Big Data and Economics

Bootstrapping, Functions, and Parallel Processing

Kyle Coombs
Bates College | ECON/DCS 368

Table of contents

Prologue

Plan for the week

- Computers closed: Explanation of bootstrapping, basics of functions, iteration, and parallel processing
- Computers open: Work through practical examples
- By the end of the week you will:
    - Understand the intuition of bootstrapping
    - Be able to write basic functions in R
    - Be able to iterate tasks serially and in parallel in R
    - Be able to bootstrap in R

Bootstrapping

Bootstrapping: Motivating example

- Imagine you gain powers to view every parallel, distinct universe where the world[1]
- With these powers, you **obviously** decide to replicate critical results in economics
  - You collect equivalent sample sizes
  - You run the same regressions to estimate the same parameters
- Do you think the results will be the same in each parallel universe?

Bootstrapping: Motivating example

- Imagine you gain powers to view every parallel, distinct universe where the world[1]
- With these powers, you **obviously** decide to replicate critical results in economics
  - You collect equivalent sample sizes
  - You run the same regressions to estimate the same parameters
- Do you think the results will be the same in each parallel universe?
- No! They'll differ a lot or a little, depending on how variable the data generating process is

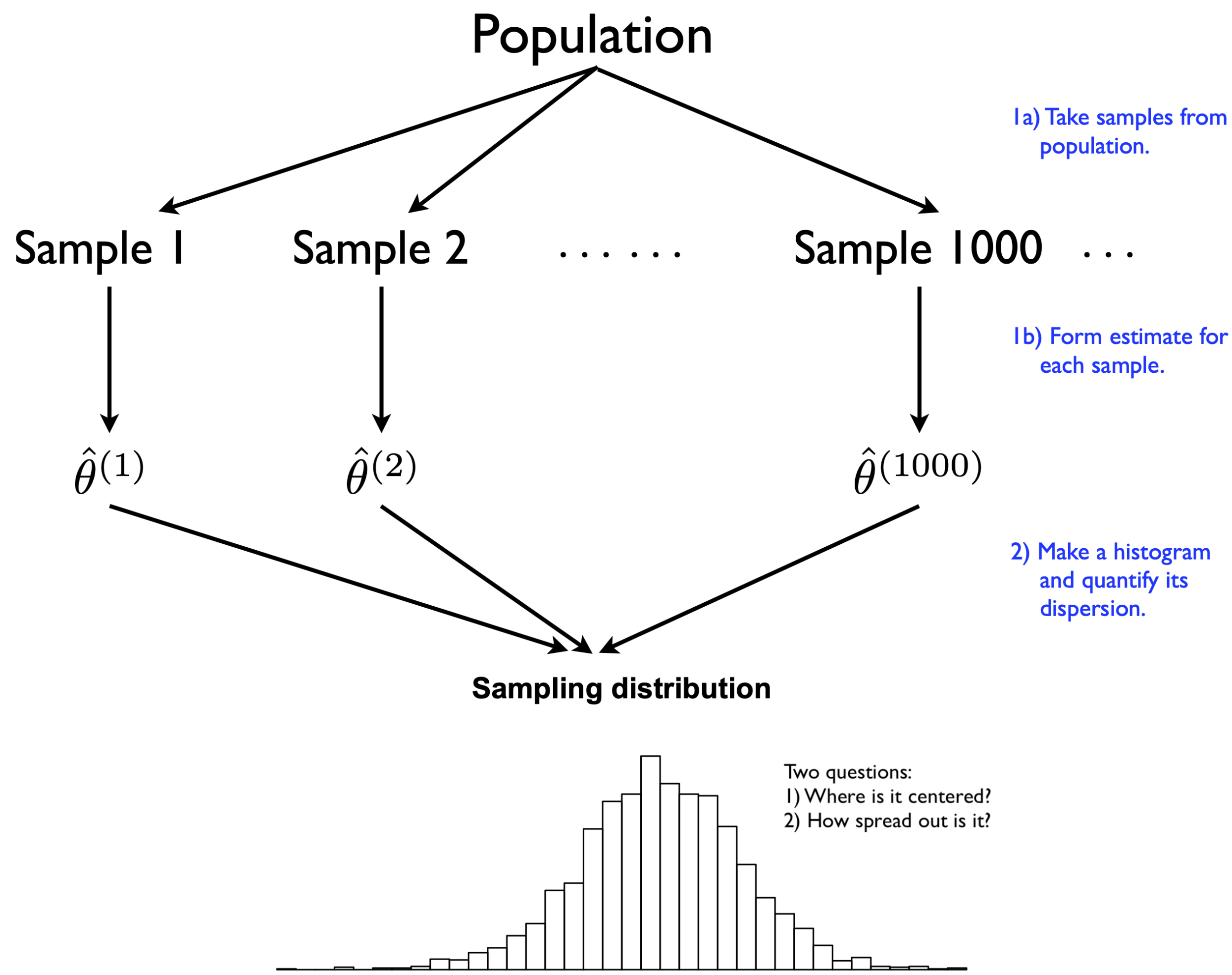
[1] Think *Everything, Everywhere, All At Once*, *Into The Spiderverse*, etc.

Return to earth

- We don't have powers to view parallel universes
- But we can view different random samples of a population of interest
- And each sample will provide a distinct estimate of the true parameters of interest
- We have two ways to use these samples to get close to our parallel universe powers:

1. **Mathematical approximations**: Make simple assumptions that randomness obeys mathematical regularities for large samples
    - e.g. *Central Limit Theorem* allows us to use the normal distribution to approximate the sampling distribution of the mean
2. **Resampling**: Use the same sample to estimate the variability of our estimates
    - e.g. *bootstrapping* which we will cover today

Visualizing samples



What is bootstrapping?

- Bootstrapping is named for "pulling yourself up by your bootstraps," a joke[2] because the method seems preposterous and impossible

- Bootstrapping has two repeated steps:

1. Draw a random sample **with replacement** of size *N* from your sample.
2. Perform the same analysis on the new sample.

- Repeat steps 1 and 2 a bunch of times saving each, the 2.5th and 97.th percentiles show the 95% confidence interval

Then plot the distribution of the estimates from each sample

[2] Not a great one.

What is a bunch of times?

How many bootstraps is enough?

What is a bunch of times?

How many bootstraps is enough?

It depends. Intuively:

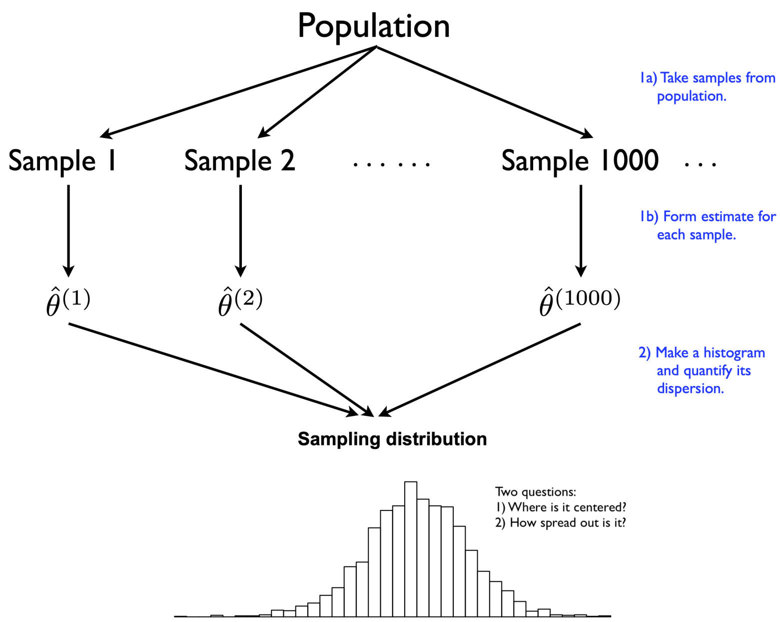
- The more bootstraps, the better
- But the more bootstraps, the longer it takes to run
- Many econometricians and statisticians have purported to solve for "optimal" bootstrapping, but it is still an open question

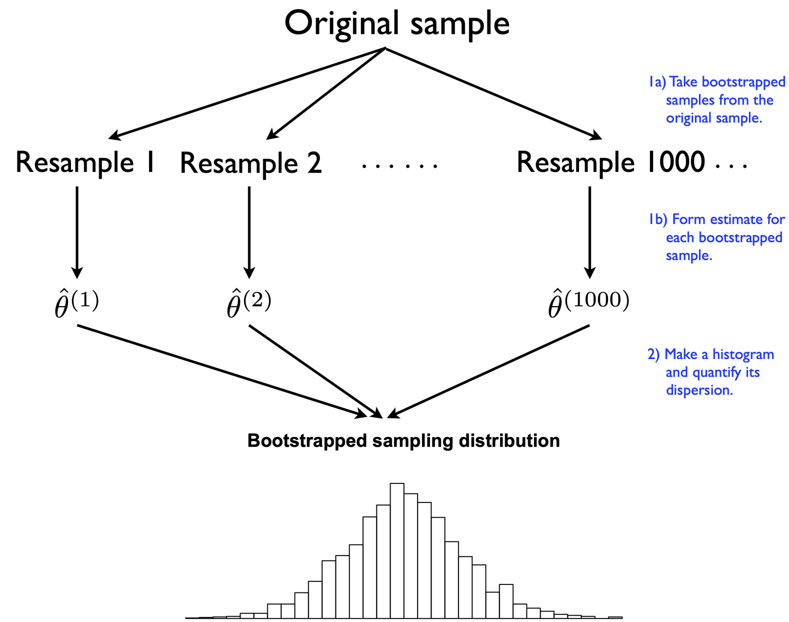
- Arguably, you should do 1000s, if not more times!

  - In this lecture, I did not do that because it would take too long to generate my slides

- See parallel processing before for speed ups!

Visualizing Bootstrapping vs. population





Population samples

Bootstrap analog

Schematics taken from Data Science in R: A Gentle Introduction by James Scott

What does bootstrapping show?

- Bootstrapping shows how much your estimates vary across samples
- It shows the **sampling distribution** of your estimates
- The 95% confidence interval is the 2.5th and 97.5th percentile of the sampling distribution

What does bootstrapping show?

- Bootstrapping shows how much your estimates vary across samples
- It shows the **sampling distribution** of your estimates

- The 95% confidence interval is the 2.5th and 97.5th percentile of the sampling distribution

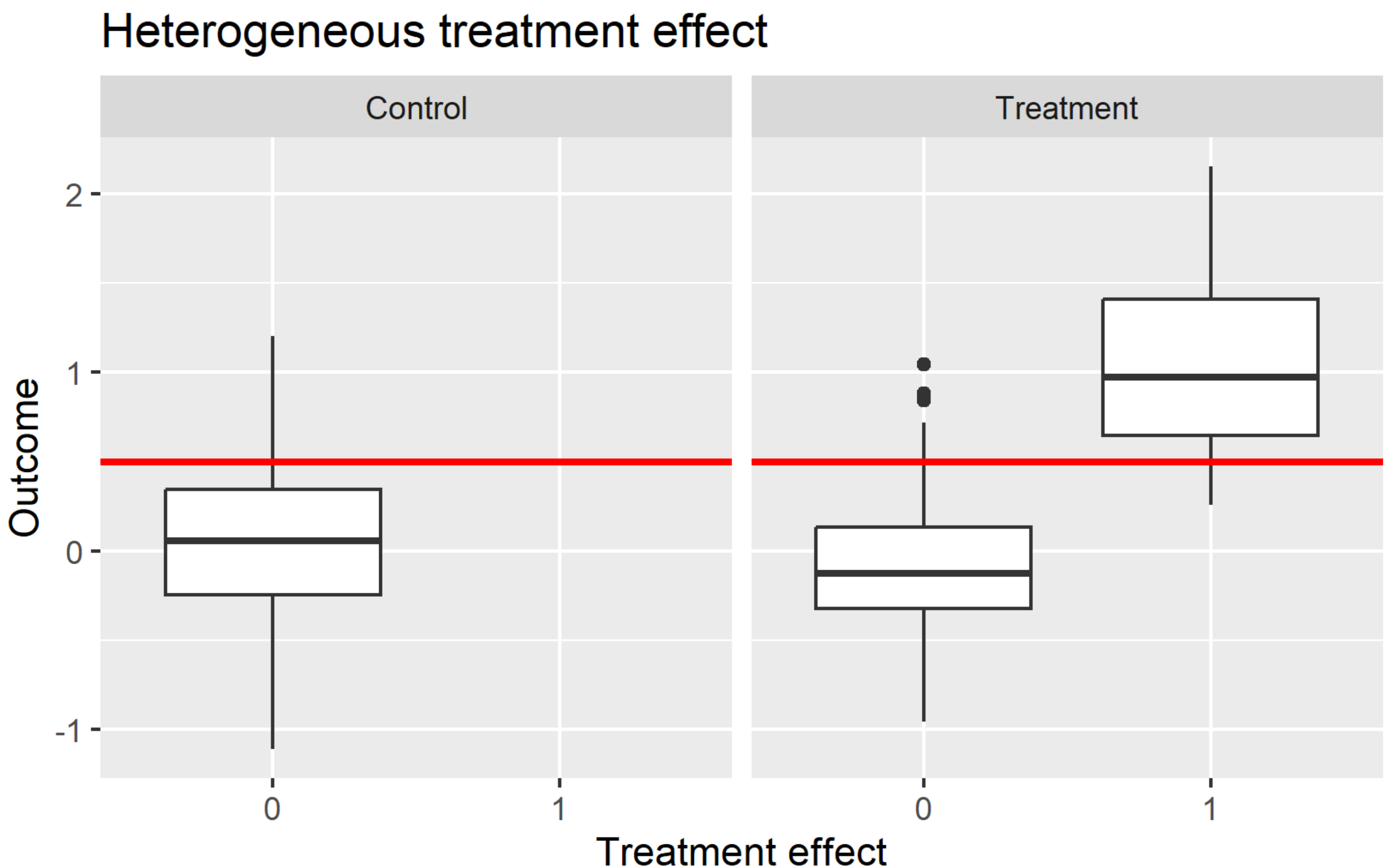
- **Intuition**: Bootstrapping simulates the process of collecting new samples
    - If your sample is truly representative, then any shuffled sample should be representative too!
    - Your own sample is itself a random sample generated from some other random sample

A "simple" example

- Imagine you have an RCT, and you want to estimate the effect of a treatment on some outcome
- There are two groups of treated users:
  - Those for whom treatment effect is zero
  - Those for whom treatment effect is one

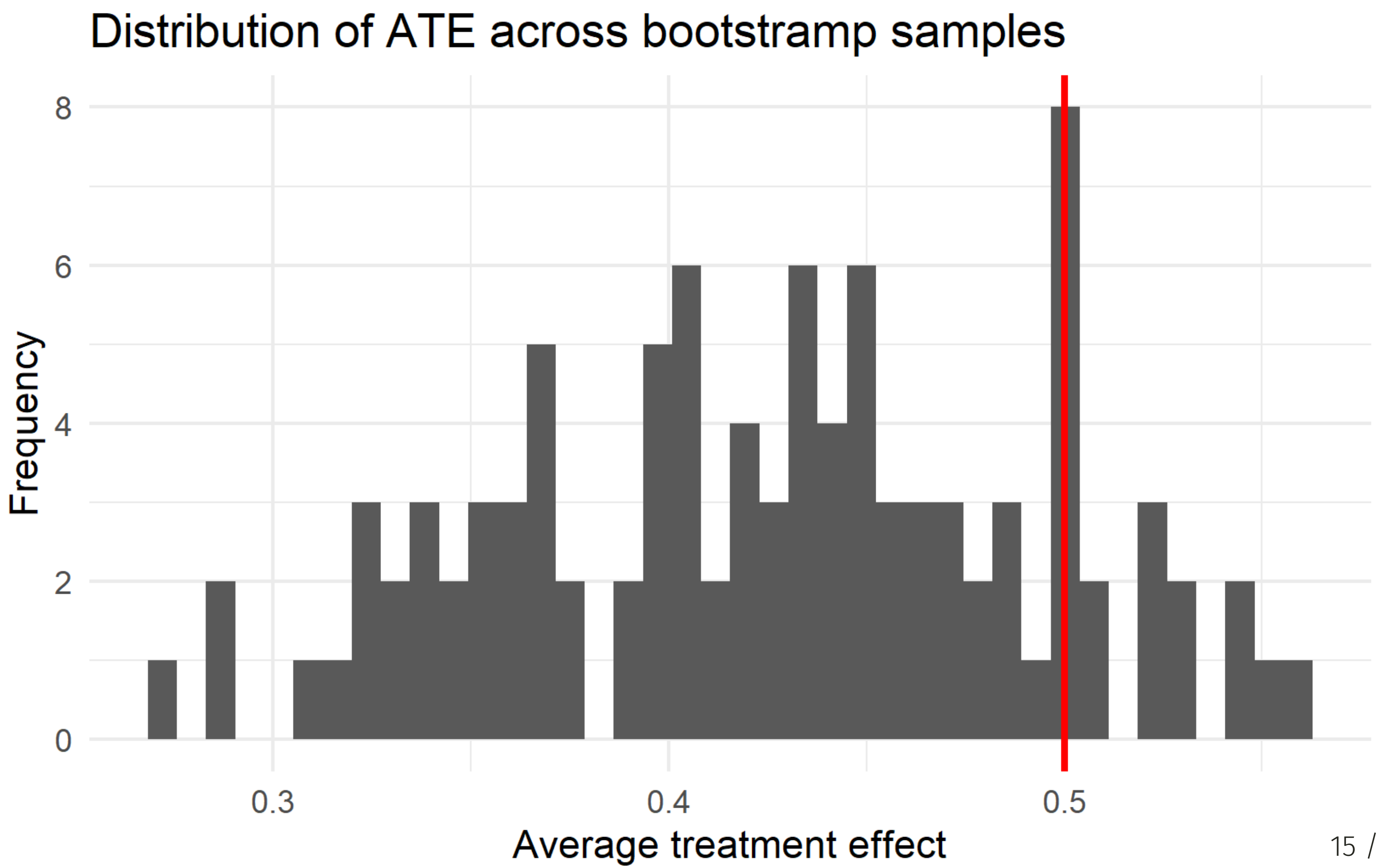
On average the treatment effect is 0.5, but there is a lot of variation!

Visualizing simple example



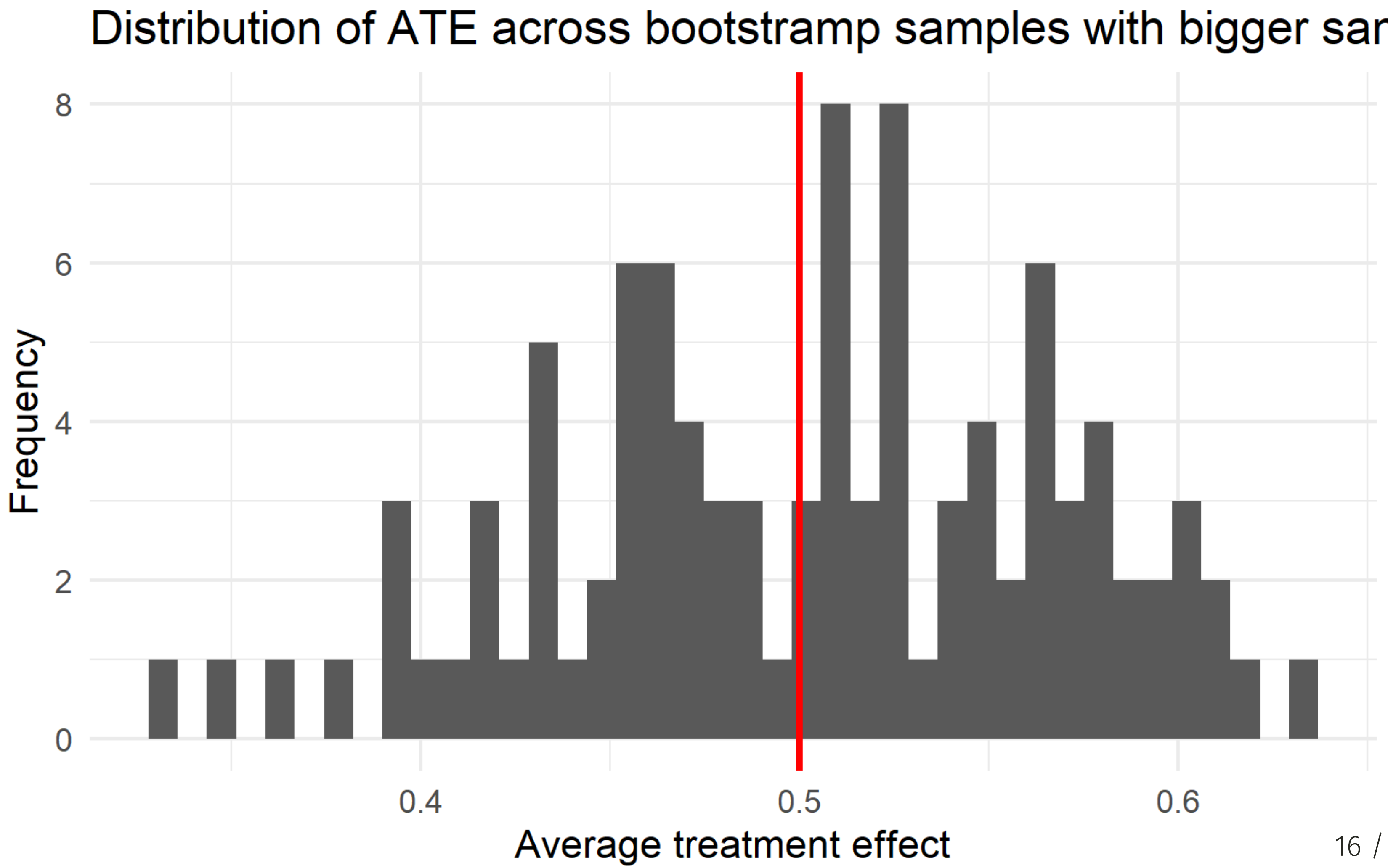
Bootstrap to get random samples

- Let's take a bunch of random samples and see how the average treatment effect varies



Now with a bigger sample

- With a larger baseline sample, the distribution of the average treatment effect is tighter



Bootstrapping assumptions

- Your sample is just one random sample from the population of interest
- Bootstrapping assumes that randomness in data is driven by sampling
- Bootstrapping assumes a distribution that is not "highly" skewed
- (Basic) bootstrapping assumes independent and identically distributed
    - But you can do clustering and other forms of correlation, etc.
- Other technical assumptions!

When should I do it?

- The bootstrap simulates the sample distribution of your estimates
- Use it to:

1. Calculate the standard error of your estimates
    - Especially when you can't use analytical formulas (e.g. the median)
2. Do power simulations on pilot data
3. Generate "training data sets" for machine learning models
4. Explore variation in model predictions as the sample changes
5. Other robustness checks and more

How do I bootstrap?

There are two main requirements:

1. Always use the same sample size as your original sample
    - This ensures the "same" data-generating process and approximates the same randomness
2. Always sample **with replacement**
    - That means you may sample the same observation twice

How many samples?

- The more samples you take, the

Limitations of bootstrapping

- Bootstrapping cannot save you if your sample is biased
- Bootstrapping cannot save you if your sample is too small
- Bootstrapping cannot save you if your sample is not representative

Limitations of packages

- I used the boot package above!
- That handles many cases, but it can get a little slow for big data
- And it has built-in parallel processing, but it may not work on different systems
- Best to know how to do it yourself as well cause it is pretty easy once you get the hang of it!

Functions

What is a function?

- In math, a function is a mapping from domain to range

$$f(x) = x^2 \quad \text{Takes a number from the domain and returns its square in the range}$$
$$f(2) = 4 \quad \text{The function applied to 2 returns 4}$$

- In programming, a function is a mapping from input to output

```r
square <- function(x) {
  x^2
}

square(2) # Returns 4
```

```
## [1] 4
```

Why use functions?

- **Abstraction**
  - They allow you to summarize complex details into a single line of code, so you only need to understand them once (instead of repeating yourself)
- **Automation**
  - Automate a task to happen many times without having to write the same code over and over
- **Documentation**
  - Well-written functions codify the steps you take to do something, so you can easily remember what you did

How do I write a function?

In R, functions are defined using the `function` keyword

```
some_function <- function(positional_input1=1,positional_input2="two",keyword_inputs) {
  # Do something with these inputs
  # Create output or ouputs
  return(output) # Return the output
  # If you do not specify return, it returns the last object
}
```

`function` takes keyword inputs and positional inputs. It does not require a specific order for these unlike in Python. But generally, position comes first.

Control flow: If/else logic

Functions make great use of if/else logic

```
square =
  function(x = NULL) {

    if (is.null(x)) { ## Start multi-line IF statement with {
      x = 1
      ## Message to users:
      message("No input value provided. Using default value of 1.")
      }               ## Close multi-line if statement with }

    x_sq = x^2
    d = data.frame(value = x, value_squared = x_sq)

    return(d)
  }
square()
```

```
## No input value provided. Using default value of 1.
```

```
##   value value_squared
## 1     1             1
```

This function has a default value of 1 for when you fail to provide a value.

Each step of bootstrap

```r
# library(tidyverse) # Already loaded
set.seed(1)
df ← tibble(x = rnorm(1000, mean = 0, sd = 1),
  y= x+rnorm(1000, mean = 0, sd = 1))

bootstrap_sample ← function(df) {
  # 1. Draw a random sample with replacement of size N from your sample.
  sample ← df %>% slice_sample(n = nrow(df), replace = TRUE)
  # 2. Perform the same analysis, here a median, on the new sample.
  return(coef(lm(y ~ x, data = sample))[2])
}

bootstrap_sample(df)
```

```
##         x
## 0.9671832
```

Wrapping up a function

- You can wrap functions inside of other functions
- This is a great way to make your code more readable and modular
- Also useful for various iteration tasks that need to take an iterated input

```
wrapper_bootstrap <- function(i,df) {
  # print(i) # if you want to visualize the i.
  bootstrap_sample(df)
}
wrapper_bootstrap(1,df)
```

```
##         x
## 0.9824952
```

More on functions

- There is a lot more to functions than we can cover today
- Check out Grant McDermott's Introudctory and Advanced chapters on functions
- There are some incredible tips on how to:
    - Debug functions
    - Write functions that are easy to read
    - Catch errors
    - Cache or `memoise` big functions

Iteration

Iteration: For loops

- You've likely heard of for loops before!
- They're the most common way to iterate across programming languages
- In R, the syntax is fairly simple: you iterative over a vector or list of values, and do stuff with those values

```
for(i in 1:10) {
  square(i)
}
```

Bootstrapping for loop

To save output, you have to pre-define a list where you deposit the output

```
deposit <- vector("list",10) # preallocate list of 10 values
set.seed(1)
for (i in 1:10) {
  # perform bootstrap
  deposit[[i]] <- bootstrap_sample(df)
}

bootstrapped_median <- bind_rows(deposit)
head(bootstrapped_median)
```

```
## # A tibble: 6 × 1
##       x
##   <dbl>
## 1 1.02
## 2 0.987
## 3 0.997
## 4 0.987
## 5 0.947
## 6 0.999
```

Binding output

- Did you notice the `bind_rows()` function I called?
- After any iteration that leaves you a bunch of dataframes in a list, you'll want to put them together
- The `bind_rows` function is a great way to bind together a list of data frames
- Other options include:
  - `do.call(rbind, list_of_dataframes)`
  - `data.table::rbindlist()`

Issues with for loops

- For loops are slow in R
- They clutter up your environment with extra variables (like the `i` indexer)
- They can also be an absolute headache to debug if they get too nested
- Look at the example below: this is a nested for loop that is hard to read and debug
- In some languages, this is all you have, but not in R!

```
for (i in 1:5) {
  for (k in 1:5) {
    if (i > k) {
      print(i*k)
    }
    else {
      for (j in 1:5) {
        print(i*j*k)
      }
    }
  }
}
```

Iteration: apply family

- R has a much more commonly used approach to iteration: the `*apply` family of functions: `apply`, `sapply`, `vapply`, `lapply`, `mapply`
- The `*apply` family takes a function and applies it to each element of a list or vector
- `lapply` is the most commonly used and returns a list back

```
lapply(1:10, square)
```

```
## [[1]]
##   value value_squared
## 1     1             1
##
## [[2]]
##   value value_squared
## 1     2             4
##
## [[3]]
##   value value_squared
## 1     3             9
##
## [[4]]
##   value value_squared
## 1     4            16
##
## [[5]]
##   value value_squared
## 1     5            25
##
## [[6]]
##   value value_squared
## 1     6            36
##
## [[7]]
##   value value_squared
```

`*apply` syntax

- The `*apply` family is a little confusing at first, but it's very powerful
- The syntax is `*apply(list_or_vector, function, other_arguments)`
- The `function` is a function that takes a single argument
- The `other_arguments` are arguments that are passed to the function

Bootstrapping lapply

- One trick: `*apply` insists on iterating over some sequence indexed `i` like a for-loop
- But you can ignore it by using `function(i)` and then not using `i` in the function

```
set.seed(1)
lapply(1:10, function(i) bootstrap_sample(df=df)) %>%
  bind_rows()
```

```
## # A tibble: 10 × 1
##         x
##     <dbl>
##   1 1.02
##   2 0.987
##   3 0.997
##   4 0.987
##   5 0.947
##   6 0.999
##   7 0.966
##   8 0.983
##   9 0.987
##  10 0.987
```

Wrapper functions to get around `*apply`

- Maybe you don't like the ugly syntax of `function(i)` and then not using `i` in the function
- Well you can write a wrapper function to get around that

```
set.seed(1)
lapply(1:10, wrapper_bootstrap, df=df) %>%
  bind_rows()
```

```
## # A tibble: 10 × 1
##        x
##    <dbl>
##  1 1.02
##  2 0.987
##  3 0.997
##  4 0.987
##  5 0.947
##  6 0.999
##  7 0.966
##  8 0.983
##  9 0.987
## 10 0.987
```

Iteration: map

- Sometimes the `*apply` syntax is a little confusing
- The **purrr** package in the tidyverse has more intuitive syntax for iteration: `map`
- The variant `map_df` is especially useful beause it automatically binds the output into a data frame
    - The same iteration syntax applies here too.

```
set.seed(1)
map_df(1:10, function(i) bootstrap_sample(df=df))
```

```
## # A tibble: 10 × 1
##        x
##    <dbl>
##  1 1.02
##  2 0.987
##  3 0.997
##  4 0.987
##  5 0.947
##  6 0.999
##  7 0.966
##  8 0.983
##  9 0.987
## 10 0.987
```

While loops

- I'm largely skipping while loops, but they're also important!
- While loops iterate until one or more conditions are met
  - Typically one condition is a max number of iterations
  - Another conditions is that the some value of the loop is within a small amount of a target value
- These are critical for numerical solvers, which are common in computational economics and machine learning

Parallel Processing

Motivating example: Parallel Processing

- Imagine you get home from the grocery store with 100 bags of groceries
- You have to bring them all inside, but you can only carry 2 at a time
- That's 50 trips back and forth
- How can you speed things up?

Motivating example: Parallel Processing

- Imagine you get home from the grocery store with 100 bags of groceries
- You have to bring them all inside, but you can only carry 2 at a time
- That's 50 trips back and forth

- How can you speed things up?

    - Ask friends to carry to at a time with you (parallel processing)
    - Get a cart and carry 10 at a time (more RAM and a better processor)

A warning

- Parallel processing is an incredibly powerful tool, but it is full of pitfalls
- A friend of mine from the PhD said that he did not understand it until the 4th year of his PhD
- Many economists understand the intuition, but not the details and only do it if absolutely necessary
- That used to be me until I started teaching this class!
- So if it is hard, that's normal. But it is worth learning!

"One trip?" okay

One trip? Okay, sure

Parallel processing: What?

- Your computer has multiple cores, which are like multiple brains
- Each of these is capable of doing the same tasks
- Parallel processing is the act of using multiple cores to do the same task at the same time

Parallel processing: What?

- Your computer has multiple cores, which are like multiple brains
- Each of these is capable of doing the same tasks
- Parallel processing is the act of using multiple cores to do the same task at the same time

- Many coding tasks are "embarassingly parallel"
  - That means they can be broken up into many small tasks that can be done at the same time
  - Bootstrapping is one such example

- Some tasks are not embarrassingly parallel
  - These are called "serial" tasks
  - Parts of these tasks may be possible to do in parallel

Parallel processing vocab

The vocab for parallel processing can get a little confusing:

- **Socket**: A socket is a physical connection between a processor and the motherboard
- **Core**: A core is a physical processor that can do computations
- **Process**: A process is a task that is being done by a core (Windows users may know this from Task Manager)
- **Thread**: A thread is a subtask of a process that can be done in parallel and share memory with other threads
- **Cluster**: A cluster is a group of computers that can be used to do parallel processing
- **Node**: One computer within a cluster

Parallel Processing in R

- In R, there are many ways to parallel process, I'll introduce you to the **future.apply** package
- There are many parallel processing packages in R, but **future.apply** follows the `*apply` family syntax

Trivial example: square numbers

- Let's start with some trivial to understand examples

- Here is a function called `slow_square`, which takes a number and squares it, but after a pause.

```
## Emulate slow function
slow_square =
  function(x = 1) {
    x_sq = x^2
    d = data.frame(value = x, value_squared = x_sq)
    Sys.sleep(2) # literally do nothing for two seconds
    return(d)
    }
```

Let's time that quickly.

```
# library(tictoc) ## Already loaded

tic()
serial_ex = lapply(1:12, slow_square)
toc(log = TRUE)
```

```
## 24.82 sec elapsed
```

Now in parallel

```r
# library(future.apply)  ## Already loaded
# plan(multisession)     ## Already set above

tic()
future_ex = future_lapply(1:12, slow_square)
toc(log = TRUE)
```

```
## 5.64 sec elapsed
```

```r
all.equal(serial_ex, future_ex)
```

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

Example: bootstrapping in parallel

- The future_lapply works the same, but now I have to set the seed inside the function

```
set.seed(1)
tic()
serial_boot <- lapply(1:1e3, function(i) bootstrap_sample(df)) %>%
  bind_rows()
toc(log = TRUE)
```

```
## 1.91 sec elapsed
```

```
tic()
parallel_boot <- future_lapply(1:1e3,
  function(i) bootstrap_sample(df),
  future.seed=1) %>%
  bind_rows()
toc(log = TRUE)
```

```
## 3.17 sec elapsed
```

Want to use `map`? Try **furrr**

The **furrr** package, i.e. future **purrrr** is a parallel processing version of **purrr**

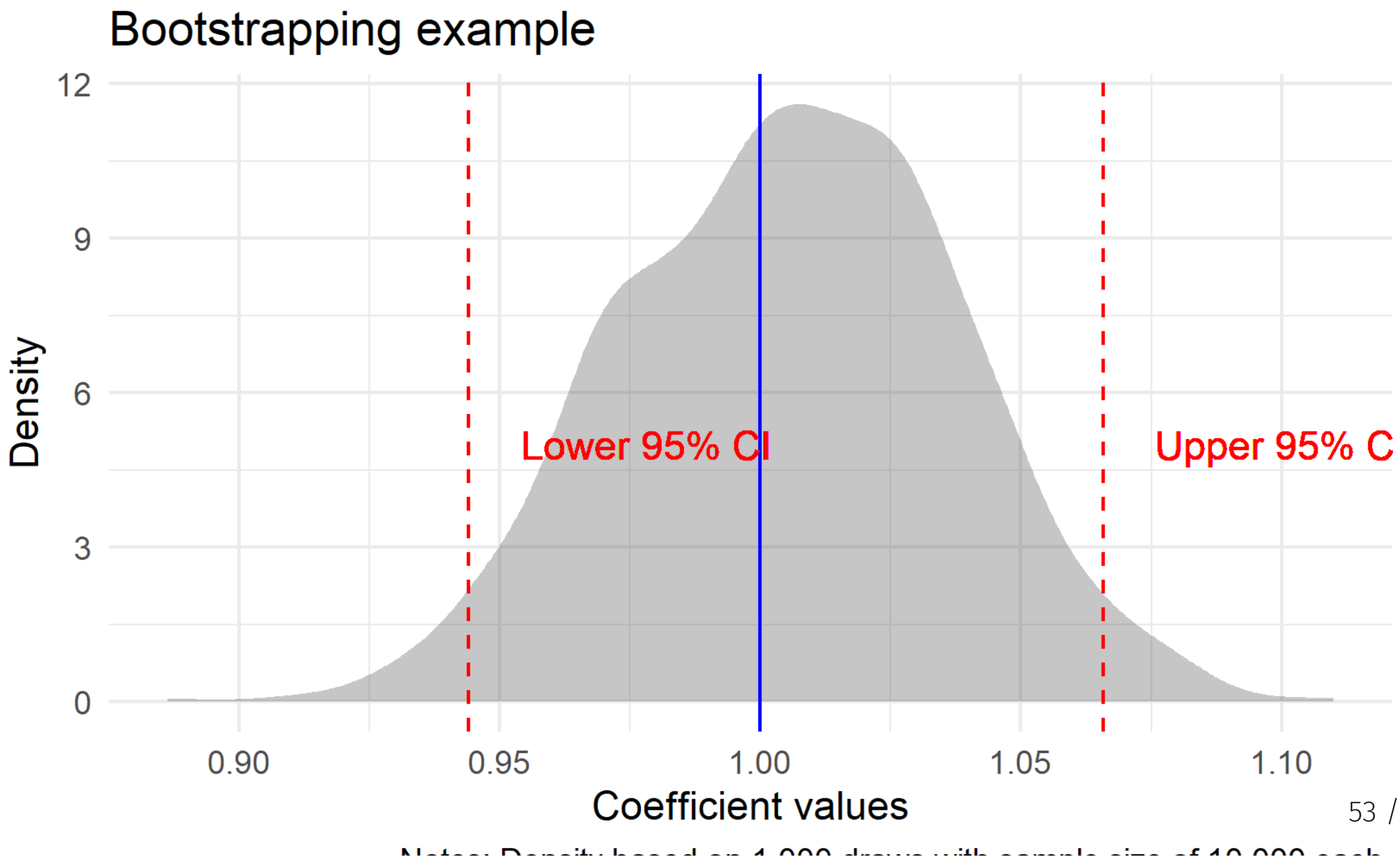
- Again, the syntax is the same, but you have to set the seed inside the function with `.options`.

```
tic()
furrr_boot = future_map_dfr(1:1e3,
  function(i) bootstrap_sample(df),
  .options = furrr_options(seed=1))
toc(log = TRUE)
```

```
## 1.39 sec elapsed
```

Get standard errors from results

- Now that we have a bunch of estimates, we can get the standard error of our estimates
- The 95% confidence interval is just the 2.5th and 97.5th percentile of the sampling distribution



R packages that use parallel processing

- Many R packages already use parallel processing
- `feols()` from **fixest** uses parallel processing to speed up OLS estimation
  - You can control how using the `nthreads` argument
- **data.table** uses parallel processing to speed up data wrangling
- **boot** and **sandwich** can use parallel processing to speed up bootstrapping
- And many others do the same

Parallel processing: Why?

- Parallel processing is a great way to speed up your code and often there are straight-forward ways to do it
- It is not always worth doing:
  - Theoretically, the gain should be linear: each additional node should speed up your code by the same amount
  - In practice, there are "overhead" costs to parallel processing that can slow things down
  - **Overhead costs**: reading in and subsetting data, tracking each node

Across computer clusters

- Parallel processing is also a way to speed up your code across multiple computers
- This is called "distributed computing"
- It is a way to speed up your code when you have a lot of data and a lot of computers
- Imagine you have 1000 computers, each with 1/1000th of your data
- You can run the same code on each computer, and then combine the results
- Same logic, but the "overhead" costs are higher

What next?

- Go try how to bootstrap in R!
- Better yet, learn to do it in parallel
- Navigate to the lecture activity 13a-bootstrapping-functions-practice

Next lecture: Decision Trees