Introduction to Data Wrangling & Descriptive Statistics

Josh Carrell - Ph.D. Student | Forest Sciences - Colorado State University

Contents

Data

1. The iris dataset. Already pre-built in R.
2. beaver1. This dataset comes from the **datasets** package.

I'll show you how to load all this data as we go through the code below.

Data Wrangling

Data Wrangling is the process of cleaning and unifying messy and complex data sets for easy access and analysis. When working on a project with many stakeholders, our data often comes from various sources. This means you may be working with several data structuring formats, missing values, and incorrect information.

We're going to walk through some of the very common methods and functions used for Data Wrangling.

Explore the Data

Here we will use the *iris* dataset, pre-built in R.

dim()

The dim() function stands for dimensions and allows us to look at the dimensions of a dataset. The results look like this: Number of Rows x Number of Columns.

```
dim(iris) # view dimensions
```

```
## [1] 150   5
```

---

names()

a simple function for looking at the column names of the dataset.

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

---

head() and tail()

Head and tail sounds like a horse shampoo. They're actually made for viewing the beginnings and endings of datasets. head() looking at the beginning; tail() looks at the end.

Lets view the first 10 rows and the last 7 rows of our data.

```
head(iris, 10) # looks at first 10
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          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           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

```
tail(iris, 7) # looks at last 7
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 144          6.8         3.2          5.9         2.3 virginica
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

---

str()

str() stands for structure. This function allows to compactly display the structure of an R object, in this case our iris data.

Running str(iris) will give us the column names, the class type of each column (remember week 1) and the first few rows of data for each column.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

---

is.na()

is.na() stands for "are there any null/not available values in the data?". Using this function means we're just looking to see if anything is missing. If there is a missing value, the response will be TRUE. If there is a value, the response is FALSE.

I wont run the code in this document but you will in your R script.

Run the following code in your R script: is.na(iris)

What happened? It ran through the entire dataset responding to each cell whether it was null or had information. Pretty terrible to look at and not really helpful if we want to see how many null values or where those null values are. Luckily, table() helps us solve that.

---

table()

table() identifies the quantity of values in a given dataset and column. Let's run table() around our is.na() function on the iris data.

```
table(is.na(iris))
```

```
##
## FALSE
##   750
```

Way freaking easier. There are 1892 missing values in our dataset.

Lets use table() to check individual columns.

```
table(is.na(iris$Age)) # checking age for NA's
```

```
## < table of extent 0 >
```

```
table(is.na(iris$comments)) # checking comments for NA's
```

```
## < table of extent 0 >
```

So for age, there were zero rows left blank. How many blanks were left in the comments?

NOTE: table() isn't only for finding NA's. This function allows you to see each row data per column and their quantity. Lets see the gender reponse of each iris participant.

```
table(iris$Gender) # Gender responses
```

```
## < table of extent 0 >
```

```
length(unique(iris$Gender)) # code bonus: view how many unique responses there were.
```

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

There were quite a few different answers and some misspellings. We can use the tidyverse to work with that.

Manipulate the Data

For this section, we're going to use another dataset, **iris**. If you remember from week 1, we briefly used this dataset. To refresh yourself, feel free to code: ?iris

---

The tidyverse

We will be using what is called **the tidyverse**.

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. - tidyverse.org

You'll notice when you install and load the tidyverse packages, several packages will upload. Each package brings functions that are helpful in data science and data wrangling.

Let's install.

```
# install.packages("tidyverse") # if never done before
library(tidyverse) # load
```

```
## Warning: package 'tidyverse' was built under R version 4.2.2
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.8     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

---

dplyr

dplyr is a package that makes part of the larger tidyverse. dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges. - https://dplyr.tidyverse.org/

We will go through the major functions of dplyr.

---

piping (%>%)

%>% is called the forward pipe operator. Essentially it means by using this symbol after a dataset, we want to directly add functions applied to that dataset. Pipes let you take the output of one function and send it directly to the next.

Pay attention to the code and look for the %>%. As we go througt the code it should be much more clear on how to use them. If not, please feel free to come to zoom hours or Google "piping in dplyr".

Note: a short cut to making the %>% symbol:

- on windows: ctrl+shift+m
- on mac: command+shift+m

---

select()

select() picks variables based on their names.

Lets create a new variable called iris_select and let's say we only want to look at the following:

- Species
- Petal Length
- Petal Width

```
names(iris) # Call column names
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
iris_select <- iris %>%              # create new variable from iris data,add pipe to add function
  select(Species, Petal.Length, Petal.Width)# select() function. Select the specific columns of interes
names(iris_select) # column names of newly created dataset of selected columns
```

```
## [1] "Species"      "Petal.Length" "Petal.Width"
```

```
head(iris_select, 5) # check out the new data with 5 rows
```

```
##   Species Petal.Length Petal.Width
## 1  setosa          1.4         0.2
## 2  setosa          1.4         0.2
## 3  setosa          1.3         0.2
## 4  setosa          1.5         0.2
## 5  setosa          1.4         0.2
```

We see that using the piping and the select function, we were able to create a new dataset with our desired columns directly from the iris dataset.

---

mutate()

mutate() allows for the creation of new variables that are functions of existing variables. Basically we can create new columns in our existing data based on conditions or equations from other columns.

Lets create a new column in iris_select called Times.Bigger. This column should reflect how many times bigger in centimeters the petal length is than the petal width for each flower.

```
iris_select <- iris %>% # Call from iris
  select(Species, Petal.Length, Petal.Width) %>%    # Select our columns of interest
  mutate(Times.Bigger = (Petal.Length/Petal.Width)) # Create new column, set = to condition/function

names(iris_select) # Check names, notice the new column
```

```
## [1] "Species"      "Petal.Length" "Petal.Width"  "Times.Bigger"
```

```
head(iris_select, 5) # Does the column look right? it worked!
```

```
##   Species Petal.Length Petal.Width Times.Bigger
## 1  setosa          1.4         0.2          7.0
## 2  setosa          1.4         0.2          7.0
## 3  setosa          1.3         0.2          6.5
## 4  setosa          1.5         0.2          7.5
## 5  setosa          1.4         0.2          7.0
```

filter()

filter() does exactly what it sounds like. This function allows us to filter through our dataset based on columns, values, and conditions.

For this example, lets use filter() to make a new dataset called setosa_small that is populated with only data of the setosa species with sepal lengths smaller than 5.

We can do this by running conditions inside the filter() function.

To filter by specific value, use 2 equal signs (==) and set the column name to the value. In this case, Species will equal setosa (Species == "setosa"). NOTE: Since setosa is a character string, you must include quotation marks.

To filter in a numeric condition, use a mathematical operator. In this case, we want sepal length less than 5 (Sepal.Length $<$ 5). NOTE: Sepal length has numeric values, no quotations as it is not a character.

Lets run both conditions in the same function. String conditions together with the % symbol.

```r
setosa_small <- iris %>%         # create from iris dataset
  filter(Species == "setosa" & Sepal.Length < 5) # filter for species name and sepal length under 5

setosa_small # view the new dataset
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           4.9         3.0          1.4         0.2  setosa
## 2           4.7         3.2          1.3         0.2  setosa
## 3           4.6         3.1          1.5         0.2  setosa
## 4           4.6         3.4          1.4         0.3  setosa
## 5           4.4         2.9          1.4         0.2  setosa
## 6           4.9         3.1          1.5         0.1  setosa
## 7           4.8         3.4          1.6         0.2  setosa
## 8           4.8         3.0          1.4         0.1  setosa
## 9           4.3         3.0          1.1         0.1  setosa
## 10          4.6         3.6          1.0         0.2  setosa
## 11          4.8         3.4          1.9         0.2  setosa
## 12          4.7         3.2          1.6         0.2  setosa
## 13          4.8         3.1          1.6         0.2  setosa
## 14          4.9         3.1          1.5         0.2  setosa
## 15          4.9         3.6          1.4         0.1  setosa
## 16          4.4         3.0          1.3         0.2  setosa
## 17          4.5         2.3          1.3         0.3  setosa
## 18          4.4         3.2          1.3         0.2  setosa
## 19          4.8         3.0          1.4         0.3  setosa
## 20          4.6         3.2          1.4         0.2  setosa
```

summarise()

summarise() creates a new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarising all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified. - https://dplyr.tidyverse.org/reference/summarise.html

Basically summarise() allows us to view statistics of specific columns in a new dataset. You can use any of the descriptive statistic functions you will learn below in the Descriptive Statistics portion.

let's make a new data frame called species_sepals that has the mean, max, and min values of sepal width.

```r
iris %>%  # call from iris data
  summarise(mean(Sepal.Width), max(Sepal.Width), min(Sepal.Width)) # run functions, each will be new co
```

```
##   mean(Sepal.Width) max(Sepal.Width) min(Sepal.Width)
## 1          3.057333              4.4                2
```

How did it turn out? We got the values we needed but the values are for the whole dataset, not broken up by species. Let's use the group_by function.

---

group_by()

The group_by() function allows us to break up our dataset by a specific value. Above we wanted to see the mean, max, and min values of sepa width by species. We can use group_by(Species) for this to work.

```
iris %>%  # call from iris data
  group_by(Species) %>%
  summarise(mean(Sepal.Width), max(Sepal.Width), min(Sepal.Width)) # run functions, each will be new co
```

```
## # A tibble: 3 x 4
##   Species    `mean(Sepal.Width)` `max(Sepal.Width)` `min(Sepal.Width)`
##   <fct>                    <dbl>              <dbl>              <dbl>
## 1 setosa                    3.43                4.4                2.3
## 2 versicolor                2.77                3.4                2
## 3 virginica                 2.97                3.8                2.2
```

---

arrange()

arrange() allows us to rearrange our data based on certain conditions. Let's create a new dataset called virginica that contains only the virginica species, has sepal widths greater than 2.9, and is arranged in descending sepal lengths (we will use the desc() function).

NOTE: desc() stands for descending. It will places the rows in descending order.

```
iris_new <- iris %>% # call from iris data
  filter(Species == "virginica" & Sepal.Width > 2.9) %>% # filter species and sepal width
  arrange(desc(Sepal.Length)) # arrange the data by descending values in sepal length

iris_new # view the new data frame
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1           7.9         3.8          6.4         2.0 virginica
## 2           7.7         3.8          6.7         2.2 virginica
## 3           7.7         3.0          6.1         2.3 virginica
## 4           7.6         3.0          6.6         2.1 virginica
## 5           7.2         3.6          6.1         2.5 virginica
## 6           7.2         3.2          6.0         1.8 virginica
## 7           7.2         3.0          5.8         1.6 virginica
## 8           7.1         3.0          5.9         2.1 virginica
## 9           6.9         3.2          5.7         2.3 virginica
## 10          6.9         3.1          5.4         2.1 virginica
## 11          6.9         3.1          5.1         2.3 virginica
## 12          6.8         3.0          5.5         2.1 virginica
## 13          6.8         3.2          5.9         2.3 virginica
## 14          6.7         3.3          5.7         2.1 virginica
## 15          6.7         3.1          5.6         2.4 virginica
## 16          6.7         3.3          5.7         2.5 virginica
## 17          6.7         3.0          5.2         2.3 virginica
## 18          6.5         3.0          5.8         2.2 virginica
## 19          6.5         3.2          5.1         2.0 virginica
```

```
## 20          6.5         3.0          5.5         1.8 virginica
## 21          6.5         3.0          5.2         2.0 virginica
## 22          6.4         3.2          5.3         2.3 virginica
## 23          6.4         3.1          5.5         1.8 virginica
## 24          6.3         3.3          6.0         2.5 virginica
## 25          6.3         3.4          5.6         2.4 virginica
## 26          6.2         3.4          5.4         2.3 virginica
## 27          6.1         3.0          4.9         1.8 virginica
## 28          6.0         3.0          4.8         1.8 virginica
## 29          5.9         3.0          5.1         1.8 virginica
```

Descriptive Statistics

"Descriptive statistics are brief descriptive coefficients that summarize a given data set, which can be either a representation of the entire population or a sample of a population. Descriptive statistics are broken down into measures of central tendency and measures of variability (spread)" - The first thing to appear on google, www.investopedia.com

Basically, descriptive statistics gives us quick snapshots of our data. R has loaded in functions (and a variety of packages to use) to generate simple and fast descriptive statistics.

Data

We're going to be using the beaver1 dataset from the datasets package. For more information, code ?beaver1 in your console or script after installing the package.

```
# install.packages("datasets") # install if you never had used it before
library(datasets)

head(beaver1) # initial look
```

```
##   day time  temp activ
## 1 346  840 36.33     0
## 2 346  850 36.34     0
## 3 346  900 36.35     0
## 4 346  910 36.42     0
## 5 346  920 36.55     0
## 6 346  930 36.69     0
```

Basic Statistics of base R

R comes with a basic package for looking at descriptive statistics. The following functions are base functions meaning we don't need to load any packages, its already there.

mean()

Calculates mean.

```
mean(beaver1$temp)
```

```
## [1] 36.86219
```

min()

Minimum value of dataset column or list.

```
min(beaver1$temp)
```

```
## [1] 36.33
```

max

Maximum value of dataset column or list.

```
max(beaver1$temp)
```

```
## [1] 37.53
```

range()

The range of values in a column or list. Code is also included for finding difference in range of numbers and working with square brackets.

```
range(beaver1$temp) # returns minimum and max
```

```
## [1] 36.33 37.53
```

```
# If you would like to find the difference, run code below.
range(beaver1$temp)[2] - range(beaver1$temp)[1]
```

```
## [1] 1.2
```

```
# Notes: Range returns 2 values. If you have a list (just like the list of 2 values above),
# you can use square brackets [] to choose which value in the list you want to work with.
# So we can choose the max value [2 or the second value in list] and subtract the min value
# [1 or the first value in list].

# For more on square brackets: https://www.dataanalytics.org.uk/r-object-elements-brackets-double-brack
```

median()

Finds median of data column or list.

```
median(beaver1$temp)
```

```
## [1] 36.87
```

quantile()

Finds the indicated quantile. Quantile amount is in percent format (50% = 0.5)

```
# 30 percent quantile
quantile(beaver1$temp, 0.3)
```

```
##    30%
## 36.789
```

```
# 50 percent
quantile(beaver1$temp, 0.5)
```

```
##   50%
## 36.87
```

NOTE: You can also find the quantile of many values by including a list in the function.

```
numbers <- c(.25, .5, .75, .90) # I want these quantiles calculated
quantile(beaver1$temp, numbers) # running function with numbers variable as condition
```

```
##     25%     50%     75%     90%
## 36.7600 36.8700 36.9575 37.0970
```

sd() and var()

sd() stands for standard deviation. var() stands for the variance.

```
sd(beaver1$temp) # standard deviation
```

```
## [1] 0.1934217
```

```
var(beaver1$temp) # variance
```

```
## [1] 0.03741196
```

summary()

summary() is best the bang for your buck and shows almost (except sd and variance) everything we have covered so far in one single function. You can run it on entire datasets.

```
summary(beaver1)
```

```
##       day             time             temp           activ
##  Min.   :346.0   Min.   :   0.0   Min.   :36.33   Min.   :0.00000
##  1st Qu.:346.0   1st Qu.: 932.5   1st Qu.:36.76   1st Qu.:0.00000
##  Median :346.0   Median :1415.0   Median :36.87   Median :0.00000
##  Mean   :346.2   Mean   :1312.0   Mean   :36.86   Mean   :0.05263
##  3rd Qu.:346.0   3rd Qu.:1887.5   3rd Qu.:36.96   3rd Qu.:0.00000
##  Max.   :347.0   Max.   :2350.0   Max.   :37.53   Max.   :1.00000
```

mode()

mode() doesn't exist.. I'm not sure why but there are other ways to find this value through using table() and sort.

```
sort(table(beaver1$temp), decreasing = TRUE)[1] # table() to find how many per value, sort() to sort!
```

```
## 36.89
##     7
```

```
# then just select the first value (which should be our most occurring value based on a decreasing
# sort) and use square brackets [1] to find the first value in the list.
```

The psych package and describeBy()

There are a few packages that do some great descriptive analyses but psych is awesome. Install pysch and we are going to use describeBy(). This function can give us most of the descriptive statistics we need in a few lines of code. It can save you some time!

Let's view the descriptive statistics for each column in iris and group them by species.

```
# install.packages("psych") # install if needed!
library(psych)
```

```
##
## Attaching package: 'psych'
```

```
## The following objects are masked from 'package:ggplot2':
##
##     %+%, alpha
```

```
describeBy(iris, # define what data you will be using
           group = iris$Species) # define how you want to group your statistics
```

```
##
##  Descriptive statistics by group
## group: setosa
##              vars  n mean   sd median trimmed  mad min max range skew kurtosis
## Sepal.Length    1 50 5.01 0.35    5.0    5.00 0.30 4.3 5.8   1.5 0.11    -0.45
```

```
## Sepal.Width     2 50 3.43 0.38    3.4    3.42 0.37 2.3 4.4   2.1 0.04     0.60
## Petal.Length    3 50 1.46 0.17    1.5    1.46 0.15 1.0 1.9   0.9 0.10     0.65
## Petal.Width     4 50 0.25 0.11    0.2    0.24 0.00 0.1 0.6   0.5 1.18     1.26
## Species*        5 50 1.00 0.00    1.0    1.00 0.00 1.0 1.0   0.0  NaN      NaN
##                 se
## Sepal.Length 0.05
## Sepal.Width  0.05
## Petal.Length 0.02
## Petal.Width  0.01
## Species*     0.00
## ------------------------------------------------------------
## group: versicolor
##              vars  n mean   sd median trimmed  mad min max range  skew kurtosis
## Sepal.Length    1 50 5.94 0.52   5.90    5.94 0.52 4.9 7.0   2.1  0.10    -0.69
## Sepal.Width     2 50 2.77 0.31   2.80    2.78 0.30 2.0 3.4   1.4 -0.34    -0.55
## Petal.Length    3 50 4.26 0.47   4.35    4.29 0.52 3.0 5.1   2.1 -0.57    -0.19
## Petal.Width     4 50 1.33 0.20   1.30    1.32 0.22 1.0 1.8   0.8 -0.03    -0.59
## Species*        5 50 2.00 0.00   2.00    2.00 0.00 2.0 2.0   0.0   NaN      NaN
##                 se
## Sepal.Length 0.07
## Sepal.Width  0.04
## Petal.Length 0.07
## Petal.Width  0.03
## Species*     0.00
## ------------------------------------------------------------
## group: virginica
##              vars  n mean   sd median trimmed  mad min max range  skew kurtosis
## Sepal.Length    1 50 6.59 0.64   6.50    6.57 0.59 4.9 7.9   3.0  0.11    -0.20
## Sepal.Width     2 50 2.97 0.32   3.00    2.96 0.30 2.2 3.8   1.6  0.34     0.38
## Petal.Length    3 50 5.55 0.55   5.55    5.51 0.67 4.5 6.9   2.4  0.52    -0.37
## Petal.Width     4 50 2.03 0.27   2.00    2.03 0.30 1.4 2.5   1.1 -0.12    -0.75
## Species*        5 50 3.00 0.00   3.00    3.00 0.00 3.0 3.0   0.0   NaN      NaN
##                 se
## Sepal.Length 0.09
## Sepal.Width  0.05
## Petal.Length 0.08
## Petal.Width  0.04
## Species*     0.00
```

Basic Plots for Descriptive Statistics

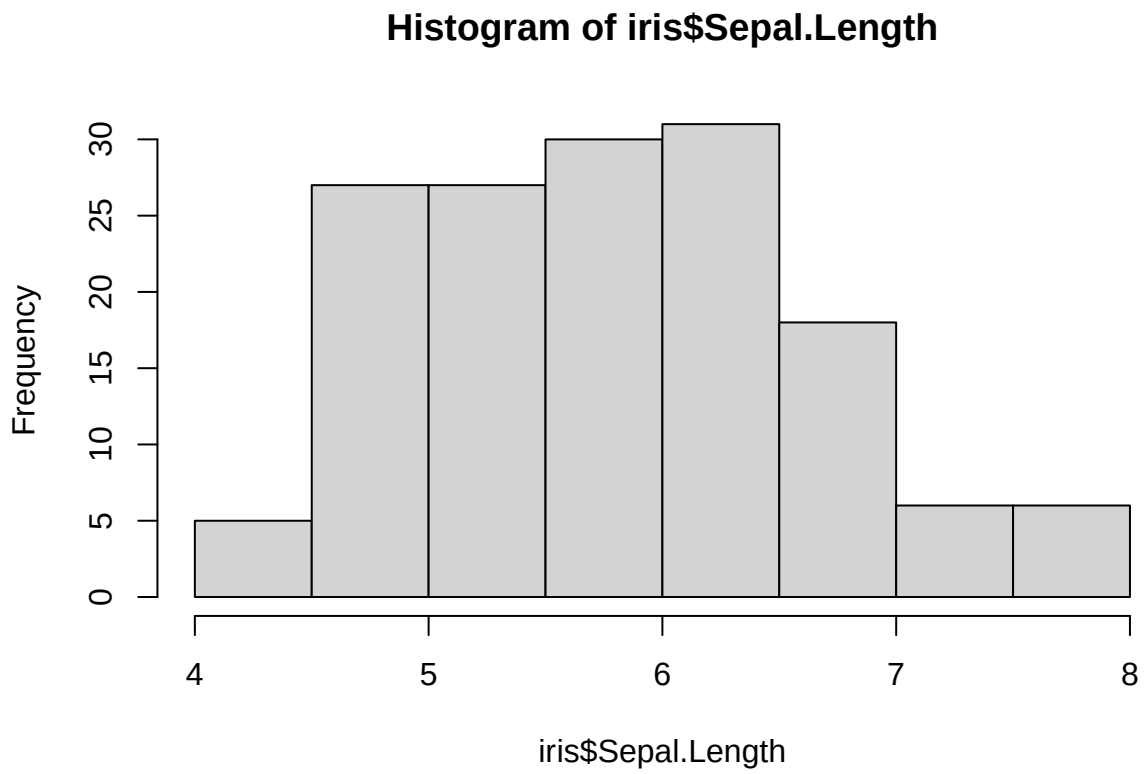
We will spend week 3 going over data visualization and I'll provide with several different ways to make some excellent graphs (and some maps! - that comes later too) but here are some simple ways to visualize your data.

---

hist()

Histograms give us insight into the spread of our data.

```
hist(iris$Sepal.Length)
```



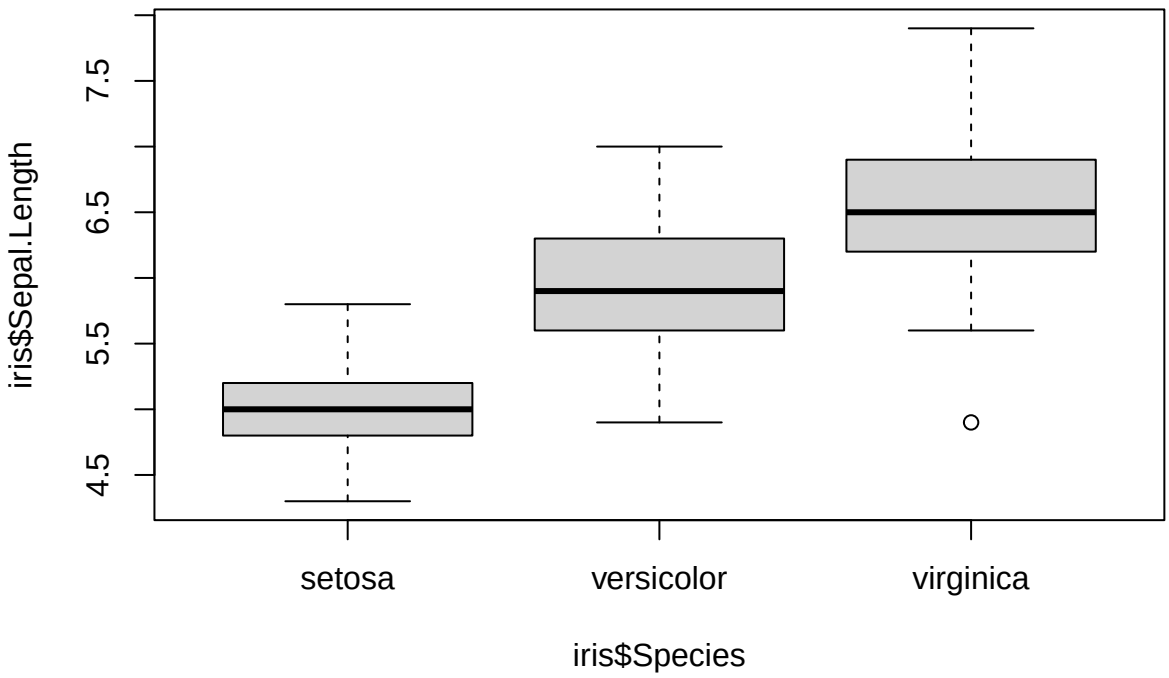
---

boxplot()

Boxplots are very similar to histograms but can only measure the quantitative values of a qualitative variable. For example, we can look at the distribution of sepal lengths for each iris species.

Boxplot code always follows this format: boxplot(quantitative variable ~ qualitative variable) or boxplot(y axis ~ x axis).

```
boxplot(iris$Sepal.Length ~ iris$Species) # first y axis then x axis
```



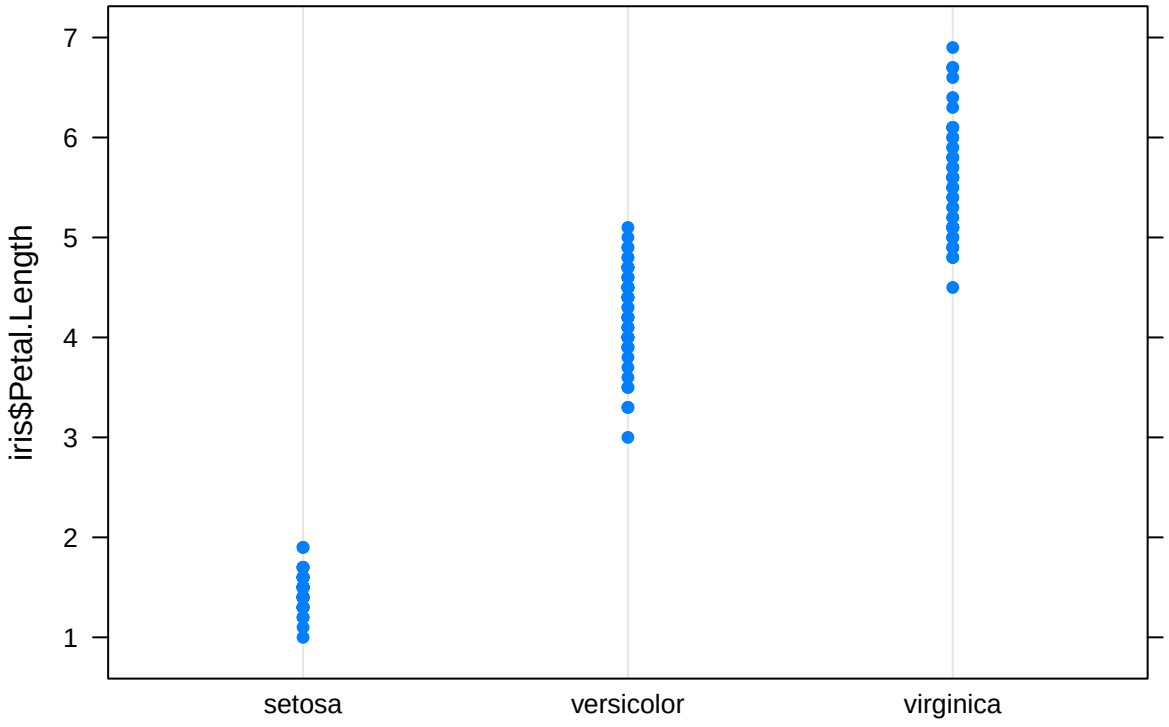
---

dotplot()

Dotplots are very similar to boxplots only instead of boxes showing the spread of data, they show the actual value as a dot. Dotplots follow the same format as boxplots (y~x).

You'll find dotplot() in the package: lattice. Install lattice and run a dotplot.

```
# install.packages(lattice) # install lattice if never used before!
library(lattice)
dotplot(iris$Petal.Length ~ iris$Species)
```

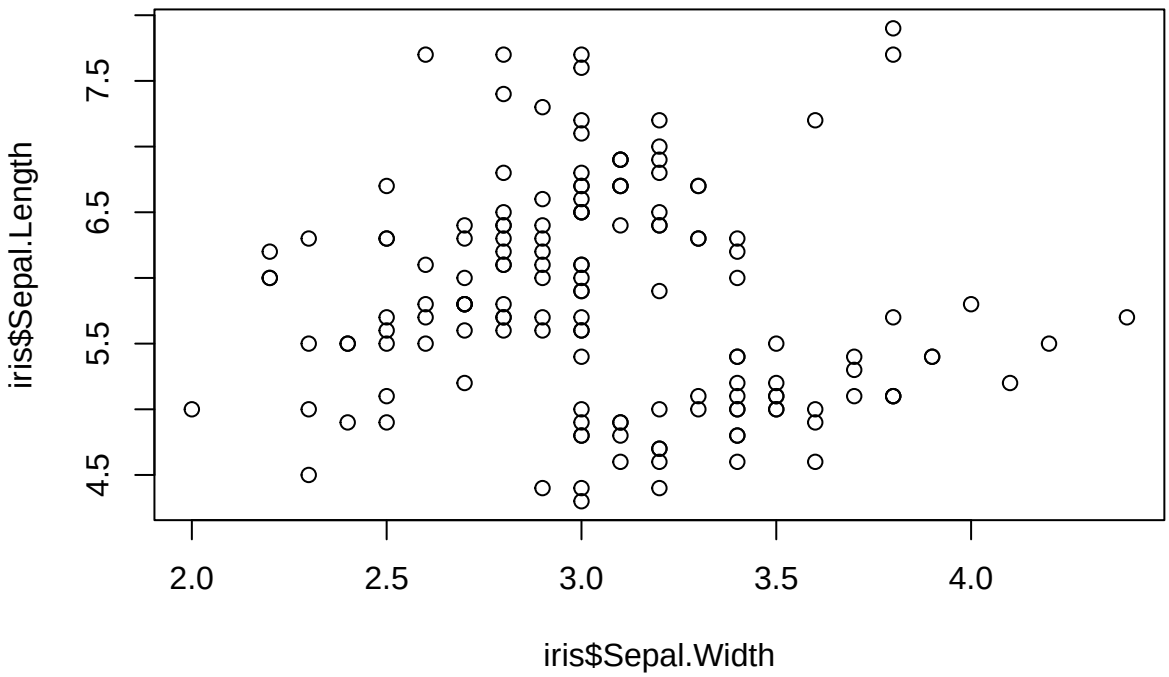


---

scatterplots

A scatterplot requires two quantitative variables and is used in linear regression to measure the relationships among variables. Scatterplots follow the same format as the other (y~x) but we only need to use the function, plot().

```
plot(iris$Sepal.Length ~ iris$Sepal.Width)
```



Done!

That was fairly long but I hope it helped you learn (or reinforce) some skills in getting to know your data. The week 2 assignment is in week 2 module.

As always, if you need any help feel free to contact me or come to the zoom office hours.