Assignment 3

Landon Wilson

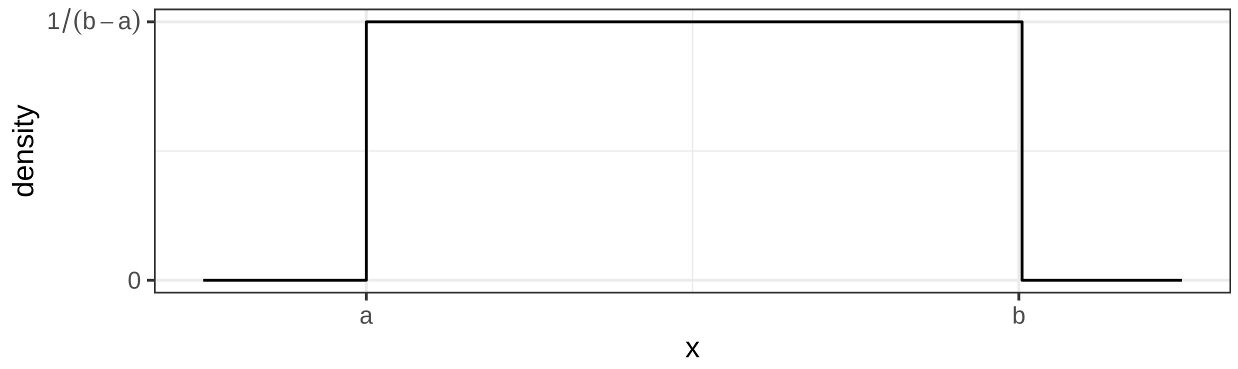
2024-10-18

Exercise 1

Write a function that calculates the density function of a Uniform continuous variable on the interval (a, b). The uniform density function is defined as

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \le x \le b \\ 0 & \text{otherwise} \end{cases}$$

which looks like this



Your goal for this exercise is to write a function `duniform(x, a, b)` that takes an arbitrary value of `x` and parameters `a` and `b` and returns the appropriate height of the density function $(1/(b-a))$. For various values of `x`, `a`, and `b`, demonstrate that your function returns the correct density value.

a) Write your function without regard for it working with vectors of data. Demonstrate that it works by calling the function three times, once where $x < a$, once where $a < x < b$, and finally once where $b < x$.

```
duniform <- function(x,a,b){
  aw <- ifelse(x >= a & x <= b, 1/(b - a), 0)
  return(aw)
}
a <- 5
b <- 10
x.1 <- 0
x.2 <- 7.5
x.3 <- 15
duniform(x.1,5,10) #x less than A
```

```
## [1] 0
```

```r
duniform(x.2,5,10) #x between a and b
```

```
## [1] 0.2
```

```r
duniform(x.3,5,10) #x larger than b
```

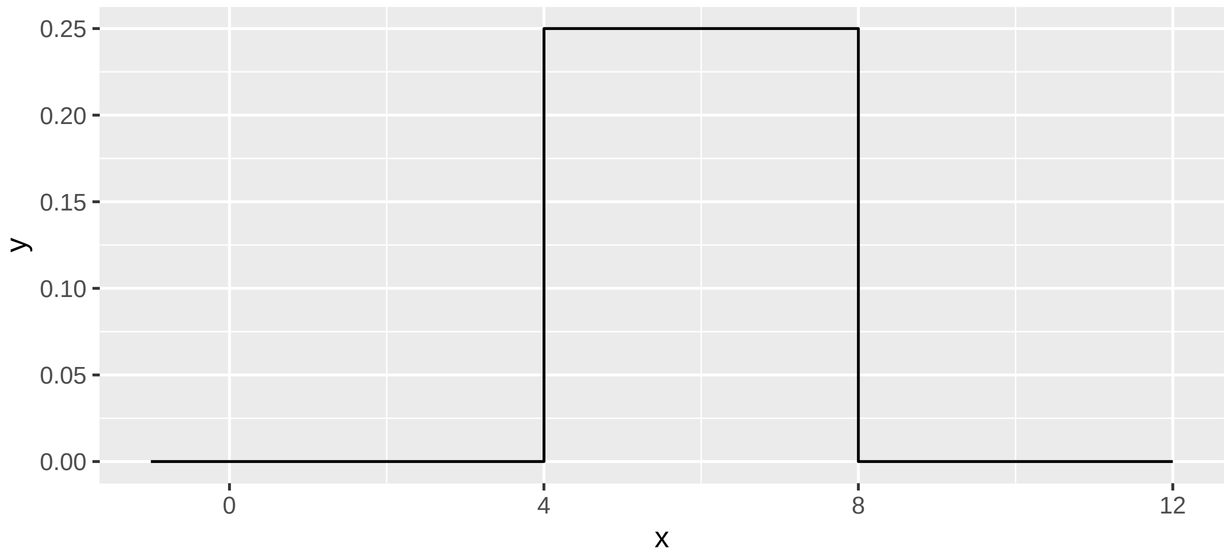
```
## [1] 0
```

b) Next lets improve our function to work correctly for a vector of `x` values. Modify your function in part (a) so that the core logic uses a `for`-loop statement and the loop moves through each element of `x` in succession. Since this is a bit more of a complex task, your function should look something like this:

```r
duniform <- function(x, a, b){
  output <- NULL
  for( i in 1:length(x) ){
    if( x[i] > a & x[i] < b ){
     output[i] <- 1/(b-a)
    }else{
     output[i] <- 0
    }
  }
  return(output)
}
```

Verify that your function works correctly by running the following code:

```r
data.frame(x = seq(-1, 12, by = 0.001)) %>%
  mutate(y = duniform(x, 4, 8)) %>%
  ggplot(aes(x = x, y = y)) +
  geom_step()
```



c) Install the R package `microbenchmark`. We will use this to discover the average duration (time) your function takes to execute code. Execute the following

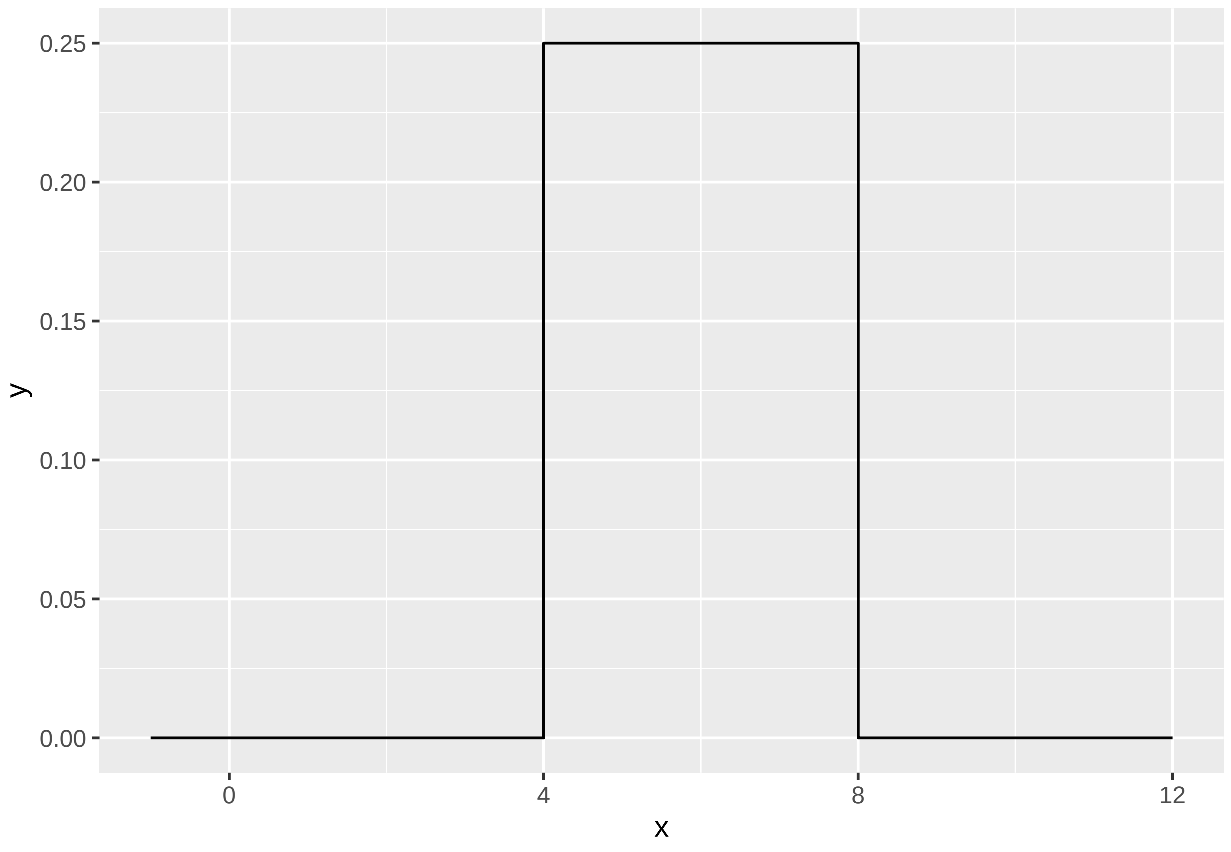
```
microbenchmark::microbenchmark( duniform( seq(-4,12,by=.0001), 4, 8), times=100)
```

```
## Unit: milliseconds
##                                      expr     min       lq     mean  median
##  duniform(seq(-4, 12, by = 1e-04), 4, 8) 45.2756 48.49765 54.43971 50.4143
##        uq      max neval
##  55.18805 132.7172   100
```

This will call the input R expression (your `duniform` function on a rather large vector of data) 100 times and report summary statistics on how long it took for the code to run. In particular, look at the median time for evaluation.

d) Instead of using a `for` loop, it might have been easier to use an `ifelse()` command, which inherently accepts vectors. Rewrite your function one last time, this time avoiding the `for` loop and instead introducing the *vectorizable* `ifelse()` command. Verify that your function works correctly by producing a plot of a uniform density. Finally, run the `microbenchmark()` code above again.

```
duniform <- function(x, a, b) {
 output <- ifelse(x >= a & x <= b, 1 / (b - a), 0)
  return(output)
}

data.frame(x = seq(-1, 12, by = 0.001)) %>%
  mutate(y = duniform(x, 4, 8)) %>%
  ggplot(aes(x = x, y = y)) +
  geom_step()
```



```
microbenchmark::microbenchmark( duniform( seq(-4,12,by=.0001), 4, 8), times=100)
```

```
## Unit: milliseconds
##                                     expr    min      lq     mean   median
##  duniform(seq(-4, 12, by = 1e-04), 4, 8) 3.5843 5.29335 6.895244 5.863252
##        uq     max neval
##  7.649802 61.5481   100
```

e) Comment on Which version of your function was easier to write? Which ran faster?

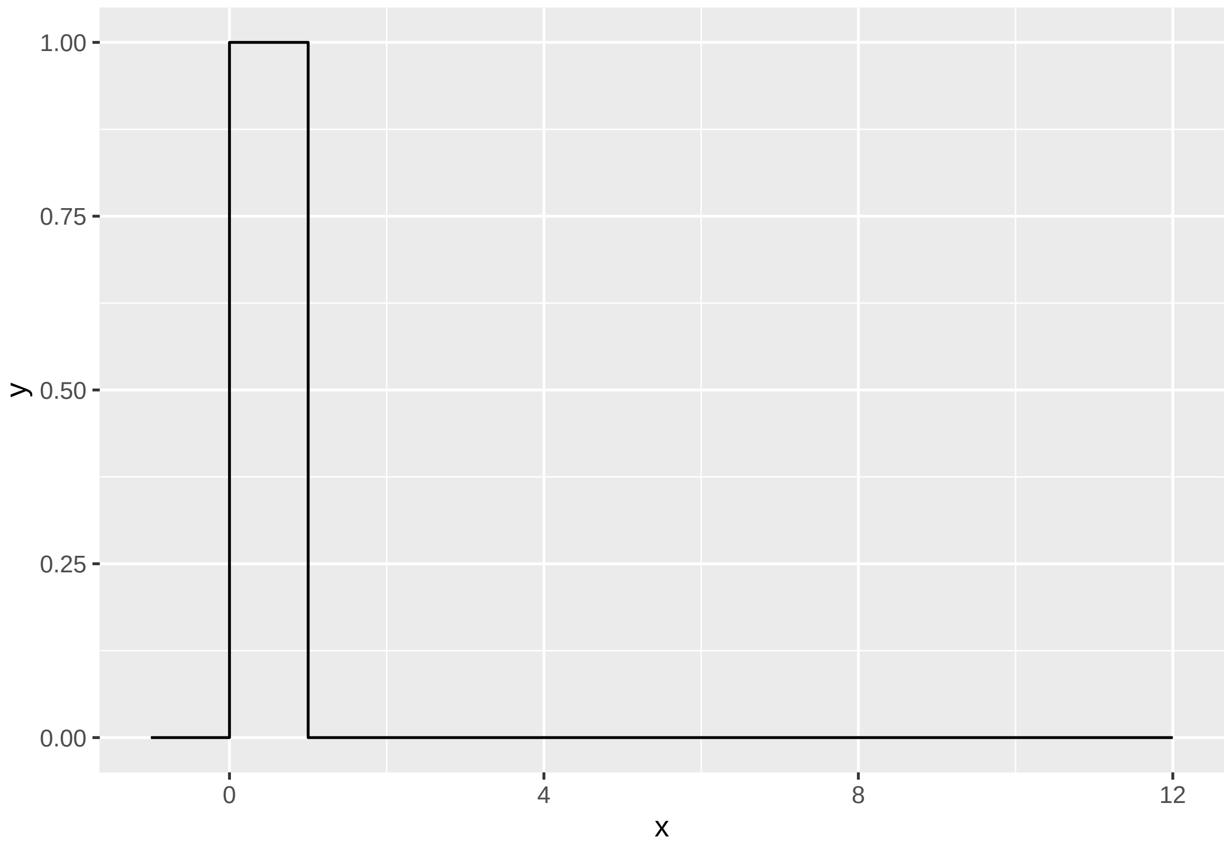
The simplier ifelse function was significantly faster and easier to write.

Exercise 2

I very often want to provide default values to a parameter that I pass to a function. For example, it is so common for me to use the `pnorm()` and `qnorm()` functions on the standard normal, that R will automatically use `mean=0` and `sd=1` parameters unless you tell R otherwise. This was discussed significantly in the chapter above. To get that behavior, we can set the default parameter values in the definition of a function. When the function is called, the user specified value is used, but if none is specified, the defaults are used. Look at the help page for the functions `dunif()`, and notice that there are a number of default parameters.

For your `duniform()` function provide default values of `0` and `1` for the arguments `a` and `b`. Demonstrate that your function is appropriately using the given default values by producing a graph of the density without specifying the `a` or `b` arguments.

```
duniform <- function(x, a=0, b=1) {
 output <- ifelse(x >= a & x <= b, 1 / (b - a), 0)
  return(output)
}

data.frame(x = seq(-1, 12, by = 0.001)) %>%
  mutate(y = duniform(x)) %>%
  ggplot(aes(x = x, y = y)) +
  geom_step()
```



Exercise 3

A common data processing step is to *standardize* numeric variables by subtracting the mean and dividing by the standard deviation. Mathematically, the standardized value is defined as

$$z = \frac{x - \bar{x}}{s}$$

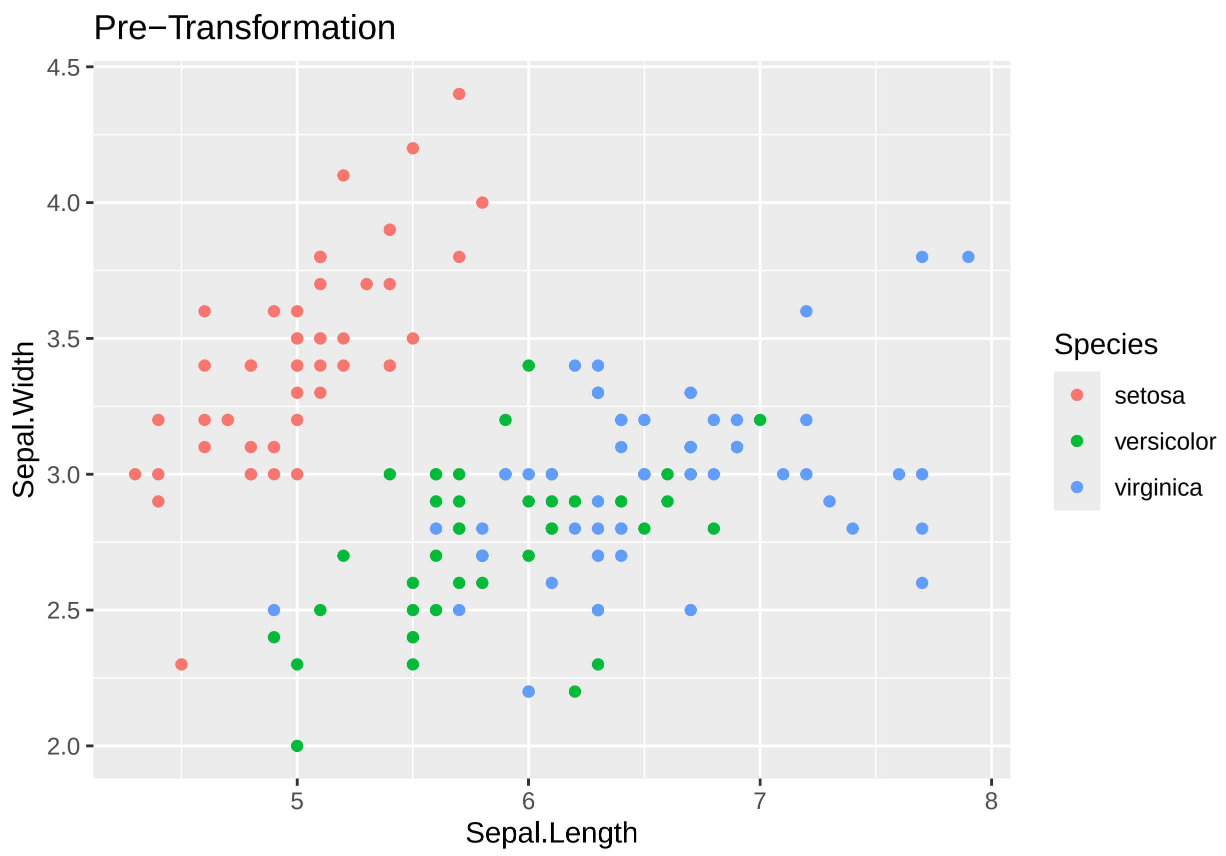
where $\bar{x}$ is the mean and s is the standard deviation.

a) Create a function that takes an input vector of numerical values and produces an output vector of the standardized values.

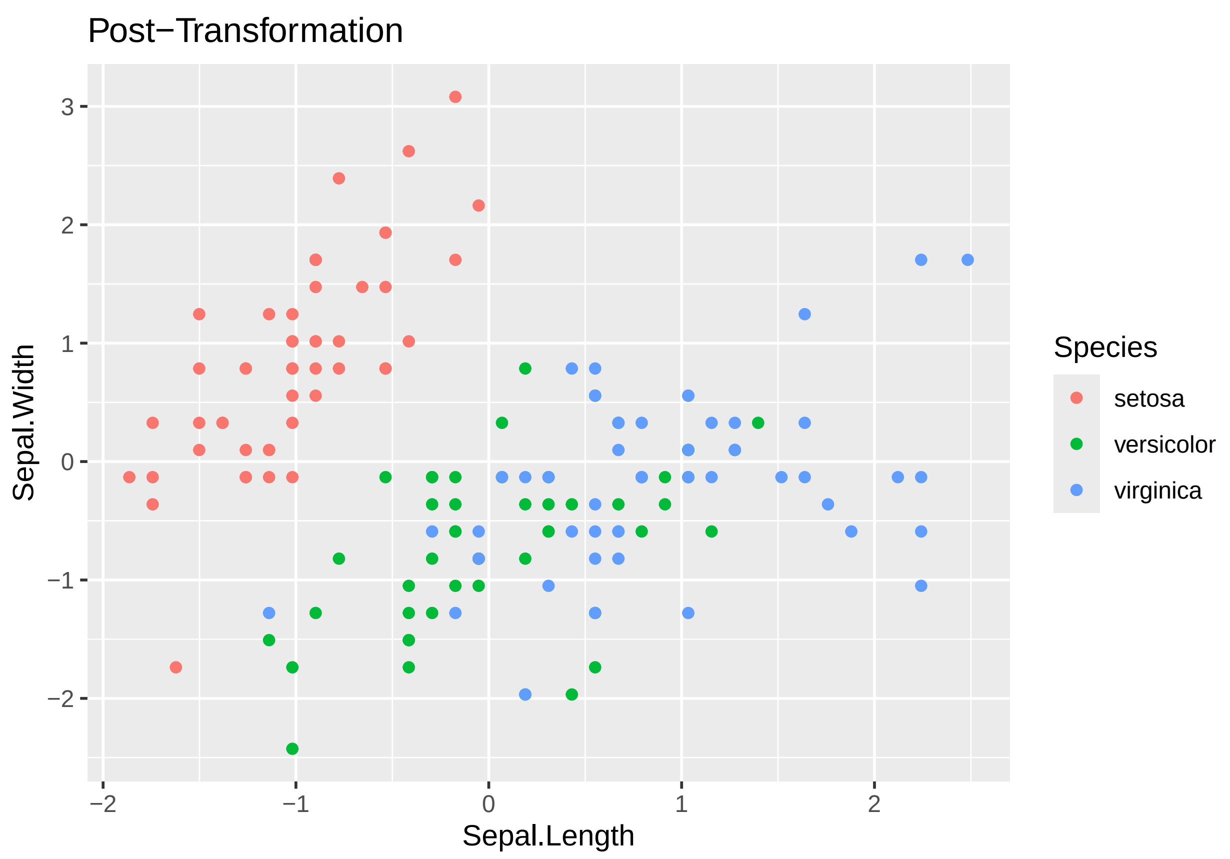
```
standardize <- function(x){
  m <- mean(x)
  sd <- sd(x)
  output <- (x-m)/sd
  return(output)
}
```

b) Apply this function to each numeric column in a data frame using the `dplyr::across()` or the `dplyr::mutate_if()` commands. *This is often done in model algorithms that rely on numerical optimization methods to find a solution. By keeping the scales of different predictor covariates the same, the numerical optimization routines generally work better.* Below is some code that should really help once your `standardize()` function is working. The graphs may not look very different, but pay attention to the x- and y-axis scales!

```r
data( 'iris' )
# Graph the pre-transformed data.
ggplot(iris, aes(x=Sepal.Length, y=Sepal.Width, color=Species)) +
  geom_point() +
  labs(title='Pre-Transformation')
```



```r
# Standardize all of the numeric columns
# across() selects columns and applies a function to them
# there column select requires a dplyr column select command such
# as starts_with(), contains(), or where().  The where() command
# allows us to use some logical function on the column to decide
# if the function should be applied or not.
iris.z <- iris %>% mutate( across(where(is.numeric), standardize) )

# Graph the post-transformed data.
ggplot(iris.z, aes(x=Sepal.Length, y=Sepal.Width, color=Species)) +
  geom_point() +
  labs(title='Post-Transformation')
```



Exercise 4

In this exercise, you'll write a function that will output a vector of the first n terms in the child's game *Fizz Buzz*. Your function should only accept the argument n, the number to which you wish to count.

Here is a description of the game. The goal is to count as high as you can but substitute in the words `Fizz`, `Buzz` or `Fizz-Buzz` depending on the divisors of the number. Specifically, any number evenly divisible by 3 should be substituted by "Fizz", any number evenly divisible by 5 substituted by "Buzz", and if the number is divisible by both 3 and 5 (i.e. by 15) substitute "Fizz-Buzz". So a sequence of integers output by your function should look like

$$1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, \ldots$$

Hint: The `paste()` function will squish strings together. The remainder operator is `%%` where it is used as `9 %% 3 = 0`.

This problem was inspired by a wonderful YouTube video that describes how to write an appropriate loop to do this in JavaScript, but it should be easy enough to interpret what to do in R. I encourage you to try to write your function first before watching the video.

```
Fizz.Buzz <- function(n){
  output <- NULL
  n.1 <- c(1:n)
  for( i in 1:length(n.1) ){
    if (n.1[i] %% 3 == 0 & n.1[i] %% 5 == 0) {
    output[i] <-  "FizzBuzz"
    }else{
    if (n.1[i] %% 3 == 0) {
    output[i] <- "Fizz"
    }else{
    if (n.1[i] %% 5 == 0) {
    output[i] <- "Buzz"
    }
    else{output[i] <- n.1[i]  }
    }
      }
        }
  return(output)
}

Fizz.Buzz(30)
```

```
##  [1] "1"        "2"        "Fizz"     "4"        "Buzz"     "Fizz"
##  [7] "7"        "8"        "Fizz"     "Buzz"     "11"       "Fizz"
## [13] "13"       "14"       "FizzBuzz" "16"       "17"       "Fizz"
## [19] "19"       "Buzz"     "Fizz"     "22"       "23"       "Fizz"
## [25] "Buzz"     "26"       "Fizz"     "28"       "29"       "FizzBuzz"
```

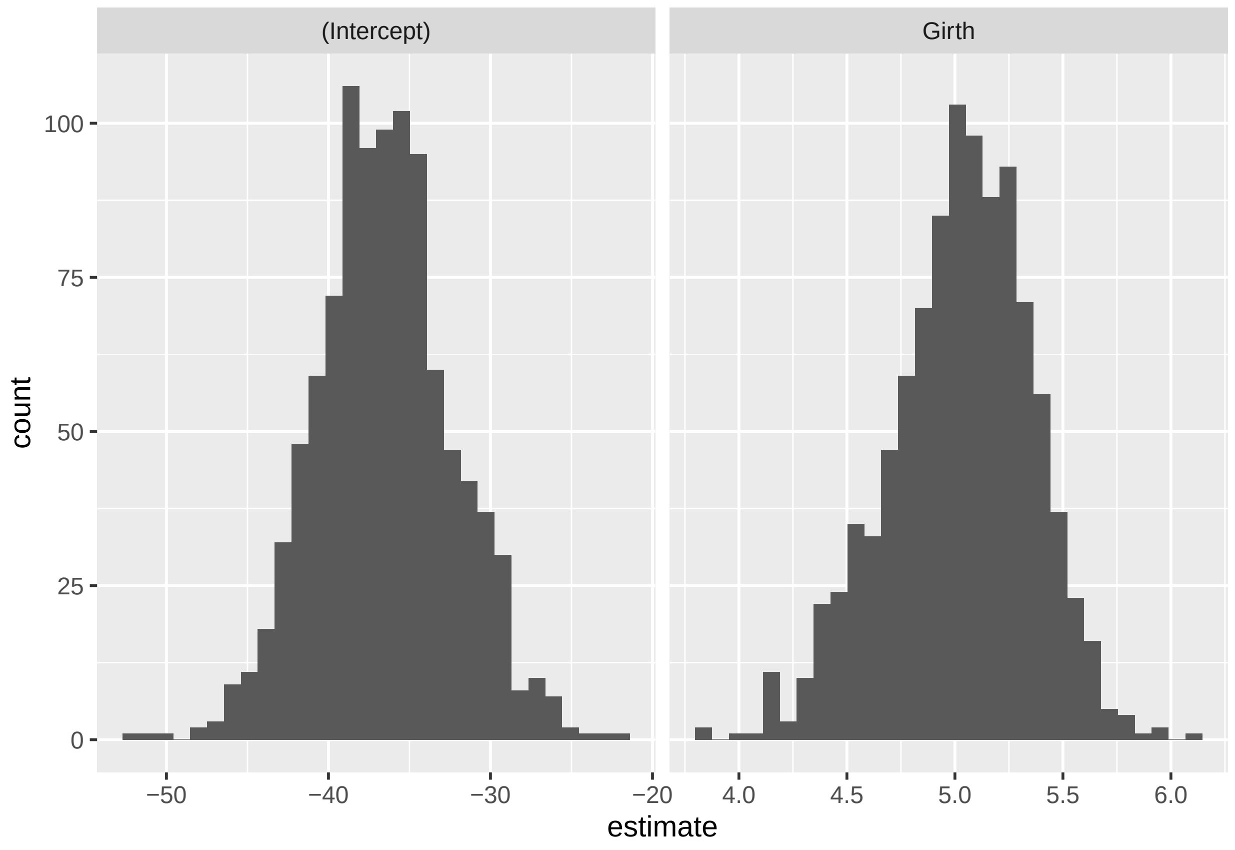
Exercise 6 (Optional)

A common statistical requirement is to create bootstrap confidence intervals for a model statistic. This is done by repeatedly re-sampling with replacement from our original sample data, running the analysis for each re-sample, and then saving the statistic of interest. Below is a function `boot.lm` that bootstraps the linear model using case re-sampling.

```
#' Calculate bootstrap CI for an lm object
#'
#' @param model
#' @param N
boot.lm <- function(model, N=1000){
  og.data    <- model$model  # Extract the original data
  formula <- model$terms  # and model formula used

  # Start the output data frame with the full sample statistic
  output <- broom::tidy(model) %>%
    select(term, estimate) %>%
    pivot_wider(names_from=term, values_from=estimate)

  for( i in 1:N ){
    data <- og.data %>% sample_frac( replace=TRUE )
    model.boot <- lm( formula, data=data)
    coefs <- broom::tidy(model.boot) %>%
      select(term, estimate) %>%
      pivot_wider(names_from=term, values_from=estimate)
    output <- output %>% rbind( coefs )
  }

  return(output)
```

```r
}

# Run the function on a model
m <- lm( Volume ~ Girth, data=trees )
boot.dist <- boot.lm(m)

# If boot.lm() works, then the following produces a nice graph
boot.dist %>% gather('term','estimate') %>%
  ggplot( aes(x=estimate) ) +
  geom_histogram() +
  facet_grid(.~term, scales='free')
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Unfortunately, the code above does not correctly calculate a bootstrap sample for the model coefficients. It has a bug… Figure out where the mistake is and fix it! *Hint: Even if you haven't studied the bootstrap, my description above gives enough information about the bootstrap algorithm to figure this out.*