

Iterating on Data and Models

Modern Plain Text Social Science
Week 07

Kieran Healy
kieran.healy@duke.edu

October 2026

Iterating on data with purrr

Load the packages, as always

```
library(here)      # manage file paths
```

here() starts at /Users/kjhealy/Documents/courses/mptc

```
library(socviz)    # data and some useful functions
library(tidyverse) # your friend and mine
```

```
— Attaching core tidyverse packages — tidyverse 2.0.0 —
✓ dplyr      1.2.1      ✓ readr      2.2.0
✓ forcats    1.0.1      ✓ stringr    1.6.0
✓ ggplot2    4.0.3      ✓ tibble     3.3.1
✓ lubridate  1.9.5      ✓ tidyr      1.3.2
✓ purrr      1.2.2
```

```
— Conflicts — tidyverse_conflicts() —
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()     masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

Split
Apply
Combine

The Core Idea

Split or group the data into pieces

Apply a function, i.e. act on the data

Combine the results back into a table

The Core Idea

Split or group the data into pieces

Apply a function, i.e. act on the data

Combine the results back into a table

Sometimes we start out in a “split” state, as when we have multiple data files, and we need to read them in and then combine them.

More than one data file

Inside `files/examples/` is a folder named `congress/`

```
# A little trick from the fs package:  
fs::dir_tree(here("files", "examples", "congress"))
```

```
/Users/kjhealy/Documents/courses/mptc/files/examples/congress
```

```
├── 01_79_congress.csv  
├── 02_80_congress.csv  
├── 03_81_congress.csv  
├── 04_82_congress.csv  
├── 05_83_congress.csv  
├── 06_84_congress.csv  
├── 07_85_congress.csv  
├── 08_86_congress.csv  
├── 09_87_congress.csv  
├── 10_88_congress.csv  
├── 11_89_congress.csv  
├── 12_90_congress.csv  
├── 13_91_congress.csv  
├── 14_92_congress.csv  
├── 15_93_congress.csv  
├── 16_94_congress.csv  
├── 17_95_congress.csv  
├── 18_96_congress.csv  
├── 19_97_congress.csv  
├── 20_98_congress.csv  
├── 21_99_congress.csv  
├── 22_100_congress.csv  
├── 23_101_congress.csv  
└── 24_102_congress.csv
```

More than one data file

Let's look at one.

```
read_csv(here("files", "examples", "congress", "17_95_congress.csv")) >
  janitor::clean_names() >
  head()

# A tibble: 6 × 25
  last      first  middle suffix nickname born  death sex  position party state
  <chr>    <chr>   <chr>  <chr>   <chr>   <chr> <chr> <chr> <chr>   <chr> <chr>
1 Abdnor   James   <NA>   <NA>   <NA>    02/1... 11/0... M    U.S. Re... Repu... SD
2 Abourezk James   George <NA>   <NA>    02/2... <NA>   M    U.S. Se... Demo... SD
3 Adams    Brockm... <NA>   <NA>   Brock   01/1... 09/1... M    U.S. Re... Demo... WA
4 Addabbo  Joseph  Patri... <NA>   <NA>    03/1... 04/1... M    U.S. Re... Demo... NY
5 Aiken    George  David  <NA>   <NA>    08/2... 11/1... M    U.S. Se... Repu... VT
6 Akaka    Daniel  Kahik... <NA>   <NA>    09/1... 04/0... M    U.S. Re... Demo... HI
# i 14 more variables: district <chr>, start <chr>, end <chr>, religion <chr>,
#   race <chr>, educational_attainment <chr>, job_type1 <chr>, job_type2 <chr>,
#   job_type3 <chr>, job_type4 <chr>, job_type5 <lgl>, mil1 <chr>, mil2 <chr>,
#   mil3 <chr>
```

We often find ourselves in this situation. We know each file has the same structure, and we would like to use them all at once.

Loops?

How to read them all in?

One traditional way, which we could do in R, is to write an explicit *loop* that iterated over a vector of filenames, read each file, and then stacked the results all together in a tall/long rectangle.

```
## Pseudocode (i.e. will not really run)
## Also, if you do write loops, do not use them to grow dataframes in this way.
filenames <- c("01_79_congress.csv", "02_80_congress.csv", "03_81_congress.csv",
               "04_82_congress.csv" [etc etc])

collected_files <- NULL

for(i in 1:length(filenames)) {
  new_file <- read_file(filenames[i])
  collected_files <- append_to(collected_files, new_files)
}
```

A **for** loop initializes a counter, a variable, and then loops around and around, doing the thing and incrementing the counter until the counter reaches a predefined maximum value. A **while** loop keeps going indefinitely until some condition is met.

Loops?

You may have noticed we have not written any loops, however.

While loops are still lurking there underneath the surface, what we will do instead is to take advantage of the combination of functions and vectors (or tables) and *apply* or *map* one to the other in order to generate results.

Speaking loosely, think of `map()` as a way of *iterating* without explicitly writing loops. You start with a vector or list of something. You feed it one element at a time to some function. The function does whatever it does. You get back output that is the same length as your input, and of a specific type.

The reason we do this is that it leads to  code we can  pipeline as a  sequence of  function calls.

Remember

Functions take actions on objects.

They take *in* objects of a certain kind (tibbles, model objects, other classes of list, vectors of various types)

They *do something* to those objects, or pieces of them

They may *return* objects of the same or different kinds

Our Goal

We want some abstractions that allow us to do things repeatedly to objects, or parts of objects, in a way that may depend on one or more variables changing as we go.

We want to know with confidence what kinds of output the actions we take will result in, and whether it succeeded.

We want to be able to manage these pieces and the actions we take on them in an efficient, tidy way.

Mapping is just a kind of iteration

Mapping is just a kind of iteration

The `purrr` package provides a big family of mapping functions. One reason there are a lot of them is that `purrr`, like the rest of the tidyverse, is picky about data types.

Mapping is just a kind of iteration

The `purrr` package provides a big family of mapping functions. One reason there are a lot of them is that `purrr`, like the rest of the tidyverse, is picky about data types.

So in addition to the basic `map()`, which always returns a *list*, we also have `map_chr()`, `map_int()`, `map_dbl()`, `map_lgl()` and others.

Mapping is just a kind of iteration

The `purrr` package provides a big family of mapping functions. One reason there are a lot of them is that `purrr`, like the rest of the tidyverse, is picky about data types.

So in addition to the basic `map()`, which always returns a *list*, we also have `map_chr()`, `map_int()`, `map_dbl()`, `map_lgl()` and others.

There are also variants for mapping over multiple inputs at once, like `map2()`, for when we are passing two arguments to some function, and `pmap()`, when we want to pass a list of any number of arguments.

Mapping is just a kind of iteration

The `purrr` package provides a big family of mapping functions. One reason there are a lot of them is that `purrr`, like the rest of the tidyverse, is picky about data types.

So in addition to the basic `map()`, which always returns a *list*, we also have `map_chr()`, `map_int()`, `map_dbl()`, `map_lgl()` and others.

There are also variants for mapping over multiple inputs at once, like `map2()`, for when we are passing two arguments to some function, and `pmap()`, when we want to pass a list of any number of arguments.

The first part of the name tells you how many arguments are passing (1, 2, many). The part after the underscore tells you what type of output to expect. If there's no suffix, you get a list back. These functions return their promised type or die trying.

Vectorized arithmetic again

The simplest cases are not that different from the vectorized arithmetic we're already familiar with.

```
a ← c(1, 3, 5, 7, 9)
```

```
b ← 1
```

```
# You know what R will do here
```

```
a + b
```

```
[1] 2 4 6 8 10
```

Vectorized arithmetic again

The simplest cases are not that different from the vectorized arithmetic we're already familiar with.

```
a ← c(1, 3, 5, 7, 9)
```

```
b ← 1
```

```
# You know what R will do here
```

```
a + b
```

```
[1] 2 4 6 8 10
```

R's vectorized rules add **b** to every element of **a**. The **+** operation can be thought of as a function that takes each element of **a** and does something with it. In this case “add **b**”. This is already a kind of iteration, in the sense that the operation recycles **b** down the vector **a** (unless they are the same length).

Vectorized arithmetic again

We can make this explicit by writing a function:

```
add_b ← function(x) {  
  b ← 1 # fixed  
  x + b # for any valid x  
}
```

Vectorized arithmetic again

We can make this explicit by writing a function:

```
add_b ← function(x) {  
  b ← 1  
  x + b # for any x  
}
```

Now:

```
add_b(x = a)
```

```
[1]  2  4  6  8 10
```

Vectorized arithmetic again

Again, R's vectorized approach means it automatically adds **b** to every element of the **x** we give it.

```
add_b(x = 10)
```

```
[1] 11
```

```
add_b(x = c(1, 99, 1000))
```

```
[1]    2  100 1001
```

Iterating in a pipeline

Some operations can't directly be vectorized in this way, which is why we need to manually iterate, which will make us want to write loops.

```
library(gapminder)

# Not what we want in this case
gapminder > n_distinct()
```

```
[1] 1704
```

When you find yourself saying “No, I meant *for each* ...”

```
gapminder >
  summarize(country_n = n_distinct(country),
            continent_n = n_distinct(continent),
            year_n = n_distinct(year),
            lifeExp_n = n_distinct(lifeExp),
            population_n = n_distinct(population))
```

```
# A tibble: 1 × 5
  country_n continent_n year_n lifeExp_n population_n
  <int>      <int>    <int>    <int>      <int>
1      142         5      12     1626      4060
```

That's tedious to write! Computers are meant to let us avoid this sort of thing.

Iterating in a pipeline

So how would we iterate this? What we want is to apply the `n_distinct()` function to each column of `gapminder`, but in a way that still allows us to use pipelines and so on. Here we leave out the names so we can see that the result is literally the same function being applied to each column.

```
library(gapminder)
gapminder >
  summarize(n_distinct(country),
            n_distinct(continent),
            n_distinct(year),
            n_distinct(lifeExp),
            n_distinct(population))
```

A tibble: 1 × 5

	`n_distinct(country)`	`n_distinct(continent)`	`n_distinct(year)`	
	<int>	<int>	<int>	
1	142	5	12	

i 2 more variables: `n\_distinct(lifeExp)` <int>,
`n\_distinct(population)` <int>

Using `n_distinct()` in this context is an idea I got from Rebecca Barter's discussion of `purrr`.

Iterating in a pipeline

Now, in practice you'd use `across()`, like this:

```
gapminder >
  summarize(across(everything(), n_distinct))
```

A tibble: 1 × 6

	country	continent	year	lifeExp	pop	gdpPercap
	<int>	<int>	<int>	<int>	<int>	<int>
1	142	5	12	1626	1704	1704

In this context, the function is applied across the specified columns; you're always getting a tibble back; and the names are taken care of automatically.

Iterating in a pipeline

But you could also write this:

```
map(gapminder, n_distinct)
```

```
$country  
[1] 142  
  
$continent  
[1] 5  
  
$year  
[1] 12  
  
$lifeExp  
[1] 1626  
  
$pop  
[1] 1704  
  
$gdpPercap  
[1] 1704
```

Read it as “Feed each column of `gapminder` to the `n_distinct()` function.

(This is pretty much what `across()` is doing more nicely.)

Iterating in a pipeline

Or, in pipeline form:

You can see we are getting a *list* back.

```
gapminder ►  
  map(n_distinct)
```

```
$country  
[1] 142
```

```
$continent  
[1] 5
```

```
$year  
[1] 12
```

```
$lifeExp  
[1] 1626
```

```
$pop  
[1] 1704
```

```
$gdpPercap  
[1] 1704
```

Iterating in a pipeline

Or, in pipeline form:

```
result ← gapminder ►  
  map(n_distinct)  
  
class(result)
```

```
[1] "list"
```

```
result$continent
```

```
[1] 5
```

```
result[[2]]
```

```
[1] 5
```

Iterating in a pipeline

But we know `n_distinct()` should always return an integer. We just want a named vector of integers back, not a list of them. So we use `map_int()` instead of the generic `map()`.

```
gapminder ►  
map_int(n_distinct)
```

country	continent	year	lifeExp	pop	gdpPercap
142	5	12	1626	1704	1704

The `map()` family can deal with all kinds of input types and output types. We choose some specific variant of `map` based on two things: what (and how many) things we are feeding the function we're mapping, and what type of thing it should return.

Get a vector of **filenames**

```
filenames ← fs::dir_ls(path = here("files", "examples", "congress"),  
  glob = "*.csv")
```

```
filenames[1:15] # Just displaying the first 15, to save slide space
```

```
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/01_79_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/02_80_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/03_81_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/04_82_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/05_83_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/06_84_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/07_85_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/08_86_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/09_87_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/10_88_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/11_89_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/12_90_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/13_91_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/14_92_congress.csv  
/Users/kjhealy/Documents/courses/mptc/files/examples/congress/15_93_congress.csv
```

And feed it to `read_csv()`

... using `map()` and binding the resulting list into a tibble.

```
df <- filenames >
map(read_csv) >
# This function takes in a list and binds its
# elements by row returning a tibble
list_rbind(names_to = "congress") >
janitor::clean_names()
```

df

A tibble: 20,580 × 26

	congress	last	first	middle	suffix	nickname	born	death	sex	position	party
	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
1	/Users/k...	Aber...	Thom...	Gerst...	<NA>	<NA>	05/1...	01/2...	M	U.S. Re...	Demo...
2	/Users/k...	Adams	Sher...	<NA>	<NA>	<NA>	01/0...	10/2...	M	U.S. Re...	Repu...
3	/Users/k...	Aiken	Geor...	David	<NA>	<NA>	08/2...	11/1...	M	U.S. Se...	Repu...
4	/Users/k...	Allen	Asa	Leona...	<NA>	<NA>	01/0...	01/0...	M	U.S. Re...	Demo...
5	/Users/k...	Allen	Leo	Elwood	<NA>	<NA>	10/0...	01/1...	M	U.S. Re...	Repu...
6	/Users/k...	Almo...	J.	Linds...	Jr.	<NA>	06/1...	04/1...	M	U.S. Re...	Demo...
7	/Users/k...	Ande...	Herm...	Carl	<NA>	<NA>	01/2...	07/2...	M	U.S. Re...	Repu...
8	/Users/k...	Ande...	Clin...	Presba	<NA>	<NA>	10/2...	11/1...	M	U.S. Re...	Demo...
9	/Users/k...	Ande...	John	Zuing...	<NA>	<NA>	03/2...	02/0...	M	U.S. Re...	Repu...
10	/Users/k...	Andr...	Augu...	Herman	<NA>	<NA>	10/1...	01/1...	M	U.S. Re...	Repu...

i 20,570 more rows

i 15 more variables: state <chr>, district <chr>, start <chr>, end <chr>,
religion <chr>, race <chr>, educational_attainment <chr>, job_type1 <chr>,
job_type2 <chr>, job_type3 <chr>, job_type4 <chr>, job_type5 <chr>,
mil1 <chr>, mil2 <chr>, mil3 <chr>

Now witness the firepower of this fully armed and operational



method of type-safe functional iteration

read_csv() can do this directly

In fact `map()` is not required for this particular use-case. We want to do this often enough that it's built-in to all the `read_` functions:

```
tmp <- read_csv(filenames, id = "path",
               name_repair = janitor::make_clean_names)

tmp >
  mutate(congress = stringr::str_extract(path, "_\\d{2,3}_congress"),
         congress = stringr::str_extract(congress, "\\d{2,3}")) >
  relocate(congress)
```

```
# A tibble: 20,580 × 27
  congress path    last first middle suffix nickname born death sex position
  <chr>    <chr>   <chr> <chr> <chr>  <chr>  <chr>   <chr> <chr> <chr>  <chr>
1 79      /User... Aber... Thom... Gerst... <NA>   <NA>   05/1... 01/2... M    U.S. Re...
2 79      /User... Adams Sher... <NA>   <NA>   <NA>   01/0... 10/2... M    U.S. Re...
3 79      /User... Aiken Geor... David  <NA>   <NA>   08/2... 11/1... M    U.S. Se...
4 79      /User... Allen Asa   Leona... <NA>   <NA>   01/0... 01/0... M    U.S. Re...
5 79      /User... Allen Leo   Elwood <NA>   <NA>   10/0... 01/1... M    U.S. Re...
6 79      /User... Almo... J.     Linds... Jr.    <NA>   <NA>   06/1... 04/1... M    U.S. Re...
7 79      /User... Ande... Herm... Carl   <NA>   <NA>   01/2... 07/2... M    U.S. Re...
8 79      /User... Ande... Clin... Presba <NA>   <NA>   10/2... 11/1... M    U.S. Re...
9 79      /User... Ande... John  Zuing... <NA>   <NA>   03/2... 02/0... M    U.S. Re...
10 79     /User... Andr... Augu... Herman <NA>   <NA>   10/1... 01/1... M    U.S. Re...

# i 20,570 more rows
# i 16 more variables: party <chr>, state <chr>, district <chr>, start <chr>,
#   end <chr>, religion <chr>, race <chr>, educational_attainment <chr>,
#   job_type1 <chr>, job_type2 <chr>, job_type3 <chr>, job_type4 <chr>,
#   job_type5 <chr>, mil1 <chr>, mil2 <chr>, mil3 <chr>
```

Wrangling Models

This is not a **statistics** seminar!

I'll just give you an example of the sort of thing that many other modeling packages implement for all kinds of modeling techniques.

Again, the principle is tidy incorporation of models and their output.

Which means, as always: tibbles. We want a *data frame* of our results. A nice rectangular table that we know what to do with regardless of what its contents mean.

Tidy regression output with **broom**

```
library(broom)  
library(gapminder)
```

```
out ← lm(formula = lifeExp ~ gdpPercap + log(pop) + continent,  
          data = gapminder)
```

Tidy regression output with **broom**

```
summary(out)
```

Call:

```
lm(formula = lifeExp ~ gdpPercap + log(pop) + continent, data = gapminder)
```

Residuals:

Min	1Q	Median	3Q	Max
-47.490	-4.614	0.250	5.293	26.094

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	3.816e+01	2.050e+00	18.618	< 2e-16 ***
gdpPercap	4.557e-04	2.345e-05	19.435	< 2e-16 ***
log(pop)	6.394e-01	1.329e-01	4.810	1.64e-06 ***
continentAmericas	1.308e+01	6.063e-01	21.579	< 2e-16 ***
continentAsia	7.784e+00	5.810e-01	13.398	< 2e-16 ***
continentEurope	1.695e+01	6.350e-01	26.691	< 2e-16 ***
continentOceania	1.764e+01	1.779e+00	9.916	< 2e-16 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

We can't do anything with this output, programmatically. A summary like this is essentially a *printout*, once again a legacy of the teletype.



Look inside the box

総本山金剛峯寺御用達



Poke around inside

What about the object created by `summary()`?

```
out
├── coefficients
├── residuals
├── effects
├── rank
├── qr
│   ├── qr
│   ├── pivot
│   ├── graux
│   ├── tol
│   └── rank
├── df.residual
├── contrasts
├── xlevels
├── call
├── terms
└── model.frame
```

Use the Object Inspector to take a look

There's a bunch of different stuff in here. It's a little awkward to work with directly all the time.

Tidy regression output with **broom**

```
library(broom)
```

```
tidy(out)
```

```
# A tibble: 7 × 5
```

	term <chr>	estimate <dbl>	std.error <dbl>	statistic <dbl>	p.value <dbl>
1	(Intercept)	38.2	2.05	18.6	1.50e- 70
2	gdpPercap	0.000456	0.0000234	19.4	3.98e- 76
3	log(pop)	0.639	0.133	4.81	1.64e- 6
4	continentAmericas	13.1	0.606	21.6	1.85e- 91
5	continentAsia	7.78	0.581	13.4	5.52e- 39
6	continentEurope	16.9	0.635	26.7	2.43e-131
7	continentOceania	17.6	1.78	9.92	1.43e- 22

That's a *lot* nicer. Now it's just a tibble. We know those. This is only a small piece of what's in the summary object, but we'll get to the rest in a minute.

Tidy regression output with **broom**

`tidy()` has some optional arguments.

```
out_conf ← tidy(out, conf.int = TRUE)
out_conf
```

```
# A tibble: 7 × 7
```

	term <chr>	estimate <dbl>	std.error <dbl>	statistic <dbl>	p.value <dbl>	conf.low <dbl>	conf.high <dbl>
1	(Intercept)	38.2	2.05	18.6	1.50e- 70	34.1	42.2
2	gdpPercap	0.000456	0.0000234	19.4	3.98e- 76	0.000410	0.000502
3	log(pop)	0.639	0.133	4.81	1.64e- 6	0.379	0.900
4	continentAmericas	13.1	0.606	21.6	1.85e- 91	11.9	14.3
5	continentAsia	7.78	0.581	13.4	5.52e- 39	6.64	8.92
6	continentEurope	16.9	0.635	26.7	2.43e-131	15.7	18.2
7	continentOceania	17.6	1.78	9.92	1.43e- 22	14.2	21.1

Tidy regression output with **broom**

```
out_conf >
  filter(term %nin% "(Intercept)") >
  mutate(nicelabs = prefix_strip(term, "continent")) >
  select(nicelabs, everything())
```

```
# A tibble: 6 × 8
  nicelabs term      estimate std.error statistic  p.value conf.low conf.high
  <chr>    <chr>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 gdpPercap gdpPercap  4.56e-4 0.0000234 19.4 3.98e- 76 4.10e-4 0.000502
2 log(Pop) log(pop)    6.39e-1 0.133      4.81 1.64e- 6 3.79e-1 0.900
3 Americas continent... 1.31e+1 0.606     21.6 1.85e- 91 1.19e+1 14.3
4 Asia      continent... 7.78e+0 0.581     13.4 5.52e- 39 6.64e+0 8.92
5 Europe    continent... 1.69e+1 0.635     26.7 2.43e-131 1.57e+1 18.2
6 Oceania   continent... 1.76e+1 1.78       9.92 1.43e- 22 1.42e+1 21.1
```

Three ways to tidy

Component level: `tidy()`

Three ways to tidy

Component level: `tidy()`

Observation level: `augment()`

Three ways to tidy

Component level: `tidy()`

Observation level: `augment()`

Model level: `glance()`

Model output again

```
> summary(out)
```

Call:

```
lm(formula = lifeExp ~ gdpPercap + log(pop) + continent, data = gapminder)
```

Residuals:

Min	1Q	Median	3Q	Max
-47.490	-4.614	0.250	5.293	26.094

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	3.816e+01	2.050e+00	18.618	< 2e-16 **
gdpPercap	4.557e-04	2.345e-05	19.435	< 2e-16 **
log(pop)	6.394e-01	1.329e-01	4.810	1.64e-06 **
continentAmericas	1.308e+01	6.063e-01	21.579	< 2e-16 **
continentAsia	7.784e+00	5.810e-01	13.398	< 2e-16 **
continentEurope	1.695e+01	6.350e-01	26.691	< 2e-16 **
continentOceania	1.764e+01	1.779e+00	9.916	< 2e-16 **

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.336 on 1697 degrees of freedom

Multiple R-squared: 0.585, Adjusted R-squared: 0.5835

F-statistic: 398.7 on 6 and 1697 DF, p-value: < 2.2e-16

Component level

```
> summary(out)
```

Call:

```
lm(formula = lifeExp ~ gdpPercap + log(pop) + continent, data = gapminder)
```

Residuals:

Min	1Q	Median	3Q	Max
-47.490	-4.614	0.250	5.293	26.094

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	3.816e+01	2.050e+00	18.618	< 2e-16	***
gdpPercap	4.557e-04	2.345e-05	19.435	< 2e-16	***
log(pop)	6.394e-01	1.329e-01	4.810	1.64e-06	***
continentAmericas	1.308e+01	6.063e-01	21.579	< 2e-16	***
continentAsia	7.784e+00	5.810e-01	13.398	< 2e-16	***
continentEurope	1.695e+01	6.350e-01	26.691	< 2e-16	***
continentOceania	1.764e+01	1.779e+00	9.916	< 2e-16	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.336 on 1697 degrees of freedom

Multiple R-squared: 0.585, Adjusted R-squared: 0.5835

F-statistic: 398.7 on 6 and 1697 DF, p-value: < 2.2e-16

Component level

```
tidy(out)
```

```
# A tibble: 7 × 5
```

	term <chr>	estimate <dbl>	std.error <dbl>	statistic <dbl>	p.value <dbl>
1	(Intercept)	38.2	2.05	18.6	1.50e- 70
2	gdpPercap	0.000456	0.0000234	19.4	3.98e- 76
3	log(pop)	0.639	0.133	4.81	1.64e- 6
4	continentAmericas	13.1	0.606	21.6	1.85e- 91
5	continentAsia	7.78	0.581	13.4	5.52e- 39
6	continentEurope	16.9	0.635	26.7	2.43e-131
7	continentOceania	17.6	1.78	9.92	1.43e- 22

Observation level

```
> summary(out)
```

Call:

```
lm(formula = lifeExp ~ gdpPercap + log(pop) + continent, data = gapminder)
```

Residuals:

Min	1Q	Median	3Q	Max
-47.490	-4.614	0.250	5.293	26.094

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	3.816e+01	2.050e+00	18.618	< 2e-16 **
gdpPercap	4.557e-04	2.345e-05	19.435	< 2e-16 **
log(pop)	6.394e-01	1.329e-01	4.810	1.64e-06 **
continentAmericas	1.308e+01	6.063e-01	21.579	< 2e-16 **
continentAsia	7.784e+00	5.810e-01	13.398	< 2e-16 **
continentEurope	1.695e+01	6.350e-01	26.691	< 2e-16 **
continentOceania	1.764e+01	1.779e+00	9.916	< 2e-16 **

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.336 on 1697 degrees of freedom

Multiple R-squared: 0.585, Adjusted R-squared: 0.5835

F-statistic: 398.7 on 6 and 1697 DF, p-value: < 2.2e-16

Observation level

```
augment(out)
```

```
# A tibble: 1,704 × 10
```

	lifeExp	gdpPercap	`log(pop)`	continent	.fitted	.resid	.hat	.sigma	.cooks
	<dbl>	<dbl>	<dbl>	<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	28.8	779.	15.9	Asia	56.5	-27.7	0.00302	8.31	0.00479
2	30.3	821.	16.0	Asia	56.6	-26.2	0.00299	8.31	0.00426
3	32.0	853.	16.1	Asia	56.7	-24.7	0.00296	8.32	0.00372
4	34.0	836.	16.3	Asia	56.7	-22.7	0.00294	8.32	0.00313
5	36.1	740.	16.4	Asia	56.8	-20.7	0.00294	8.32	0.00259
6	38.4	786.	16.5	Asia	56.9	-18.4	0.00292	8.33	0.00205
7	39.9	978.	16.4	Asia	56.9	-17.0	0.00291	8.33	0.00174
8	40.8	852.	16.4	Asia	56.9	-16.0	0.00292	8.33	0.00155
9	41.7	649.	16.6	Asia	56.9	-15.2	0.00294	8.33	0.00140
10	41.8	635.	16.9	Asia	57.1	-15.3	0.00297	8.33	0.00144

```
# i 1,694 more rows
```

```
# i 1 more variable: .std.resid <dbl>
```

Observation level

For OLS models:

`.fitted` — The fitted values of the model.

`.se.fit` — The standard errors of the fitted values.

`.resid` — The residuals.

`.hat` — The diagonal of the hat matrix.

`.sigma` — An estimate of the residual standard deviation when the corresponding observation is dropped from the model.

`.cooks` — Cook's distance, a common regression diagnostic.

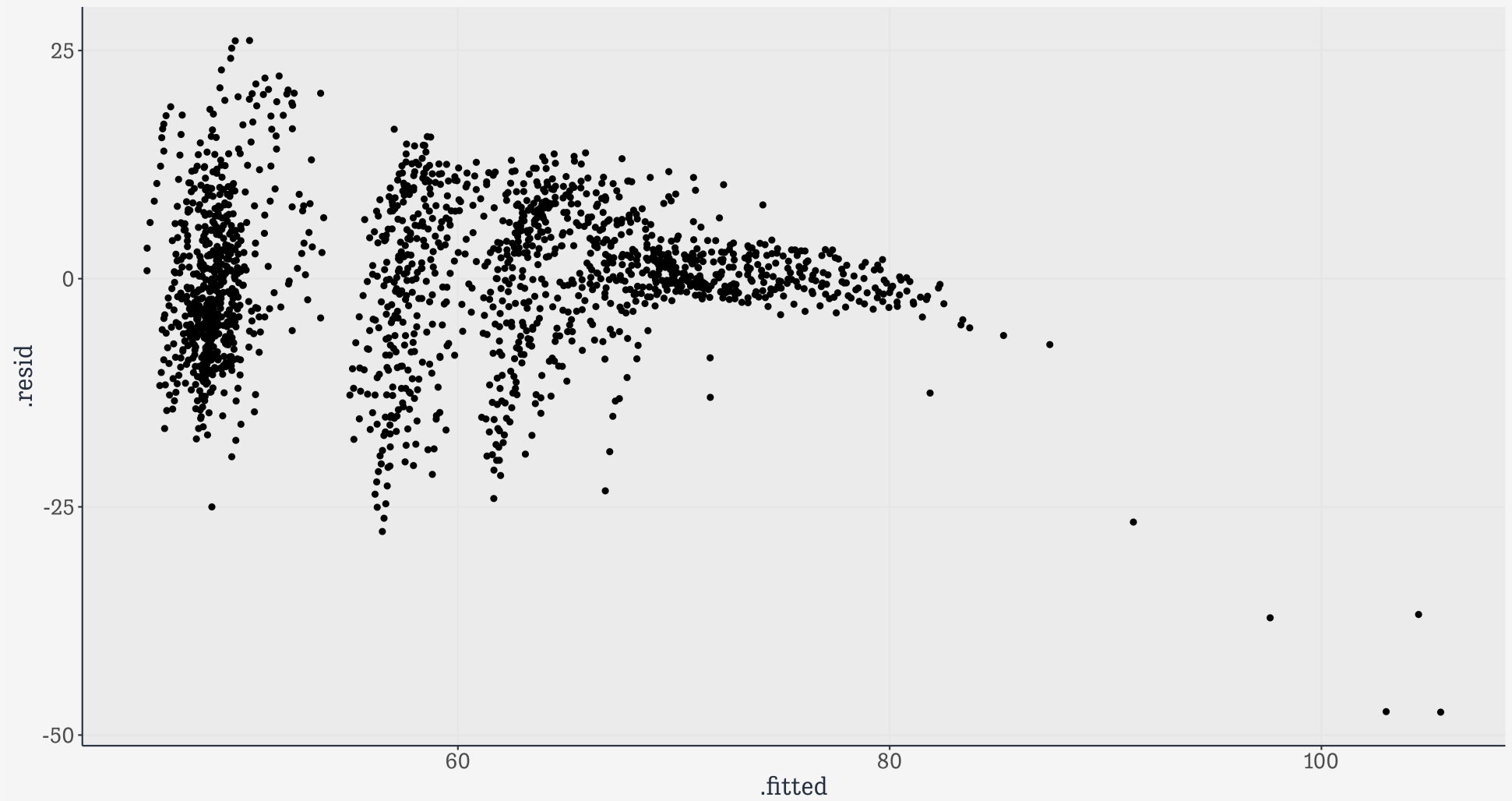
`.std.resid` — The standardized residuals.

Observation level

```
# Adding the data argument puts back any additional columns from the original  
# tibble  
out_aug ← augment(out, data = gapminder)  
head(out_aug)
```

```
# A tibble: 6 × 12  
  country continent  year lifeExp    pop gdpPercap .fitted .resid    .hat .sigma  
  <fct>    <fct>    <int>  <dbl>  <int>    <dbl>   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  
1 Afghan... Asia      1952   28.8 8.43e6    779.    56.5  -27.7 0.00302  8.31  
2 Afghan... Asia      1957   30.3 9.24e6    821.    56.6  -26.2 0.00299  8.31  
3 Afghan... Asia      1962   32.0 1.03e7    853.    56.7  -24.7 0.00296  8.32  
4 Afghan... Asia      1967   34.0 1.15e7    836.    56.7  -22.7 0.00294  8.32  
5 Afghan... Asia      1972   36.1 1.31e7    740.    56.8  -20.7 0.00294  8.32  
6 Afghan... Asia      1977   38.4 1.49e7    786.    56.9  -18.4 0.00292  8.33  
# i 2 more variables: .cooksd <dbl>, .std.resid <dbl>
```

```
## Residuals vs Fitted Values  
p ← ggplot(data = out_aug,  
           mapping = aes(x = .fitted, y = .resid))  
p_out ← p + geom_point()
```



(I told you it was misspecified)

Model level

```
> summary(out)
```

Call:

```
lm(formula = lifeExp ~ gdpPercap + log(pop) + continent, data = gapminder)
```

Residuals:

Min	1Q	Median	3Q	Max
-47.490	-4.614	0.250	5.293	26.094

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	3.816e+01	2.050e+00	18.618	< 2e-16 **
gdpPercap	4.557e-04	2.345e-05	19.435	< 2e-16 **
log(pop)	6.394e-01	1.329e-01	4.810	1.64e-06 **
continentAmericas	1.308e+01	6.063e-01	21.579	< 2e-16 **
continentAsia	7.784e+00	5.810e-01	13.398	< 2e-16 **
continentEurope	1.695e+01	6.350e-01	26.691	< 2e-16 **
continentOceania	1.764e+01	1.779e+00	9.916	< 2e-16 **

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.336 on 1697 degrees of freedom

Multiple R-squared: 0.585, Adjusted R-squared: 0.5835

F-statistic: 398.7 on 6 and 1697 DF, p-value: $\leq 2.2e-16$

Model level

```
glance(out)
```

```
# A tibble: 1 × 12
  r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC    BIC
  <dbl>      <dbl> <dbl>      <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>
1   0.585      0.584  8.34       399. 1.01e-319     6 -6028. 12072. 12115.
# i 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

The usefulness of `glance()` becomes clearer when dealing with ensembles of models.

Grouped Analysis

Grouped analysis and **list columns**

European countries in 1977:

```
gapminder >  
  filter(continent == "Europe", year == 1977)
```

```
# A tibble: 30 × 6
```

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Albania	Europe	1977	68.9	2509048	3533.
2	Austria	Europe	1977	72.2	7568430	19749.
3	Belgium	Europe	1977	72.8	9821800	19118.
4	Bosnia and Herzegovina	Europe	1977	69.9	4086000	3528.
5	Bulgaria	Europe	1977	70.8	8797022	7612.
6	Croatia	Europe	1977	70.6	4318673	11305.
7	Czech Republic	Europe	1977	70.7	10161915	14800.
8	Denmark	Europe	1977	74.7	5088419	20423.
9	Finland	Europe	1977	72.5	4738902	15605.
10	France	Europe	1977	73.8	53165019	18293.

```
# i 20 more rows
```

Let's fit a simple model predicting life expectancy from GDP per capita for just these countries.

Grouped analysis and list columns

```
eu77 <- gapminder > filter(continent = "Europe", year = 1977)
fit <- lm(lifeExp ~ log(gdpPercap), data = eu77)
```

```
summary(fit)
```

Call:

```
lm(formula = lifeExp ~ log(gdpPercap), data = eu77)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-7.4956	-1.0306	0.0935	1.1755	3.7125

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	29.489	7.161	4.118	0.000306	**
log(gdpPercap)	4.488	0.756	5.936	2.17e-06	**

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.114 on 28 degrees of freedom

Multiple R-squared: 0.5572, Adjusted R-squared: 0.5414

F-statistic: 35.24 on 1 and 28 DF, p-value: 2.173e-06

What if we want to do this for every continent in every year?

Grouped analysis and **list columns**

```
df_nest ← gapminder ▷  
  group_by(continent, year) ▷  
  nest()
```

```
df_nest
```

```
# A tibble: 60 × 3  
# Groups:   continent, year [60]  
  continent year data  
  <fct>      <int> <list>  
1 Asia      1952 <tibble [33 × 4]>  
2 Asia      1957 <tibble [33 × 4]>  
3 Asia      1962 <tibble [33 × 4]>  
4 Asia      1967 <tibble [33 × 4]>  
5 Asia      1972 <tibble [33 × 4]>  
6 Asia      1977 <tibble [33 × 4]>  
7 Asia      1982 <tibble [33 × 4]>  
8 Asia      1987 <tibble [33 × 4]>  
9 Asia      1992 <tibble [33 × 4]>  
10 Asia     1997 <tibble [33 × 4]>  
# i 50 more rows
```

Think of nesting as a kind of “super-grouping”. Look in the object inspector.

Grouped analysis and **list columns**

It's all still in there.

```
df_nest > filter(continent = "Europe" & year = 1977) >  
  unnest(cols = c(data))
```

```
# A tibble: 30 × 6
```

```
# Groups:   continent, year [1]
```

	continent	year	country	lifeExp	pop	gdpPercap
	<fct>	<int>	<fct>	<dbl>	<int>	<dbl>
1	Europe	1977	Albania	68.9	2509048	3533.
2	Europe	1977	Austria	72.2	7568430	19749.
3	Europe	1977	Belgium	72.8	9821800	19118.
4	Europe	1977	Bosnia and Herzegovina	69.9	4086000	3528.
5	Europe	1977	Bulgaria	70.8	8797022	7612.
6	Europe	1977	Croatia	70.6	4318673	11305.
7	Europe	1977	Czech Republic	70.7	10161915	14800.
8	Europe	1977	Denmark	74.7	5088419	20423.
9	Europe	1977	Finland	72.5	4738902	15605.
10	Europe	1977	France	73.8	53165019	18293.

```
# i 20 more rows
```

Grouped analysis and **list columns**

Here we **map()** a custom function to every row in the **data** column.

```
fit_ols ← function(df) {  
  lm(lifeExp ~ log(gdpPercap), data = df)  
}  
  
df_nest ← gapminder ▷  
  group_by(continent, year) ▷  
  nest() ▷  
  mutate(model = map(data, fit_ols))
```

Grouped analysis and **list columns**

```
df_nest
```

```
# A tibble: 60 × 4
# Groups:   continent, year [60]
  continent year data          model
  <fct>      <int> <list>          <list>
1 Asia      1952 <tibble [33 × 4]> <lm>
2 Asia      1957 <tibble [33 × 4]> <lm>
3 Asia      1962 <tibble [33 × 4]> <lm>
4 Asia      1967 <tibble [33 × 4]> <lm>
5 Asia      1972 <tibble [33 × 4]> <lm>
6 Asia      1977 <tibble [33 × 4]> <lm>
7 Asia      1982 <tibble [33 × 4]> <lm>
8 Asia      1987 <tibble [33 × 4]> <lm>
9 Asia      1992 <tibble [33 × 4]> <lm>
10 Asia     1997 <tibble [33 × 4]> <lm>
# i 50 more rows
```

Grouped analysis and **list columns**

We can tidy the nested models, too.

```
fit_ols <- function(df) {  
  lm(lifeExp ~ log(gdpPercap), data = df)  
}  
  
out_tidy <- gapminder >  
  group_by(continent, year) >  
  nest() >  
  mutate(model = map(data, fit_ols),  
         tidied = map(model, tidy)) >  
  unnest(cols = c(tidied)) >  
  filter(term %nin% "(Intercept)" &  
         continent %nin% "Oceania")
```

Grouped analysis and list columns

```
out_tidy
```

```
# A tibble: 48 × 9
```

```
# Groups:   continent, year [48]
```

	continent	year	data	model	term	estimate	std.error	statistic	p.value
	<fct>	<int>	<list>	<list>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	Asia	1952	<tibble>	<lm>	log(gdp...	4.16	1.25	3.33	2.28e-3
2	Asia	1957	<tibble>	<lm>	log(gdp...	4.17	1.28	3.26	2.71e-3
3	Asia	1962	<tibble>	<lm>	log(gdp...	4.59	1.24	3.72	7.94e-4
4	Asia	1967	<tibble>	<lm>	log(gdp...	4.50	1.15	3.90	4.77e-4
5	Asia	1972	<tibble>	<lm>	log(gdp...	4.44	1.01	4.41	1.16e-4
6	Asia	1977	<tibble>	<lm>	log(gdp...	4.87	1.03	4.75	4.42e-5
7	Asia	1982	<tibble>	<lm>	log(gdp...	4.78	0.852	5.61	3.77e-6
8	Asia	1987	<tibble>	<lm>	log(gdp...	5.17	0.727	7.12	5.31e-8
9	Asia	1992	<tibble>	<lm>	log(gdp...	5.09	0.649	7.84	7.60e-9
10	Asia	1997	<tibble>	<lm>	log(gdp...	5.11	0.628	8.15	3.35e-9

```
# i 38 more rows
```

Grouped analysis and **list columns**

```
out_tidy ▷  
  ungroup() ▷  
  sample_n(5)
```

```
# A tibble: 5 × 9
```

	continent	year	data	model	term	estimate	std.error	statistic	p.value
	<fct>	<int>	<list>	<list>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	Africa	1982	<tibble>	<lm>	log(gdpP...	5.61	0.898	6.25	8.79e-8
2	Americas	1987	<tibble>	<lm>	log(gdpP...	7.10	1.14	6.25	2.22e-6
3	Europe	1967	<tibble>	<lm>	log(gdpP...	5.43	0.796	6.82	2.09e-7
4	Europe	1957	<tibble>	<lm>	log(gdpP...	7.45	0.945	7.88	1.39e-8
5	Africa	1977	<tibble>	<lm>	log(gdpP...	4.51	0.920	4.90	1.04e-5

Example: Iterating on the US Census

Iterating on the US Census

Mapped iteration is very general, and not just for local files

```
## Register for a free Census API key  
library(tidycensus)
```

One county in one year:

```
out ← get_acs(geography = "county",  
              variables = "B19013_001", # Median HH income  
              state = "NY",  
              county = "New York",  
              survey = "acs1",  
              year = 2005)
```

out

```
# A tibble: 1 × 5  
  GEOID NAME          variable estimate moe  
  <chr> <chr>          <chr>      <dbl> <dbl>  
1 36061 New York County, New York B19013_001    55973  1462
```

Iterating on the US Census

We can get all counties in New York State for a specific year:

```
out ← get_acs(geography = "county",
              variables = "B19013_001",
              state = "NY",
              survey = "acs1",
              year = 2005)
```

out

```
# A tibble: 38 × 5
  GEOID NAME          variable estimate moe
  <chr> <chr>          <chr>      <dbl> <dbl>
1 36001 Albany County, New York B19013_001 50054 2030
2 36005 Bronx County, New York  B19013_001 29228  853
3 36007 Broome County, New York B19013_001 36394 2340
4 36009 Cattaraugus County, New York B19013_001 37580 2282
5 36011 Cayuga County, New York  B19013_001 42057 2406
6 36013 Chautauqua County, New York B19013_001 35495 2077
7 36015 Chemung County, New York  B19013_001 37418 3143
8 36019 Clinton County, New York  B19013_001 44757 3500
9 36027 Dutchess County, New York B19013_001 61889 2431
10 36029 Erie County, New York   B19013_001 41967 1231
# i 28 more rows
```

Iterating on the US Census

What if we want the results for *every* available year? The `get_acs()` function can't do this. We'll need to repeatedly apply it.

First, a handy function: `set_names()`

```
x ← c(1:10)
```

```
x
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
x ← set_names(x, nm = letters[1:10])
```

```
x
```

```
a b c d e f g h i j  
1 2 3 4 5 6 7 8 9 10
```

Iterating on the US Census

By default, `set_names()` will label a vector with that vector's values:

```
c(1:10) ►  
  set_names()
```

```
1  2  3  4  5  6  7  8  9 10  
1  2  3  4  5  6  7  8  9 10
```

Iterating on the US Census

This works with `map()` just fine:

```
df <- 2005:2019 >
  map(\(x) get_acs(geography = "county",
                  variables = "B19013_001",
                  state = "NY",
                  survey = "acs1",
                  year = x)) >
list_rbind(names_to = "year")
```

df

```
# A tibble: 580 × 6
  year GEOID NAME          variable estimate moe
  <int> <chr> <chr>          <chr>      <dbl> <dbl>
1     1  1 36001 Albany County, New York B19013_001  50054  2030
2     1  1 36005 Bronx County, New York   B19013_001  29228   853
3     1  1 36007 Broome County, New York  B19013_001  36394  2340
4     1  1 36009 Cattaraugus County, New York B19013_001  37580  2282
5     1  1 36011 Cayuga County, New York   B19013_001  42057  2406
6     1  1 36013 Chautauqua County, New York B19013_001  35495  2077
7     1  1 36015 Chemung County, New York   B19013_001  37418  3143
8     1  1 36019 Clinton County, New York  B19013_001  44757  3500
9     1  1 36027 Dutchess County, New York B19013_001  61889  2431
10    1  1 36029 Erie County, New York   B19013_001  41967  1231
# i 570 more rows
```

Iterating on the US Census

Our `id` column *tracks* the year. But we'd like it to *be* the year. So, we use `set_names()`:

```
df <- 2005:2019 >
  set_names() >
  map(\(x) get_acs(geography = "county",
                   variables = "B19013_001",
                   state = "NY",
                   survey = "acs1",
                   year = x)) >
  list_rbind(names_to = "year") >
  mutate(year = as.integer(year))
```

Iterating on the US Census

```
df
```

```
# A tibble: 580 × 6
  year GEOID NAME          variable estimate moe
  <int> <chr> <chr>          <chr>      <dbl> <dbl>
1  2005 36001 Albany County, New York B19013_001  50054  2030
2  2005 36005 Bronx County, New York  B19013_001  29228   853
3  2005 36007 Broome County, New York B19013_001  36394  2340
4  2005 36009 Cattaraugus County, New York B19013_001  37580  2282
5  2005 36011 Cayuga County, New York  B19013_001  42057  2406
6  2005 36013 Chautauqua County, New York B19013_001  35495  2077
7  2005 36015 Chemung County, New York  B19013_001  37418  3143
8  2005 36019 Clinton County, New York  B19013_001  44757  3500
9  2005 36027 Dutchess County, New York B19013_001  61889  2431
10 2005 36029 Erie County, New York     B19013_001  41967  1231
# i 570 more rows
```

Now **year** is just the year. The **year** column will be created as a character vector, so we converted it back to an integer again at the end.

Iterating on the US Census

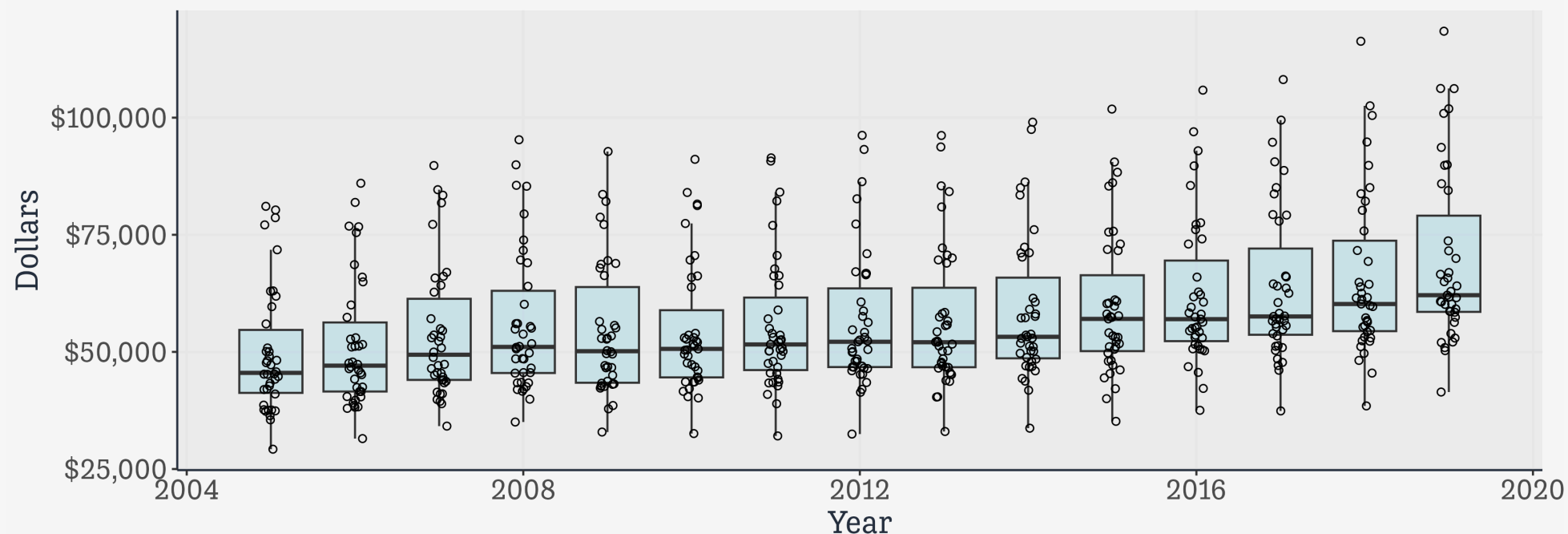
```
p_out <- 2005:2019 >
  set_names() >
  map(\(x) get_acs(geography = "county",
                  variables = "B19013_001",
                  state = "NY",
                  survey = "acs1",
                  year = x)) >
  list_rbind(names_to = "year") >
  mutate(year = as.integer(year)) >
  ggplot(mapping = aes(x = year, y = estimate, group = year)) +
  geom_boxplot(fill = "lightblue", alpha = 0.5, outlier.alpha = 0) +
  geom_jitter(position = position_jitter(width = 0.1), shape = 1) +
  scale_y_continuous(labels = scales::label_dollar()) +
  labs(x = "Year", y = "Dollars",
       title = "Median Household Income by County in New York State, 2005-2019",
       subtitle = "ACS 1-year estimates", caption = "Data: U.S. Census Bureau.")
```

Iterating on the US Census

```
print(p_out)
```

Median Household Income by County in New York State, 2005-2019

ACS 1-year estimates



Data: U.S. Census Bureau.

Midwest PCA Example

Midwest data

midwest

A tibble: 437 × 28

	PID	county	state	area	poptotal	popdensity	popwhite	popblack	popamerindian
	<int>	<chr>	<chr>	<dbl>	<int>	<dbl>	<int>	<int>	<int>
1	561	ADAMS	IL	0.052	66090	1271.	63917	1702	98
2	562	ALEXAN...	IL	0.014	10626	759	7054	3496	19
3	563	BOND	IL	0.022	14991	681.	14477	429	35
4	564	BOONE	IL	0.017	30806	1812.	29344	127	46
5	565	BROWN	IL	0.018	5836	324.	5264	547	14
6	566	BUREAU	IL	0.05	35688	714.	35157	50	65
7	567	CALHOUN	IL	0.017	5322	313.	5298	1	8
8	568	CARROLL	IL	0.027	16805	622.	16519	111	30
9	569	CASS	IL	0.024	13437	560.	13384	16	8
10	570	CHAMPA...	IL	0.058	173025	2983.	146506	16559	331

i 427 more rows

i 19 more variables: popasian <int>, popother <int>, percwhite <dbl>,
percblack <dbl>, percamerindian <dbl>, percasian <dbl>, percother <dbl>,
popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
percchildbelowpovert <dbl>, percadultpoverty <dbl>,

Extract numeric vars

```
mw_pca <- midwest %>%  
  group_by(state) %>%  
  select(where(is.numeric)) %>%  
  select(-PID)
```

```
mw_pca
```

```
# A tibble: 437 × 25
```

```
# Groups:   state [5]
```

	state	area	poptotal	popdensity	popwhite	popblack	popamerindian	popasian
	<chr>	<dbl>	<int>	<dbl>	<int>	<int>	<int>	<int>
1	IL	0.052	66090	1271.	63917	1702	98	249
2	IL	0.014	10626	759	7054	3496	19	48
3	IL	0.022	14991	681.	14477	429	35	16
4	IL	0.017	30806	1812.	29344	127	46	150
5	IL	0.018	5836	324.	5264	547	14	5
6	IL	0.05	35688	714.	35157	50	65	195
7	IL	0.017	5322	313.	5298	1	8	15
8	IL	0.027	16805	622.	16519	111	30	61
9	IL	0.024	13437	560.	13384	16	8	23
10	IL	0.058	173025	2983.	146506	16559	331	8033

```
# i 427 more rows
```

```
# i 17 more variables: popother <int>, percwhite <dbl>, percblack <dbl>,  
#   percamerindian <dbl>, percasian <dbl>, percother <dbl>, popadults <int>,  
#   perchsd <dbl>, percollege <dbl>, percprof <dbl>, poppovertyknown <int>,  
#   percpovertyknown <dbl>, percbelowpoverty <dbl>, percchildbelowpovert <dbl>,
```

PCA Helper function

We could do this anonymously, with a lambda function, too.

```
do_pca ← function(df){  
  prcomp(df,  
    center = TRUE, scale = TRUE)  
}
```

PCA on the whole dataset

```
out_pca ← mw_pca ▷  
  ungroup() ▷  
  select(-state) ▷  
  do_pca()
```

out_pca

Standard deviations (1, .., p=24):

```
[1] 3.10e+00 2.21e+00 1.65e+00 1.19e+00 1.12e+00 8.98e-01 8.86e-01 8.19e-01  
[9] 6.92e-01 5.65e-01 5.44e-01 4.85e-01 3.80e-01 3.58e-01 3.09e-01 2.50e-01  
[17] 2.09e-01 1.92e-01 9.65e-02 3.47e-02 1.33e-02 3.86e-03 2.89e-09 9.54e-16
```

Rotation (n x k) = (24 x 24):

	PC1	PC2	PC3	PC4	PC5	PC6
area	-0.02619	0.0150	-0.10362	0.2508	-0.626804	0.48689
poptotal	-0.31299	0.0515	0.11077	0.0544	0.002497	0.04058
popdensity	-0.29205	0.0236	0.04558	-0.1042	0.139053	0.15186
popwhite	-0.31410	0.0238	0.08117	0.0278	0.016318	0.07259
popblack	-0.28773	0.1074	0.14902	0.0598	0.006193	0.04880
popamerindian	-0.26919	0.0916	-0.01213	-0.1236	-0.196022	0.14136
popasian	-0.28406	0.0578	0.14522	0.2132	-0.083397	-0.16286
popother	-0.25831	0.0807	0.19651	0.2163	-0.110491	-0.26225
percwhite	0.18251	-0.1756	0.25486	0.4371	0.082341	0.07589
percblack	-0.19880	0.1119	-0.15319	-0.2542	0.343979	0.25296
percamerindian	-0.00153	0.1745	-0.18427	-0.4065	-0.522049	-0.28835
percasian	-0.19122	-0.1272	-0.33296	0.2078	0.057549	-0.06347

Tidier view, whole dataset

```
tidy_pca ← tidy(out_pca, matrix = "pcs")
```

```
tidy_pca
```

```
# A tibble: 24 × 4
```

	PC	std.dev	percent	cumulative
	<dbl>	<dbl>	<dbl>	<dbl>
1	1	3.10	0.400	0.400
2	2	2.21	0.203	0.603
3	3	1.65	0.113	0.717
4	4	1.19	0.0593	0.776
5	5	1.12	0.0524	0.829
6	6	0.898	0.0336	0.862
7	7	0.886	0.0327	0.895
8	8	0.819	0.0280	0.923
9	9	0.692	0.0200	0.943
10	10	0.565	0.0133	0.956

```
# i 14 more rows
```

Using `augment()`

This puts the original data points back in.

```
aug_pca <- augment(out_pca, data = mw_pca[,-1])
aug_pca <- aug_pca >
  tibble::add_column(midwest$state, midwest$county, .before = TRUE) >
  rename(state = `midwest$state`, county = `midwest$county`)
```

aug_pca

A tibble: 437 × 51

	state	county	.rownames	area	poptotal	popdensity	popwhite	popblack
	<chr>	<chr>	<chr>	<dbl>	<int>	<dbl>	<int>	<int>
1	IL	ADAMS	1	0.052	66090	1271.	63917	1702
2	IL	ALEXANDER	2	0.014	10626	759	7054	3496
3	IL	BOND	3	0.022	14991	681.	14477	429
4	IL	BOONE	4	0.017	30806	1812.	29344	127
5	IL	BROWN	5	0.018	5836	324.	5264	547
6	IL	BUREAU	6	0.05	35688	714.	35157	50
7	IL	CALHOUN	7	0.017	5322	313.	5298	1
8	IL	CARROLL	8	0.027	16805	622.	16519	111
9	IL	CASS	9	0.024	13437	560.	13384	16
10	IL	CHAMPAIGN	10	0.058	173025	2983.	146506	16559

i 427 more rows

i 43 more variables: popamerindian <int>, popasian <int>, popother <int>,
percwhite <dbl>, percblack <dbl>, percamerindian <dbl>, percasian <dbl>,
percother <dbl>, popadults <int>, perchsd <dbl>, percollege <dbl>,
percprof <dbl>, poppovertyknown <int>, percpovertyknown <dbl>,
percbelowpoverty <dbl>, percchildbelowpovert <dbl>, percadultpoverty <dbl>,

Grouped PCA

Let's do that grouped by State

```
mw_pca <- mw_pca >
  group_by(state) >
  nest()
```

```
mw_pca
```

```
# A tibble: 5 × 2
# Groups:   state [5]
  state data
  <chr> <list>
1 IL    <tibble [102 × 24]>
2 IN    <tibble [92 × 24]>
3 MI    <tibble [83 × 24]>
4 OH    <tibble [88 × 24]>
5 WI    <tibble [72 × 24]>
```

Grouped PCA

```
state_pca ← mw_pca ▷  
  mutate(pca = map(data, do_pca))
```

```
state_pca
```

```
# A tibble: 5 × 3
```

```
# Groups:   state [5]
```

	state	data	pca
	<chr>	<list>	<list>
1	IL	<tibble [102 × 24]>	<prcomp>
2	IN	<tibble [92 × 24]>	<prcomp>
3	MI	<tibble [83 × 24]>	<prcomp>
4	OH	<tibble [88 × 24]>	<prcomp>
5	WI	<tibble [72 × 24]>	<prcomp>

Grouped PCA

Alternatively, write a lambda function:

```
mw_pca ▷  
  mutate(pca = map(data, \(x) prcomp(x, center = TRUE, scale = TRUE)))
```

```
# A tibble: 5 × 3  
# Groups:   state [5]  
  state data          pca  
  <chr> <list>         <list>  
1 IL    <tibble [102 × 24]> <prcomp>  
2 IN    <tibble [92 × 24]>  <prcomp>  
3 MI    <tibble [83 × 24]>  <prcomp>  
4 OH    <tibble [88 × 24]>  <prcomp>  
5 WI    <tibble [72 × 24]>  <prcomp>
```

Tidy it

```
# broom has methods for different models, including prcomp, and they take  
# arguments specific to those models. Here we want the principal components.
```

```
do_tidy ← function(pr){  
  broom::tidy(pr, matrix = "pcs")  
}
```

```
state_pca ← mw_pca ▷  
  mutate(pca = map(data, do_pca),  
         pcs = map(pca, do_tidy))
```

```
state_pca
```

```
# A tibble: 5 × 4
```

```
# Groups:   state [5]
```

	state	data	pca	pcs
	<chr>	<list>	<list>	<list>
1	IL	<tibble [102 × 24]>	<prcomp>	<tibble [24 × 4]>
2	IN	<tibble [92 × 24]>	<prcomp>	<tibble [24 × 4]>
3	MI	<tibble [83 × 24]>	<prcomp>	<tibble [24 × 4]>
4	OH	<tibble [88 × 24]>	<prcomp>	<tibble [24 × 4]>
5	WI	<tibble [72 × 24]>	<prcomp>	<tibble [24 × 4]>

Tidy it

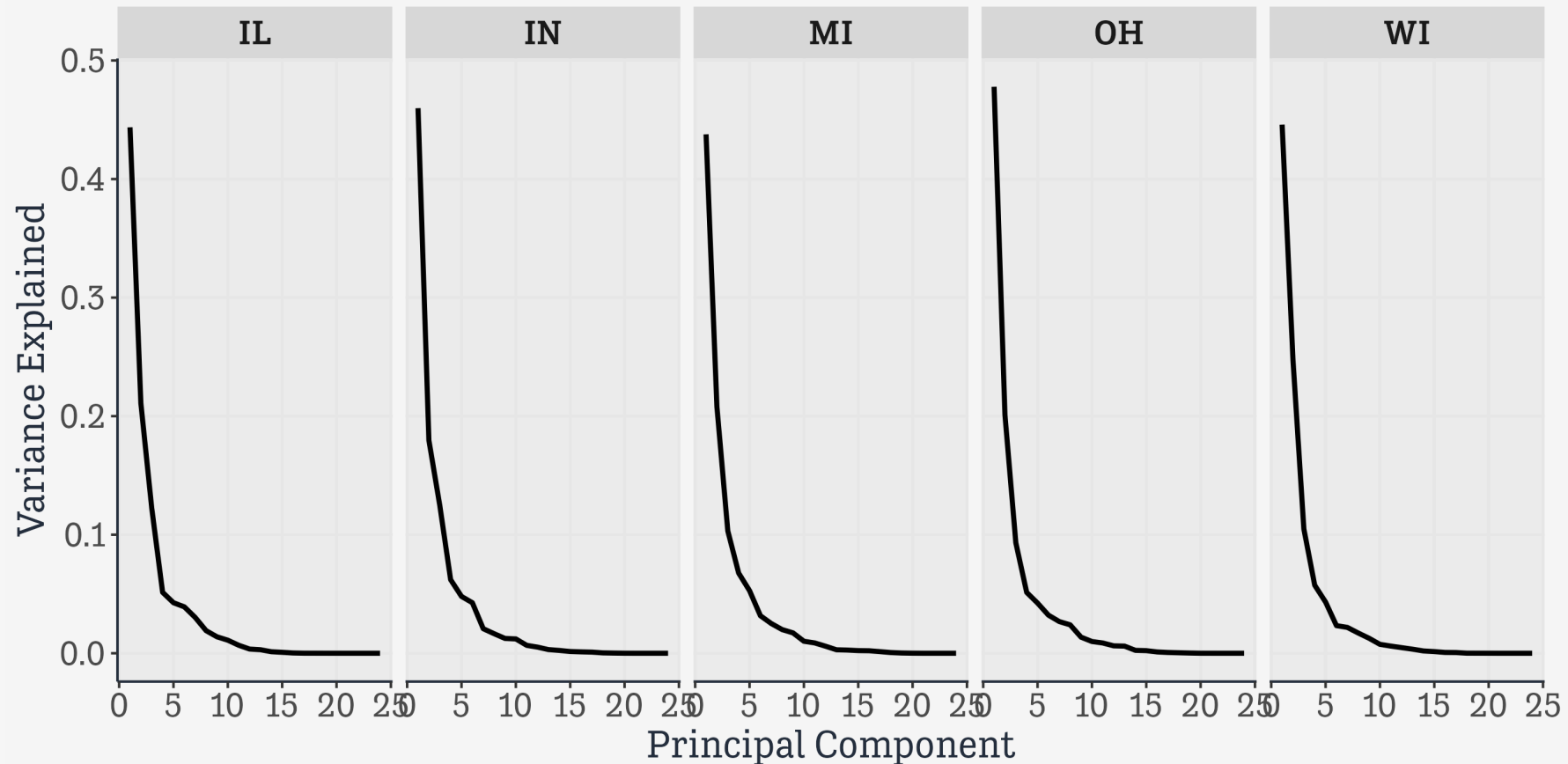
Or again, write a lambda function

```
mw_pca ▷  
  mutate(pca = map(data, \(x) prcomp(x, center = TRUE, scale = TRUE)),  
         pcs = map(pca, \(x) tidy(x, matrix = "pcs")))
```

```
# A tibble: 5 × 4  
# Groups:   state [5]  
  state data          pca      pcs  
  <chr> <list>         <list>  <list>  
1 IL    <tibble [102 × 24]> <prcomp> <tibble [24 × 4]>  
2 IN    <tibble [92 × 24]>  <prcomp> <tibble [24 × 4]>  
3 MI    <tibble [83 × 24]>  <prcomp> <tibble [24 × 4]>  
4 OH    <tibble [88 × 24]>  <prcomp> <tibble [24 × 4]>  
5 WI    <tibble [72 × 24]>  <prcomp> <tibble [24 × 4]>
```

Unnest to inspect

```
mw_pca >
  mutate(pca = map(data, \(x) prcomp(x, center = TRUE, scale = TRUE)),
         pcs = map(pca, \(x) tidy(x, matrix = "pcs"))) >
  unnest(cols = c(pcs)) >
  ggplot(aes(x = PC, y = percent)) +
  geom_line(linewidth = 1.1) +
  facet_wrap(~ state, nrow = 1) +
  labs(x = "Principal Component",
       y = "Variance Explained")
```



And finally, `augment()`

```
do_aug ← function(pr){  
  broom::augment(pr)  
}
```

```
state_pca ← mw_pca ▷  
  mutate(pca = map(data, do_pca),  
         pcs = map(pca, do_tidy),  
         fitted = map(pca, do_aug))
```

```
state_pca
```

```
# A tibble: 5 × 5
```

```
# Groups:   state [5]
```

	state	data	pca	pcs	fitted
	<chr>	<list>	<list>	<list>	<list>
1	IL	<tibble [102 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [102 × 25]>
2	IN	<tibble [92 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [92 × 25]>
3	MI	<tibble [83 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [83 × 25]>
4	OH	<tibble [88 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [88 × 25]>
5	WI	<tibble [72 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [72 × 25]>

... or a lambda function

```
mw_pca ►  
  mutate(pca = map(data, \(x) prcomp(x, center = TRUE, scale = TRUE)),  
         pcs = map(pca, \(x) tidy(x, matrix = "pcs")),  
         fitted = map(pca, \(x) augment(x)))
```

```
# A tibble: 5 × 5
```

```
# Groups:   state [5]
```

	state	data	pca	pcs	fitted
	<chr>	<list>	<list>	<list>	<list>
1	IL	<tibble [102 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [102 × 25]>
2	IN	<tibble [92 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [92 × 25]>
3	MI	<tibble [83 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [83 × 25]>
4	OH	<tibble [88 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [88 × 25]>
5	WI	<tibble [72 × 24]>	<prcomp>	<tibble [24 × 4]>	<tibble [72 × 25]>

In one breath

```
out ← midwest ▷
  group_by(state) ▷
  select_if(is.numeric) ▷
  select(-PID) ▷
  nest() ▷
  mutate(pca = map(data, \(x) prcomp(x, center = TRUE, scale = TRUE)),
         pcs = map(pca, \(x) tidy(x, matrix = "pcs")),
         fitted = map(pca, \(x) augment(x))) ▷
  unnest(cols = c(fitted)) ▷
  add_column(county = midwest$county)
```

out

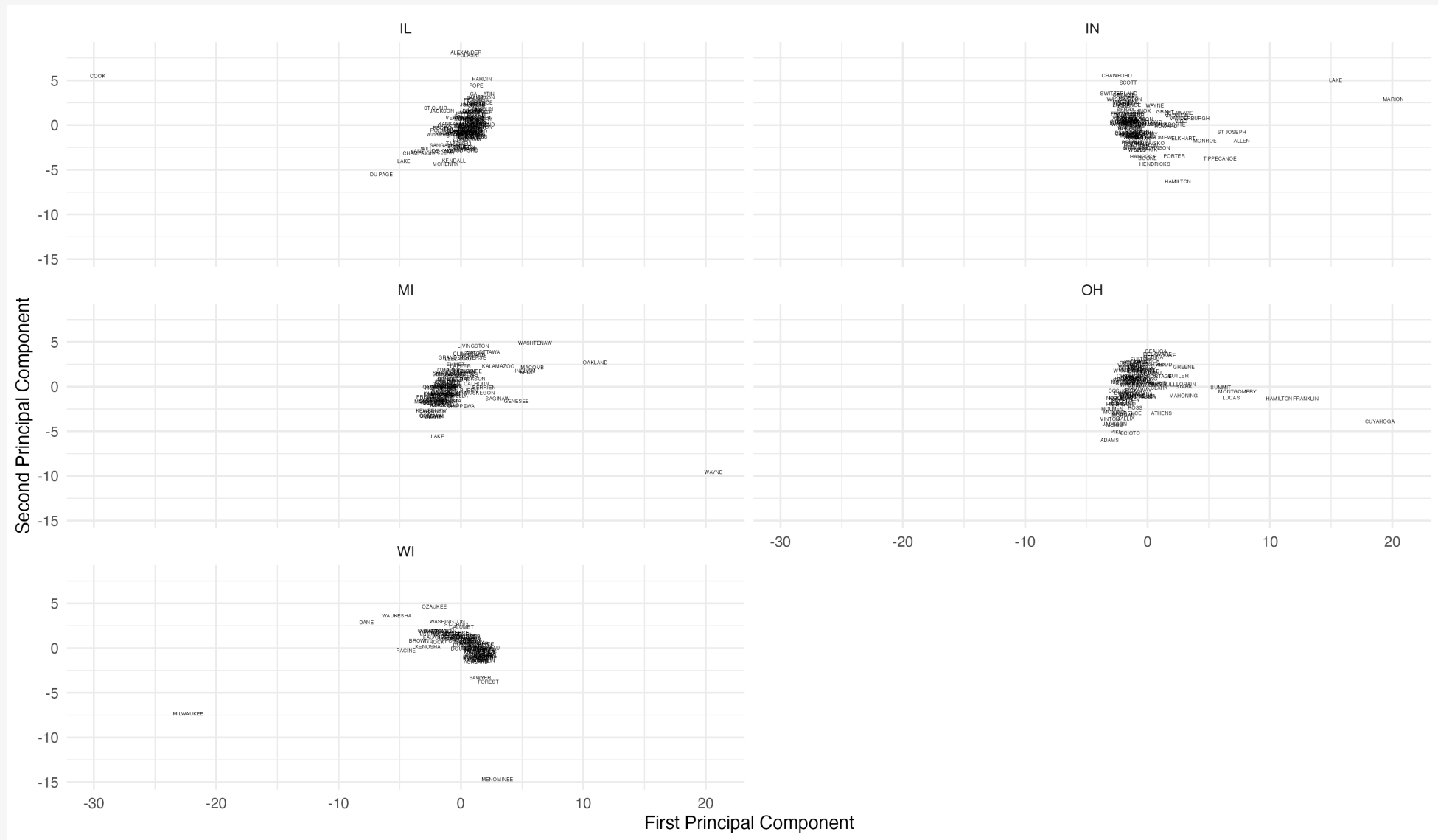
A tibble: 437 × 30

Groups: state [5]

	state	data	pca	pcs	.rownames	.fittedPC1	.fittedPC2	.fittedPC3
	<chr>	<list>	<list>	<list>	<chr>	<dbl>	<dbl>	<dbl>
1	IL	<tibble>	<prcomp>	<tibble>	1	0.403	-0.233	0.00488
2	IL	<tibble>	<prcomp>	<tibble>	2	0.413	8.14	3.33
3	IL	<tibble>	<prcomp>	<tibble>	3	0.908	0.326	-0.303
4	IL	<tibble>	<prcomp>	<tibble>	4	-0.344	-2.37	-0.907
5	IL	<tibble>	<prcomp>	<tibble>	5	0.940	1.61	1.06
6	IL	<tibble>	<prcomp>	<tibble>	6	0.482	-1.07	-0.744
7	IL	<tibble>	<prcomp>	<tibble>	7	1.61	1.83	-1.29
8	IL	<tibble>	<prcomp>	<tibble>	8	0.835	-0.428	-0.839
9	IL	<tibble>	<prcomp>	<tibble>	9	1.28	0.408	-1.08
10	IL	<tibble>	<prcomp>	<tibble>	10	-3.49	-3.13	7.39

i 427 more rows

i 22 more variables: .fittedPC4 <dbl>, .fittedPC5 <dbl>, .fittedPC6 <dbl>,
.fittedPC7 <dbl>, .fittedPC8 <dbl>, .fittedPC9 <dbl>, .fittedPC10 <dbl>,
.fittedPC11 <dbl>, .fittedPC12 <dbl>, .fittedPC13 <dbl>, .fittedPC14 <dbl>,



Join us now for the



Crossover Event



of the Year

ATT estimates

```
library(WeightIt)
library(MatchIt)
data("lalonde", package = "MatchIt")
lalonde
```

	treat	age	educ	race	married	nodegree	re74	re75	re78
NSW1	1	37	11	black	1	1	0.0	0.0	9930.0
NSW2	1	22	9	hispan	0	1	0.0	0.0	3595.9
NSW3	1	30	12	black	0	0	0.0	0.0	24909.5
NSW4	1	27	11	black	0	1	0.0	0.0	7506.1
NSW5	1	33	8	black	0	1	0.0	0.0	289.8
NSW6	1	22	9	black	0	1	0.0	0.0	4056.5
NSW7	1	23	12	black	0	0	0.0	0.0	0.0
NSW8	1	32	11	black	0	1	0.0	0.0	8472.2
NSW9	1	22	16	black	0	0	0.0	0.0	2164.0
NSW10	1	33	12	white	1	0	0.0	0.0	12418.1
NSW11	1	19	9	black	0	1	0.0	0.0	8173.9
NSW12	1	21	13	black	0	0	0.0	0.0	17094.6
NSW13	1	18	8	black	0	1	0.0	0.0	0.0
NSW14	1	27	10	black	1	1	0.0	0.0	18739.9
NSW15	1	17	7	black	0	1	0.0	0.0	3023.9
NSW16	1	19	10	black	0	1	0.0	0.0	3228.5
NSW17	1	27	13	black	0	0	0.0	0.0	14581.9
NSW18	1	23	10	black	0	1	0.0	0.0	7693.4

I heard you were working on these in Steve's class.

ATT estimates

```
n_replicates ← 1e3
set.seed(22102026)

m_orig ← matchit(treat ~ age + educ + race + married + nodegree + re74 + re75,
  data = lalonde, method = "nearest",
  estimand = "ATT")

dr_orig ← lm(re78 ~ treat + age + educ + race + married + nodegree + re74 + re75,
  data = lalonde, weights = m_orig$weights)

dr_tidy ← tidy(dr_orig) ▷
  filter(term = "treat")
dr_tidy
```

```
# A tibble: 1 × 5
  term estimate std.error statistic p.value
<chr>    <dbl>    <dbl>    <dbl>    <dbl>
1 treat    1345.      790.      1.70    0.0894
```

Bootstrapped ATT estimates

Tidily bootstrap our estimates.

```
## Keep everything, do the matching and weighting inside the boot
boot_out <- tibble(boot_id = 1:n_replicates) >
  mutate(
    boot_data = map(boot_id, \(x) slice_sample(lalonde,
                                              n = nrow(lalonde),
                                              replace = TRUE)),

    match_obj = map(boot_data, \(bd) {
      matchit(treat ~ age + educ + race + married + nodegree + re74 + re75,
              data = bd,
              method = "nearest",
              estimand = "ATT")
    }),

    model = map2(boot_data, match_obj, \(bd, mo) {
      lm(re78 ~ treat + age + educ + race + married + nodegree + re74 + re75,
         data = bd,
         weights = mo$weights)
    }),

    coefs = map(model, tidy)
  )
```

Four `map()` calls including one to `map2()`. We could separately write named functions for the steps inside the mapping calls, but we use lambdas instead and write them on the fly.

Bootstrapped ATT estimates

What we get:

```
boot_out
```

```
# A tibble: 1,000 × 5
  boot_id boot_data      match_obj model  coefs
  <int>   <list>      <list>    <list> <list>
1       1 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
2       2 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
3       3 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
4       4 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
5       5 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
6       6 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
7       7 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
8       8 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
9       9 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
10      10 <df [614 × 9]> <matchit> <lm>   <tibble [10 × 5]>
# i 990 more rows
```

Bootstrapped ATT estimates

Unnest the coefs column:

```
boot_out ►  
  unnest(coefs)
```

```
# A tibble: 10,000 × 9
```

	boot_id	boot_data	match_obj	model	term	estimate	std.error	statistic	p.value
	<int>	<list>	<list>	<list>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	1	<df>	<matchit>	<lm>	(Int...	-8.28e+3	3318.	-2.50	1.30e-2
2	1	<df>	<matchit>	<lm>	treat	1.07e+3	811.	1.32	1.88e-1
3	1	<df>	<matchit>	<lm>	age	9.08e+1	45.1	2.01	4.50e-2
4	1	<df>	<matchit>	<lm>	educ	8.82e+2	221.	3.98	8.25e-5
5	1	<df>	<matchit>	<lm>	race...	1.54e+3	1384.	1.11	2.67e-1
6	1	<df>	<matchit>	<lm>	race...	6.24e+2	964.	0.648	5.17e-1
7	1	<df>	<matchit>	<lm>	marr...	7.09e+2	1065.	0.666	5.06e-1
8	1	<df>	<matchit>	<lm>	node...	2.34e+3	1225.	1.91	5.67e-2
9	1	<df>	<matchit>	<lm>	re74	3.10e-2	0.102	0.305	7.61e-1
10	1	<df>	<matchit>	<lm>	re75	4.24e-1	0.169	2.52	1.23e-2

```
# i 9,990 more rows
```

Bootstrapped ATT estimates

```
# Simplest way to calculate bootstrap ci's; could get  
# much fancier using boot() or any number of other fns for the  
# bootstrap stage  
boot_tbl ← boot_out ▷  
  unnest(coefs) ▷  
  filter(term = "treat") ▷  
  summarize(  
    boot_att = mean(estimate),  
    boot_se = sd(estimate),  
    boot_ci_lower = quantile(estimate, 0.025),  
    boot_ci_upper = quantile(estimate, 0.975)  
  )  
  
tinytable::tt(boot_tbl, digits = 3)
```

boot_att	boot_se	boot_ci_lower	boot_ci_upper
1276	761	-152	2744

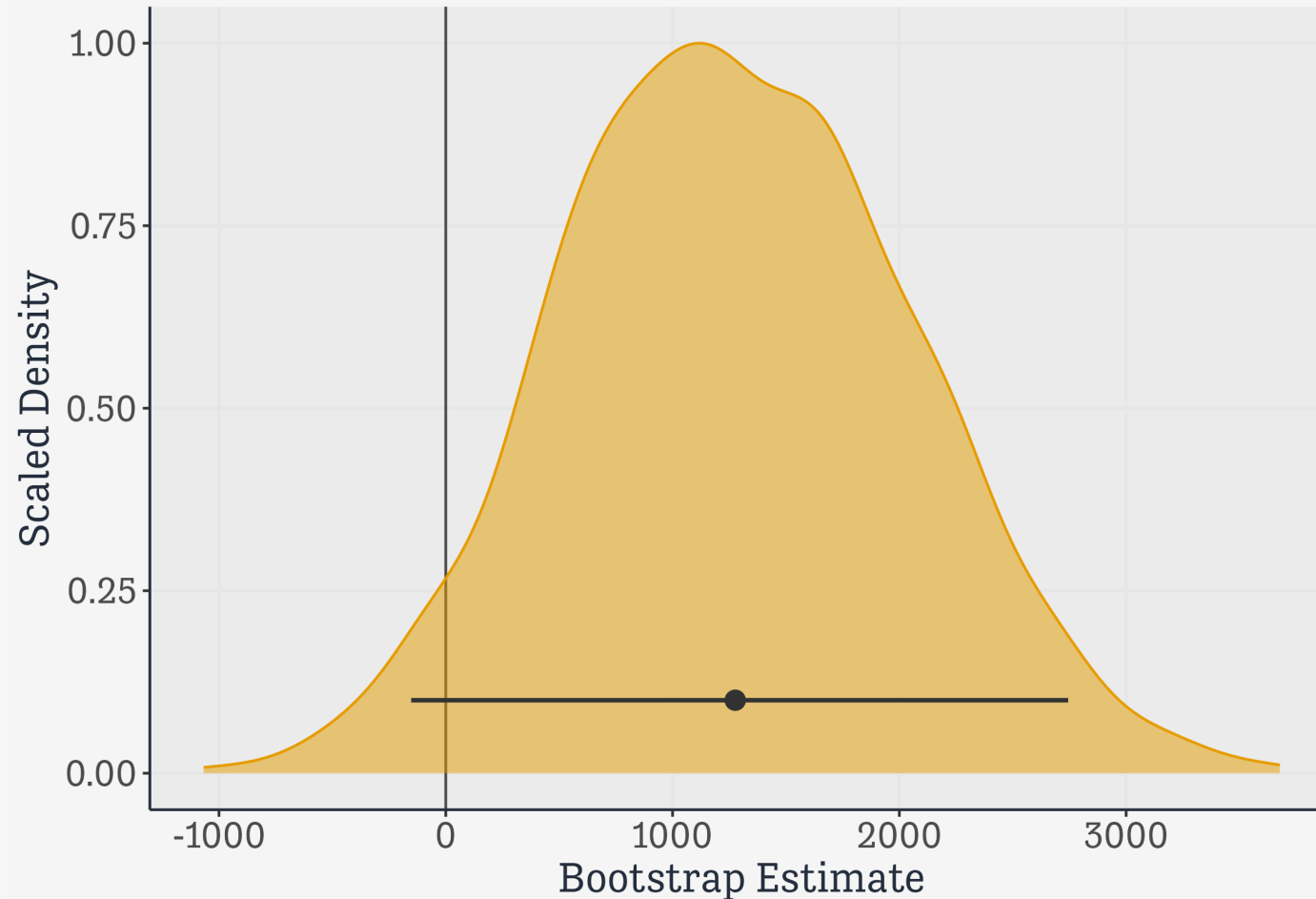
Bootstrapped ATT estimates

```
tmp_tbl ← boot_tbl ▷  
  mutate(y = 0.1)  
  
boot_out ▷  
  unnest(coefs) ▷  
  select(boot_id, term, estimate) ▷  
  filter(term = "treat") ▷  
  ggplot(mapping = aes(x = estimate, y = after_stat(scaled),  
    color = term, fill = term)) +  
  geom_vline(xintercept = 0, color = "gray30") +  
  geom_density(alpha = 0.5) +  
  geom_pointrange(data = tmp_tbl,  
    mapping = aes(x = boot_att, y = y, xmin = boot_ci_lower,  
      xmax = boot_ci_upper),  
    color = "gray20", size = rel(0.7), linewidth = rel(0.9),  
    inherit.aes = FALSE) +  
  labs(title = "Distribution of Bootstrapped Treatment Estimates",  
    subtitle = "Pointrange shows point estimate and 95% quantile range.",  
    x = "Bootstrap Estimate", y = "Scaled Density") +  
  guides(color = "none", fill = "none")
```

Bootstrapped ATT estimates

Distribution of Bootstrapped Treatment Estimates

Pointrange shows point estimate and 95% quantile range.



Distribution of Treatment Estimates