# Programming Homework 4

Trever Yoder

### Load Packages

```
library(tidyverse)
```

```
Warning: package 'ggplot2' was built under R version 4.4.2
```

```
Warning: package 'lubridate' was built under R version 4.4.3
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.2
-- 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 becor
```

### Task 1: Conceptual Questions

```
#Create a list with the requested questions
my_list1 <- list("1. What is the purpose of the lapply() function?
                 What is the equivalent purrr function?",
"2. Suppose we have a list called my_list.
Each element of the list is a numeric data frame (all columns
are numeric). We want use lapply() to run the code
cor(numeric_matrix, method = kendall)
```

```
on each element of the list. Write code to do this below!
(I'm really trying to ask you how you specify
method = kendall when calling lapply())",
"3. What are two advantages of using purrr functions
instead of the BaseR apply family?",
"4. What is a side-effect function?",
"5. Why can you name a variable sd in a function and
not cause any issues with the sd function?")

#print the questions/list
my_list1
```

```
[[1]]
[1] "1. What is the purpose of the lapply() function? \n                 What is the equivale

[[2]]
[1] "2. Suppose we have a list called my_list.\nEach element of the list is a numeric data fi

[[3]]
[1] "3. What are two advantages of using purrr functions\ninstead of the BaseR apply family?"

[[4]]
[1] "4. What is a side-effect function?"

[[5]]
[1] "5. Why can you name a variable sd in a function and \nnot cause any issues with the sd f
```

### Question 1

lapply() is used to apply functions across many rows/columns and the "l" ensures R always outputs a list.

### Question 2

```
#create data frames
df1 <- data.frame(a = c(1:3), b = c(4:6))
df2 <- data.frame(x = c(7:9), y = c(10:12), z = c(13:15))

#create list
```

```r
my_list <- list(df1, df2)

#write/show lapply function
lapply(my_list, cor, method = "kendall")
```

```
[[1]]
  a b
a 1 1
b 1 1

[[2]]
  x y z
x 1 1 1
y 1 1 1
z 1 1 1
```

### Question 3

Two advantages of using `purr` functions instead of `BaseR` apply family are: 1. Greater consistency between functions. For example, you can predict the output type exclusively from the function name, which isn't always true for `BaseR` apply functions. 2. `Purr` also has some functions to fill in some gaps such as `imap()` where you can map simultaneously over `x` and its indices.

### Question 4

A side-effect function does something beyond it's function return value. For example, write files to a disk. If we want the side effect of `hist` (which is the visual part) we can use the `walk()` function to only print the histogram.

### Question 5

We can name a variable `sd` in a function and not cause any issues with the `sd` function because functions have their own temporary environment. So, once the function executes, all variables within that function are "gone" or in other words, not saved to the main environment.

## Task 2: Writing R Functions

### Question 1

Here I will create a function that calculates RMSE. There is an ellipses in the function to allow for additional arguments.

```
getRMSE <- function(response, predicted, ...){
  add <- list(...)
  remove <- isTRUE(add$na.rm)
  if (remove) {
    dropmissing <- !is.na(response) & !is.na(predicted)
    response <- response[dropmissing]
    predicted <- predicted[dropmissing]
  }
  sqrt(mean((response-predicted)^2))
}
```

### Question 2

Let's run some code to create some response values and predictions.

```
set.seed(10)
n <- 100
x <- runif(n)
resp <- 3 + 10*x + rnorm(n)
pred <- predict(lm(resp ~ x), data.frame(x))

#now let's test our RMSE function using this data!
getRMSE(resp, pred)
```

```
[1] 0.9581677
```

```
#Add 2 missing vlaues to resp
resp[c(1,5)] <- NA_real_

#Test with and without specifying what R should do with the missing values
getRMSE(resp, pred)
```

```
[1] NA
```

```
getRMSE(resp, pred, na.rm = TRUE)
```

```
[1] 0.9646971
```

### Question 3

Let's create a similar function, except it calculates the MAE instead of the RMSE.

```
getMAE <- function(response, predicted, ...){
  add <- list(...)
  remove <- isTRUE(add$na.rm)
  if (remove) {
    dropmissing <- !is.na(response) & !is.na(predicted)
    response <- response[dropmissing]
    predicted <- predicted[dropmissing]
  }
  mean(abs(response-predicted))
}
```

### Question 4

Now let's create some data and test our MAE function using the data.

```
set.seed(10)
n <- 100
x <- runif(n)
resp <- 3 + 10*x + rnorm(n)
pred <- predict(lm(resp ~ x), data.frame(x))

#Let's test our MAE function
getMAE(resp, pred)
```

```
[1] 0.8155776
```

```
#Add 2 missing vlaues to resp
resp[c(1,5)] <- NA_real_

#Test with and without specifying what R should do with the missing values
getMAE(resp, pred)
```

```
[1] NA
```

```
getMAE(resp, pred, na.rm = TRUE)
```

```
[1] 0.8210863
```

### Question 5

Now we want to create a wrapper function that can be used to get either both metrics or a single functioned called.

```
#Create the wrapper function
my_wrapper <- function(response, predicted, metrics = c("MAE", "RMSE"), ...) {
  if (!(is.vector(response) && is.atomic(response) && is.numeric(response))) {
    message("Error: 'response' must be a numeric atomic vecotr.")
    return(invisible(NULL))
  }
  if (!(is.vector(predicted) && is.atomic(predicted) && is.numeric(predicted))) {
    message("Error: 'predicted' must be a numeric atomic vecotr.")
    return(invisible(NULL))
  }

  #create empty list for results
  results <- list()

  #calculate metrics
  if ("MAE" %in% toupper(metrics)) {
    results$MAE <- getMAE(response, predicted, ...)
  }
  if ("RMSE" %in% toupper(metrics)) {
    results$RMSE <- getRMSE(response, predicted, ...)
  }
  return(results)
}
```

### Question 6

Now let's test our wrapper function in a similar way that we tested our previous functions.

```r
#create data
set.seed(10)
n <- 100
x <- runif(n)
resp <- 3 + 10*x + rnorm(n)
pred <- predict(lm(resp ~ x), data.frame(x))

#call individually then together
my_wrapper(resp, pred, metrics = "MAE")
```

```
$MAE
[1] 0.8155776
```

```r
my_wrapper(resp, pred, metrics = "RMSE")
```

```
$RMSE
[1] 0.9581677
```

```r
my_wrapper(resp, pred, metrics = c("MAE", "RMSE"))
```

```
$MAE
[1] 0.8155776

$RMSE
[1] 0.9581677
```

```r
#test again but with 2 NA values in resp
resp[c(1,5)] <- NA_real_
my_wrapper(resp, pred, metrics = "MAE")
```

```
$MAE
[1] NA
```

```r
my_wrapper(resp, pred, metrics = "RMSE", na.rm = TRUE) #tested excluding NA
```

```
$RMSE
[1] 0.9646971
```

```
my_wrapper(resp, pred, metrics = c("MAE", "RMSE"))
```

```
$MAE
[1] NA

$RMSE
[1] NA
```

```
#Test by passing a data frame created in Task 1 Question 2.
my_wrapper(df1, df1, metrics = c("MAE", "RMSE"))
```

```
Error: 'response' must be a numeric atomic vecotr.
```

```
my_wrapper(resp, df1, metrics = c("MAE", "RMSE"), na.rm = TRUE)
```

```
Error: 'predicted' must be a numeric atomic vecotr.
```

## Task 3: Querying an API and a Tidy-Style Function

### Question 1

### Question 2

### Question 3