Assignment 3

Jack Scherer

2024-10-09

```r
library(ggplot2)
library(dplyr)
```

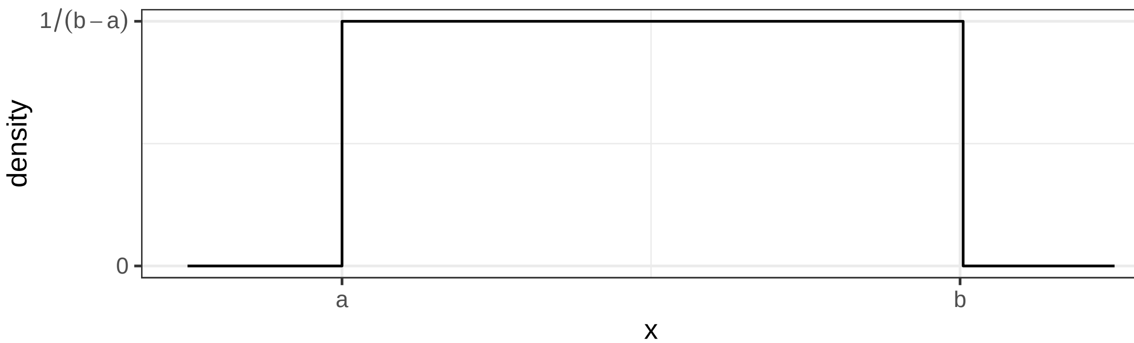
```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(microbenchmark)
```

```
## Warning: package 'microbenchmark' was built under R version 4.3.3
```

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 $b < x$.

```r
duniform <- function( x, a, b ){
  if( a<=x & x <=b ){ #if x is between a and b
    return( 1/(b-a) ) #return the uniform density
  }else{              #otherwise
    return( 0 )       #return zero
  }
}

duniform( 3, 6, 15 )
```

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

```r
duniform( 9, 6, 15 )
```

```
## [1] 0.1111111
```

```r
duniform( 20, 6, 15 )
```

```
## [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:

```r
x=c(1:20)

duniform <- function( x, a, b ){
  y = c( length(x) )
  m = c( 1:length(x) )
  for( i in m ){
    if( a<=x[i] & x[i] <=b ){ #if i is between a and b
    y[i] = ( 1/(b-a) ) #set it to the uniform density
  }else{              #otherwise
    y[i] = 0       #set it to zero
  }
  }
  return(y)
}

duniform(x,5,10)
```

```
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.

```
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?

```
duniform <- function( x, a, b ){
  y = c( length(x) )                      #create an empty vector to later return, same length as x
  y = ifelse( a<=x & x<=b, 1/(b-a), 0 ) #conditionally fill y with values
  return(y)                               #return y
}

#test this function
x=c(1:20)
duniform( x, 4, 9 )
```

```
##  [1] 0.0 0.0 0.0 0.2 0.2 0.2 0.2 0.2 0.2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
## [20] 0.0
```

```
#test the speed of this function
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) 3.4196 4.00945 5.346621 4.2164 7.1189
##     max neval
##  9.3604   100
```

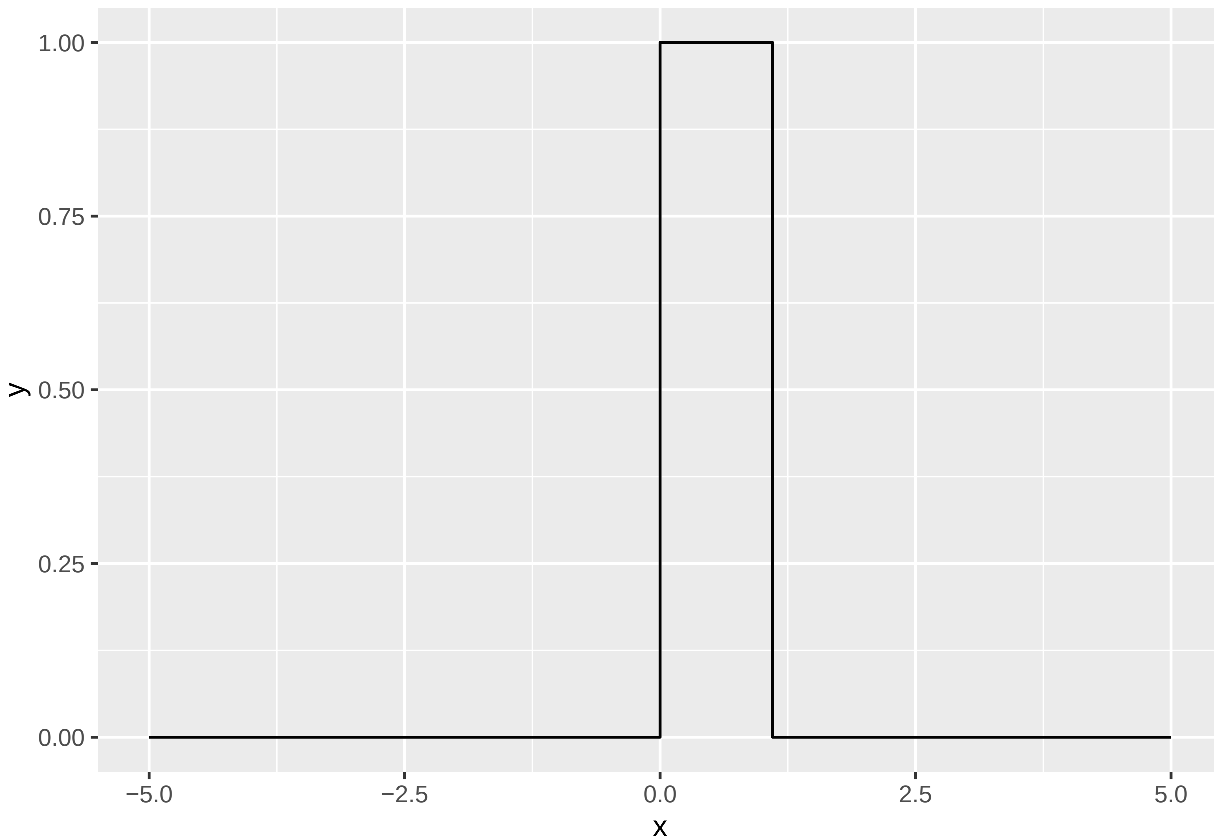
e) Comment on Which version of your function was easier to write? Which ran faster? The version of the function that uses the ifelse command is much more streamlined as it utilizes an existing function that can operate on a vector, thus avoiding the for loop. It is also faster, as evidenced by the microbenchmark function outputs.

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.

```r
duniform <- function( x, a=0, b=1 ){
  y = c( length(x) )                    #create an empty vector to later return, same length as x
  y = ifelse( a<=x & x<=b, 1/(b-a), 0 ) #conditionally fill y with values
  return(y)  #return y
}

values <- seq( -5, 5, 0.1) #produce a vector of values to test the duniform function on
d.values <- duniform(values) #perform the duniform function on the test vector
df <- data.frame( x=values, y=d.values )

ggplot( df, 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.

```r
standardize <- function(x){
  (x-mean(x))/sd(x)  ## subtract the mean and divide by the standard deviation
}
```

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')

# 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.

```r
fizzbuzz <- function( n ){

  output <- n   #vector to store output values
```

```r
  for( i in 1:length( n ) ){
    if( i %% 3 == 0 & i %% 5 == 0 ){          #divisible by 3 and 5
      output[i] <- "Fizz-Buzz"
    } else if( i %% 3 == 0 ){        #divisible by 3
      output[i] <- "Fizz"
    } else if( i %% 5 == 0 ){          #divisible by 5
      output[i] <- "Buzz"
    }
  }
  return( output )                   #return modified vector
}

integers <- c( 1:30 )
fizzbuzz(integers)
```

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