Assignment Two

Rebecca Chavez

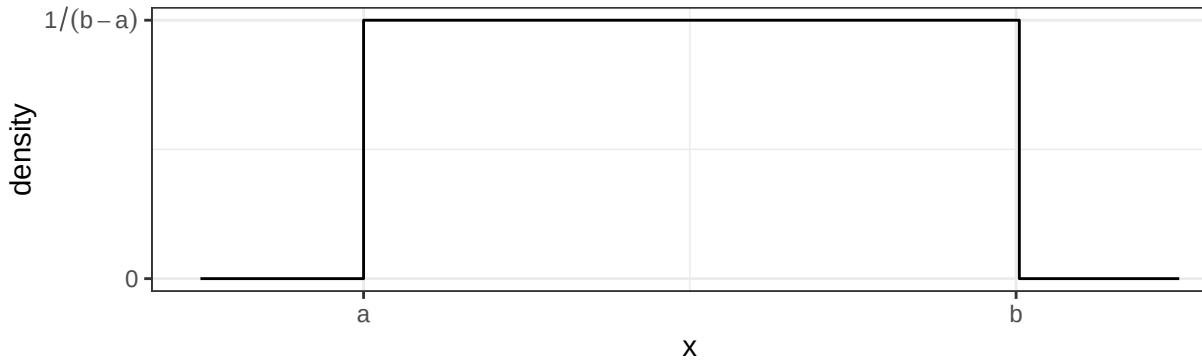
2023-10-10

Chapter Ten

Question One

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

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



which looks like this
We want to write a function `duniform(x, a, b)` that takes an arbitrary value of `x` and parameters a and b and return the appropriate height of the density function. 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 with a three times, once where $x< a$, once where $a < x < b$, and finally once where $b < x$.`

```
# use function() with parameters to define a function
duniform <- function(x, a, b) {
  # check if x is between a and b
  if(x >= a && x <= b) {
    # assign to output
    output <- 1/(b-a)
  }

  # assume x is not between a and b
  else {
```

```
    # assign to output
    output <- 0
  }

  # return output
  return(output)
}

# check function with three test vals
duniform(3, 5, 10)
```

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

```
duniform(6, 5, 10)
```

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

```
duniform(12, 5, 10)
```

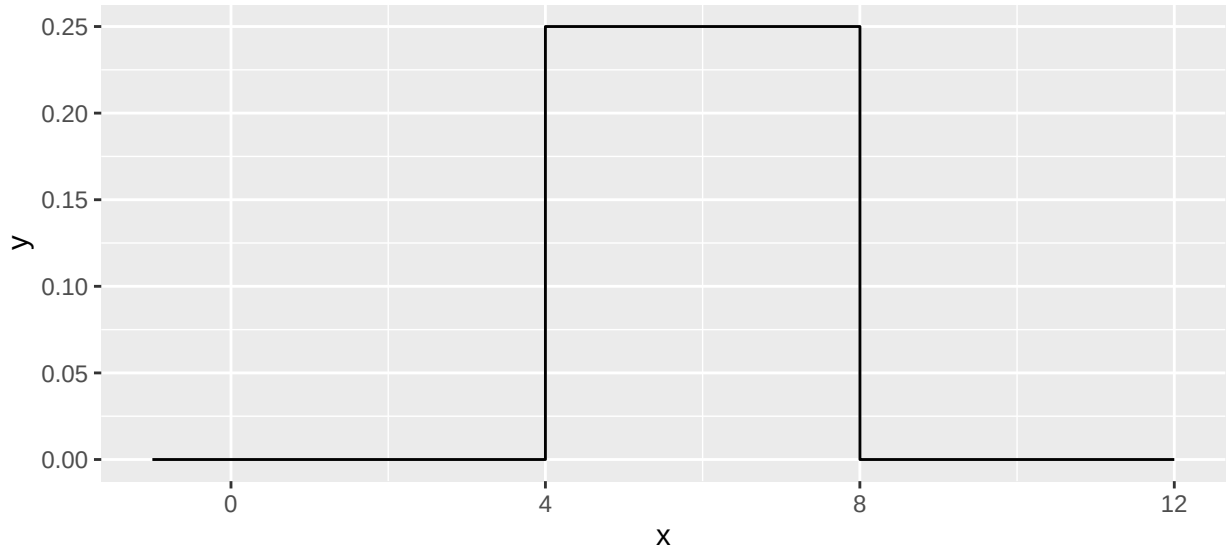
```
## [1] 0
```

b) Next we force our function to work correctly for a vector of ‘x‘ values. Modify your function in part (a) so that the core logic is inside a ‘for‘ statement and the loop moves through each element of ‘x‘ in succession. Your function should look something like this:

```
duniform <- function(x, a, b) {
  output <- NULL
  # get length of x for loop variable
  n <- length(x)

  for( i in 1:n ) {  # Set the for loop to look at each element of x
    # check if element of x is between a and b
    if( x[i] >= a & x[i] <= b) {
      # assign to element of output
      output[i] <- 1/(b-a)
    }

    # assume element of x is not between a and b
    else{
      # assign to element of output
      output[i] <- 0
    }
  }

  # return output vector
  return(output)
}
```

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

```r
data.frame( x=seq(-1, 12, by=.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 your function takes.

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

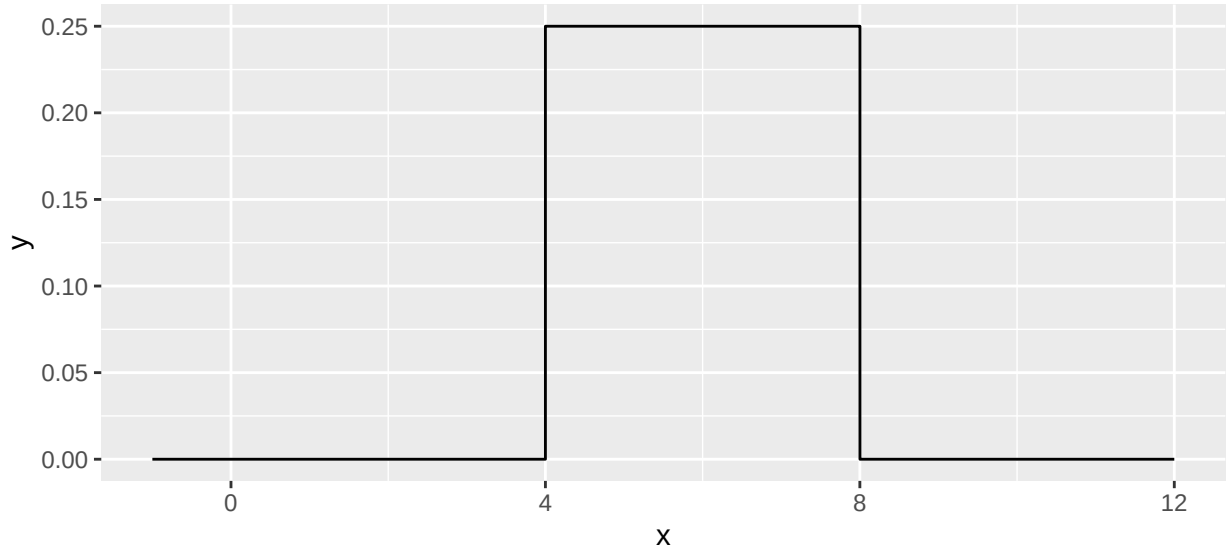
```
## Unit: milliseconds
##                                      expr      min       lq     mean   median
##  duniform(seq(-4, 12, by = 1e-04), 4, 8) 70.48141 74.83689 79.18177 77.14366
##        uq      max neval
##  79.66417 177.5081   100
```

This will call the input R expression 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. Rewrite your function to avoid the ‘for‘ loop and just use an ‘ifelse()‘ command. Verify that your function works correctly by producing a plot, and also run the ‘microbenchmark()‘. Which version of your function was easier to write? Which ran faster?

```r
duniform <- function(x, a, b) {
  # ifelse() will perform comparison for each element of the vector
  # without using a for loop
  output <- ifelse(x >= a & x <= b, 1/(b-a), 0)
```

```r
  # return output vector
  return(output)
}

# check graph
data.frame( x=seq(-1, 12, by=.001) ) %>%
  mutate( y = duniform(x, 4, 8) ) %>%
  ggplot( aes(x=x, y=y) ) +
  geom_step()
```




```r
# check median time
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) 6.276899 8.659947 11.07824 9.628695
##        uq     max neval
##  12.22627 96.2436   100
```

The version using the `ifelse()` statement was easier to write, as it was accomplished by only one line as opposed to the several lines necessary when using the `for()` loop. The `ifelse()` version was also much faster, with a median speed of 8 milliseconds instead of 78 milliseconds.

Question Two

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. To get that behavior, we just set the default parameter values in the definition. 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 `a` and `b`. Demonstrate that your function is appropriately using the given default values.

```r
# define default variables in the function header
duniform <- function(x, a=0, b=1) {
  output <- ifelse(x >= a & x <= b, 1/(b-a), 0)

  return(output)
}

# check using only an x value
duniform(-2)
```

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

```r
duniform(.5)
```

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

```r
duniform(3)
```

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

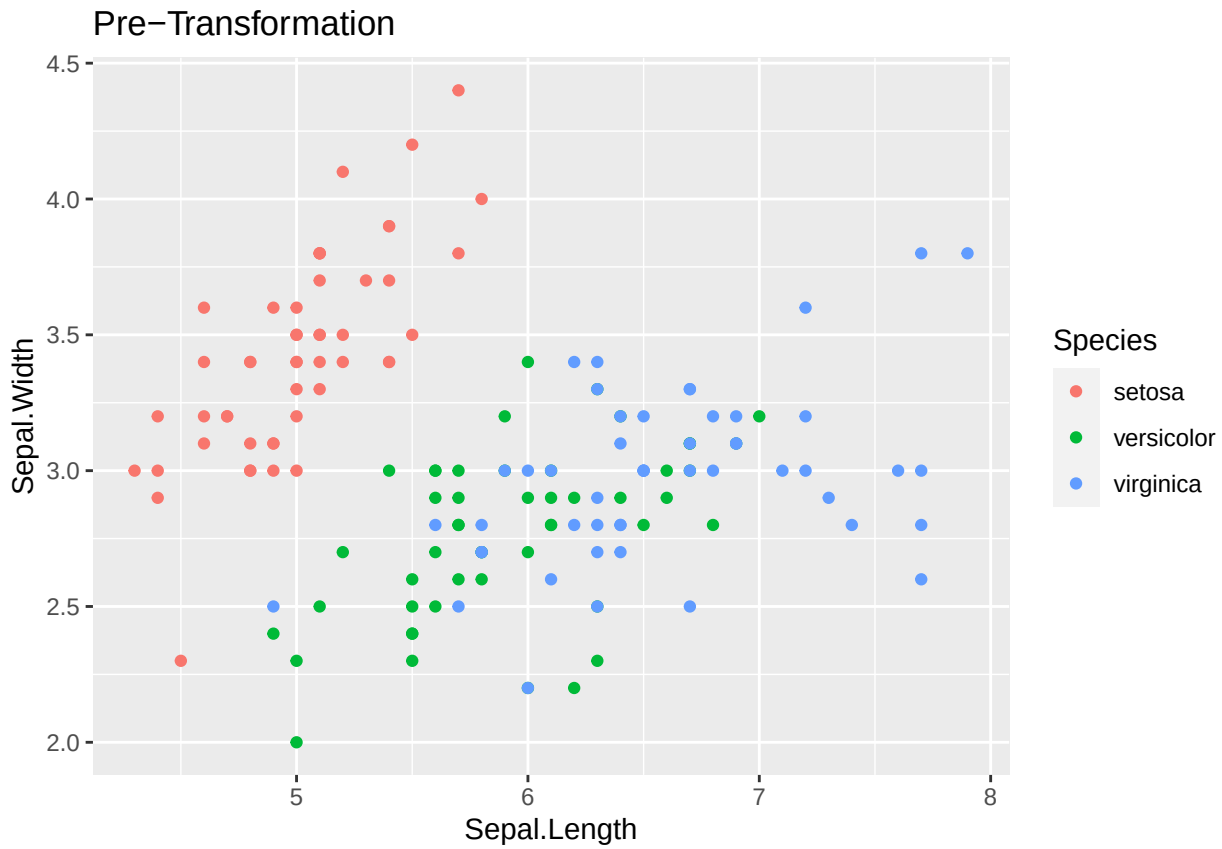
Question Three

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

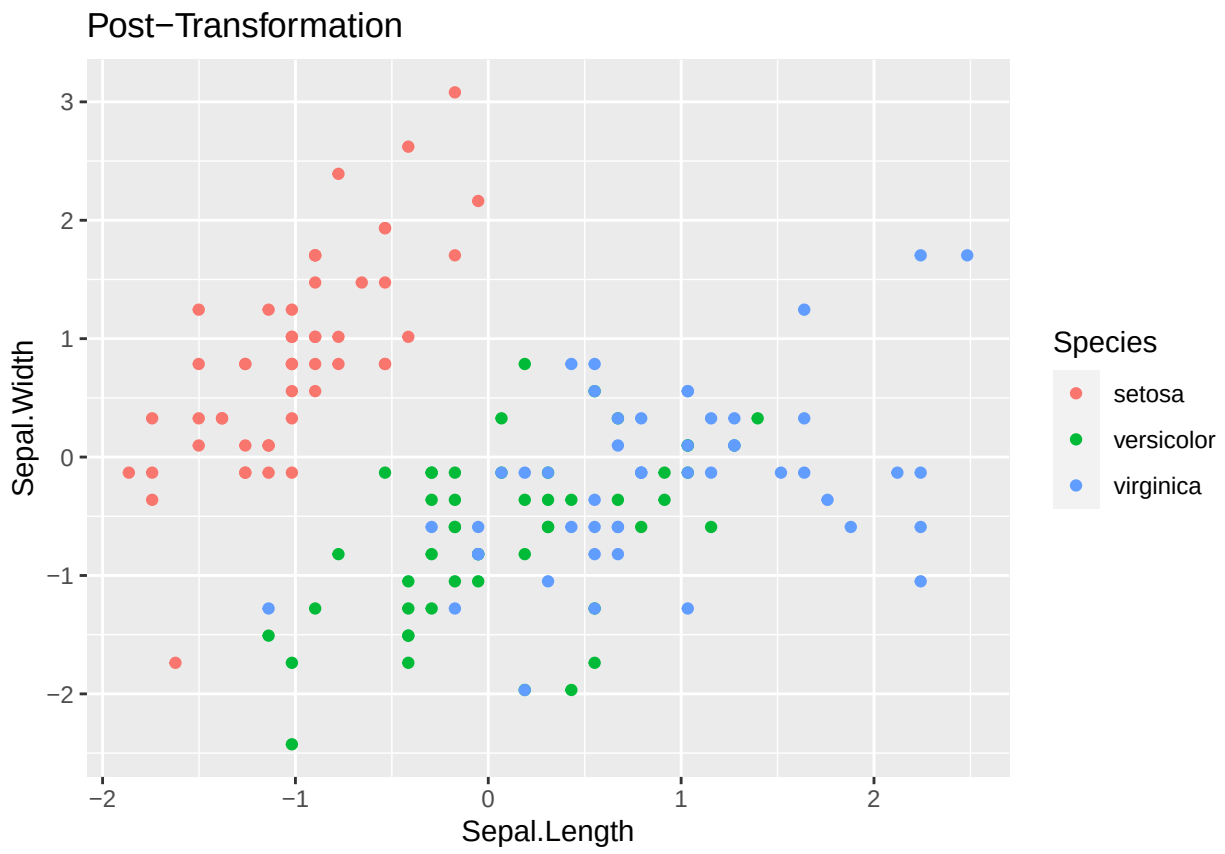
$$z = \frac{x - \bar{x}}{s}$$

where $\bar{x}$ is the mean and s is the standard deviation. Create a function that takes an input vector of numerical values and produces an output vector of the standardized values. We will then 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.*

```r
standardize <- function(x){
  # get average of x vector
  avg <- mean(x)
  # get std dev of x vector
  stdDev <- sd(x)
  # standardize
  standardOut <- (x - avg) / stdDev

  return(standardOut)
}

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




```
# 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')
```



Question Four

In this example, we'll write a function that will output a vector of the first n terms in the child's game *Fizz Buzz*. The goal is to count as high as you can, but for any number evenly divisible by 3, substitute "Fizz" and any number evenly divisible by 5, substitute "Buzz", and if it is divisible by both, substitute "Fizz Buzz". So the sequence will look like 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, ... *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.*

```
fizzBuzz <- function(x) {
  outVector <- NULL

  # loop for x numbers
  for(i in 1:x) {
    # check if divisible by both three and five
    if((i %% 3) == 0 & (i %% 5) == 0) {
      # add fizz to vector
      outVector[i] <- "Fizz Buzz"
    }

    # check if only divisible by five
    else if((i %% 5) == 0) {
      # add buzz to vector
```

```
      outVector[i] <- "Buzz"
    }

    # check if only divisible by three
    else if((i %% 3) == 0) {
      # add fizz buzz to vector
      outVector[i] <- "Fizz"
    }

    # assume not divisible by three or five
    else {
      # add number to vector
      outVector[i] <- i
    }
  }

  return(outVector)
}

fizzBuzz(16)
```

```
##  [1] "1"         "2"         "Fizz"      "4"         "Buzz"      "Fizz"
##  [7] "7"         "8"         "Fizz"      "Buzz"      "11"        "Fizz"
## [13] "13"        "14"        "Fizz Buzz" "16"
```

Question Five

The `dplyr::fill()` function takes a table column that has missing values and fills them with the most recent non-missing value. For this problem, we will create our own function to do the same.

```
#' Fill in missing values in a vector with the previous value.
#'
#' @parm x An input vector with missing values
#' @result The input vector with NA values filled in.
myFill <- function(x){
  lastUsed <- NULL
  outVector <- NULL
  n <- length(x)

  # loop through input vector
  for(i in 1:n) {
    # check for not na value
    if(!is.na(x[i])) {
      # set last used
      lastUsed <- x[i]
    }

    # set out vector to last used
    outVector[i] <- lastUsed
  }

  return(outVector)
}
```

The following function call should produce the following ouput

```
test.vector <- c('A',NA,NA, 'B','C', NA,NA,NA)
myFill(test.vector)
```

```
## [1] "A" "A" "A" "B" "C" "C" "C" "C"
```

```r
[1] "A" "A" "A" "B" "C" "C" "C" "C"
```

Challenge

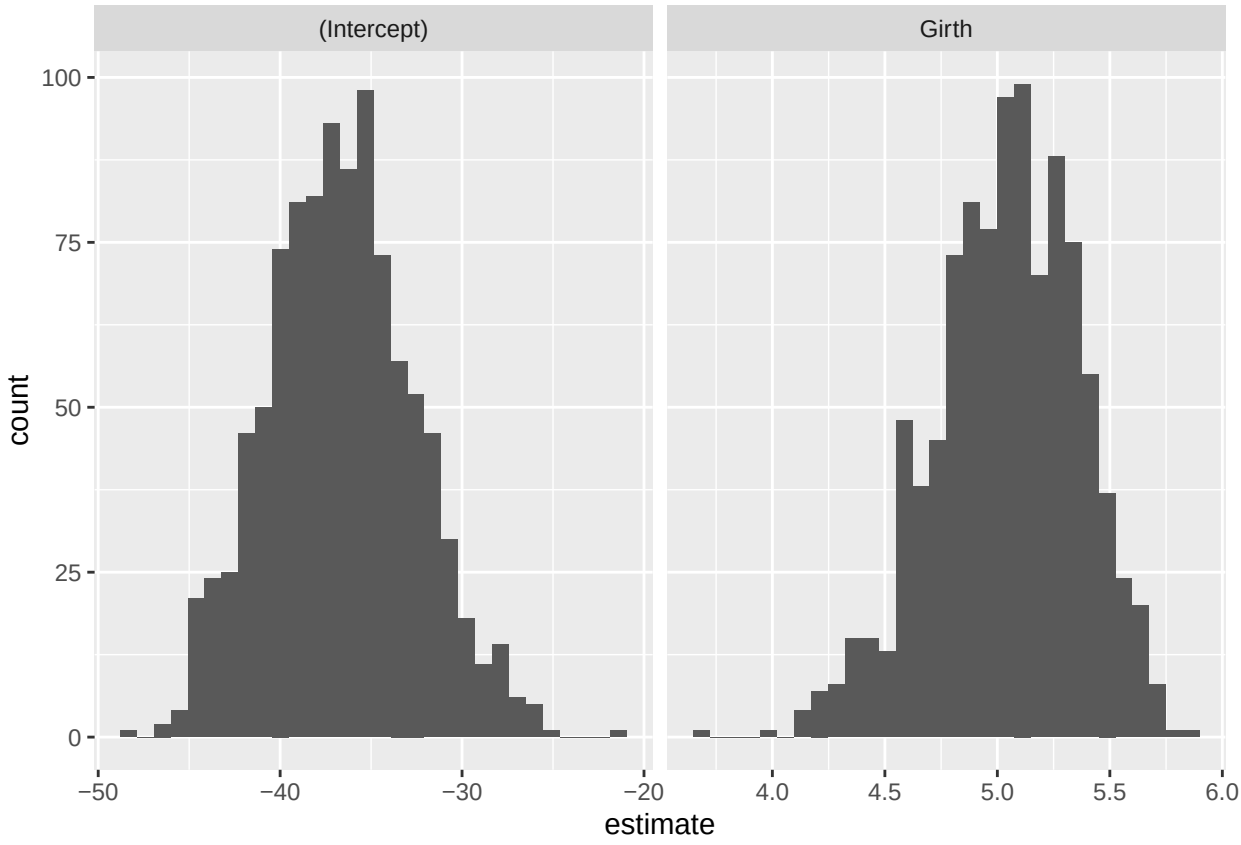
Question Six

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){
      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 ){
        # save into new variable instead of back into itself
        dataSample <- data %>% sample_frac( replace=TRUE )
        model.boot <- lm( formula, data=dataSample)
        coefs <- broom::tidy(model.boot) %>%
          select(term, estimate) %>%
          pivot_wider(names_from=term, values_from=estimate)
        output <- output %>% rbind( coefs )
      }

      return(output)
    }

    # 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`.
```



This code does not correctly calculate a bootstrap sample for the model coefficients. Figure out where the mistake is.
Hint: Even if you haven't studied the bootstrap, my description above gives
enough information about the bootstrap algorithm to figure this out.