

Homework 2

```
# Initial packages required (we'll be adding more)
library(tidyverse)
library(nycflights13)
```

Filter/remove the NA values before rescaling

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
df$a[5] = NA
df
```

```
# A tibble: 10 x 4
      a      b      c      d
  <dbl> <dbl> <dbl> <dbl>
1  1.61 -0.616  1.05 -0.164
2  0.306  0.783 -0.364  1.18
3  0.407 -0.254  1.32  0.716
4  1.21 -0.615  0.425 -0.188
5 NA     -0.512  0.387 -0.0745
6  0.516  0.157  2.21  0.663
7 -0.755 -0.659  1.13 -0.631
8  0.799  0.576 -0.824 -0.101
9  0.949 -0.710  0.712  1.87
10 0.446  1.18  1.21  0.0700
```

```
rescale_basic <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}
```

```
df |>
  filter(!is.na(a)) |>
  mutate(new_a = rescale_basic(a))
```

```
# A tibble: 9 x 5
      a      b      c      d new_a
  <dbl> <dbl> <dbl> <dbl> <dbl>
1  1.61 -0.616  1.05 -0.164  1
2  0.306  0.783 -0.364  1.18  0.448
3  0.407 -0.254  1.32  0.716  0.491
4  1.21 -0.615  0.425 -0.188  0.829
5  0.516  0.157  2.21  0.663  0.537
6 -0.755 -0.659  1.13 -0.631  0
7  0.799  0.576 -0.824 -0.101  0.657
8  0.949 -0.710  0.712  1.87  0.720
9  0.446  1.18  1.21  0.0700 0.508
```

[Pause to Ponder 1:] Do you notice anything in the output above that gives you pause?

We've deleted an entire row whose other data we may still want to use.

Create an if statement to check if there are NAs; return an error if NAs exist

First, here's an example involving weighted means:

```
# Create function to calculate weighted mean
wt_mean <- function(x, w) {
  sum(x * w) / sum(w)
}
wt_mean(c(1, 10), c(1/3, 2/3))
```

```
[1] 7
```

```
wt_mean(1:6, 1:3)
```

```
[1] 7.666667
```

[Pause to Ponder 2:] Why is the answer to the last call above 7.67? Aren't we taking a weighted mean of 1-6, all of which are below 7?

This is happening because we gave it a vector of weights that doesn't match the length of the vector of numbers to take the mean for, so R is recycling the vector of weights to multiply the first vector by, but it is not recycling when calculating the sum to divide by. Thus, we aren't really calculating weights properly, and instead making all the values in our data larger.

```
# update function to handle cases where data and weights of unequal length
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  } else {
    sum(w * x) / sum(w)
  }
}
wt_mean(1:6, 1:3)
```

Error: `x` and `w` must be the same length

```
# should produce an error now if weights and data different lengths
# - nice example of if and else
```

[Pause to Ponder 3:] What does the `call.` option do? If `call. = TRUE`, it will tell you what line/function the error is occurring in; if false, it just gives the error message.

Now let's apply this to our rescaling function

```
rescale_w_error <- function(x) {
  if (is.na(sum(x))) {
    stop("`x` cannot have NAs", call. = FALSE)
  } else {
    (x - min(x)) / (max(x) - min(x))
  }
}

temp <- c(4, 6, 8, 9)
rescale_w_error(temp)
```

```
[1] 0.0 0.4 0.8 1.0
```

```
temp <- c(4, 6, 8, 9, NA)
rescale_w_error(temp)
```

Error: `x` cannot have NAs

[Pause to Ponder 4:] Why can't we just use `if (is.na(x))` instead of `is.na(sum(x))`? `x` is going to be an entire vector, so `is.na(x)` would return a logical vector of TRUEs and FALSEs, which would not work properly with the `if` function, which is checking for a single TRUE/FALSE. If we do `is.na(sum(x))`, then `sum(x)` will become NA if `x` has any NAs, so `is.na` will essentially return TRUE for any NAs in `x`, which is what we want.

[Pause to Ponder 5:] Predict what the code below will do, and (only) then run it to check. Think about: why do we have `sort = sort`? why not embrace `df`? why didn't we need `n` in the arguments?

This code is going to create a function that takes an input of a dataset, variable in that dataset, and set of conditions. It will filter for only rows in the dataset that match that condition, then count all the different factors of the variable and calculate the proportions. With `mpg`, we will get a table of the counts and proportions of cars from each of the seven manufacturers. `sort = sort` is because the function can optionally take an input for `sort` as either TRUE or FALSE, so the user can control it if desired. We don't need to embrace `df` because it is an environment variable that lives in the environment, so it does not need to be embraced like data-variables inside a data frame need to be. The `n` column is automatically created by the `count` function, so we didn't need to manually put it anywhere.

```
new_function <- function(df, var, condition, sort = TRUE) {
  df |>
    filter({{ condition }}) |>
    count({{ var }}, sort = sort) |>
    mutate(prop = n / sum(n))
}

mpg |> new_function(var = manufacturer,
                    condition = manufacturer %in% c("audi",
                                                    "honda",
                                                    "hyundai",
                                                    "nissan",
                                                    "subaru",
                                                    "toyota",
                                                    "volkswagen")
                    )
```

[Pause to Ponder 6:] Here's another nice use of `pick()`. Predict what the function will do, then run the code to see if you are correct.

The function will take a dataframe, variables for the rows, and variables for the columns. It will make an `n` column for counts of the values in the rows and columns, then pivot wider the table to essentially show for each combination of the row variables what the counts are for each option from the `cols` variable.

```
# Source: https://twitter.com/pollicipes/status/1571606508944719876
new_function <- function(data, rows, cols) {
  data |>
    count(pick(c({{ rows }}, {{ cols }}))) |>
    pivot_wider(
      names_from = {{ cols }},
      values_from = n,
      names_sort = TRUE,
      values_fill = 0
    )
}

mpg |> new_function(c(manufacturer, model), cyl)
```

On Your Own

1. Rewrite this code snippet as a function: `x / sum(x, na.rm = TRUE)`. This code creates weights which sum to 1, where NA values are ignored. Test it for at least two different vectors. (Make sure at least one has NAs!)

```
weights_one <- function(x) {
  x / sum(x, na.rm = TRUE)
}

test1 <- c(NA, 1, 2, 3, 4)
weights_one(test1)
```

```
[1] NA 0.1 0.2 0.3 0.4
```

```
test2 <- c(5, 8, 20, 3, 72)
weights_one(test2)
```

```
[1] 0.04629630 0.07407407 0.18518519 0.02777778 0.66666667
```

```
sum(weights_one(test2))
```

```
[1] 1
```

2. Create a function to calculate the standard error of a variable, where $SE = \text{square root of the variance divided by the sample size}$. Hint: start with a vector like `x <- 0:5` or `x <- gss_cat$age` and write code to find the SE of `x`, then turn it into a function to handle any vector `x`. Note: `var` is the function to find variance in R and `sqrt` does square root. `length` may also be handy. Test your function on two vectors that do not include NAs (i.e. do **not** worry about removing NAs at this point).

```
se <- function(x) {  
  sqrt(var(x) / length(x))  
}  
  
test1 <- c(0,1,2,3,4,5)  
se(test1)
```

```
[1] 0.7637626
```

```
test2 <- c(7,7,7)  
se(test2)
```

```
[1] 0
```

3. Use your `se` function within `summarize` to get a table of the mean and s.e. of `hwy` and `cty` by `class` in the `mpg` dataset.

```
mpg |>  
  group_by(class) |>  
  summarize(  
    mean_hwy = mean(hwy),  
    se_hwy = se(hwy),  
    mean_cty = mean(cty),  
    se_cty = se(cty)  
  )
```

```
# A tibble: 7 x 5  
  class      mean_hwy se_hwy mean_cty se_cty  
  <chr>      <dbl>  <dbl>   <dbl>  <dbl>
```

| | | | | | |
|---|------------|------|-------|------|-------|
| 1 | 2seater | 24.8 | 0.583 | 15.4 | 0.245 |
| 2 | compact | 28.3 | 0.552 | 20.1 | 0.494 |
| 3 | midsize | 27.3 | 0.334 | 18.8 | 0.304 |
| 4 | minivan | 22.4 | 0.622 | 15.8 | 0.553 |
| 5 | pickup | 16.9 | 0.396 | 13 | 0.356 |
| 6 | subcompact | 28.1 | 0.909 | 20.4 | 0.778 |
| 7 | suv | 18.1 | 0.378 | 13.5 | 0.307 |

4. Use your `se` function within `summarize` to get a table of the mean and s.e. of `arr_delay` and `dep_delay` by carrier in the `flights` dataset. Why does the output look like this?

```
flights |>
  group_by(carrier) |>
  summarize(
    mean_arr_delay = mean(arr_delay),
    se_arr_delay = se(arr_delay),
    mean_dep_delay = mean(dep_delay),
    se_dep_delay = se(dep_delay)
  )
```

```
# A tibble: 16 x 5
  carrier mean_arr_delay se_arr_delay mean_dep_delay se_dep_delay
  <chr>         <dbl>         <dbl>         <dbl>         <dbl>
1 9E             NA             NA             NA             NA
2 AA             NA             NA             NA             NA
3 AS             NA             NA             NA             NA
4 B6             NA             NA             NA             NA
5 DL             NA             NA             NA             NA
6 EV             NA             NA             NA             NA
7 F9             NA             NA             NA             NA
8 FL             NA             NA             NA             NA
9 HA          -6.92           4.06           4.90           4.01
10 MQ            NA             NA             NA             NA
11 OO            NA             NA             NA             NA
12 UA            NA             NA             NA             NA
13 US            NA             NA             NA             NA
14 VX            NA             NA             NA             NA
15 WN            NA             NA             NA             NA
16 YV            NA             NA             NA             NA
```

There is at least one NA for almost every carrier in the `arr_delay` and `dep_delay` columns, which spreads into the mean and se calculations and makes them NAs, too.

5. Make your `se` function handle NAs with an `na.rm` option. Test your new function (you can call it `se` again) on a vector that doesn't include NA and on the same vector with an added NA. **Be sure to check that it gives the expected output with `na.rm = TRUE` and `na.rm = FALSE`.** Make `na.rm = FALSE` the default value. Repeat #4. (Hint: be sure when you divide by sample size you don't count any NAs)

```
se <- function(x, na.rm = FALSE) {  
  sqrt(var(x, na.rm = na.rm) / (length(x) - sum(is.na(x))))  
}  
  
se(0:5)
```

```
[1] 0.7637626
```

```
se(0:5, na.rm = TRUE)
```

```
[1] 0.7637626
```

```
test <- c(0,1,2,3,4,5,NA)  
se(test)
```

```
[1] NA
```

```
se(test, na.rm = TRUE)
```

```
[1] 0.7637626
```

```
flights |>  
  group_by(carrier) |>  
  summarize(  
    mean_arr_delay = mean(arr_delay, na.rm = TRUE),  
    se_arr_delay = se(arr_delay, na.rm = TRUE),  
    mean_dep_delay = mean(dep_delay, na.rm = TRUE),  
    se_dep_delay = se(dep_delay, na.rm = TRUE)  
  )
```



```
# A tibble: 16 x 5
  carrier mean_arr_delay se_arr_delay mean_dep_delay se_dep_delay
  <chr>      <dbl>      <dbl>      <dbl>      <dbl>
1 9E         7.38        0.381        16.7        0.348
2 AA         0.364        0.238         8.59        0.209
3 AS        -9.93        1.37         5.80        1.18
4 B6         9.46        0.184        13.0        0.165
5 DL         1.64        0.203         9.26        0.182
6 EV        15.8        0.221        20.0        0.205
7 F9        21.9        2.36        20.2        2.23
8 FL        20.1        0.960        18.7        0.933
9 HA        -6.92        4.06         4.90        4.01
10 MQ        10.8        0.273        10.6        0.247
11 OO        11.9        9.02        12.6        8.00
12 UA         3.56        0.170        12.1        0.148
13 US         2.13        0.235         3.78        0.199
14 VX         1.76        0.699        12.9        0.626
15 WN         9.65        0.427        17.7        0.394
16 YV        15.6        2.27        19.0        2.11
```

6. Create `both_na()`, a function that takes two vectors of the same length and returns how many positions have an NA in both vectors. Hint: create two vectors like `test_x <- c(1, 2, 3, NA, NA)` and `test_y <- c(NA, 1, 2, 3, NA)` and write code that works for `test_x` and `test_y`, then turn it into a function that can handle any `x` and `y`. (In this case, the answer would be 1, since both vectors have NA in the 5th position.) Test it for at least one more combination of `x` and `y`.

```
both_na <- function(x, y) {
  is_x <- is.na(x)
  is_y <- is.na(y)
  both <- (is_x & is_y)
  sum(both)
  #or just sum(is.na(x) & is.na(y))
}

testx <- c(1,2,3,NA,NA)
testy <- c(NA,1,2,3,NA)
both_na(testx, testy)
```

```
[1] 1
```

```
testx2 <- c(NA, NA, 1, 3, 5, 6, 7, 7)
testy2 <- c(NA, NA, NA, NA, NA, NA, NA, NA)
both_na(testx2, testy2)
```

```
[1] 2
```

7. Run your code from (6) with the following two vectors: `test_x <- c(1, 2, 3, NA, NA, NA)` and `test_y <- c(NA, 1, 2, 3, NA)`. Did you get the output you wanted or expected? Modify your function using `if`, `else`, and `stop` to print an error if `x` and `y` are not the same length. Then test again with `test_x`, `test_y` and the sets of vectors you used in (6).

```
test_x <- c(1, 2, 3, NA, NA, NA)
test_y <- c(NA, 1, 2, 3, NA)
both_na(test_x, test_y)
```

Warning in `is_x & is_y`: longer object length is not a multiple of shorter object length

```
[1] 2
```

We get an error because the two vectors are not of the same length.

```
both_na <- function(x, y) {
  if(length(x) == length(y)) {
    is_x <- is.na(x)
    is_y <- is.na(y)
    both <- (is_x & is_y)
    sum(both)
  } else {
    stop("'x' and 'y' are not the same length", call. = FALSE)
  }
}

both_na(test_x, test_y)
```

Error: 'x' and 'y' are not the same length

```
both_na(testx, testy)
```

```
[1] 1
```

8. Here is a way to get `not_cancelled` flights in the flights dataset:

```
not_cancelled <- flights |>
  filter(!is.na(dep_delay), !is.na(arr_delay))
```

Is it necessary to check `is.na` for both departure and arrival? Using `summarize`, find the number of flights missing departure delay, arrival delay, and both. (Use your new function!)

```
flights |>
  summarize(dep_missing = sum(is.na(dep_delay)),
            arr_missing = sum(is.na(arr_delay)),
            both_missing = both_na(dep_delay, arr_delay))
```

```
# A tibble: 1 x 3
  dep_missing arr_missing both_missing
      <int>      <int>      <int>
1      8255      9430      8255
```

It looks like every row missing its departure delay is also missing its arrival delay, but not vice versa, so we could just filter out rows missing arrival delay in this case.

9. Read the code for each of the following three functions, puzzle out what they do, and then brainstorm better names.

```
time.diff.min <- function(time1, time2) {
  hour1 <- time1 %/% 100
  min1 <- time1 %% 100
  hour2 <- time2 %/% 100
  min2 <- time2 %% 100

  (hour2 - hour1)*60 + (min2 - min1)
}
#difference between two times in minutes

area.cm.to.inch <- function(lengthcm, widthcm) {
  (lengthcm / 2.54) * (widthcm / 2.54)
```

```

}
#area in inches

combine.nas <- function(x) {
  fct_collapse(x, "non answer" = c("No answer", "Refused",
                                   "Don't know", "Not applicable"))
}
#takes variations of "don't know" and makes them all one factor level

```

10. Explain what the following function does and demonstrate by running `foo1(x)` with a few appropriately chosen vectors `x`. (Hint: set `x` and run the “guts” of the function piece by piece.)

```

num.desc <- function(x) {
  diff <- x[-1] - x[1:(length(x) - 1)]
  sum(diff < 0)
}
test1[-1]

```

```
[1] 1 2 3 4 5
```

```

test1 <- c(5,4,3,2,1)
num.desc(test1)

```

```
[1] 4
```

This function will return the number of values in a list that are less than the one before them.

11. The `foo1()` function doesn’t perform well if a vector has missing values. Amend `foo1()` so that it produces a helpful error message and stops if there are any missing values in the input vector. Show that it works with appropriately chosen vectors `x`. Be sure you add `error = TRUE` to your R chunk, or else knitting will fail!

```

foo1 <- function(x) {
  if(is.na(sum(x))) {
    stop("'x' has missing values", call. = FALSE)
  } else{
    diff <- x[-1] - x[1:(length(x) - 1)]
    sum(diff < 0)
  }
}

```

```
}  
  
foo1(c(1,2,3,4))
```

```
[1] 0
```

```
foo1(c(1, NA, 2, 3, 4))
```

Error: 'x' has missing values

12. Write a function called `greet` using `if`, `else if`, and `else` to print out “good morning” if it’s before 12 PM, “good afternoon” if it’s between 12 PM and 5 PM, and “good evening” if it’s after 5 PM. Your function should work if you input a time like: `greet(time = "2018-05-03 17:38:01 CDT")` or if you input the current time with `greet(time = Sys.time())`. [Hint: check out the `hour` function in the `lubridate` package]

```
greet <- function(time) {  
  hr <- hour(time)  
  if(hr < 12) {  
    "good morning"  
  } else if(12 <= hr & hr < 17) {  
    "good afternoon"  
  } else {  
    "good evening"  
  }  
}  
  
greet("2018-05-03 17:38:01 CDT")
```

```
[1] "good evening"
```

```
greet(time = Sys.time())
```

```
[1] "good morning"
```

13. Modify the `summary6()` function from earlier to add an argument that gives the user an option to remove missing values, if any exist. Show that your function works for (a) the `hwy` variable in `mpg_tbl <- as_tibble(mpg)`, and (b) the `age` variable in `gss_cat`.

```
summary6 <- function(data, var, na.rm = FALSE) {
  data |> summarize(
    mean = mean({{ var }}, na.rm = na.rm),
    median = median({{ var }}, na.rm = na.rm),
    sd = sd({{ var }}, na.rm = na.rm),
    IQR = IQR({{ var }}, na.rm = na.rm),
    n = n(),
    n_miss = sum(is.na({{ var }})),
    .groups = "drop"
  )
}
mpg_tbl <- as_tibble(mpg)
summary6(mpg_tbl, hwy, na.rm = TRUE)
```

```
# A tibble: 1 x 6
  mean median    sd  IQR      n n_miss
<dbl> <dbl> <dbl> <dbl> <int> <int>
1  23.4     24  5.95     9   234     0
```

```
summary6(gss_cat, age, na.rm = TRUE)
```

```
# A tibble: 1 x 6
  mean median    sd  IQR      n n_miss
<dbl> <int> <dbl> <dbl> <int> <int>
1  47.2     46  17.3    26 21483     76
```

14. Add an argument to (13) to produce summary statistics by group for a second variable (you should now have 4 possible inputs to your function). Show that your function works for (a) the hwy variable in `mpg_tbl <- as_tibble(mpg)` grouped by `drv`, and (b) the age variable in `gss_cat` grouped by `partyid`.

```
summary6 <- function(data, var, var2, na.rm = TRUE) {
  data |>
    group_by(pick({{var2}})) |>
    summarize(
      mean = mean({{ var }}, na.rm = na.rm),
      median = median({{ var }}, na.rm = na.rm),
      sd = sd({{ var }}, na.rm = na.rm),
      IQR = IQR({{ var }}, na.rm = na.rm),
      n = n(),
      n_miss = sum(is.na({{ var }})),

```

```

    .groups = "drop"    # to leave the data in an ungrouped state
  )
}

summary6(mpg_tbl, hwy, drv)

```

```

# A tibble: 3 x 7
  drv    mean median    sd   IQR     n n_miss
<chr> <dbl>  <dbl> <dbl> <dbl> <int>  <int>
1 4      19.2    18  4.08     5   103     0
2 f      28.2    28  4.21     3   106     0
3 r       21    21  3.66     7    25     0

```

```

party_age <- summary6(gss_cat, age, partyid)
party_age

```

```

# A tibble: 10 x 7
  partyid          mean median    sd   IQR     n n_miss
<fct>         <dbl>  <dbl> <dbl> <dbl> <int>  <int>
1 No answer      50.8    48  18.7  28    154     9
2 Don't know     34     34   NA    0      1     0
3 Other party    45.2   44.5  15.8  23    393     3
4 Strong republican 51.9    51  17.0  26   2314     8
5 Not str republican 47.2    45  17.2  26   3032     8
6 Ind,near rep   47.1    46  17.1  27   1791     2
7 Independent    43.3    41  16.3  24   4119    18
8 Ind,near dem   44.9    43  17.1  27   2499     2
9 Not str democrat 46.5    44  17.3  26.5 3690    11
10 Strong democrat 51.2    50  17.4  27   3490    15

```

15. Create a function that has a vector as the input and returns the last value. (Note: Be sure to use a name that does not write over an existing function!)

```

final_val <- function(x) {
  x[[length(x)]]
}

final_val(c(1,2,3,4))

```

```
[1] 4
```

```
final_val(c(1,2,3,7,10, 5, 6, 9))
```

```
[1] 9
```

16. Save your final table from (14) and write a function to draw a scatterplot of a measure of center (mean or median - user can choose) vs. a measure of spread (sd or IQR - user can choose), with points sized by sample size, to see if there is constant variance. Each point should be labeled with partyid, and the plot title should reflect the variables chosen by the user.

Hint: start with a ggplot with no user input, and then functionize:

```
library(ggrepel)
```

Warning: package 'ggrepel' was built under R version 4.4.3

```
center.spread <- function(data, center, spread, count, group) {  
  label <- rlang::englue("{{center}} versus {{spread}} by party affiliation")  
  data |>  
    ggplot(aes(x = {{spread}}, y = {{center}}, label = {{group}}, color = {{group}})) +  
    geom_point(aes(size = {{count}}), alpha = 0.75) +  
    geom_text_repel(aes(label = {{group}})) +  
    geom_smooth(method = lm) +  
    labs(title = label) +  
    guides(color = "none")  
}  
  
center.spread(party_age, mean, IQR, n, partyid)
```

`geom_smooth()` using formula = 'y ~ x'

