ENTMLGY 6707 Entomological Techniques and Data Analysis

R Activity 2: Data Wrangling

1 Introduction

We are going to get some practice cleaning, exploring, and graphing data. Quality control checks are extremely important before you begin any formal analysis (e.g., are all the rows and columns loaded? are the missing values identified correctly?). For the actual activity, head to the last page. Otherwise, this document is comprised of advice on and examples of coding and analyses.

2 Loading in data and initial checks

Typically, we start by loading in some data (at which you all are now professionals)

```
library("readxl")
```

```
iris_df <- read_excel(path= "Iris_data.xlsx",
                              sheet="some_famous_data",range="A1:E151")
```

It is a good idea to look at the `summary()`. I often use `nrow(data_name)` to examine the number of rows I have loaded.

```
summary(iris_df)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
```

```
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
   Species
 Length:150
 Class :character
 Mode  :character
```

```
nrow(iris_df)
```

```
[1] 150
```

You may also want to look at the structure of the data using `str()`, which provides insights into number of observations (= number of rows), types of variables (numeric vs. character vs. logical [`TRUE`/`FALSE`]), and the first few rows of data.

```
str(iris_df)
```

```
tibble [150 x 5] (S3: tbl_df/tbl/data.frame)
 $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : chr [1:150] "setosa" "setosa" "setosa" "setosa" ...
```

It's a good idea to ensure you've correctly loaded in the top and bottom of the data (i.e., that your entire rectangular spreadsheet arrived safely into R).

```
head(iris_df)
```

```
# A tibble: 6 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl> <chr>
1          5.1         3.5          1.4         0.2 setosa
```

```
2          4.9         3            1.4         0.2 setosa
3          4.7         3.2          1.3         0.2 setosa
4          4.6         3.1          1.5         0.2 setosa
5          5           3.6          1.4         0.2 setosa
6          5.4         3.9          1.7         0.4 setosa
```

```
tail(iris_df)
```

```
# A tibble: 6 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl> <chr>
1          6.7         3.3          5.7         2.5 virginica
2          6.7         3            5.2         2.3 virginica
3          6.3         2.5          5           1.9 virginica
4          6.5         3            5.2         2   virginica
5          6.2         3.4          5.4         2.3 virginica
6          5.9         3            5.1         1.8 virginica
```

The below code opens up a new tab in R studio displaying your entire data frame. It is nice to view the data this way, but you typically still need the above functions to find erroneous `NA` values, typographical errors, missing rows/columns, etc.

```
View(iris_df)
```

3 Converting variables

As a reminder, you can access specific columns in data frames using a `$`.

```
head(iris_df$Species)
```

```
[1] "setosa" "setosa" "setosa" "setosa" "setosa" "setosa"
```

It is very common in practice to convert a variable from a number to a factor. For example, maybe you recorded your sites as 1, 2, 3, etc., and R will read that column (incorrectly, as far as we are concerned) as `numeric`. In the `tidyverse` (of which `readxl` is a part), some factors are read in as characters. You will notice `chr` in the `str()` output above for the `Species` column, indicating that `Species` is a `character` according to R.

You can quickly convert such variables to factors. Notice below how the output of `summary()` changes after the conversion. Specifying the variable type incorrectly can cause a lot of problems when it comes to graphing and analysis, so always check that R has correctly identified numbers as numeric and categorical variables as factors (or characters). The difference(s) between factors and characters in R will not cause problems when graphing and analyzing, as the main difference is just that factors have predefined levels. This distinction can cause problems if you are trying to add a new level to an existing factor, but that is so uncommon we are not going to worry about the differences in this course again.

```
summary(iris_df$Species)
```

```
   Length     Class      Mode
      150 character character
```

```
is.character(iris_df$Species)
```

```
[1] TRUE
```

```
iris_df$Species <- as.factor(iris_df$Species)
summary(iris_df$Species)
```

```
    setosa versicolor  virginica
        50         50         50
```

4 Logical operators

R has several built-in logical operators that can make it easier to access subsets of data.

Let's create a logical vector called `sepal_greater5` and feed it to `table()`. We are asking each value of `iris_df$Sepal.Length` if it is greater than 5 and getting an answer of `TRUE` or `FALSE`.

```
sepal_greater5 <- iris_df$Sepal.Length > 5
table(sepal_greater5)
```

```
sepal_greater5
FALSE  TRUE
   32   118
```

Let's do something similar, asking if each value of `Sepal.Lenth` is equal to 5.1.

```
sepal_equal5.1 <- iris_df$Sepal.Length == 5.1
table(sepal_equal5.1)
```

```
sepal_equal5.1
FALSE  TRUE
  141     9
```

You should notice that our output is a string of `TRUE` and `FALSE` values exactly equal to the length of `iris_df$Sepal.Length`.

Note the `==` means "exactly equal to" whereas `=` is often used as a substitute for `<-`. Below is a quick example. Also note the use of semicolons (`;`), which you can use to split up code on the same line. I use it when I make graphs or create vectors and want the output printed immediately every time I run the code. Use it sparingly.

```
x <- 5+5;x
```

```
[1] 10
```

```
y =  7+3;y
```

```
[1] 10
```

```
y==x
```

```
[1] TRUE
```

One other operator I frequently use is `%in%`, which you read aloud as 'that occurs in'. It can come in really handy when extracting information from a data set. This code determines if each observation of `Sepal.Length` equals 5.1, 4.6 or 6.0, again returning a `TRUE` or `FALSE`.

```
head(iris_df$Sepal.Length %in% c(5.1,4.6,6.0))
```

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

I will often use it with the `which()` command. Instead of returning `TRUE` or `FALSE`, this now returns the location of observations for which the operator is `TRUE`.

```
head(which(iris_df$Sepal.Length %in% c(5.1,4.6,6.0)))
```

```
[1]  1  4  7 18 20 22
```

5 Subsetting data using base R

You will often need to inspect or extract subsets of your data, and here we'll cover several useful functions for doing so. We can extract specific rows and columns by entering the name of a data frame followed by brackets: `iris_df[,]`. There are two "arguments" inside the brackets, the first indicates which rows you want and the second indicates which columns you want:

```
iris_df[25,2:3] # row 25, columns 2 and 3
iris_df[c(26,31),] # rows 26 and 31 for all columns
iris_df[,] # when left blank, the entire dataframe is returned
iris_df[-5,1:2] # remove the 5th row, return all of columns 1 and 2
iris_df[which(iris_df$Sepal.Length>5),] # observations w/ Sepal.Length > 5
iris_df[which(iris_df$Sepal.Length==5.1),] # obsrvations w/ Sepal.Length = 5.1
```

I prefer to use `which()` because I like inputting numbers into the `dataframe[,]` arguments, rather than `TRUE` and `FALSE` values. Both should work, but ALWAYS ALWAYS ALWAYS check your output each time you run code to make sure R is doing exactly what you think it is doing. Likewise, if you have a nested function like I have above (i.e., `which(iris_df$Sepal.Length==5.1)` nested in `iris_df[,]`), it is a good idea to check the output of that nested function before entering it into another function.

The tapply() function is useful for viewing information in one column by levels of another. This gives us the mean of `Sepal.Width` by `Species`. I use this all the time when I am writing papers and need to type out the means and standard errors of treatment groups.

```
tapply(iris_df$Sepal.Width, iris_df$Species,  mean)
```

```
    setosa versicolor  virginica
     3.428      2.770      2.974
```

6 Cleaning/viewing data with `tidyverse`

Install the `tidyverse` package and load it into R

```
install.packages("tidyverse")
```

```
library(tidyverse)
```

We will now use the `tidyverse` functions and its pipe operators. First, we will use the `filter()` function to subset rows for sepal length > 5.

```
iris_df_subset_1 <-  iris_df %>% filter(Sepal.Length > 5)
summary(iris_df_subset_1)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :5.100   Min.   :2.200   Min.   :1.200   Min.   :0.100
 1st Qu.:5.600   1st Qu.:2.800   1st Qu.:3.925   1st Qu.:1.200
 Median :6.100   Median :3.000   Median :4.700   Median :1.500
 Mean   :6.130   Mean   :3.048   Mean   :4.315   Mean   :1.432
 3rd Qu.:6.575   3rd Qu.:3.300   3rd Qu.:5.375   3rd Qu.:1.900
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :22
 versicolor:47
 virginica :49
```

Let's also get the subset of observations with `Sepal.Length` values equal to 5.

```
iris_df_subset_2 <- filter(iris_df, Sepal.Length == 5)
summary(iris_df_subset_2)
```

```
  Sepal.Length  Sepal.Width     Petal.Length   Petal.Width          Species
 Min.   :5     Min.   :2.000   Min.   :1.20   Min.   :0.20   setosa    :8
 1st Qu.:5     1st Qu.:3.050   1st Qu.:1.40   1st Qu.:0.20   versicolor:2
 Median :5     Median :3.350   Median :1.55   Median :0.25   virginica :0
```

```
 Mean   :5     Mean   :3.120   Mean   :1.84   Mean   :0.43
 3rd Qu.:5     3rd Qu.:3.475   3rd Qu.:1.60   3rd Qu.:0.55
 Max.   :5     Max.   :3.600   Max.   :3.50   Max.   :1.00
```

Let's get the mean, standard deviation, and max value of `Sepal.Width` by `Species`. Notice it is pretty straightforward to get multiple pieces of information for each level of a factor.

```
iris_df %>%
 group_by(Species) %>%
   summarise(Means = mean(Sepal.Width),
             SD = sd(Sepal.Width),
             max_sep = max(Sepal.Width))
```

```
# A tibble: 3 x 4
  Species    Means    SD max_sep
  <fct>      <dbl> <dbl>   <dbl>
1 setosa      3.43 0.379     4.4
2 versicolor  2.77 0.314     3.4
3 virginica   2.97 0.322     3.8
```

Do the same thing as the previous example, but arrange the output by mean `Sepal.Width`.

```
iris_df %>%
 group_by(Species) %>%
   summarise(Means = mean(Sepal.Width),
             SD = sd(Sepal.Width),
             max_sep = max(Sepal.Width)) %>%
   arrange(Means)
```

```
# A tibble: 3 x 4
  Species    Means    SD max_sep
  <fct>      <dbl> <dbl>   <dbl>
1 versicolor  2.77 0.314     3.4
2 virginica   2.97 0.322     3.8
3 setosa      3.43 0.379     4.4
```

The `tidyverse` has its own command for converting variables into factors. If I have not convinced you yet, R has a LOT of ways of doing the same thing. Find the commands you

need, make sure they are working as you intend (i.e, always check your output!), and try not to get bogged down in all the options.

```
iris_df$Species <- as_factor(iris_df$Species)
```

7 Dealing with NA values

Now let's add some `NA` values to the column `Sepal.Length`.

```
set.seed(123) # set location of random number generator
iris_df[sample(1:nrow(iris_df),10),"Sepal.Length"] <- NA
```

Here is a logical operator for `NA` values. The `NA` values of `Sepal.Length` are indicated by the values of `TRUE`.

```
tail(is.na(iris_df$Sepal.Length))
```

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

Some functions in R do not know how to handle `NA` values.

```
mean(iris_df$"Sepal.Length") # oof
```

```
[1] NA
```

```
mean(na.omit(iris_df$"Sepal.Length")) # yay
```

```
[1] 5.855714
```

```
mean(iris_df$"Sepal.Length", na.rm=T) # also yay
```

```
[1] 5.855714
```

Let's create a subset of data with only observations of `Sepal.Length` that are greater than the mean `Sepal.Length`. Notice that `NA` values can cause problems.

```
try1 <-  iris_df %>% filter(Sepal.Length > mean(Sepal.Length))
summary(try1) # oof
```

```
  Sepal.Length  Sepal.Width   Petal.Length  Petal.Width        Species
 Min.   : NA   Min.   : NA   Min.   : NA   Min.   : NA   setosa    :0
 1st Qu.: NA   1st Qu.: NA   1st Qu.: NA   1st Qu.: NA   versicolor:0
 Median : NA   Median : NA   Median : NA   Median : NA   virginica :0
 Mean   :NaN   Mean   :NaN   Mean   :NaN   Mean   :NaN
 3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA   3rd Qu.: NA
 Max.   : NA   Max.   : NA   Max.   : NA   Max.   : NA
```

So, tell the `mean()` function how to handle `NA` values...

```
try2 <-  iris_df %>% filter(Sepal.Length > mean(Sepal.Length, na.rm = TRUE))
summary(try2) # yay!
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :5.900   Min.   :2.200   Min.   :4.000   Min.   :1.000
 1st Qu.:6.225   1st Qu.:2.800   1st Qu.:4.700   1st Qu.:1.500
 Median :6.450   Median :3.000   Median :5.100   Median :1.800
 Mean   :6.582   Mean   :2.956   Mean   :5.229   Mean   :1.809
 3rd Qu.:6.800   3rd Qu.:3.175   3rd Qu.:5.700   3rd Qu.:2.100
 Max.   :7.900   Max.   :3.800   Max.   :6.900   Max.   :2.500
       Species
 setosa    : 0
 versicolor:25
 virginica :41
```

8 Proof of concept on the random number generator.

The random number generator in R can be very helpful for assigning treatments, running simulations, and calling on students during class. Run the following code all at once. Then just run the `sample()` function a few times. Then run the all the code all at once again. You should notice that the output of `sample()` is exactly the same when you "set the seed."

```r
set.seed(123) # set random number generator to same place
sample(1:100,3)
```

```
[1] 31 79 51
```

9 R Activity

1. Load in the `breakfast` data set. Remember to check the file type so that you can identify the right command for loading in the data.
2. Use a single command to count the number of columns in the `breakfast` data.
3. Use base R (i.e., no external packages should be used for this question) to create a subset of the data that has only observations with values of `Calories` greater than 130.
4. Use the pipe operators in `tidyverse` to create a subset of the data that has only observations with values of `Calories` greater than 130.
5. Use the pipe operators in `tidyverse` to calculate the mean `Sugars` by `Company`.