445_Assignment2

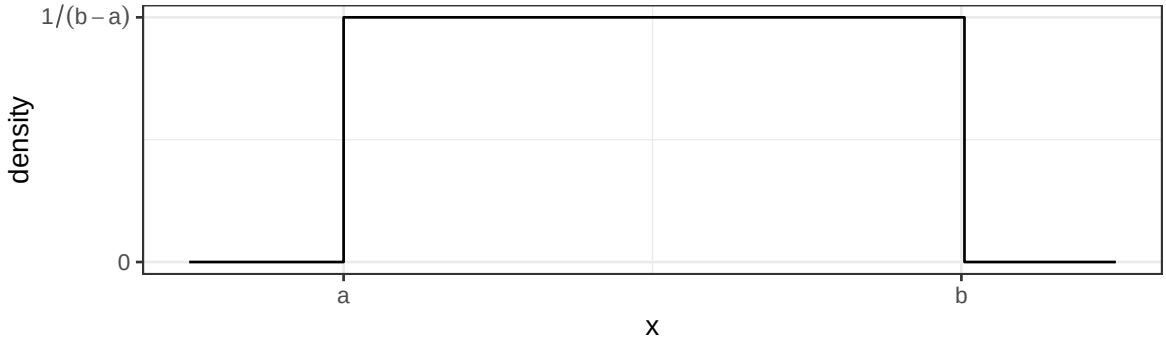
2023-10-12

Exercises

1. 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 $b < x$.

x < a <: | a < x < b: | x > b: duniform(0, 1, 3) | duniform(2, 1, 3) | duniform(4, 1, 3) [1] 0 | [1] 0.5 | [1] 0

```
duniform <- function(x, a, b){
  if( x >= a & x<= b){ # use density formula if x is between/equal to a & b
    density <- 1/(b-a)
  }else{
    density <- 0 # if x above/below a & b, then value is 0
  }
  return(density)
}
# duniform(4, 1, 3)
```

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.

```
duniform <- function(x, a, b){
  density <- NULL
  for( i in 1:length(x) ){
    if( x[i] >= a & x[i] <= b ){  # a <= x <= b
      density[i] <- 1/(b-a)
    }else{
      density[i] <- 0 # density = 0 if first if is not satisfied
    }
  }
  return(density)
}
test.vector <- c(1, 3, 3.5, 6)
duniform(test.vector, 2, 4)
```

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.

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

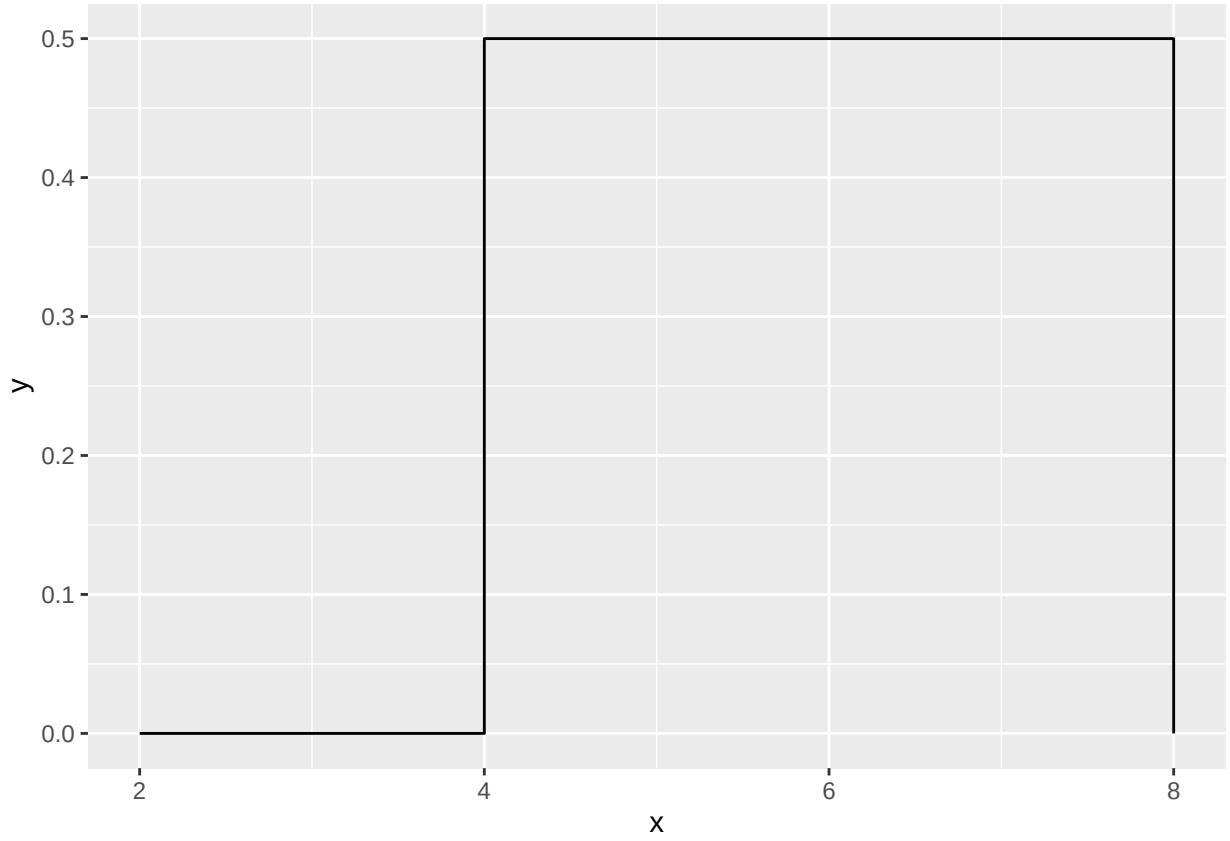
d) Instead of using a ‘for‘ loop, it might have been easier to use an ‘ifelse()‘ command. Rewrite your function to avoid the ‘for‘ loop and just use an ‘ifelse()‘ command. Verify that your function works correctly by producing a plot, and also run the ‘microbenchmark()‘. Which version of your function was easier to write? Which ran faster?

The if_else statement was easier to write than the for loop but the for loop ran faster.

```
duniform <- function(x, a, b){ # function with if_else
    if_else( x >= a & x <= b, 1/(b-a), 0 )
  }
test.vector <- c(1, 3, 3.5, 6)
duniform(test.vector, 2, 4)
```

```
## [1] 0.0 0.5 0.5 0.0
```

```
data.frame( x=seq(2, 8, by=2) ) %>% # plot
  mutate( y = duniform(test.vector, 2, 4) ) %>%
  ggplot( aes(x=x, y=y) ) +
  geom_step()
```




```
microbenchmark::microbenchmark( duniform( seq(2, 8, by=2), 2, 4), times=100)
```

```
## Warning in microbenchmark::microbenchmark(duniform(seq(2, 8, by = 2), 2, : less
## accurate nanosecond times to avoid potential integer overflows
```

```
## Unit: microseconds
##                               expr    min      lq     mean  median     uq
##  duniform(seq(2, 8, by = 2), 2, 4) 47.601 48.8515 50.64935 49.3845 50.102
##      max neval
##  140.917   100
```

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.

```r
# ?dunif()
duniform <- function(x, a=0, b=1){
    if_else( x >= a & x <= b, 1/(b-a), 0 ) # if_else(condition,true,false)
}
# duniform(.5)
# [1] 1

# duniform(2)
# [1] 0
```

3. A common data processing step is to *standardize* numeric variables by subtracting the mean and dividing by the standard deviation. Mathematically, the standardizedvalue 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){
  (x - mean(x))/sd(x) # x minus mean over sd
}

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')
```

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.*

```
fizz.buzz <- function(x){
  output <- NULL
  for( i in 1:length(x) ){
    if( x[i] %% 3 == 0 & x[i] %% 5 != 0 ){ output[i] <- "Fizz"} # div by 3 only
    if( x[i] %% 5 == 0 & x[i] %% 3 != 0 ){output[i] <- "Buzz"} # div by 5 only
    if(x[i] %% 3 == 0 & x[i] %% 5 == 0){output[i] <- "FizzBuzz"} # div by 3 & 5
    if(x[i] %% 3 != 0 & x[i] %% 5 != 0){output[i] <- x[i]} # not div by 3 or 5
  }
  return(output)
}
fizz.buzz(1:100) # show first 100 numbers
```

```
##   [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"
##  [31] "31"       "32"       "Fizz"     "34"       "Buzz"     "Fizz"
##  [37] "37"       "38"       "Fizz"     "Buzz"     "41"       "Fizz"
##  [43] "43"       "44"       "FizzBuzz" "46"       "47"       "Fizz"
##  [49] "49"       "Buzz"     "Fizz"     "52"       "53"       "Fizz"
##  [55] "Buzz"     "56"       "Fizz"     "58"       "59"       "FizzBuzz"
##  [61] "61"       "62"       "Fizz"     "64"       "Buzz"     "Fizz"
##  [67] "67"       "68"       "Fizz"     "Buzz"     "71"       "Fizz"
##  [73] "73"       "74"       "FizzBuzz" "76"       "77"       "Fizz"
##  [79] "79"       "Buzz"     "Fizz"     "82"       "83"       "Fizz"
##  [85] "Buzz"     "86"       "Fizz"     "88"       "89"       "FizzBuzz"
##  [91] "91"       "92"       "Fizz"     "94"       "Buzz"     "Fizz"
##  [97] "97"       "98"       "Fizz"     "Buzz"
```

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){
  output <- NULL
  for( i in 1:length(x)){
    if(!is.na(x[i])){output[i] <- x[i]} # non na values remain same
    if(is.na(x[i])){output[i] <- output[i-1]} # na values replaced by previous
  }                                          # non na value stored in output
  return(output)
}
```

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"
```

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