Data Visualizations

Neil J. Hatfield

1/22/2021

Welcome to the latest installment of using R and RStudio. This guide will be looking at data visualizations. As you look through this guide, attempt to create the visualizations for yourself using the example code. Do this with both the demo data and a new data set (say from HW #1.1). Also see what element you can customize.

Getting Started

To get started we need to do two major tasks: load the necessary packages and load our data.

Load Packages

When you start a new session of R, one of the things that you should do is to load the extra packages that will bring new functions and new capabilities. If you don't load the necessary packages, your code might not work.

That being said, I'm going to make use of three key packages in this guide: `ggplot2`, `car`, and `lattice`. The `ggplot2` package will produce the most professional looking graphics, but can be more complicated for coding.

To load these packages, you'll want to make a code chunk with either of the following:

```r
library(ggplot2)
library(car)
library(lattice)
```

```r
packages <- c("tidyverse", "car", "lattice")
lapply(
  X = packages,
  FUN = library,
  character.only = TRUE
)
```

The `tidyverse` package will load `ggplot2` along with some other packages that are useful for data cleaning and manipulation such as the pipe, `%>%`. Either of the above methods work. I often use the second method as this lets me create a single vector with all of the packages I want (and I can quickly add more packages to the list) and then apply the `library` function to all of them.

Load Data

For this guide I'm going to use the class demo Oreo data set. That way you can see an example that will translate well to your Homework #1.1.

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

I want to point out an important feature of the code above: you'll notice that the first bit of the code was `oreoData`. This will become the name of the object in R. If I ever want to use this data in R, I will need to include the name `oreoData`. Thus, you'll want to use a name that is 1) meaningful and 2) unique. If I were to run `oreoData <- 2` right after the above code, I'll replace the data with the number 2.

To help you keep track, I'll often run individual chunks as I go by pressing the play button that appears in the upper right corner of each chunk. This will load things into my Environment and allow me to see what the chunk will produce when I knit the document.

The second thing that I want to point out is the assignment operator `<-`. This two character symbol says "take the output of the what's on the right and call that output what's on the left. In other words,"assign the output the name I'm pointing to". This is the preferred method for creating objects in R rather than using the equals sign.

Making Data Visualizations

Now that we have loaded both the packages and our data, we can turn to making data visualizations. If you can imagine the data visualization, there is a way to make that visualization in R. (Provided you have the patience and perseverance to figure out how.)

I will point out that what I have below are just a few of the many different types of data visualizations you can create using R. And for each one, there are many different ways you can go about creating the visualization. You are by no means bound to the methods I use here. However, the methods that I demonstrate create professional looking graphics. Thus, if you are preparing a report, I would recommend that you try to emulate these. If you are just wanting to make a quick visualization for yourself, you can use other methods.

Before we get too far into making visualizations, now would be a good time to check the structure of our data.

```
str(oreoData)
```

```
## 'data.frame':    60 obs. of  2 variables:
##  $ Filling.Mass: num  3.27 3.12 3.15 3.27 3.24 ...
##  $ Type        : chr  "Regular" "Regular" "Regular" "Regular" ...
```

Generally speaking, **DO NOT** include the output of a `str` call in your analysis reports. This information is more or less just for you as you start EDA. In this case, I notice that I have two attributes; Filling Mass which is numeric (which I know is in grams) (with variable `Filling.Mass`) and type which is character (`chr`; with variable `Type`). Please note that R is case sensitive; thus, `type` and `Type` are treated as two *different* objects.

The other thing that don't want to include in your analysis reports are raw code. You'll notice that in these guides, I'm including the raw code chunks. This is so you can see the codes that makes the resulting visualization. Raw code has only one viable place in a report–an appendix at the end. Be sure that your reports do not include raw code.

A Quick Bit of Data Cleaning

For ANOVA, we often want one of our attributes to be treated as a factor by R. From the output of the `str` call, R isn't thinking about either of the attributes as a factor. In this case, we want `Type` to be a factor. Thus, let's quickly tell R that we want `Type` to be a factor:

```
oreoData <- oreoData %>%
  dplyr::mutate(
    dplyr::across(where(is_character), as_factor)
  )
```

The above code says to do the following:

1) Grab the data frame, the `%>%` reads as "and then"
2) mutate (change) the data set by
3) looking across all of the columns to see where the character attributes are; change those character attributes to factors
4) save the altered data as `oreoData`.

You'll notice that `mutate` and `across` have `dplyr::` in front of them. This is a way to ensure that you are calling these functions from the `dplyr` package. If you know that you have loaded the package, you can leave `dplyr::` out of your code. I often use the package signifier to demonstrate what packages I'm calling different functions out of.

Check for yourself to see whether if `Type` has a new data type (i.e., no longer `chr`).

Let's now begin to make some visualizations.

Histograms and Bar Charts

Some people get very touchy about when you call a histogram a bar chart or a bar chart a histogram. I'm not one of them. The crux of the distinction (histograms are for numeric data and and bar charts are categorical data) has **nothing** to do with the information presented in the visualization nor how a person reads them. Thus, histograms are bar charts and bar charts are histograms. Sadly, software developers have enshrined the distinction without a difference into the code we use to make these kinds of visualizations.

```r
oreoData %>%
  ggplot2::ggplot(
    mapping = aes(x = Filling.Mass)
  ) +
  ggplot2::geom_histogram(
    binwidth = 0.5,
    boundary = 0,
    closed = "left",
    na.rm = TRUE,
    col = "black",
    fill = "blue"
  ) +
  ggplot2::theme_bw() +
  xlab("Filling mass (g)") +
  ylab("Frequency")
```

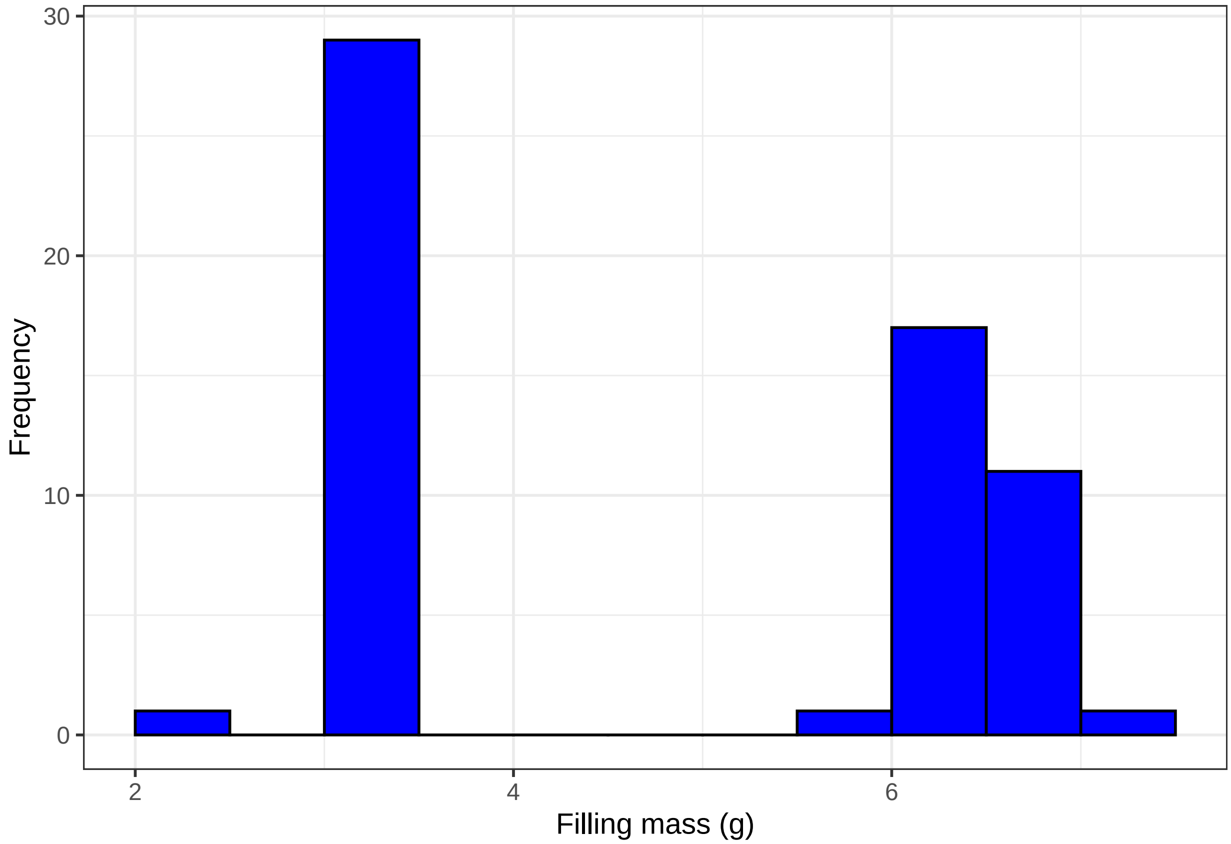

Figure 1: Histogram of Oreo Créme Filling Mass

Since we aren't storing this plot in an object, we don't need to use the assignment operator. Additionally, you don't have to use the pipe (`%>%`) as I'll demonstrate later. The first important portion is `ggplot2::ggplot`. This function is what initiates the entire visualization. Since I used the pipe to send a data frame into the function, I did not need to include the `data` argument. The other important element is the `mapping` argument. You'll use the aesthetic function `aes`, to say which variables from the data frame go where: `x` for horizontal axis, `y` for vertical, and more. The `mapping` exists as part of many other function calls (e.g., `geom_histogram`) and will get passed along to them along with the data.

This brings us to `ggplot2::geom_histogram` which says that we want to make a histogram. The arguments here are important for you think about.

- `binwidth`: how wide do you want the bins? You might need to play around with this value OR you can use the Freedman-Diaconis rule by using `binwidth = function(x){ifelse(IQR(x) == 0, 0.1, 2 * IQR(x) / (length(x)^(1/3)))}`.
    - If I know that I'm going to be making multiple histograms and that I'll be using the Freedman-Diaconis rule, I'll define a function in my file once (`freedmanDiaconis <- function(x){ifelse(IQR(x) == 0, 0.1, 2 * IQR(x) / (length(x)^(1/3)))}`) and then in my histograms I use `binwidth = freedmanDiaconis`.
- `boundry`: this is an optional argument and will set whatever value you put here to be the edge of a bin and work from there; useful to making nice looking visualizations.
- `closed`: Most people are familiar with (and except) histograms that include the value on the left edge in the bar, but not the value on the right. This argument allows you match that expectation. The default is the exact opposite (that is, the right value is included but not the left).
- `na.rm`: by saying `TRUE` you are saying "remove the missing values SILENTLY". The R will remove any missing values (`NA`) as the plot gets built but unless you say `na.rm = TRUE` R will be noisy and print messages in your report that should not appear.
- `col`: determines the color of the piping/edges of the bars.
- `fill`: determines the color of the bars.

The last three lines set a default theme (background color, font sizes, etc.) and the labels for the axes.

Here is the same data with a *relative* frequency histogram.

```r
ggplot2::ggplot(
  data = oreoData,
  mapping = aes(x = Filling.Mass)
) +
  ggplot2::geom_histogram(
    mapping = aes(y = stat(count) / sum(count)),
    binwidth = freedmanDiaconis,
    boundary = 0,
    closed = "left",
    na.rm = TRUE,
    col = "black",
    fill = "blue"
  ) +
  ggplot2::theme_bw() +
  xlab("Filling mass (g)") +
  ylab("Rel. frequency") +
  ggplot2::scale_y_continuous(labels = scales::percent)
```

Here I didn't pipe the data set into `ggplot` so I had to include `data = oreoData`. Additionally, the `mapping = aes(y = stat(count) / sum(count))` line is what will create relative frequencies. I also used the Freedman Diaconis rules for bin width. The last line `scale_y_continuous(labels = scales::percent)` calls the `percent` function from the `scales` package to make the labels on the vertical axis look nice. Be sure that you have the `scales` package installed first.

Let's now make a similar visualization for `Type`.

```r
ggplot2::ggplot(
  data = oreoData,
  mapping = aes(x = Type)
) +
  ggplot2::geom_bar(
    na.rm = TRUE,
    col = "black",
    fill = "blue"
  ) +
  ggplot2::theme_bw() +
  xlab("Type of Oreo") +
  ylab("Frequency")
```

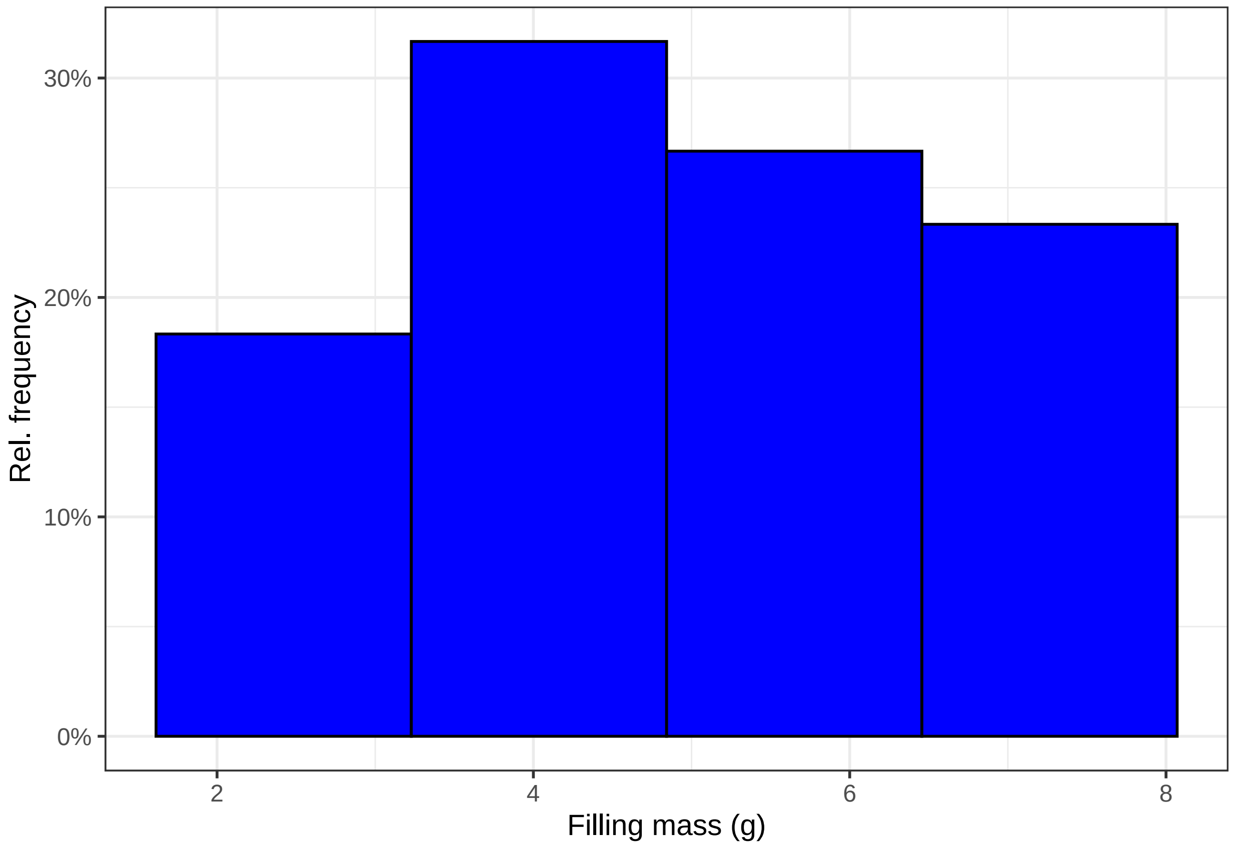

Figure 2: Histogram of Oreo Créme Filling Mass

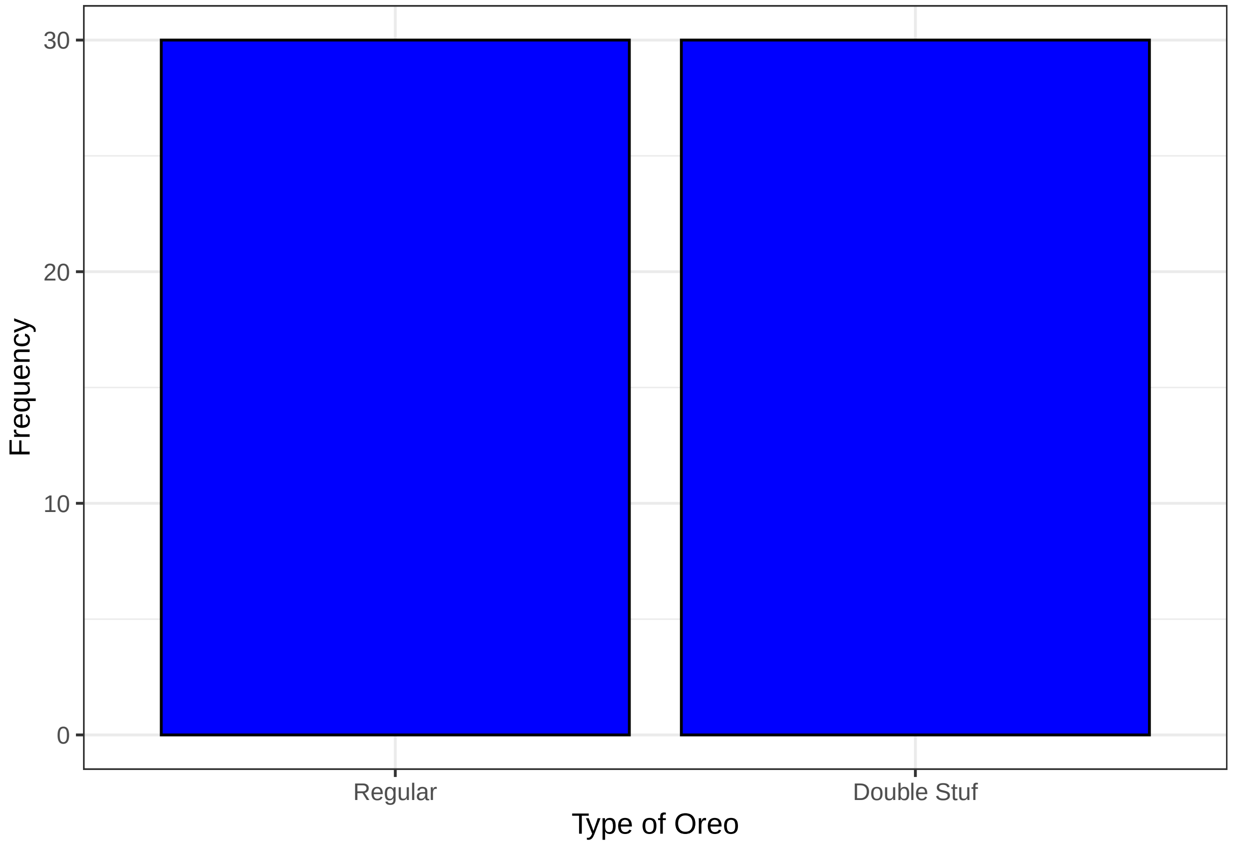

Figure 3: Break down of Oreo Types

Pretty much the only thing that changed was that instead of `geom_histogram` we used `geom_bar`. Within `ggplot2` the `geom_*` functions control the various geometries of data visualizations.

Density Plots

A density plots are a smoothed version of a histogram that can help us think about the underlying population rather than the sample in front of us. Making one in `ggplot2` is fairly straight forward.

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

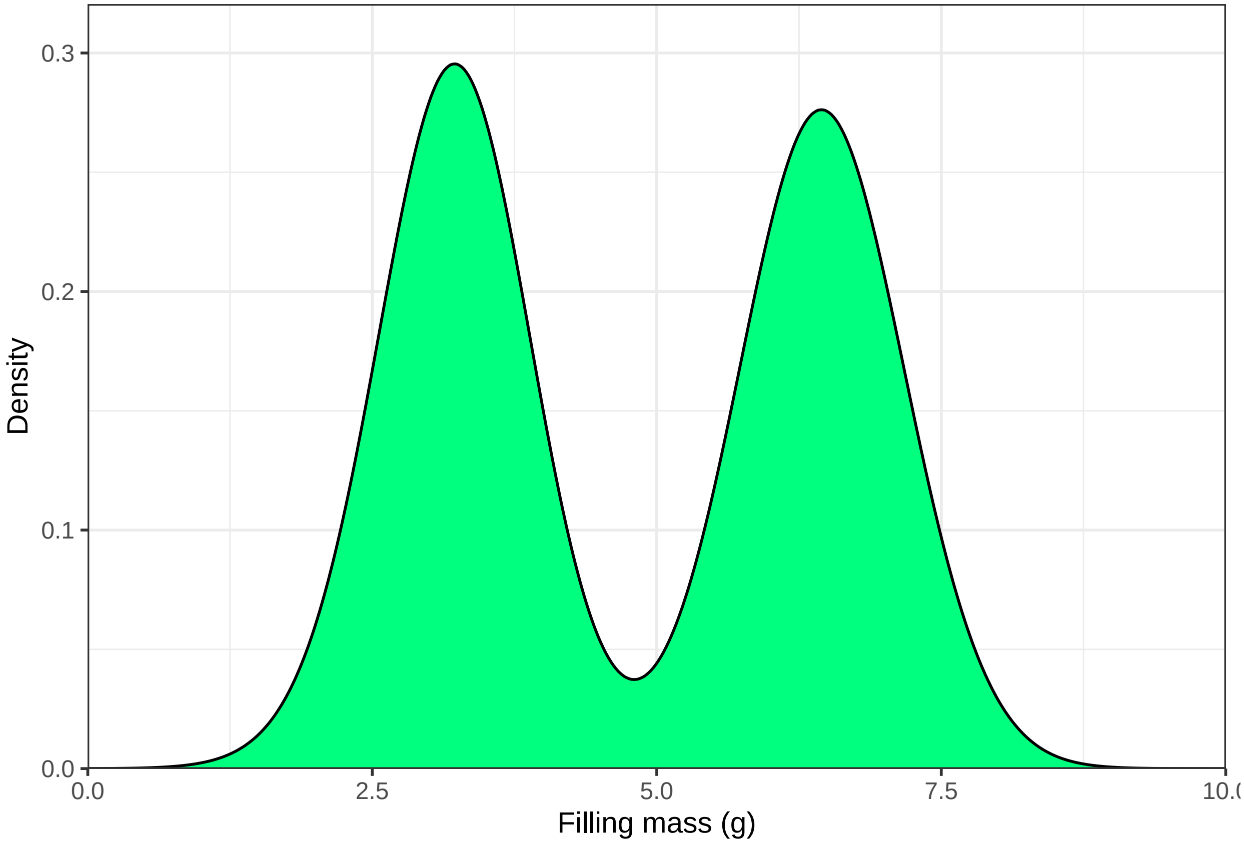

Figure 4: ggplot2 density plot of Oreo créme filling mass

Again, many of the arguments do the same thing as for the histograms/bar charts. Notice that instead of `col` you can use the whole word `color` (you can also use British spellings, `colour`) for the same effect. We changed which geometry we used to `geom_density` to get the density plot.

I also want to draw your attention to the last few lines with `scale_x/y_continuous`. This is where you can manipulate the scales of the plot. If notice in the histograms, the bars do not continue all of the way to the bottom of the plot, rather, they float above. The `expand` argument is what you need to manipulate with the `expansion` function. The `mult` is a multiplicative scaling while `add` simply adds values on. If you give these a single value, then that value is used for both sides (left/right or bottom/top). If you give two values, such as `c(0, 0.025)` the first value listed will go to the left/bottom and the second value to right/top. The `limits` argument allows you to control to what extend the plot goes for each axis. These limits will then have the expansion applied to them (if any). If you omit `limits`, then R will figure out decent limits for your data.

A second way to create a density plot is with the `lattice` package.

```
lattice::densityplot(
  x = ~ Filling.Mass,
  data = oreoData,
  na.rm = TRUE,
  xlab = "Filling mass (g)",
  col = "black"
)
```

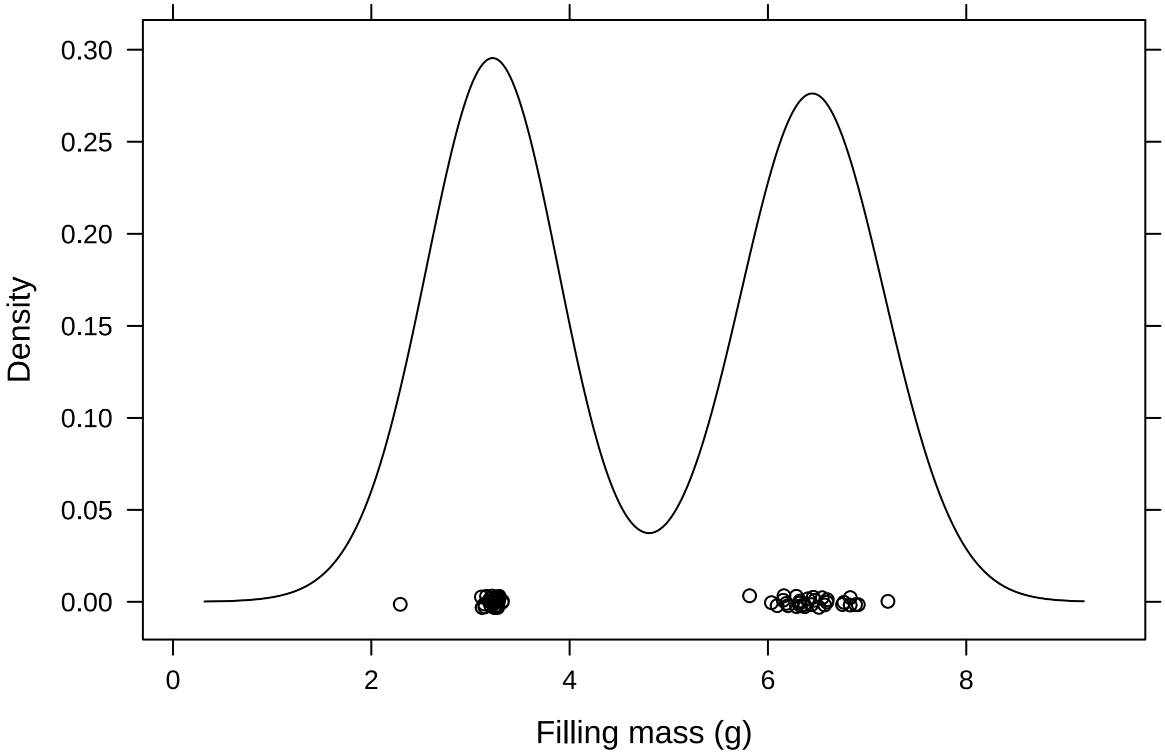

Figure 5: lattice density plot of créme filling mass

The `lattice` package's density plot includes all of the data points (the circles) along the plot's bottom. There is a new argument here: `x = ~ Filling.Mass`. The tilde, ~, is R's formula notation. We will be using throughout the semester. Using the formula notation `lattice`'s is similar to `ggplot2`'s `mapping` argument.

It is possible to use `ggplot2` to create a density plot like the `lattice` package. Think of this as a challenge for you to figure out how to do. Hint: you might need to use two geometries.

Box plots

As you might guess, making a box plot in `gglot2` will involve using `geom_boxplot`.

```
ggplot2::ggplot(
  data = oreoData,
  mapping = aes(x = Type, y = Filling.Mass, color = Type)
) +
  ggplot2::geom_boxplot(
    na.rm = TRUE
  ) +
```

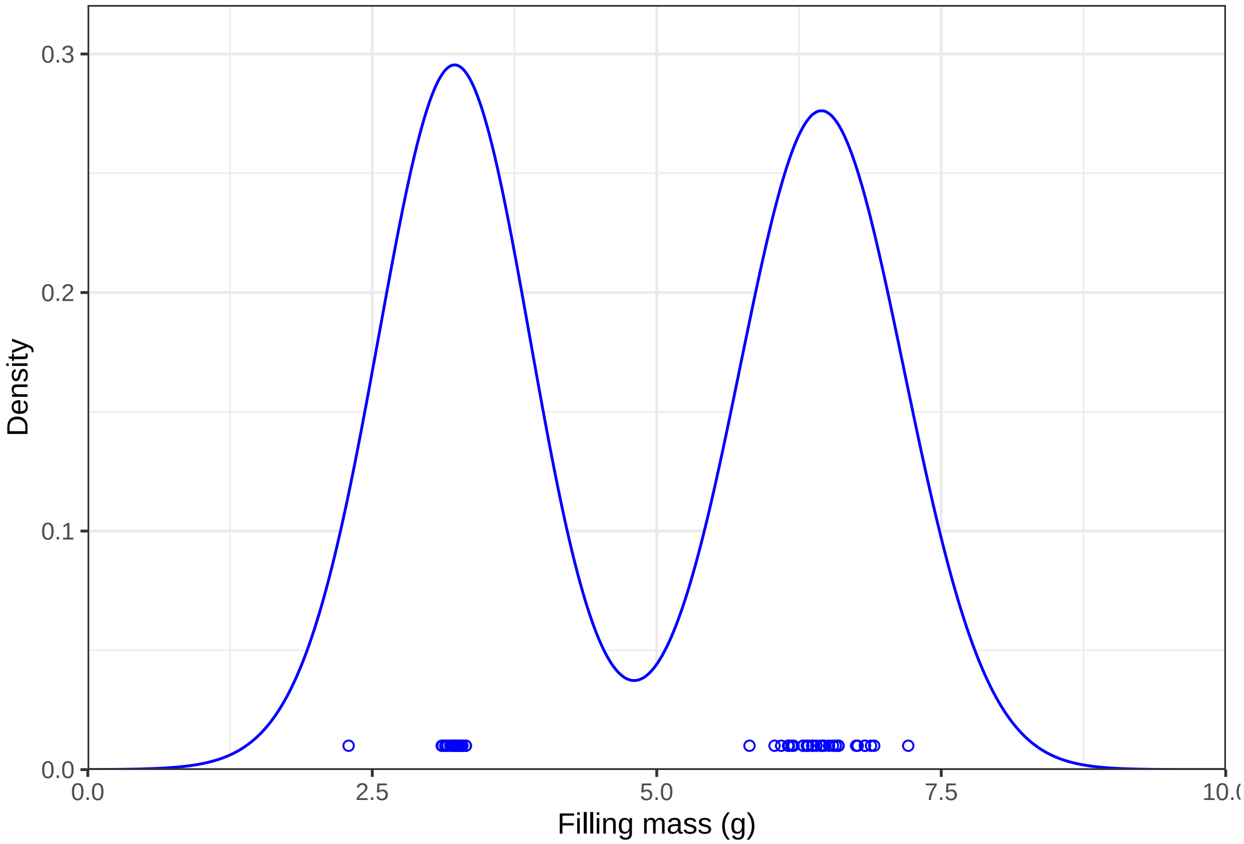

Figure 6: Challenge density plot

```
  ggplot2::theme_bw() +
  xlab(NULL) +
  ylab("Filling mass (g)") +
  theme(
    legend.position = "bottom"
  )
```

Notice that this time we defined three aesthetics: the horizontal axis (`x`) and `color` using the Type attribute and the vertical axis (`y`) with créme filling mass. When you define the color aesthetic, R will find a color scheme for to use and create a legend to include in the plot. You can alter the color scheme used. You can also position the legend with the `legend.position` argument of the `theme` call. Notice that `xlab(NULL)` does not turn off the horizontal axis; rather just the label disappears.

If you don't have any groups for the box plot, then you can omit the extra aesthetics. For vertical box plots, assign the attribute to `y`; for horizontal box plots, use `x` instead.

I should point out that `ggplot2` did NOT produce a true box plot. Rather R created an *outlier* box plot. The whiskers only extend to 1.5 times the value of the *IQR* from the first and third quariltes. Any data values beyond that appear as solid dots. A true box plot has the whiskers extend to the values of the *sample minimum* and *sample maximum*.

Strip Charts

Strip charts (or dot plots) are a type of scatter plot where instead of using two numeric attributes you use one numeric and one categorical. There are two ways to make strip charts: one using base R and the other using `ggplot2`.

The base R method is pretty easy to do:

```
stripchart(
  x = Filling.Mass ~ Type,
  data = oreoData,
  pch = 20,
  vertical = TRUE,
  xlab = "Type",
```

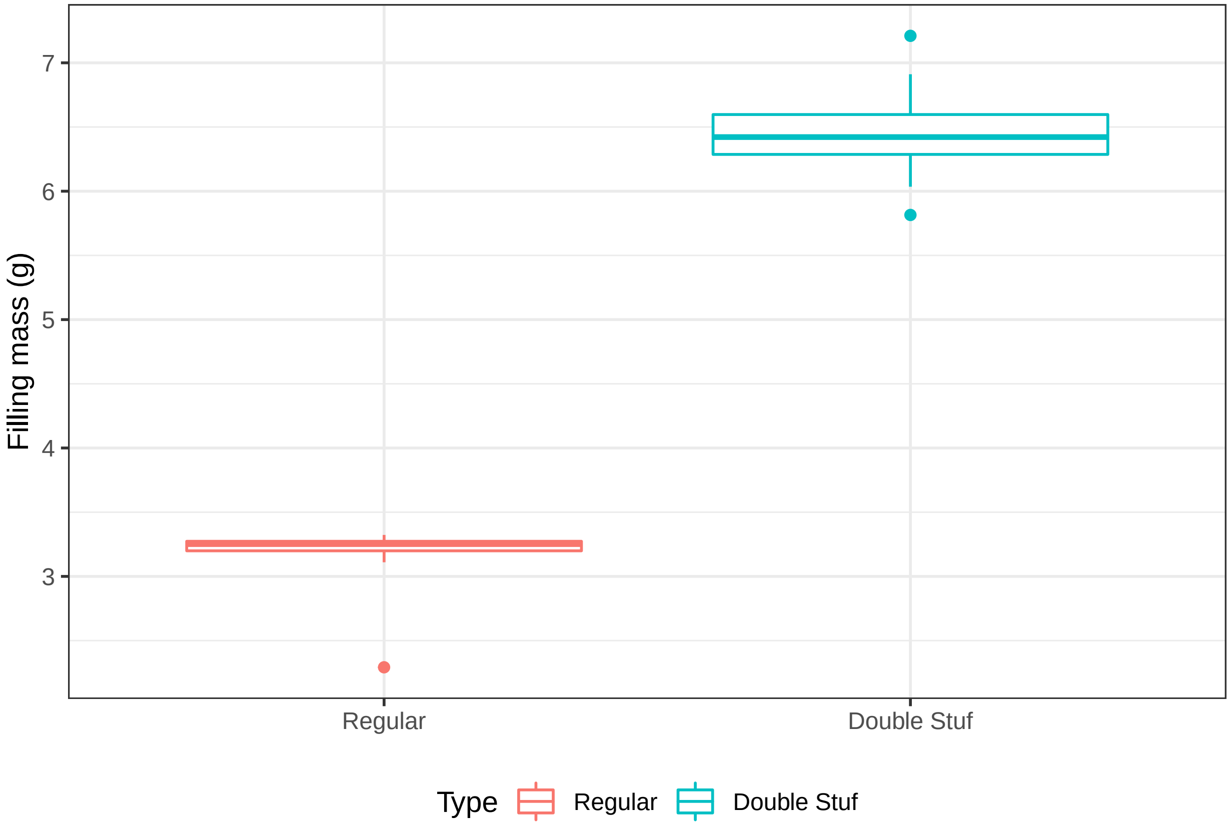

Figure 7: Box plot of créme filling mass by type

```
  ylab = "Filling mass (g)"
)
```

Notice that we used R's formula notation `Filling.Mass ~ Type`. The ordering is important: `Response ~ Factor` / `Numeric ~ ModelTerms`. The other new arguments here are `pch` which sets the shape for plotting the points. To see a list type `?points` in your R console. I typically use either 19 or 20.

To do the same thing in `ggplot2`:

```
ggplot2::ggplot(
  data = oreoData,
  mapping = aes(x = Type, y = Filling.Mass, fill = Type)
) +
  ggplot2::geom_dotplot(
    binaxis = "y",
    binwidth = 0.15,
    binpositions = "bygroup",
    stackdir = "centerwhole"
  ) +
  ggplot2::theme_bw() +
  xlab("Type") +
  ylab("Filling mass (g)")
```

Notice that I used the `fill` aesthetic with the Type attribute. Within `geom_dotplot` there are a couple of arguments we need to talk about.

- `binaxis`: states which axis R should try to create bins for; this should be whichever axis you place the numerical data on.
- `binwidth`: this value controls how wide the bins are and to a certain extent the size of the dots. If you omit this argument, `binwidth = NULL`, you might get a warning message. You can either ignore the message (and suppress the printing of the message in your report) OR you can try various values to see what works well with your data.
- `binpositions`: generally if you are displaying some type of group like I did you'll want to bin the values separately by

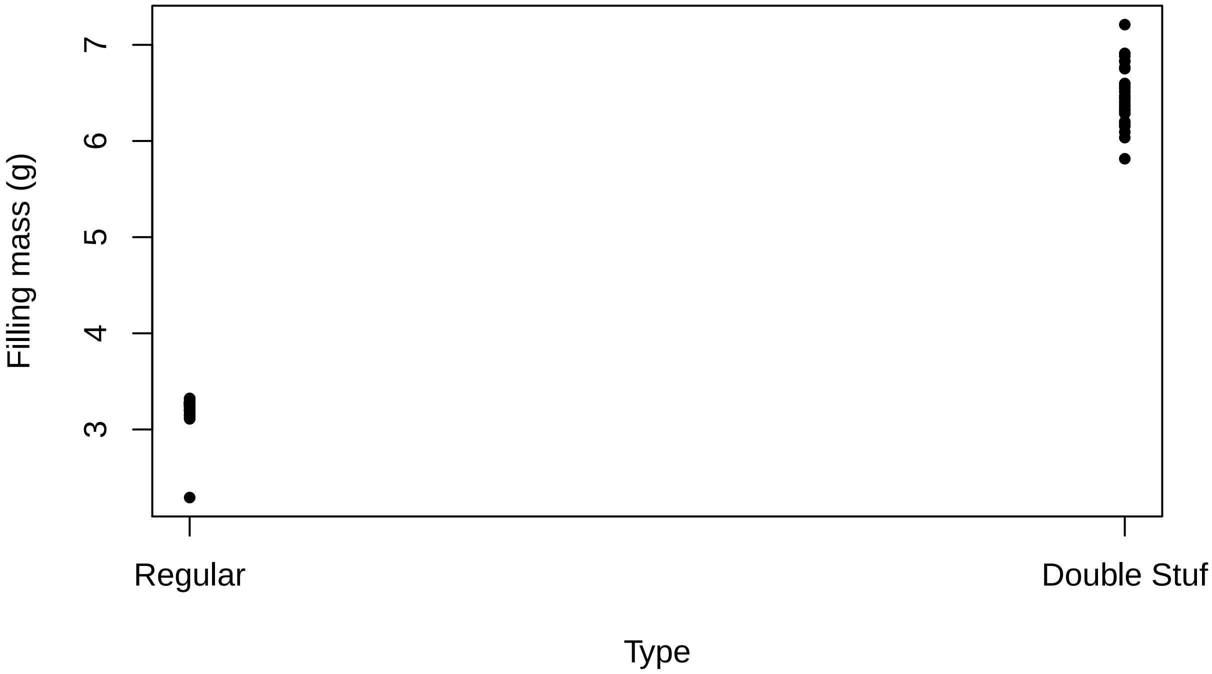

Figure 8: Strip chart for the Oreo data using base R

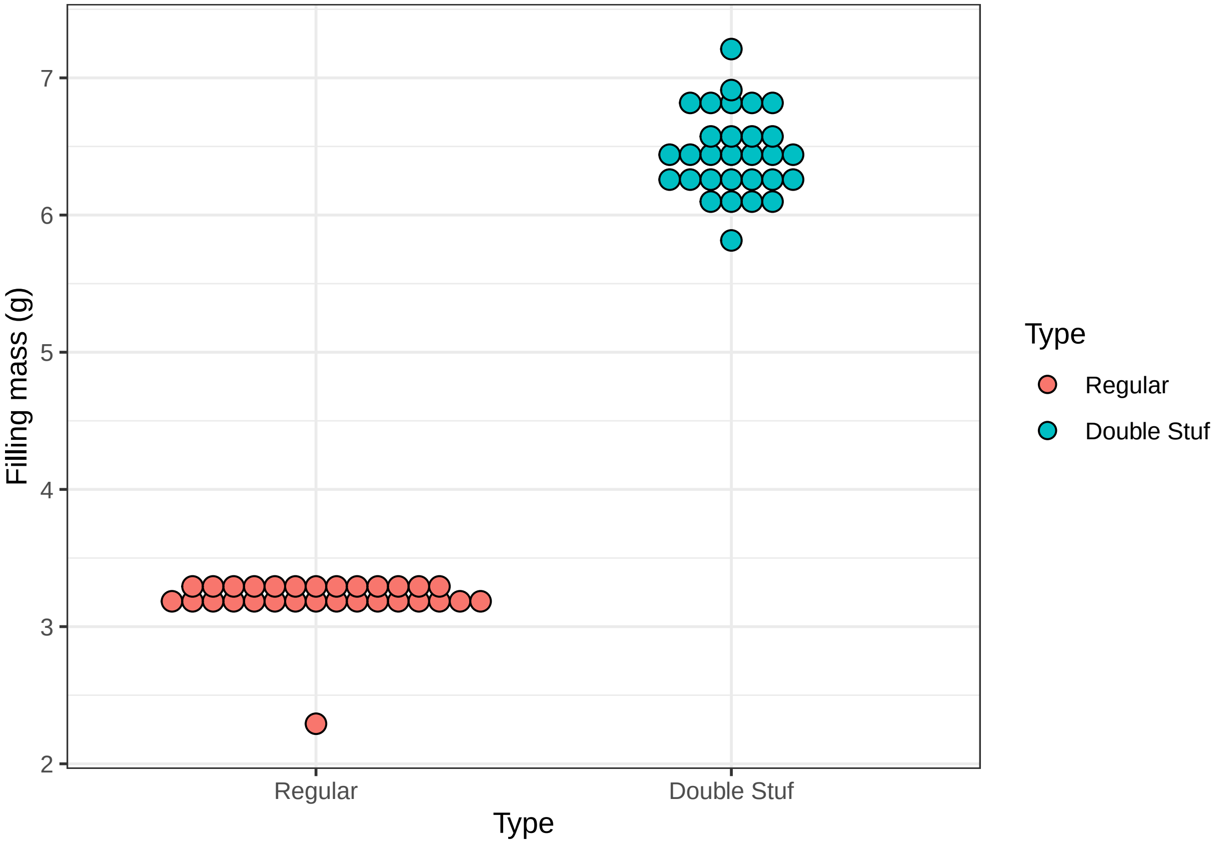

Figure 9: Strip chart for the Oreo data using ggplot2

each group.

- `stackdir`: the direction you want the dots to stack. The most common is `"up"` which will make the dots stack on top of each other (or to the right depending on orientation). Another method is `"center"` (or `"centerwhole"`) which stakes the dots in both directions from the center point.

QQ Plots

Quantile-Quantile (QQ) plots are a useful tool to see whether or not your data follow a particular distribution. Much like density plots there are two ways you can create them; one using the `car` package and one (more complicated) using `ggplot2`.

```
car::qqPlot(
  formula = ~ Filling.Mass,
  data = oreoData,
  distribution = "norm",
  envelope = 0.90,
  id = FALSE,
  pch = 19,
  ylab = "Empirical quantiles"
)
```

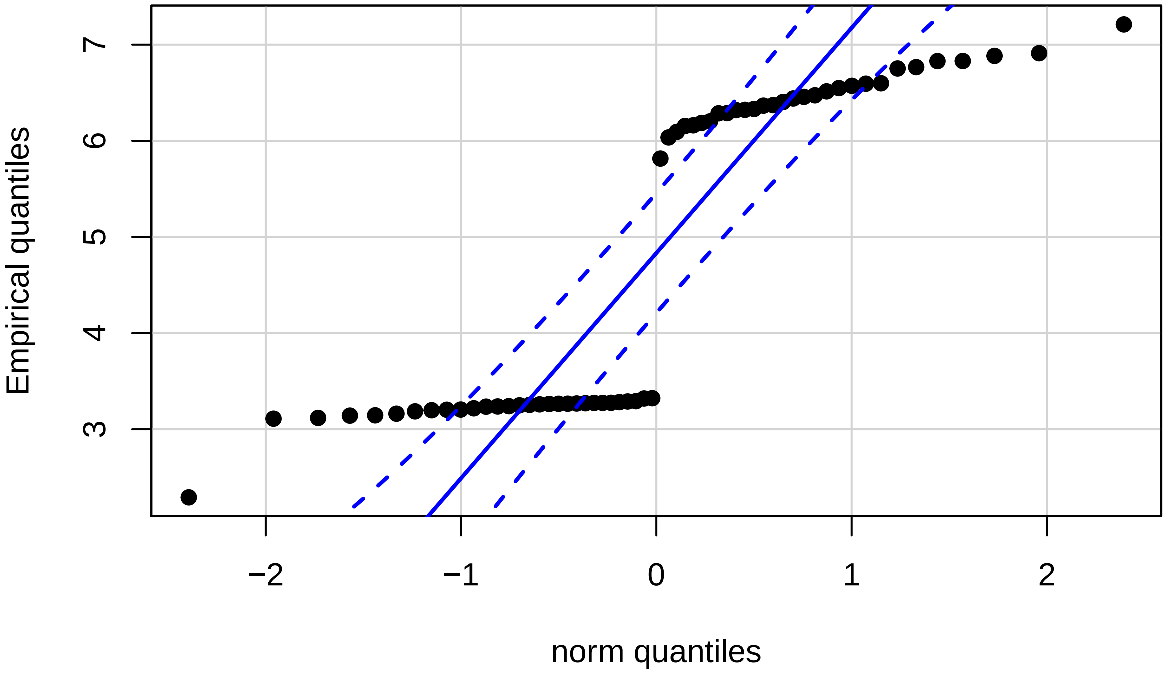

Figure 10: QQ plot for Oreo créme filling mass using car

The `car` package's method produces a scatter plot of the empirical and theoretical quantiles for each point as well as plotting the line of perfect match (solid blue) as well as a confidence envelope (dashed blue lines).

Here the key arguments are the `formula` argument to specify the attribute's variable you want to explore, `data` to set which data set to use, `distribution` for which named distribution you want to reference (`"norm"` will give you the "normal" or Gaussian distribution), and `envelope` is where you specify the confidence level (if you don't want an envelope you can use `envelope = FALSE` to turn this off).

The `id` argument when set to `TRUE` will identify the two points with the most extreme vertical values. Often times these are not useful as the labels are the row indices. If you want to label them, set `id = TRUE` and store the plot in an object (i.e, `a <- car::qqPlot(...)`). The visualization will still get displayed but the row indices will not get printed out of context in your report.

To create a similar plot in `ggplot2`, you'll need to do the following:

```
ggplot2::ggplot(
  data = oreoData,
  mapping = aes(sample = Filling.Mass)
) +
  ggplot2::geom_qq(
    na.rm = TRUE,
    distribution = stats::qnorm,
    dparams = list()
  ) +
  ggplot2::geom_qq_line(
    na.rm = TRUE,
    distribution = stats::qnorm,
    dparams = list(),
    color = "blue"
  ) +
  ggplot2::theme_bw() +
  xlab("Theoretical quantiles") +
  ylab("Empirical quantiles")
```

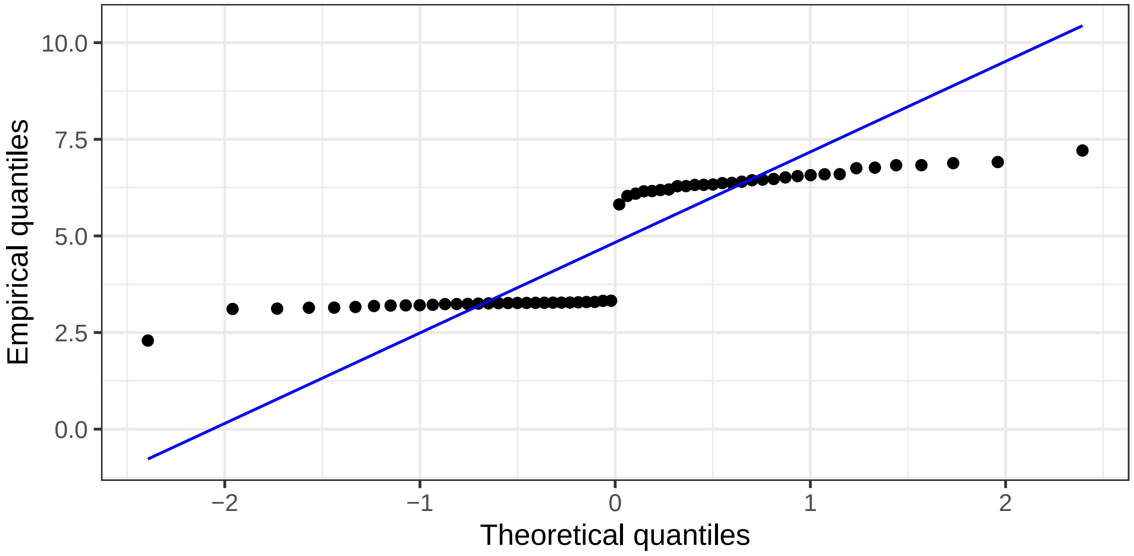

Figure 11: QQ plot for Oreo créme filling mass using ggplot2

You can add a confidence envelope through some data manipulation. Think of this as another challenge that you can choose to take up.

Notice that we used two geometries: `geom_qq` to plot the points and `geom_qq_line` to graph the line. For both of these we needed to define the aesthetic `sample` to house our data values we wanted to test. In addition, we need to specify which named distribution we're going to use as the theoretical reference. The `distribution` argument allows us to set this using the quantile functions built into the `stats` package that comes with all versions of R. You can see a list of the possible named distributions in the `stats` package help documentation (jump to the Q's).

The `dparams` argument is where you can specify particular parameters. Notice that we left ours as `list()`. This says to use the defaults for the named distribution. In our case, this would be an Expected Value of 1 and a Standard Deviation of 1. Suppose that you wanted to test whether data followed a normal/Gaussian distribution with Expected Value of 50 and Standard Deviation 5, you would then set `dparams = list(mu = 50, sd = 5)`. R would then use that distribution to make the plot.

Figure Options in R Markdown

There are a variety of options that you can apply to your data visualizations in R Markdown. If you aren't using R Markdown, your visualizations will appear in the Plots window in the lower-right of RStudio. You can click the Export button and copy the plot to your clipboard or save as a file. You can then put that picture into Word or Google Docs. You will then manipulate the following aspects there.

Basic Format

All of the figure options in R Markdown will need to be listed in the chunk options for the code chunk that creates the visualization.

When you create a chunk, you should have a some text inside of a set of curly braces immediately after the opening three graves. These are the chunk options. The first thing should be a lower case r to set the language, followed by a space, and then the *unique* name you're going to give that chunk. You will then place a comma after the name and begin listing the options you want. Here are the options for the very first histogram I created in this guide: `{r histEx1, echo=TRUE, eval=TRUE, fig.cap="Histogram of Oreo Créme Filling Mass"}`.

Both `echo` and `eval` don't deal with the visualization, just whether to print the code (`echo`) and run the code (`eval`). You can omit them; the homework template has these predefined for you. However, `fig.cap` is a figure option. If you wanted to include more figure options, you just keep adding to the list: `{r histEx1, fig.align='center', fig.height=3, fig.width=6, fig.cap="QQ plot for Oreo créme filling mass using ggplot2", fig.pos='h'}`. The order does not matter. There are many different ones that you can use but I'm going to focus on what I feel are the most useful ones.

Alignment

Alignment refers to whether the visualization appears to the left, center, or right of the page horizontally. Almost uniformly, data visualizations should appear centered. To do this, you would need to set `fig.align='center'` in each code chunk where a visualization gets made. However, you can also set this as a standard for your entire document:

```
knitr::opts_chunk$set(
  echo = FALSE,
  warning = FALSE,
  message = FALSE,
  fig.align = "center"
)
```

The above default are part of the homework template and will make all visualizations centered for you.

Size

There are two aspects to size for a visualization: height and width. All visualizations (including any legends) will be inside of a rectangle hence these two dimensions. The `fig.height` will control how tall this rectangle is while `fig.width` will control how wide. Both of these options are expecting a single number which is understood to be a number of *inches*. Thus, `fig.height=2, fig.width=4` will create a visualization that is 2 inches tall and 4 inches wide.

Caption

The `fig.cap` option adds a caption to your visualization AND will put "Figure #" out in front. This is consistent with many publication standards. If you omit the `fig.cap` option, then your visualization will not get a label. In such cases, be sure to add a title to your visualizations.

Your caption should appear in quotation marks just after the equals sign: `fig.cap="QQ plot for Oreo créme filling mass using ggplot2"`. Captions should describe the visualization by stating what kind of visualization and of what data. Try not to be overly wordy with your captions.

Referencing

One of the most useful aspects about using `fig.cap` is that your figures become numbered and referenceable. Have you ever read something and you come across something like "As shown in Figure 3.1…"? This is an example of referencing a figure. The computer figures out the appropriate number for you…provided you've set up a caption for the visualization.

In R Markdown, you need to have two things first: you have to name your code chunk and provide something as `fig.cap`. Once you have those things, you can that visualization. Start by typing the word "Figure" and then`\ref{fig:chunkName}`. When you knit your file, R will replace the ref code with the appropriate value. For example, if I were to type "Figure `\ref{fig:histEx1}`", I will get Figure @ref(fig:histEx1)—the reference to the first histogram.

You can add captions, auto numbered labels, and cross references in programs like Word. From the Insert menu select caption to set up what you want. To add a reference to a figure, select Cross-reference from the Insert menu, select the appropriate type and then the correct figure. Do set the cross reference to only include the label and number, not the whole caption.

Position

The last option I'm going to mention is one of the trickier ones to use; `fig.pos` *attempts* to control of the location of the visualization. When you knit the document, R will attempt to find the optimal location for the visualization. You can attempt to override this by setting the value of `fig.pos`. For example, `fig.pos='h'` says "hey, I want the visualization the appear"here" in my document. R will attempt to do that but if your figure is too large for the space at that place, your visualization will be moved (typically, the top of the next page).

Final Remarks

Hopefully as you've gone through this guide, you have a better idea of how to make just a few of the many different data visualizations possible in R. Further, I hope that you're getting a better idea of how to use R Markdown to write code to your advantage. Practicing these examples is the best way to improve your fluency. For some additional datasets to work with, feel free to explore the following data sets:

- Motor Trend Car Road Tests—access with `data(mtcars)`
- Weight versus Age of Chicks on Different Diets—access with `data(ChickWeight)`
- Effectiveness of Insect Sprays—access with `data(InsectSprays)`
- The Iris data set—access with `data(iris)`
- Palmer Penguin Data—install the `palmerpenguins` package first, then access with `palmerpenguins::penguins`

This concludes this guide to data visualizations in R/RStudio.