R Demo 1

Neil J. Hatfield

1/24/2020

Contents

Abstract

In this demo, I'll be demonstrating how to use `R` and RStudio to write and run code. Additionally, I'll be demonstrating using R Markdown to create output documents as PDFs.

Getting Started

Keep in mind that you may use other statistical software packages (e.g., SAS, JMP, SPSS, Minitab, etc.) as well as other programs to produce your reports (e.g., Microsoft Word, LibreOffice, Google Docs, etc.). My in-class demonstrations and my demonstration documents will be limited to `R` and RStudio.

I'm going to start with the assumption that you already have the software package installed on your machine OR you have access to the software on your computer device. If you are having problems with installation or gaining access, please come see me ASAP. Launch your software package (`R`/Rstudio, or your chosen combination) now.

Creating a R Markdown File

When you've started RStudio, you can create a new R Markdown file. This will allow you to create a professional looking document that contains all of your code plus your narrative. Click on `File`, mouse over `New File` and then select `R Markdown...`. This will launch the the wizard to help you make the appropriate selections. For this class, you'll need to either select PDF or Word as the output type. **NOTE: you might need to install additional packages in `R` and/or software–a TeX installation–on your computer.**

There is also an option where you can start from a template. You might checkout Alex Haye's R Markdown Homework Template, which will automatically make a Code Appendix for you.

Creating a Document

If you are not using R Markdown, you'll want to create a document in either Microsoft Word, Google Docs, or LibreOffice. You will want to make sure that you understand how to use that program for inserting graphics, making tables, and writing mathematics.

Using Headers and Styles

In both R Markdown and documents, you should use styles to help format your document. These often refer to Headers and are similar to HTML headers (h1, h2, h3, etc.). Headings nest with Heading 1 being the highest level.

Document Styles

Microsoft Word has a Styles menu that is part of the Home Ribbon. LibreOffice has a dedicated Styles menu. Google Docs has Styles dropdown menu as part of the main toolbar.

R Markdown Styles

In R Markdown, you'll use the octothorp (the "hash tag" or "pound sign") to tell a line to be styled as a header. The number of octothorps you use indicates the level of heading; # for Heading 1, ## for Heading 2, ### for Heading 3, etc. Return to move to a new line to start a normal paragraph after each heading.

Bold and Italics

Documents allow you to use regular commands for using bold and italics. R Markdown uses a slightly different approach. For *italics*, you'll want to enclose the text in single asterisks (*italics*) or underscores (_italics_). For **boldface**, use two asterisks or two underscores on either side of the text (**boldface** or __boldface__).

Mathematics

When you want to include mathematics in your document you'll want to use the built-in Equation Editor for Microsoft Word, Google Docs, and LibreOffice.

For R Markdown, you'll use LaTeX syntax. You may enclose mathematics either in single dollar signs (`$f(x)=x^2+\frac{3}{4}$`) or with a backslash-parenthesis combo (i.e., `\(f(x)=x^2+\frac{3}{4}\)`) for in-line math; $f(x) = x^2 + \frac{3}{4}$.

For display style math (i.e., math that gets placed on a new line and centered automatically), use double dollar signs or a backslash-square bracket combo. That is,

$$f(x)=x^2+\frac{3}{4}$$ or \[f(x)=x^2+\frac{3}{4}\]:

$$f(x) = x^2 + \frac{3}{4}$$

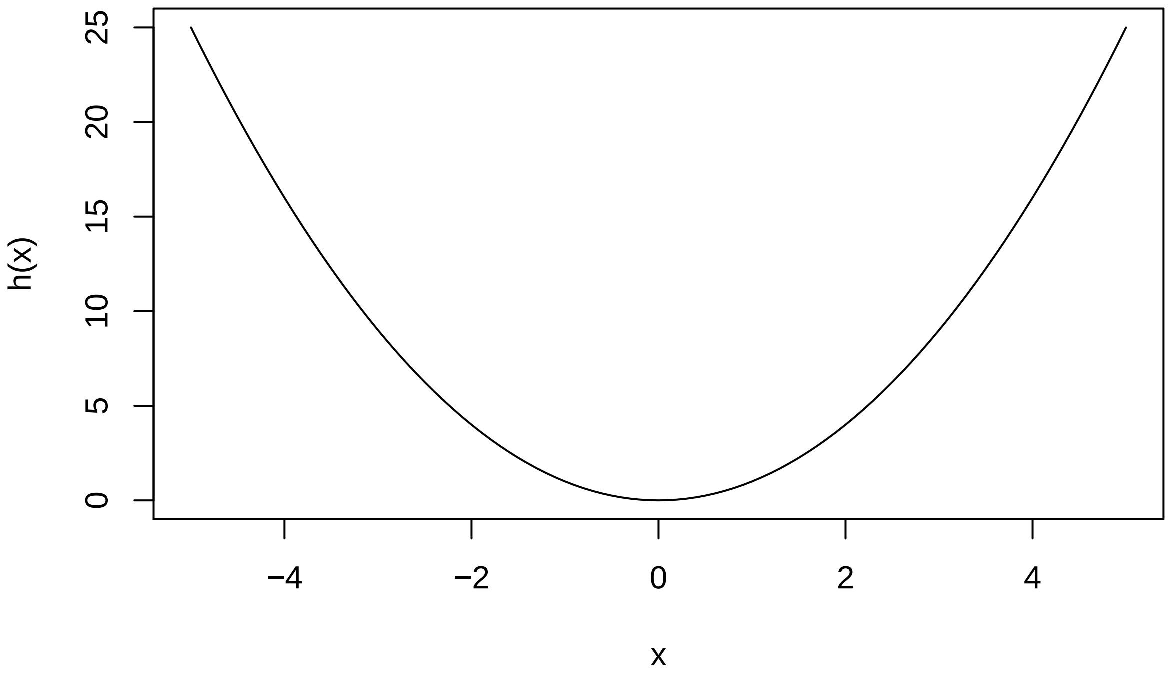
You may also use math environments such as align (for numbered, multiline statements) and align* (for unnumbered multiline statements). To do so, you will start a new line with `\\begin{align*}` and then type your set of lines. To end a line and move to the next you will need to use two backslashes, \\. When finished, you'll have one more line (no proceeding double backslash necessary) with `\\end{align*}`.

$$\begin{align*}
H_0 &: \ y = \mu_{\bullet} + \epsilon_i \\
H_1 &: \ y = \mu_{\bullet\bullet} + \alpha_i + \epsilon_{ij}
\end{align*}$$

`R` is a graphing calculator

Don't forget that `R` can be used as a graphing calculator. You can perform calculations and even plot regular mathematical functions:

```r
curve(x^2, from = -5, to = 5, xlab="x", ylab="h(x)")
```



Code-R Markdown Only

The benefit of using R Markdown is that you can embed code in your narrative directly, either as in-line or code chunks. The code will be run (provided you haven't set the `eval=FALSE`) automatically when you knit your file.

In-Line Code

For in-line code you start with a single backtick (‘; the key next to the 1 key) and then type “r” (lower case and no quotation marks) to set the language. After the “r”, you put a space and then type of the code you want run. You finish the code with another backtick (‘) When you knit, the in-line code will run and you’ll see the output rather than the code.

Code Chunks

For a code chunk, you’ll want to return twice from your current line. You’ll then type three backticks (‘ ‘ ‘) to start the chunk and then use curly braces to set options: {`r chunkName, option1=value, option2=value, etc.`}. (At minimum you must have the little r to set the code language. I recommend you set a chunk name so you can navigate easily. You’ll then type your code just was you would in the `R` Console. You close the code chunk with another set of three backticks. When you knit, your code chunk will be added to your file (if you set `echo=TRUE`) and your code’s output will be added.

Code chunks have the extra benefit that you can run them separately from the Knit command. This allows you to see a preview of what they will yield as well as populating your current environment with objects. This lets you test out code in the Console as well as view objects. To run a chunk look to the upper right of the chunk in the R Markdown file.You should see three icons: a gear (options menu) a grey triangle point to a green block, and a green triangle/play button. The middle icon will run all code chunks down to the current one, while the last icon will only run that one code chunk.

Professional looking reports will have code included in an appendix, NOT in the body of the report. Thus, you’ll want to use the option `echo=FALSE` when in the main part of your report. I’ve used `echo=TRUE` in this teaching document so you can see code paired with output.

Other Styles

Make use of the internet, RStudio’s Cheat sheets (Help->Cheatsheets) to help you discover more of the ins and outs for using R Markdown. I recommend Xie, Allaire, and Grolemund’s R Markdown: The Definitive Guide as a great resource. Further, Xie has a great resource for Kintr Options. Ismay and Kennedy also have a great resource: Getting Used to R, RStudio, and R Markdown that you can checkout.

You can also use internet for Microsoft Word, Google Docs, and LibreOffice help. You can also come see me for help.

Plot/Figure Options

There are a number of options you can use with code chunks that produce plots (figures in general). The most useful figure related code chunk options are:

- `fig.align='left', 'center', or 'right'` (sets whether the plot is left, center, or right aligned)

- `fig.width=#` (sets the width of the plot to # inches)
- `fig.height=#` (sets the height of the plot to # inches)
- `fig.caption="caption text goes here"` (automatically labels and numbers your figure while placing the caption; allows you to include an in-text reference to the figure)

Documents

Microsoft Word, Google Docs, LibreOffice should all have the same capabilities to control the size, position, and labeling of figures. In Microsoft Word, here is how you add the label and caption and then include a reference:

1. Insert your graphic into Word.
2. Click/Select your graphic, and then from the Insert Menu select Caption…
3. Fill in the appropriate information and press OK.
4. In a sentence where you want to add a reference, select Cross-reference… from the Insert Menu
5. Make the appropriate selections in the drop down menu and selection field. Press Insert when finished.

*Note: Word will update the numbering when you call up the Print menu (even if you don't print). You'll need to do this if you change the order of your figures, add/remove some, etc.

R Markdown Figure Label and Cross-reference

1. Build your code chunk to make a plot and give the chunk a name.
2. In the chunk options, you'll need to define a caption: `fig.caption="caption text goes here"` (If you do this, you can omit the graph title.)
3. In the body of your narrative, type `\ref{fig:chunkName}` to call the reference to your plot.

Stuck? Come See Me.

If you get stuck on how to do something, come see me. I'm happy to help.

Reading Data In

I will attempt to make data files available in formats that multiple software packages can read. These would include the CSV, DAT, TXT, and XLSX formats. Occasionally, I might post something in a format (e.g., RData) that might not work in one program or another. If I do, please reach out to me and I will help you.

RStudio Wizard

If you are using RStudio, you can use the Import Dataset wizard to help you. From the Environment pane, click on the Import Dataset button and select the appropriate option from the dropdown.

You can also click on the File menu, scroll down to Import Dataset, and then make your selection.

Writing Code

You can also write your own code to import a data set. The five most useful functions for reading data into `R` are:

- `read.table`; good csv, txt, dat files and web hosted files
- `read.csv`; generally good for CSV files–internally calls `read.table`
- `load`; good for RData files
- `openxlsx::read.xlsx`; good for reading xlsx files
- `data`; good for loading a dataset that is included in `R`

Code Writing Tip: If you're using a function from a non-base package, call the package in your code explicitly by including the namespace in the function call, i.e., `openxlsx::`. This will help you keep track of where particular functions are coming from and what packages you actually use.

Let's practice by loading the Demo Oreo Data.

```
demoOreo <- read.table(
  file = "https://raw.github.com/neilhatfield/STAT461/master/classDemoOreo.dat",
  header = TRUE, sep = ",")
```

Let's now read in a data set that is part of R, insect sprays:

```
data("InsectSprays")
```

Notice that we did not need to store the data in an `R` object like we did with the Oreo Data. The name "InsectSprays" becomes name of the data object.

Structure of Data

A good thing to do once you've read in your data is to look at both the structure of the data and the data. The `str` function in `R` will give you information about the structure of *any* object in `R`.

```
str(demoOreo)
```

```
## 'data.frame':    60 obs. of  2 variables:
##  $ Filling.Mass: num  3.27 3.12 3.15 3.27 3.24 ...
##  $ Type        : Factor w/ 2 levels "Double Stuf",..: 2 2 2 2 2 2 2 2 2 2 ...
```

```
str(InsectSprays)
```

```
## 'data.frame':    72 obs. of  2 variables:
##  $ count: num  10 7 20 14 14 12 10 23 17 20 ...
##  $ spray: Factor w/ 6 levels "A","B","C","D",..: 1 1 1 1 1 1 1 1 1 1 ...
```

The `View` function will display what is in the object and is only useful in the Console (i.e., don't use inside your in-line or code chunks of your R Markdown file).

Data Visualizations

There are multiple packages (and ways) to create different data visualizations in `R`. The two ways I'll list here are to use the "Base" packages that automatically come with each `R` installation and automatically load and using the `ggplots2` package.

If you're using another software package, you'll have to see how to use that package to create data visualizations. In SAS, I recommend using ODS graphics procedures such as `sgpie`, `sgplot`, and `sgscatter`.

My intent here is to NOT cover all possible ways to build all possible data visualizations. Rather, I want to present some of the basics to get you started.

Histograms

Histograms (or bar charts) are one of the most fundamental data visualizations. They are fairly easy to create. In base `R`, you can use the `hist` function:

```r
hist(demoOreo$Filling.Mass)
```

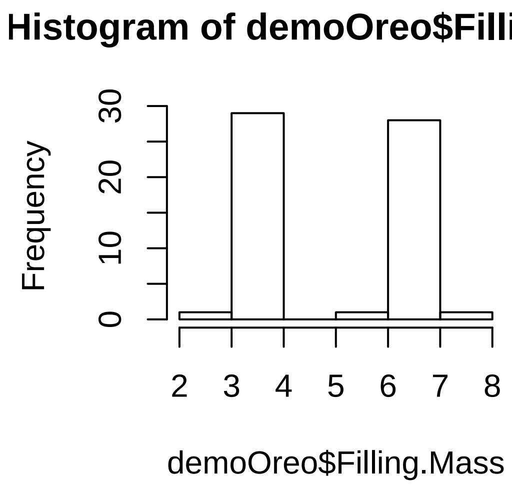


Figure 1: Histogram of Oreo Creme Filling Mass

Notice that Figure 1 is rather plain and has a few issues. The horizontal label is unprofessional looking. The title is similar, redundant since I've included a caption, and gets cut off due to our size options. However, there is subtle issue: `R` breaks with standard convention for histograms by doing open left, closed right bins. To fix these issues, we need to tweak the function call:

```r
hist(demoOreo$Filling.Mass,
     right = FALSE, #forces R to make a conventional histogram
     main = NA, #removes the title
     xlab = "Filling Mass (g)", #adds a better label
     col = "springgreen" #allows you to color the bars
     )
```

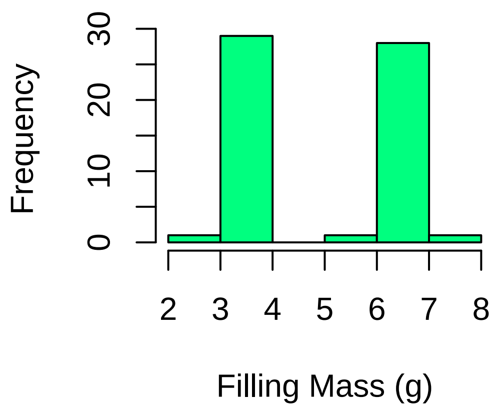


Figure 2: Fixed Histogram of Oreo Creme Filling Mass

Notice that Figure 2 is better looking histogram than Figure 1. For a listing of colors available in base R check out this page.

You can also use the `ggplots2` package as well. If you don't already have this package installed, you'll need to run `install.packages("ggplot2")` in the Console first. Then in your code you'll need to add `library(ggplot2)`; the first code chunk that RStudio built for you (i.e., "setup") is a great place to add the library call for any additional packages.

```
ggplot2::ggplot(data = demoOreo, # set the data frame to be used
                mapping = aes(x = Filling.Mass) # define which attributes are where
                ) + # the plus sign lets you add elements
  ggplot2::geom_histogram() # such as the histogram
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

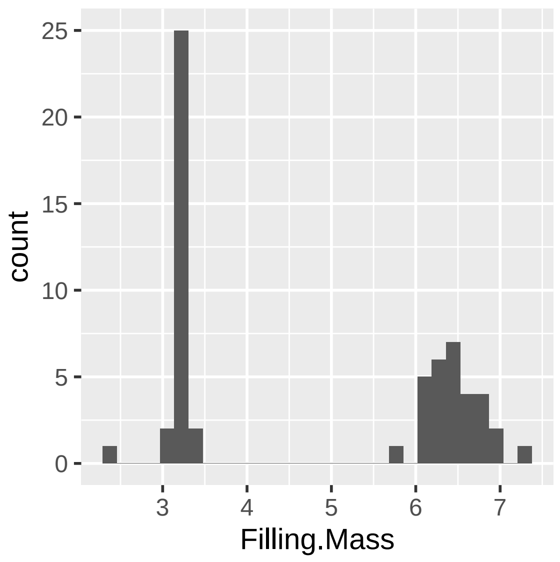


Figure 3: ggplot2 Histogram of Oreo Creme Filling Mass

Notice that just prior to the histogram in Figure 3, there was a warning message printed: "## `stat_bin()` using `bins = 3`. Pick a better value with `binwidth`." Warnings and messages should not appear in your narrative. You may suppress these with code chunk options (e.g., `warning=FALSE`).

Figure 3 can be improved as shown in Figure 4.

```
ggplot2::ggplot(data = demoOreo,
                mapping = aes(x = Filling.Mass)
                ) +
  ggplot2::geom_histogram(
    binwidth = 0.5, # define how wide each bin should be
    closed = "left", # uses a standard histogram
    na.rm = TRUE # remove missing values silently
  ) +
  ggplot2::theme_bw() + # applies a standard black and white color theme
  xlab("Filling Mass (g)") + # fixes the hoizontal label
  ggplot2::scale_x_continuous( # Horizontal axis controls
    expand = expand_scale(mult = 0, add = 0), # control buffer space
    limits = c(0,8) # limits of horizontal axis
    ) +
  ggplot2::scale_y_continuous( # Veritcal axis controls
    expand = expand_scale(mult = 0, add = 0),
    limits = c(0,20)
    )
```

Notice that the lower left corner (0,0). While not vitally important for histograms, being able to place the origin in the lower left corner is useful for other plots (e.g., scatterplots).

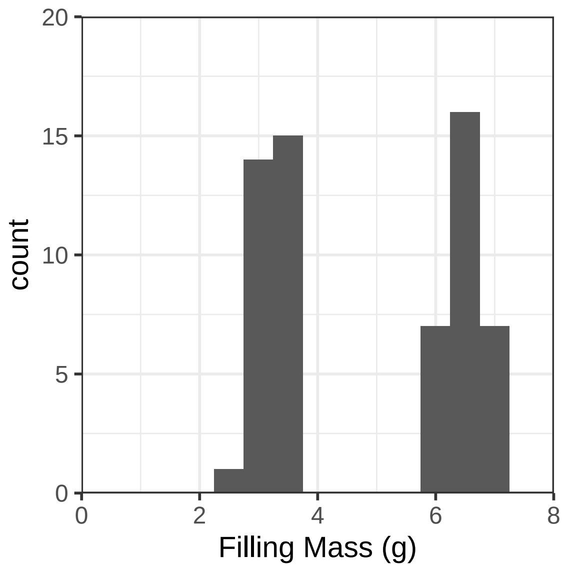


Figure 4: Fixed Histogram of Oreo Creme Filling Mass

The `ggplot2` package DOES NOT do so natively; the package will insert space on both axes by default. You will need to use `scale_x_continuous` and `scale_y_continuous` to override this.

Density Plots

Density plots are useful for looking for modal clumps that histogram might otherwise hide or over-exaggerate. Figure 5 shows the base package approach to denisty plots.

```r
d <- density(demoOreo$Filling.Mass)
plot(d, main = NA, xlab = "Filling Mass (g)")
polygon(d, col="springgreen")
```

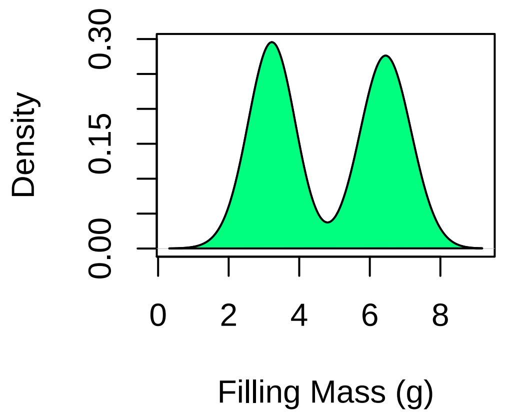


Figure 5: Density Plot of Oreo Creme Filling Mass

You may also use `ggplot2` for density plots as shown in Figure 6.

```r
ggplot2::ggplot(data = demoOreo,
                mapping = aes(x = Filling.Mass)
                ) +
  ggplot2::geom_density(
    na.rm = TRUE,
    color = "black",
    fill = "springgreen"
  ) +
  ggplot2::theme_bw() +
  xlab("Filling Mass (g)") +
  ggplot2::scale_x_continuous(
    expand = expand_scale(mult = 0, add = 0),
    limits = c(0,10)
    ) +
  ggplot2::scale_y_continuous(
    expand = expand_scale(mult = 0, add = c(0,0.025)),
    )
```

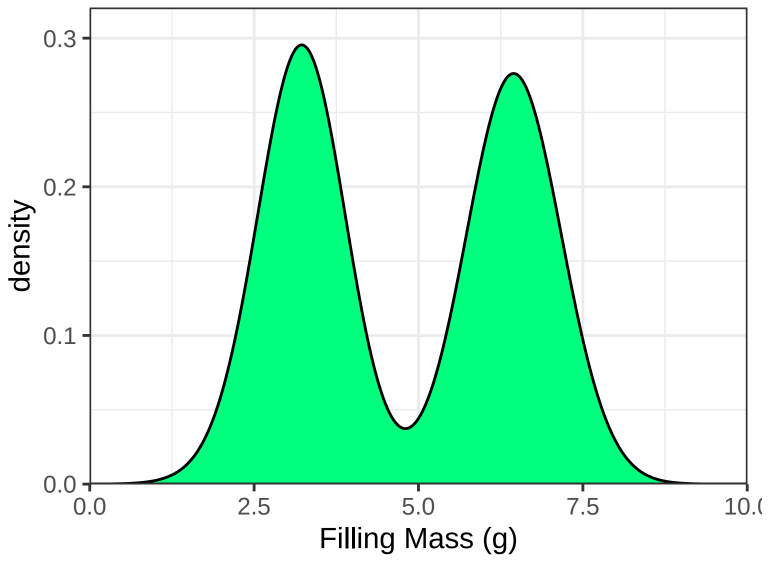


Figure 6: Density Plot of Oreo Creme Filling Mass

Another package that you can use for density plots is `lattice` as shown in Figure 7.

```r
lattice::densityplot(
  demoOreo$Filling.Mass,
  na.rm = TRUE,
  xlab = "Filling Mass (g)"
  )
```

The `lattice` package adds dots to the density plot for the observed values, allowing you to see where your observations fall.

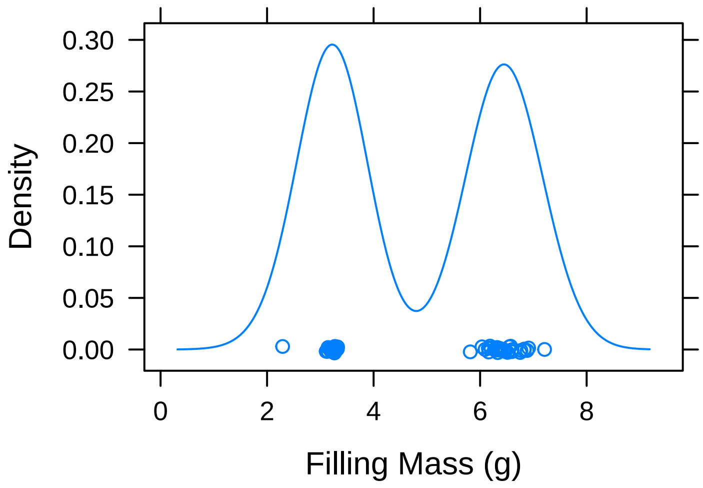


Figure 7: Lattice Density Plot of Oreo Creme Filling Mass

Boxplots

Boxplots (a.k.a. box-and-whisker plots) are another one of those useful data visualizations. The most common boxplot is Tukey's Boxplot which makes Tukey's Five Number Summary into a visualization. However, there are also outlier boxplots which do not extend the whiskers automatically to the minimum and maximum values. Rather, the whiskers extend to the larger of the [minimum value, lower IQR bound] and the smaller of [maximum value, upper IQR bound]. Observations beyond the IQR bounds appear as asterisks or dots in the plot. Figure 8 shows `R`'s default boxplot.

```r
boxplot(demoOreo$Filling.Mass,
        na.rm = TRUE,
        xlab = "Filling Mass (g)",
        horizontal = TRUE,
        range = 0) # 0 -> standard boxplot, 1.5 -> outlier boxplot
```

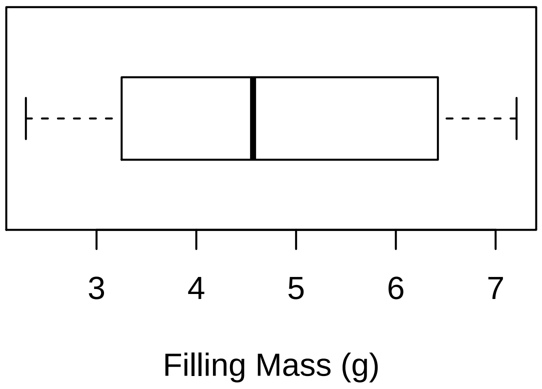


Figure 8: Boxplot of Oreo Creme Filling Mass

Figure 9 shows making a boxplot with `ggplot2`.

```r
ggplot2::ggplot(data = demoOreo,
                mapping = aes(y = Filling.Mass)
                )+
  ggplot2::geom_boxplot(
    na.rm = TRUE
  ) +
  ylab("Filling Mass (g)") +
  ggplot2::theme_bw()
```

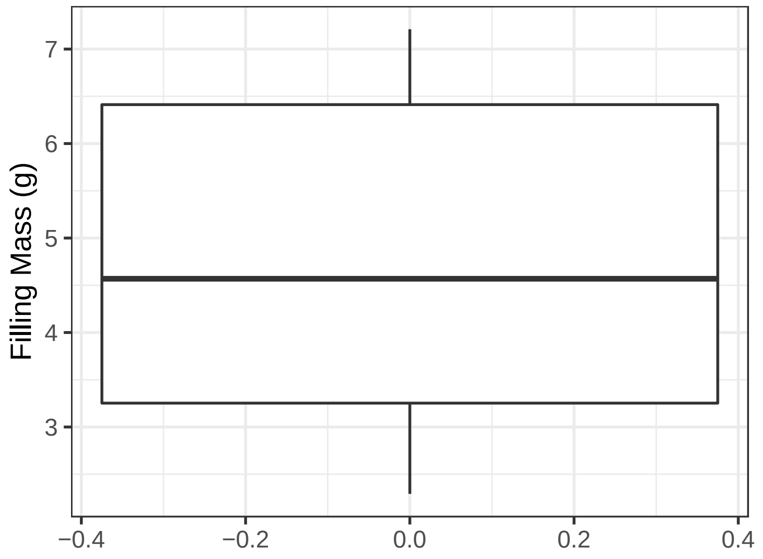


Figure 9: Boxplot of Oreo Creme Filling Mass

QQ Plots

QQ Plots or Quantile-Quantile Plots are useful visualizations for deciding whether our data violate an assumption about following a particular distribution (typicaly, a normal/Gaussian distribution).

```
qqplot(
  x = qnorm(ppoints(500),
            mean = mean(demoOreo$Filling.Mass),
            sd = sd(demoOreo$Filling.Mass)),
  y = demoOreo$Filling.Mass,
  main = NA,
  xlab = "Theoretical Quantiles",
  ylab = "Sample Quantiles"
  )
qqline(demoOreo$Filling.Mass,
       distribution = function (p){
         qnorm(p, mean = mean(demoOreo$Filling.Mass),
            sd = sd(demoOreo$Filling.Mass))
         }, # set reference distribution
       col = "blue", # line color
       lwd = 3 # line thickness
       )
```

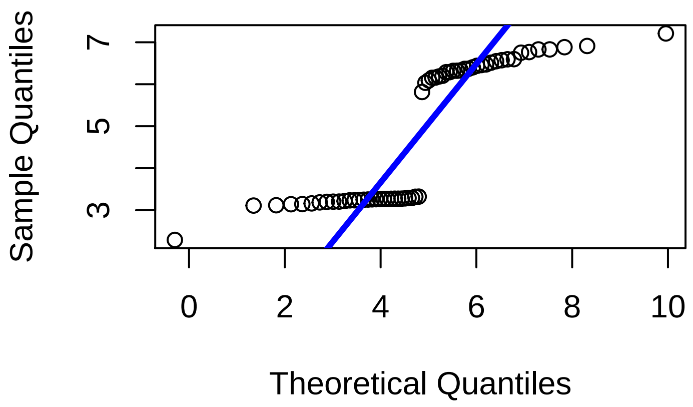


Figure 10: QQ Plot of Oreo Creme Filling Mass

If you just want to reference the normal/Guassian distribution, you can replace `qqplot` with `qqnorm` and drop the `x` parameter.

The `car` package allows you to make a qqplot that automatically adds the reference line as well as confidence limits as shown in Figure 11.

```
a <- car::qqPlot( # store as an object to get just the plot to show.
  x = demoOreo$Filling.Mass,
  distribution = "norm",
  envelope = 0.90, # sets confidence level
  ylab = "Filling Mass (g)"
)
```

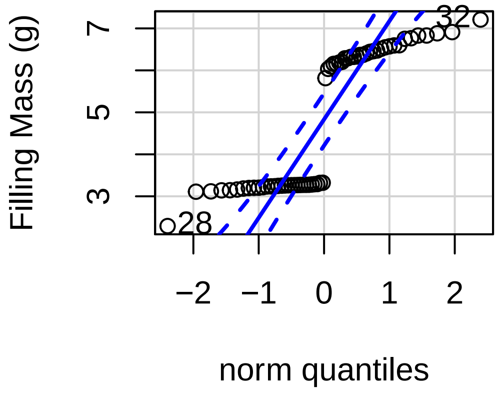


Figure 11: Car QQ Plot of Oreo Creme Filling Mass

Figure 12 is an example of a QQ plot using `ggplot2`.

```
ggplot2::ggplot(data = demoOreo,
                mapping = aes(sample = Filling.Mass)
                )+
  ggplot2::geom_qq(
    distribution = stats::qnorm, # notice the explicit package call
    dparams = list(), # defaults to sample; set custom parameters here
    na.rm = TRUE
  ) +
  ggplot2::geom_qq_line(
    color = "blue",
    na.rm = TRUE,
    distribution = stats::qnorm,
    dparams = list(),
  ) +
  ggplot2::theme_bw()
```

Using a Model Statement

If you use a model statement, you can expand your data visualizations, especially when you have multiple groups. For example, Figure 13 shows a scatterplot of filling mass, separating out the two types of oreos by using the model statement `Filling.Mass ~ Type`. The general format of a model statement is [response] ~ [factors and interactions].

Several data visualization functions support a model statment in conjunction with a `data` parameter. When you use a model statement instead of a data vector, the plot will change. For example, `boxplot` will now give you side-by-side boxplots, splitting on the factor you include to the right of the tilde (~).

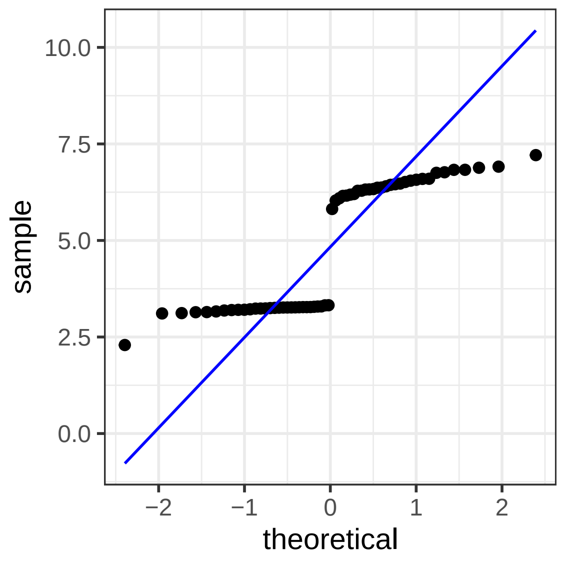


Figure 12: Car QQ Plot of Oreo Creme Filling Mass

```r
stripchart(Filling.Mass ~ Type,
                  vertical = TRUE,
                  pch = 20,
                  data = demoOreo)
```

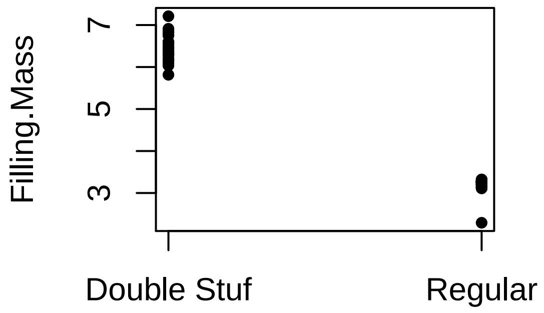


Figure 13: Scatterplot of Oreo Creme Filling Mass by Type

Figure 14 shows side-by-side box plots made by using a model statement.

```r
boxplot(Filling.Mass ~ Type,
        data = demoOreo,
        na.rm = TRUE,
        xlab = "Filling Mass (g)",
        horizontal = TRUE,
        range = 0)
```

You can also make this type of plot in `ggplot2` as shown in Figure 15.

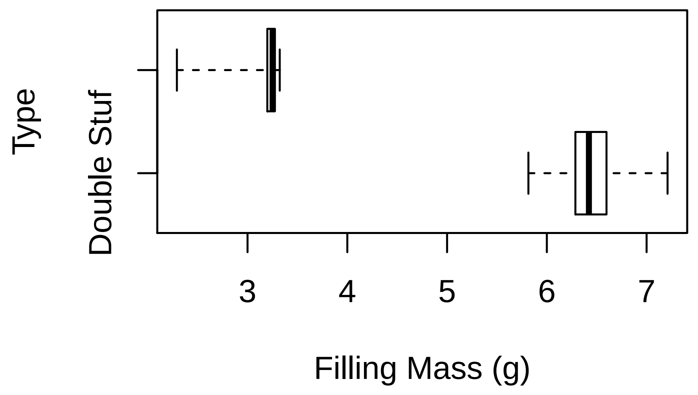


Figure 14: Boxplots of Oreo Creme Filling Mass by Type

```r
ggplot2::ggplot(data = demoOreo,
                mapping = aes(x = Type, y = Filling.Mass)
) +
  ggplot2::geom_dotplot(binaxis = "y",
                        stackdir = "center",
                        binwidth = 0.25,
                        dotsize = 0.25,
                        stackratio = 1.5) +
  ggplot2::theme_bw()
```

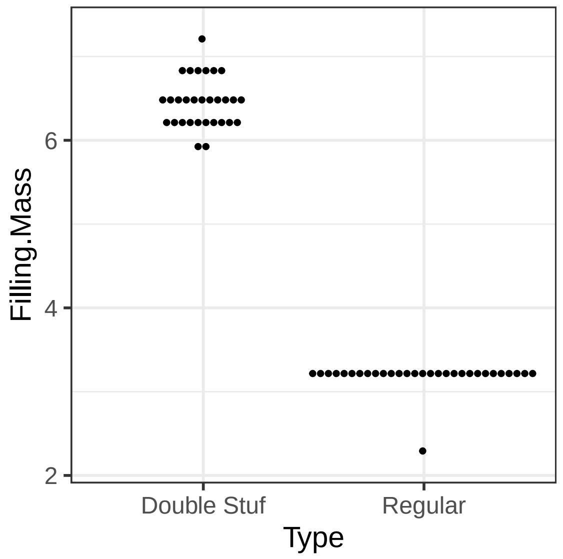


Figure 15: Scatterplot of Oreo Creme Filling Mass by Type

`ggplot2` also allows for you to make side-by-side box plots as shown in Figure 16.

```r
ggplot2::ggplot(data = demoOreo,
                mapping = aes(y = Filling.Mass,
                                x = Type,
                                fill = Type)
              )+
  ggplot2::geom_boxplot(
    na.rm = TRUE
  ) +
  ylab("Filling Mass (g)") +
  xlab("Type of Oreo") +
  ggplot2::theme_bw()
```

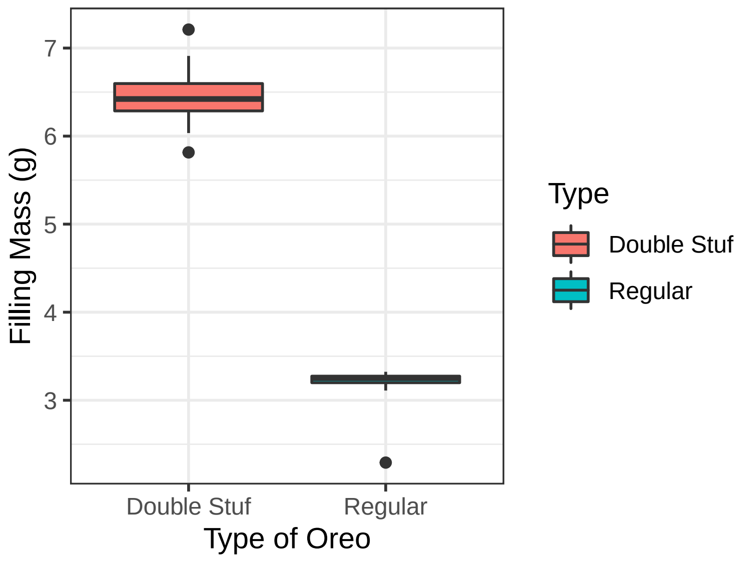


Figure 16: Boxplot of Oreo Creme Filling Mass by Type

Notice that when using `ggplot2`, you don't need to use a model statement (i.e., `Filling.Mass ~ Type`). Rather you use both the `x` and `y` parameters in the aesthetic mapping (`aes`).

Descriptive/Incisive Statistics

Remember, data visualizations can only get you so far when you are writing up a data narrative. Using descriptive/incisive statistics will help you make a more complete narrative. You should interpret at least one value of each statistic you make use of in your narrative.

Five Number Summary

Tukey's Five Number Summary is a fantastic set of 5 statistics. There are numerous ways to get the Five Number Summary in R.

```
summary(demoOreo$Filling.Mass, digits = 3)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.29    3.25    4.57    4.83    6.41    7.21
```

```r
quantile(demoOreo$Filling.Mass,
         na.rm = TRUE,
         type = 6 # There are 9 different formulas to choose from
         )
```

```
##      0%     25%     50%     75%    100%
## 2.29200 3.25025 4.56900 6.43075 7.21000
```

Sample Arithmetic Mean, and *Sample Arithmetic Variance*/*Sample Arithmetic Standard Deviation*

Notice that the `summary` function gave the value of the *sample arithmetic mean*. You may also call *SAM* directly as well as the *Sample Arithemetic Variance* and *Sample Arithmetic Standard Deviation*. There are other means (e.g., *geometric* and *harmonic*) that you can call in `R` and you can program any others you wish.

```r
mean(demoOreo$Filling.Mass, na.rm = TRUE)
```

```
## [1] 4.8304
```

```r
var(demoOreo$Filling.Mass, na.rm = TRUE)
```

```
## [1] 2.751586
```

```r
sd(demoOreo$Filling.Mass, na.rm = TRUE)
```

```
## [1] 1.65879
```

Notice that in the above calls, we used the *entire* data set; that is, we did not parse out by the type of Oreo.

Sample Skewness and *Sample Kurtosis*

Sample Skewness and *Sample Kurtosis* are two statistics that are NOT part of base `R`. You will need to use another package to call them. The `psych` package as a way to call them as does the `e1071` package.

```r
e1071::skewness(demoOreo$Filling.Mass, na.rm = TRUE)
```

```
## [1] 0.02847018
```

```r
e1071::kurtosis(demoOreo$Filling.Mass, na.rm = TRUE)
```

```
## [1] -1.931979
```

Getting Multiple Statistics' Values at Once

The `psych` package has a great function, `describe`, that allows you to call a large number of statistics at one time and store the outputs in a list. You can then call individual elements out of that as necessary. This would be great for weaving values of statistics into your narrative; run the code chunk to save the output, then call the needed output with a bit of inline code.

```
outValues <- psych::describe(
  x = demoOreo$Filling.Mass,
  na.rm =  TRUE,
  interp = TRUE, # Use interpolation for Sample Median
  quant = c(0.25, 0.75), # Include Q1 and Q3 (and others)
  skew = TRUE, # Calculuate Skewness and Kurtosis
  ranges = TRUE, # Calculate Range
  IQR = TRUE, # Calculate IQR
)
outValues
```

```
##    vars  n mean   sd median trimmed  mad  min  max range skew kurtosis   se  IQR
## 1     1 60 4.83 1.66   4.57     4.8 2.17 2.29 7.21  4.92 0.03    -1.93 0.21 3.16
##    Q0.25 Q0.75
## 1   3.25  6.41
```

```
# Calling a single statistic's value
outValues$skew
```

```
## [1] 0.02847018
```

Summaries by Group

Some statistics will support model statements or a `by` parameter. However the `psych` package has the `describeBy` function that does the same thing as `describe` but for all levels of a factor.

```
outType <- psych::describeBy(
  x = demoOreo$Filling.Mass,
  group = demoOreo$Type,
  na.rm =  TRUE,
  interp = TRUE,
  quant = c(0.25, 0.75),
  skew = TRUE,
  ranges = TRUE,
  IQR = TRUE,
  digits = 3, # Round to thousandths place
  mat = TRUE # Store output as a matrix
)
outType
```

```
##    item       group1 vars  n  mean    sd median trimmed   mad   min   max range
## 11    1 Double Stuf    1 30 6.457 0.302  6.422   6.448 0.259 5.815 7.210 1.395
## 12    2     Regular    1 30 3.204 0.181  3.252   3.236 0.047 2.292 3.323 1.031
##      skew kurtosis    se   IQR Q0.25 Q0.75
## 11  0.323   -0.200 0.055 0.311 6.286 6.597
## 12 -4.218   18.425 0.033 0.074 3.199 3.273
```

```
# Calling a single group's value of a statistic
outType[2, "IQR"] # Gives the Regular Oreo's value of the IQR
```

```
## [1] 0.074
```

Statistical Inference

If you are using a CDA approach, then you will definitely be doing statistical inference. If you're using an EDA approach, you might use statistical inference to help you test out a potential model for testing with new data later on.

In either case, you MUST check out the assumptions of any inference method. Assumption explorations and decisions are part of your narrative. I recommend that discussing your assumptions should be done BEFORE you actually run and report the results of any test.

What follows here are examples of the code needed to run various tests and get various results. I'm not providing any assumption checking or interpretations at this time.

I recommend you use the `str` (structure) function on the output objects where test results are stored. This will help you see how to store the results and then call individual elements in your narrative.

In the following examples, I'm using the `demoOreo` data…however, what I do here should NOT be taken as indicative of what you should do to answer the SRQ in HW #1.1. In other words, you will need to think through what that actual SRQ asks you to do. Hint: you might need to think about transformations.

Parametric Shortcut: Student's and Welch's *t* tests

How to call parametric t tests (One Sample, Two Sample, and Paired) in base `R`.

```
tTestOutput <- t.test(
  Filling.Mass ~ Type,
  data = demoOreo,
  alternative = "two.sided", # Which type of alt. hyp. do you have
  mu = 0, # Parameter Value under the Null
  paired = FALSE, # Paired or Unpaired Data
  var.equal = FALSE, # T -> Student's Pooled, F -> Welch's
  conf.level = 0.98, # Confidence Level
  na.action = "na.omit"
)
tTestOutput
```

```
##
```

```
##  Welch Two Sample t-test
##
## data:  Filling.Mass by Type
## t = 50.569, df = 47.456, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 98 percent confidence interval:
##  3.098256 3.408011
## sample estimates:
## mean in group Double Stuf     mean in group Regular
##                  6.456967                  3.203833
```

```
# You can extract individual pieces from the output
tTestOutput$conf.int
```

```
## [1] 3.098256 3.408011
## attr(,"conf.level")
## [1] 0.98
```

Nonparametric Shortcut: Wilcoxon/Mann-Whitney Test

How to call One and Two Sample Wilcoxon/Mann Whitney tests in base `R`.

```
wilcoxonTestOutput <- wilcox.test(
  Filling.Mass ~ Type,
  data = demoOreo,
  alternative = "two.sided", # Which type of alt. hyp. do you have
  mu = 0,
  paired = FALSE,
  exact = FALSE, # Whether exact pvalue/conf. int. should be calculated
  correct = TRUE, # Continuity Correction
  conf.int = TRUE, # Give confidence interval
  conf.level = 0.98,
  na.action = "na.omit"
)
wilcoxonTestOutput
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  Filling.Mass by Type
## W = 900, p-value = 3.018e-11
## alternative hypothesis: true location shift is not equal to 0
## 98 percent confidence interval:
##  3.076971 3.337077
## sample estimates:
## difference in location
##                 3.19898
```

Permutation Test

A Permutation test takes the notion that group membership is arbitrary to heart (i.e., the null hypothesis). We can exchange members between the groups aribtrarily since we're assuming the null model says that group membership is arbitrary. You will have to make use of an additional package to perform a permutation test, such as the `perm` package.

```
permTestOutput <- perm::permTS(
  Filling.Mass ~ Type,
  data = demoOreo,
  alternative = "two.sided",
  exact = FALSE, # Whether exact pvalue/conf. int. should be calculated
  na.action = "na.omit"
)
permTestOutput
```

```
##
##  Permutation Test using Asymptotic Approximation
##
## data:  Filling.Mass by Type
## Z = 7.5955, p-value = 3.064e-14
## alternative hypothesis: true mean Type=Double Stuf - mean Type=Regular is not equal t
## sample estimates:
## mean Type=Double Stuf - mean Type=Regular
##                                   3.253133
```

Permutation Tests generally give you p-values rather than confidence intervals. Additionally, you have to watch out for output running off the page.

Bootstrapping

Bootstrapping involves re-sampling with replacement from your sample. Like permutation tests, you will need to use another package; `boot` is the best one.

You will need to think about the structure of your data carefully as well as define a function to use as your statistic.

```
ratioSAM <- function(df, w) {
  return(
    mean(df$Double * w) / mean(df$Regular * w)
  )
}

oreo <- unstack(demoOreo, Filling.Mass ~ Type)
names(oreo) <- c("Double", "Regular")
```

```r
bootObject <- boot::boot(
  data = oreo, # transformed data set
  statistic = ratioSAM, # Our custom statistic
  stype = "w", # Tell boot to use weights in our function
  R = 1000, # Number of bootstrap replications
)
bootObject
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot::boot(data = oreo, statistic = ratioSAM, R = 1000, stype = "w")
##
##
## Bootstrap Statistics :
##     original        bias    std. error
## t1* 2.015388 0.0004291455   0.0258661
```

```r
boot::boot.ci(
  bootObject,
  conf = 0.97,
  type = "all"
  )
```

```
## Warning in boot::boot.ci(bootObject, conf = 0.97, type = "all"): bootstrap
## variances needed for studentized intervals
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot::boot.ci(boot.out = bootObject, conf = 0.97, type = "all")
##
## Intervals :
## Level      Normal              Basic
## 97%   ( 1.959,  2.071 )   ( 1.953,  2.067 )
##
## Level     Percentile            BCa
## 97%   ( 1.963,  2.077 )   ( 1.968,  2.087 )
## Calculations and Intervals on Original Scale
## Some BCa intervals may be unstable
```

Effect Sizes

Effect sizes can be a bit tricky to produce. As we progress through the course, we will use a couple packages and scripts to help get the values. For HW #1.1, you'll have to be content with the following.

Cohen's *D* and Hedge's *G*

To get Cohen's D or Hedge's G (both members of the Distance Family), we will make use of the `psych` package.

```
cohenOut <- psych::cohen.d(
  x = demoOreo$Filling.Mass,
  group = demoOreo$Type
)

cohenOut$cohen.d
```

```
##          lower    effect     upper
## [1,] -16.62061 -13.28011 -9.713948
```

```
cohenOut$hedges.g
```

```
##     data
## -13.0569
```

Probability of Superiority

I have not been able to find a package that has a Probabiltiy of Superiority function. Here's some code that you may use.

```
PS<- function(d){ # d will be a value of Cohen's D
    ps <- pnorm(-sqrt(2)/2*as.numeric(d), mean = 0, sd = 1,
                lower.tail = FALSE)
    if (ps < 0.0001) { return("PS is essentially 0")}
    else { return(paste("PS =", round(ps, 4)))  } }
PS(cohenOut$cohen.d[1,"effect"])
```

```
## [1] "PS is essentially 0"
```

Final Caution

Be careful when displaying output in R Markdown. The system does not care if code/output runs beyond the page margins or off the page. Nor the does the system guard against breaking code, tables, output across pages, increasing the reading difficulty of your narrative. I've attempted to leave these issues in this document so that you could see them yourself.