Common Challenges Working With Data

Ward, A, Rivas, S., Shenasa, A.

The Burdens of Messy Data

Working with data can be frustrating. Collecting data is difficult and humans make mistakes meaning that often times the data we first begin to analyze and interpret can be full of flaws and mistakes. Programming languages aren't perfect either and need direction on how your data is formatted to be able to interpret it properly.

We have all experienced a time when messy data (either of our own doing or others) has nearly made us pull our hair out from frustration. This can be especially true when you're beginning to interact with a programming language for the first time. Our hope is to alleviate some of that stress and burden by giving some guidance on how to handle common data issues that can arise.

Here, we've narrowed down six of the more common challenges we have experienced while analyzing and visualizing data. We have broken down how these challenges may arise and given some suggestions and examples on how to proceed.

Our Top Six Data Challenges

1. ***Missing Data***

   In R, missing values in a data set are often represented by NA. This will either be from blank cells being filled in with NAs as the data frame is uploaded or if the uploaded data frame already contains NAs rather than blank cells. However, this can get messy for a number of reasons when NAs are mislabeled (i.e. N/A vs. N.A. vs. NA - which we deal with in our *Inconsistent Formatting* section), when you don't know how many missing values there are, or when they harm the visualization or interpretation of your data.

- Checking data set for missing values

  If you have missing values in your data set, a good first step is simply to check for the presence of NAs. This will give an output of TRUE or FALSE based on the presence of missing values.

```r
x <- c(8,12,NA,19,2,NA,99,4,78,1,32,NA)

is.na(x)
```

```
 [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

- Counting missing values

  Once the presence of NA values have been identified, we can check how many NA values are present. This can be especially helpful with larger data sets with many rows of data. This will give a number as an output of how many NA values are present.

```r
y <- c(9,13,NA,24,17,15,3,NA,2,13,29,30,12,5,67,NA,92,46,73,NA,12,68,23,12,14,30,54,NA,1

sum(is.na(y))
```

```
[1] 9
```

- Excluding missing values

  Omitting NA values can be an import step to ensuring standard R functions and statistical models run properly and avoid skewed results (depending on the source of NA values). Several functions like na.omit and na.rm can be used to exclude the missing values.

  na.omit removes missing values from the data (vector, matrix, or data frame). We can then check using sum(is.na()) to see if all NA values have been removed.

```r
cleaned_y <- na.omit(y)

sum(is.na(cleaned_y))
```

```
[1] 0
```

  Alternatively, na.rm can be used to ignore missing values during a calculation rather than removing them from the entire data set. This is an argument used within a function rather than a standalone function like na.omit. For example, when we try to take the mean of our data set with missing values, it returns "NA".

```r
mean(y)
```

```
[1] NA
```

  But, if we include na.rm = TRUE, we can get the average without having to use our cleaned_y data set.

```r
mean(y, na.rm = TRUE)
```

```
[1] 29.13043
```

```
mean(cleaned_y)
```

```
[1] 29.13043
```

- Replacing NAs

  Sometimes replacing NA values can be helpful either to approximate these values or to change them to a numeric value. If we wanted to replace all NAs in our vector with an average, we would simply take a subset of the vector that includes only NAs and set that equal to the value of our mean.

```
#Here we take all NAs in vector y and set them equal to our mean of y
y[is.na(y)] <- mean(y, na.rm = TRUE)

#View y
y
```

```
 [1]  9.00000 13.00000 29.13043 24.00000 17.00000 15.00000  3.00000 29.13043
 [9]  2.00000 13.00000 29.00000 30.00000 12.00000  5.00000 67.00000 29.13043
[17] 92.00000 46.00000 73.00000 29.13043 12.00000 68.00000 23.00000 12.00000
[25] 14.00000 30.00000 54.00000 29.13043 12.00000 90.00000 43.00000 21.00000
[33] 68.00000 54.00000  3.00000  2.00000 90.00000 52.00000 12.00000 29.13043
[41] 13.00000 29.13043 24.00000 17.00000 15.00000  3.00000 29.13043  2.00000
[49] 13.00000 29.00000 30.00000 12.00000  5.00000 67.00000 29.13043
```

2. ***Inconsistent Formatting***

   One very common challenge when working with data is having inconsistent formatting in your data frame. This could mean values that mean the same thing are spelled differently, are abbreviated, or have varying capitalization. Not clearing up these issues early on can lead to confusing analyses and incorrect visualizations. Making sure your data is formatted well from the start of your data analysis journey will save many headaches in the long run.

```
messy_df <- data.frame(
  id = 1:10,
  color = c("Orange", "or", "orange ", "ORANGE", "blue", "Blue", "blu", "green", " Greer
  response = c("Yes", "yes", "YES", "No", "no", "NO", " yEs ", "nO", "Yes", "no"),
  score = c("1", "2", "three", "4", "5", "six", "7", "8", "nine", "10"),
  group = c("A", "a", "Group A", "B", "b ", "Group B", "C", "c", "Group C", "C")
)
```

- Inconsistent capitalization

Values in a column can sometimes differ only by the capitalization of the letters. “No”, “no”, and “NO” will all be interpreted differently by R. Therefore, making this consistent is very important if these responses hold the same interpretation.

```
messy_df$response <- tolower(messy_df$response)

messy_df$response
```

```
 [1] "yes"   "yes"   "yes"   "no"    "no"    "no"    " yes " "no"    "yes"
[10] "no"
```

- Leading or trailing spaces

  Inputting data manually can lead to additional spacing in places leading to unintended hidden characters. These sometimes will need to be removed to help with consistency in your data.

```
messy_df$color <- trimws(messy_df$color)
messy_df$response <- trimws(messy_df$response)

messy_df$color
```

```
 [1] "Orange" "or"     "orange" "ORANGE" "blue"   "Blue"   "blu"    "green"
 [9] "Green"  "green"
```

```
messy_df$response
```

```
 [1] "yes" "yes" "yes" "no"  "no"  "no"  "yes" "no"  "yes" "no"
```

- Misspelled words/ multiple labels for one variable

  This will also lead to categorizing specific variables differently even if they are equivalent. But you can reformat your data to read them all as a single variable.

```
messy_df$color[messy_df$color == "or"] <- "orange"
messy_df$color[messy_df$color == "blu"] <- "blue"

messy_df$color
```

```
 [1] "Orange" "orange" "orange" "ORANGE" "blue"   "Blue"   "blue"   "green"
 [9] "Green"  "green"
```

3. ***Merging Dataframes***

   To properly analyze your data, you may need to combine multiple datasets that provide additional information or add more data. This can be done by adding more variables

(i.e. columns) or by adding new observations (i.e. rows). A unified dataframe can simplify your data analysis process and lead to a cleaner workflow.

```
#Example data frames
df_1<- data.frame(ID = 1:5, species = c("tiger","lion" , "jaguar", "lion", "tiger"))
df_2<- data.frame(ID = 4:8, species= c("lion", "jaguar", "lion", "tiger", "tiger"), age
```

- Merging by columns

  Merging by columns is useful when two data frames contain different variables for the same observation. This can happen when data are collected from different sources or at different times but samples have the same ID.

  There are several types of joins available within the merge function. A left join keeps all rows from the first dataset and matching rows from the second. A right join includes all rows from the second dataset and matching rows from the first. An inner keeps only rows with matching keys in both datasets. And an outer keeps all rows from both datasets.

```
#Merging by columns by the ID

#Keeps all rows from df_1 and adds matching data from df_2
left_join <- merge(df_1, df_2, by = "ID", all.x = TRUE)

#Keeps all rows from df_2 and adds matching data from df_1
right_join <- merge(df_1, df_2, by = "ID", all.y = TRUE)

#Keeps only rows where ID appears in both df_1 and df_2
inner_join <- merge(df_1, df_2, by = "ID")

#Keeps every row from both data frames
full_join <- merge(df_1, df_2, by = "ID", all = TRUE)
```

- Merging by row

  Merging by rows is important when two dataframes have the same variables but different observations, such as data collected at different times or from different locations. Both dataframes are supposed to have the same columns.

```
# Add missing column to df_1 so the structures match
df_1$age <- NA

merged_rows <- rbind(df_1, df_2)
merged_rows
```

```
   ID species age
1   1   tiger  NA
```

```
2   2    lion  NA
3   3  jaguar  NA
4   4    lion  NA
5   5   tiger  NA
6   4    lion   4
7   5  jaguar   2
8   6    lion   6
9   7   tiger   1
10  8   tiger   4
```

- A word to the wise:
  - Merging data frames can be both messy and scary work. It is a very common tool to merge data frames, but in order to make it a useful and efficient process here are a few useful tips!
  - Know your data well! Use functions like head(), str(), and summary() to understand the column names, the type of data and the underlying structure of your data set.
  - Understand merging types and how they fit your needs. The merge function allows for left, right, inner, and outer joins that all merge your data in varying ways. Know how these function and what suits your needs.
  - Row binding errors can happen when data frames don't have the same column names or types

4. ***Mismatched Row/Column Length***

   When trying to merge dataframes, you may get errors if the dataframes have different numbers of rows or columns. It's good to review your dataframes before importing them to try to make the merge seamless if possible, but here's some ways you can merge mismatched lengths in R.

- Mismatched row lengths

```r
#example - these data frames have different numbers of values
years <- c(2023, 2024)
temps <- c(13.2, 13.8, 14.1)

df <- data.frame(years, temps) #try running this -- did you get an error?
```

```
Error in data.frame(years, temps): arguments imply differing number of rows: 2, 3
```

```r
#this can be fixed using plyr package with the function cbind.fill()
library(str2str)
cbind_fill(years, temps) #fills in missing values with NA
```

```
    X1   X2
1 2023 13.2
2 2024 13.8
3   NA 14.1
```

- Mismatched column names

  You try to bind rows, but data frames have different column names

```
df1 <- data.frame(year = 2023, temp = 13.2)
df2 <- data.frame(year = 2024, salinity = 33.4)
#column names ‘temp’ and ‘salinity’ don’t match, so you can't merge

#use plyr package, this time with function rbind.fill() to fill in missing values with NA
library(plyr)
rbind.fill(df1, df2)
```

```
  year temp salinity
1 2023 13.2       NA
2 2024   NA     33.4
```

```
#OR use dplyr package to fill in the missing column values with NA
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:plyr':

    arrange, count, desc, failwith, id, mutate, rename, summarise,
    summarize

The following object is masked from 'package:str2str':

    pick

The following objects are masked from 'package:stats':

    filter, lag
```

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
bind_rows(df1, df2)
```

```
  year temp salinity
1 2023 13.2       NA
2 2024   NA     33.4
```

5. ***Incorrect Column Types***

In R, every column in a data frame has a data type (e.g., numeric, character, factor, date). Problems happen when a column's type doesn't match how you want to use it. This often occurs when importing data from CSVs, Excel files, or mixed-format columns. Incorrect types can cause errors, misleading plots, or mistakes in calculations.

- Numbers stored as characters

  When imputting data, sometimes columns of numbers are stored as characters (chr) or factors (factor), when you actually want them to be interpreted as numbers (num) or integers (int). This can cause problems when you try to do calculations, like finding the mean or plotting a variable.

```
#made-up data frame
date <- as.Date(c("2026-01-01", "2026-01-02", "2026-01-03", "2026-01-04", "2026-01-05"))
location <- c("Beach", "Harbor", "Beach", "Harbor", "Beach")
temperature_c <- c(12.5, 13.2, 12.8, 13.5, 13.0)

temp_data <- data.frame(
  date = date,
  location = location,
  temperature_c = temperature_c)

#make sure temperature column is a numeric column
temp_data$temperature_c <- as.numeric(temp_data$temperature_c)
```

- Date/Time columns imported as character or numeric

  Dates and times in files can frequently be read as simple character strings or numeric codes (especially from Excel files) rather than R's specific date type. There's different ways to fix this issue in R, using packages or base R.

```r
df_dates <- data.frame(
  event = c("A", "B", "C"),
  date = c("15/01/2025", "24/04/2025", "10/09/2025"))

#using base R (example)
df_dates_fixed <- as.Date(df_dates$date, format = "%d/%m/%Y")

#use the lubridate package
library(lubridate)
```

```
Attaching package: 'lubridate'
```

```
The following objects are masked from 'package:str2str':

    is.Date, is.POSIXct, is.POSIXlt
```

```
The following objects are masked from 'package:base':

    date, intersect, setdiff, union
```

```r
  #ymd() for "YYYY-MM-DD" or "YYYY/MM/DD"
  #mdy() for "MM-DD-YYYY" or "MM/DD/YYYY"
  #dmy() for "DD-MM-YYYY" or "DD/MM/YYYY"
df_dates$date <- dmy(df_dates$date)


#use dplyr package
df_dates_new <- df_dates %>%
  mutate(date_new = as.Date(date, format = "%m/%d/%Y"))
```

- You can also use the readr package in tidyverse theusing col_types function
  - library(readr) my_data_readr <- read_csv("my_data.csv", col_types = list(col1 = col_character(), col2 = col_number(), col3 = col_date()))
- How to check the current data type of columns

  If you're not sure why your code isn't working, an easy thing to try first is to check what the data types of your dataframe columns are stored as using the class() function. This is also just good practice when you first load in a dataframe!

- Other useful functions

  as.numeric() – Convert to numeric (decimal numbers)

  as.integer() – Convert to integer (whole numbers) or, if you want a column to round numbers, you can use round()

  as.character() – Convert to character (text/strings)

  as.factor() – Convert to a factor (categorical data)

  as.Date() – Convert to date

6. ***Importing Data Error***

Importing your data set in R is typically the first step to begin your analysis. When importing data, there are several frustrating issues you may run into.

- File path issues

  Your working directory is the default location where R will look for your files to load. Issues can arise when the current working directory is not where your files are, resulting in R being unable to locate your files. To check where your working directory is set to, we can use the function getwd()

```
getwd()
```

```
[1] "/Users/Abbie1/Documents/Repositories/BIOE 176 DataScience4EEB/Working_with_Data_Sea
```

  If this is not the correct file path, we can then use setwd() to the intended file path.

  Alternatively, to make the code accessible to users on different computers, you can use library(here) to reference the top-level of a directory of a file project rather than setwd() which only works on your own computer.

```
library(here)
```

```
here() starts at /Users/Abbie1/Documents/Repositories/BIOE 176 DataScience4EEB/Working_w


Attaching package: 'here'

The following object is masked from 'package:plyr':

    here
```

- Data format and structure issues

  Once we are able to locate and upload a file on R, issues regarding reading the file correctly may arise. The function read.csv() is for comma-separated files.

```
read.csv("PlantGrowth.csv")
```

```
   weight group
1    4.17  ctrl
2    5.58  ctrl
3    5.18  ctrl
4    6.11  ctrl
5    4.50  ctrl
6    4.61  ctrl
7    5.17  ctrl
8    4.53  ctrl
9    5.33  ctrl
10   5.14  ctrl
11   4.81  trt1
12   4.17  trt1
13   4.41  trt1
14   3.59  trt1
15   5.87  trt1
16   3.83  trt1
17   6.03  trt1
18   4.89  trt1
19   4.32  trt1
20   4.69  trt1
21   6.31  trt2
22   5.12  trt2
23   5.54  trt2
24   5.50  trt2
25   5.37  trt2
26   5.29  trt2
27   4.92  trt2
28   6.15  trt2
29   5.80  trt2
30   5.26  trt2
```

  - Alternative data files

    If your file uses a different separator, like tabs, semicolons, etc) you need to specify what delimiter is used. For tab separated files, we can use read.delim(), or we can use the readr package with the function read_tsv()

For files with other delimiters, we can specify the correct separator using **read.csv(file, sep = “;”)**