STA 445 Ass2

Prince Kwame Gyimah

2023-10-10

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

Question 1

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 $b < x$.

```
duniform <- function(x, a, b){

  f_x  <-  ifelse(x>=a & x<=b , 1/(b-a), 0)

  return(f_x)
}

duniform(2,1,3)
```

```
## [1] 0.5
```

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 <- c()

  for( i in 1:length(x) ){  # Set the for loop to look at each element of x

    if(x[i]>=a & x[i]<=b ){  # What should this logical expression be?
```

```
        output[i] <- 1/(b-a)
    }else{
      output[i] <- 0
    }
  }
  return(output)
}
#duniform(1:10,1,3)
```

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

```
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.

```
#install.packages('microbenchmark')
```

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

```
## Unit: milliseconds
##                                      expr    min     lq     mean  median      uq
##  duniform(seq(-4, 12, by = 1e-04), 4, 8) 61.237 64.241 66.80253 66.1788 67.4199
##       max neval
##  109.7458   100
```

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?

```
duniform2 <- function(x, a, b){
  output <- ifelse(x>=a & x<=b,1/(b-a),0)
  return(output)
}
```

```
##Plot
data.frame( x=seq(-1, 12, by=.001) ) %>%
  mutate( y = duniform2(x, 4, 8) ) %>%
  ggplot( aes(x=x, y=y) ) +
  geom_step()

microbenchmark::microbenchmark( duniform2( seq(-4,12,by=.0001), 4, 8), times=100)
```

The modified function `duniform2` runs faster than `duniform`.This is because the execution time of the function `duniform2` is less than the execution time of duniform.

Question 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. 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.

```
duniform2 <- function(x, a=0, b=1){
  output <- ifelse(x>=a & x<=b,1/(b-a),0)
  return(output)
}
```

Question 3

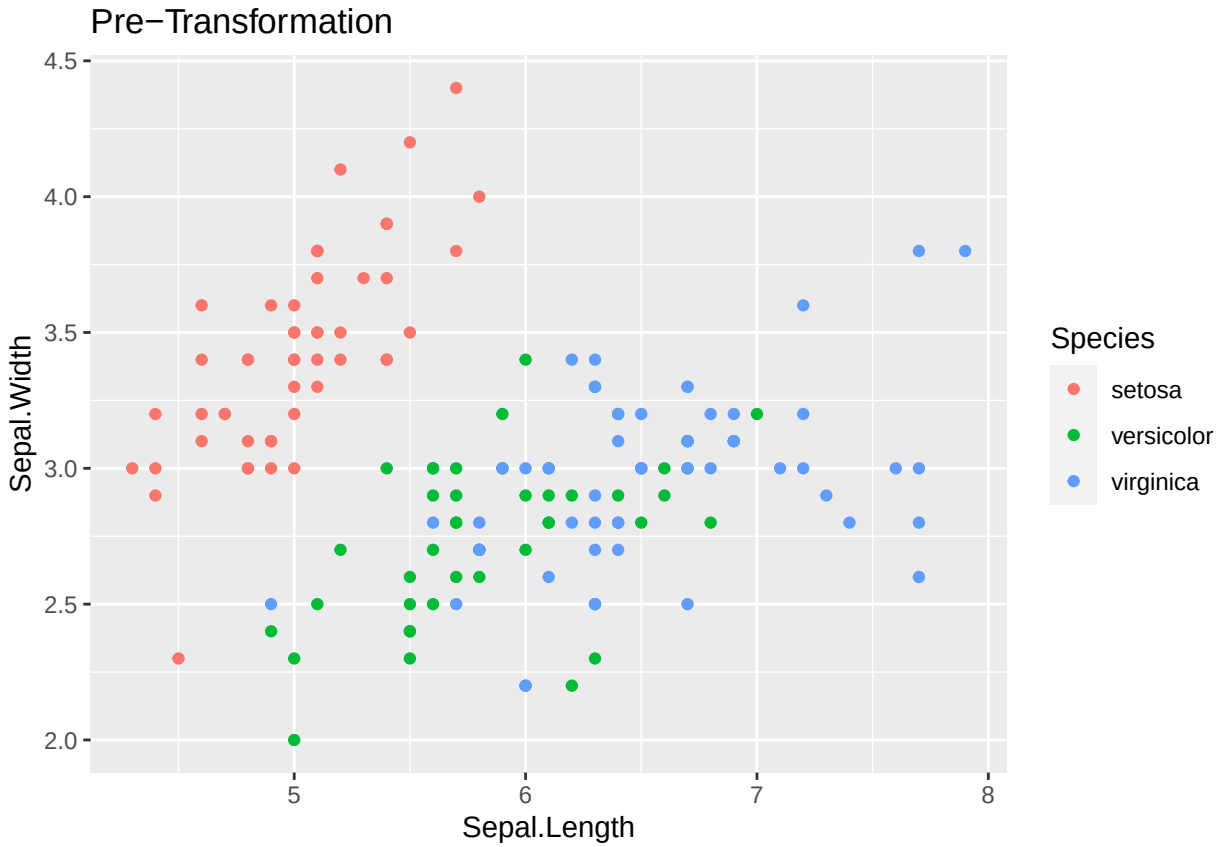
A common data processing step is to *standardize* numeric variables by subtracting the mean and dividing by the standard deviation. Mathematically, the standardized value is defined as

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

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

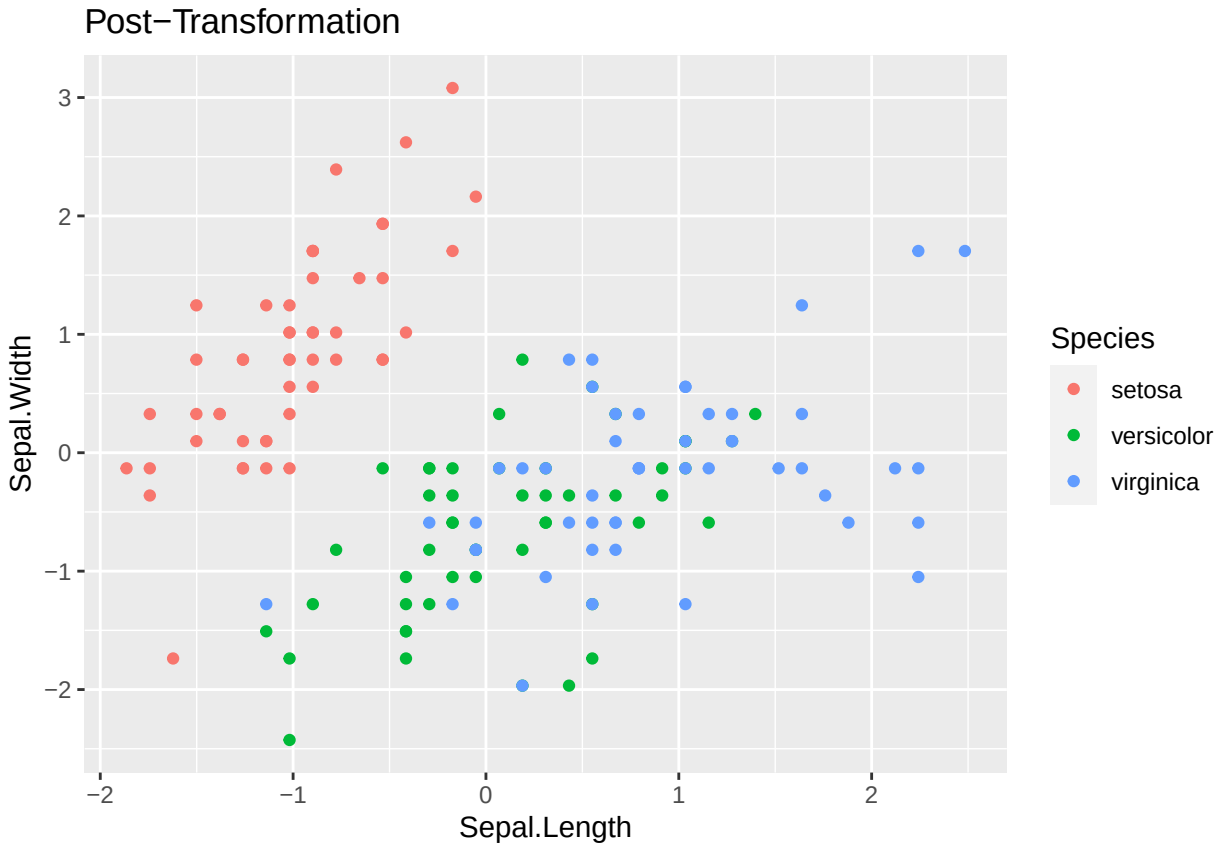
```
    standardize <- function(x){
      std <- (x-mean(x))/sd(x)
      return(std)
    }

    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 4

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.*

```
fizzBuzzGame <-  function(x){
  counter =c()

  for(i in 1:length(x)){
    if(x[i]%%3==0 & x[i]%%5==0){
      counter <- c(counter,'Fizz Buzz')
    }
     else if(x[i]%%3==0){
      counter <- c(counter,'Fizz')
    }
   else if(x[i]%%5==0) {
     counter <- c(counter,'Fizz')
   }
    else{
       counter <- c(counter,x[i])
```

```
    }
  }
  return(counter)
}
fizzBuzzGame(1:20)
```

```
##  [1] "1"         "2"         "Fizz"      "4"         "Fizz"      "Fizz"
##  [7] "7"         "8"         "Fizz"      "Fizz"      "11"        "Fizz"
## [13] "13"        "14"        "Fizz Buzz" "16"        "17"        "Fizz"
## [19] "19"        "Fizz"
```

Question 5

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){
  for(i in 1:length(x)){
    if(is.na(x[i])){
      x[i] =x[i-1]
    }
  }

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

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