Week 12 Walkthrough

A crash course on the *grammar of graphics*

Kris Wilson

March 28, 2025

Wait, what's that?

An author? These walkthroughs have *authors*?

That's right! These walkthroughs are indeed not AI-generated… to my knowledge, anyways. While I can't capture the same essence created by ~~our lovely mystery man~~ James, I can certainly try!

Just kidding. My imitation would be poor. Before we begin, here's a brief list of things that'll be covered in this walkthrough:

Learning Objectives:

- learn what the *heck* `gg` stands for in `ggplot` (and how it works)
- learn common one- and two-variable plots, including but not limited to the following: bar graphs, box plots, histograms, scatterplots
- customize plots: axis labels, titles, change color of plots
- this walkthrough will *not* be covering faceting or legends. While legends are mentioned, there won't be an explicit example of customizing a legend. Without further ado…

ggplot!

The "gg" in `ggplot` stands for "Grammar of Graphics". In the same way that grammar defines the regular structures and composition of a language, grammar of graphics outlines a framework to structure statistical graphics!

The idea of `ggplot` is that instead of being limited to a predefined set of charts and plots, you can plot just about anything! Much like how a painter starts with a canvas, then paints layers on top of one another to produce something beautiful; `ggplot` works by stacking layers on top of a canvas. We use `+` to accomplish this.

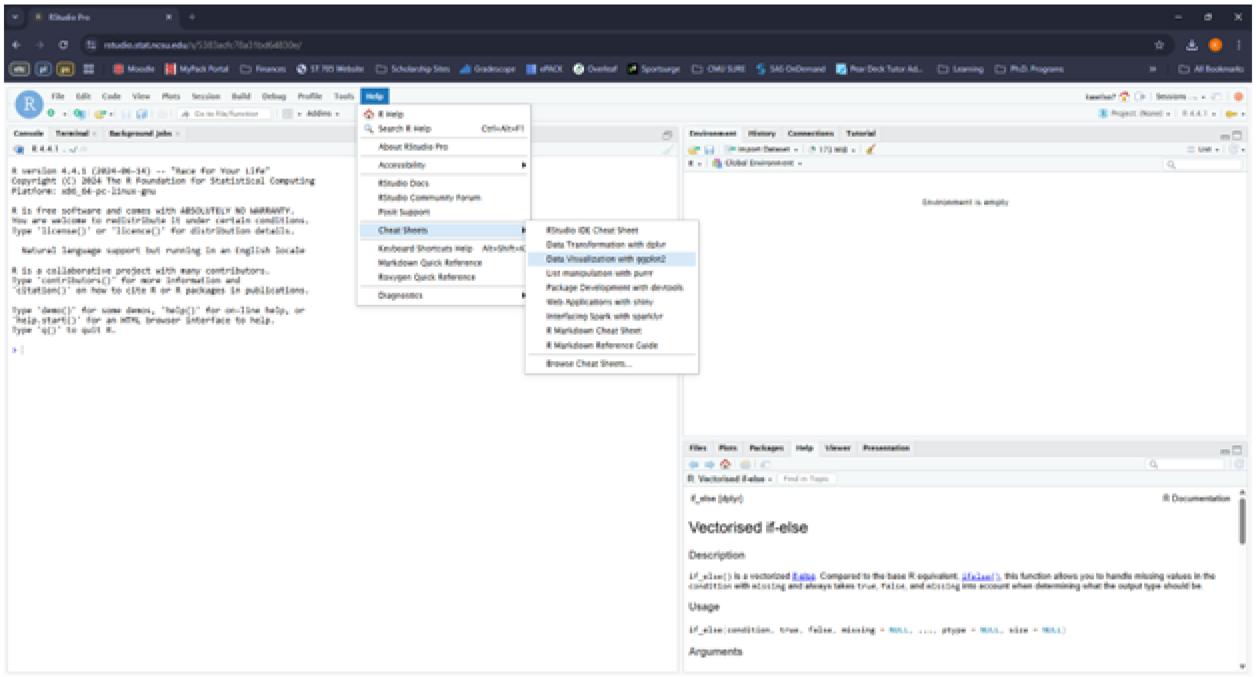
The plus operator '+' works exactly the same as our handy dandy pipe operator `%>%`. There's also the native pipe operator `|>` that I prefer to use, but they're exactly the same, I promise. With `ggplot`, though, we use `+`

The layers are as follows:

- **Data**: Every plot needs data! `ggplot` likes *tidy* data, which is usually just in the form of a data frame. We create this layer, this *canvas*, with the **`ggplot()`** function. This stores the data to be used later in other parts! Can't paint without a canvas!
  - (I mean, you could paint on the walls or something, although *technically* the wall would be the canvas then, and you'd be down *a lot* of money when you're inevitably fined.)
- **Mapping**: the next layer is called the mapping, but I prefer the fancy term **aesthetics**. This is short for "aesthetic attributes", which works as a dictionary to translate the data into the graphics system that creates our output!
  - This is where the customization *really* starts to take shape. If we wanted to create one-variable plots, we'd only specify, well, *one* variable!
- **Layers**: the heart of any graphic is the layers. This takes the mapped data and represents it as something we can interpret. You can think of the **mapping** as the specific aspects of the data we want to select: the variable(s) we want to look at, if we want to color them (perhaps by another variable!), things like that. Layers have three parts:
  1. **geometry** `geom_()`: this determines how the data is displayed. This includes points (e.g., dotplots, scatterplots), lines (e.g., lineplots, density plots, violin plots, or contour plots) or rectangles (e.g., histograms, ~~barplots~~ bar graphs) (call them what you want)
  2. **statistical transformation** `stat_()`: inside of `ggplot`, we can create new variables to plot the data we want to look at. Counts and frequencies are one such examples that we would use in bar graphs and histograms.
  3. **position**: if our plot has a lot going on, we can adjust the *position* of things on our canvas. This would include things like jittering, stacking, or dodging.

If this is *really* confusing, well, that's because it is, at first. If only there were a one-stop shop with all this information organized in a neatly manner that could be accessed whenever needed...

Fortunately for us, there is! There is a `ggplot2` cheat sheet if you go to **Help ⟶ Cheat Sheets ⟶ Data visualization with ggplot2**:



```
R version 4.4.1 (2024-06-14) -- "Race for Your Life"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

if_else {dplyr} R Documentation

Vectorised if-else

Description

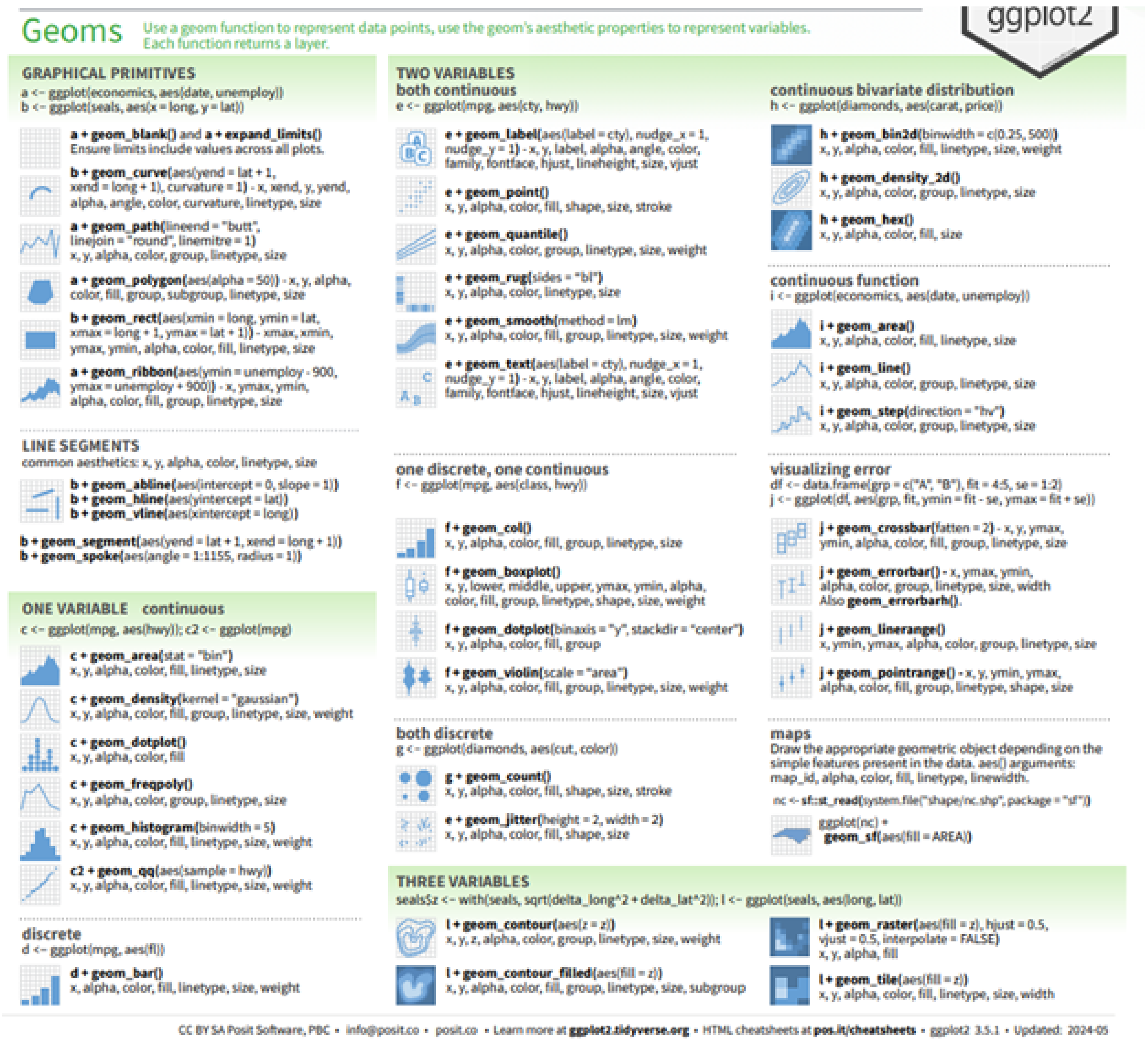
if_else() is a vectorized if-else. Compared to the base R equivalent, ifelse(), this function allows you to handle missing values in the condition with missing and always takes true, false, and missing into account when determining what the output type should be.

Usage

```
if_else(condition, true, false, missing = NULL, ..., ptype = NULL, size = NULL)
```

Arguments

It is rather quite lengthy, so I just want to highlight one part: the geoms!



Geoms Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

ggplot2

GRAPHICAL PRIMITIVES
a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))

a + geom_blank() and **a + expand_limits()**
Ensure limits include values across all plots.

b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1) - x, xend, y, yend, alpha, angle, color, curvature, linetype, size

a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1)
x, y, alpha, color, group, linetype, size

a + geom_polygon(aes(alpha = 50)) - x, y, alpha, color, fill, group, subgroup, linetype, size

b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)) - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)) - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS
common aesthetics: x, y, alpha, color, linetype, size

b + geom_abline(aes(intercept = 0, slope = 1))
b + geom_hline(aes(yintercept = lat))
b + geom_vline(aes(xintercept = long))

b + geom_segment(aes(yend = lat + 1, xend = long + 1))
b + geom_spoke(aes(angle = 1:1155, radius = 1))

ONE VARIABLE continuous
c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)

c + geom_area(stat = "bin")
x, y, alpha, color, fill, linetype, size

c + geom_density(kernel = "gaussian")
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly()
x, y, alpha, color, group, linetype, size

c + geom_histogram(binwidth = 5)
x, y, alpha, color, fill, linetype, size, weight

c2 + geom_qq(aes(sample = hwy))
x, y, alpha, color, fill, linetype, size, weight

discrete
d <- ggplot(mpg, aes(fl))

d + geom_bar()
x, alpha, color, fill, linetype, size, weight

TWO VARIABLES
both continuous
e <- ggplot(mpg, aes(cty, hwy))

e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

e + geom_point()
x, y, alpha, color, fill, shape, size, stroke

e + geom_quantile()
x, y, alpha, color, group, linetype, size, weight

e + geom_rug(sides = "bl")
x, y, alpha, color, linetype, size

e + geom_smooth(method = lm)
x, y, alpha, color, fill, group, linetype, size, weight

e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

one discrete, one continuous
f <- ggplot(mpg, aes(class, hwy))

f + geom_col()
x, y, alpha, color, fill, group, linetype, size

f + geom_boxplot()
x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

f + geom_dotplot(binaxis = "y", stackdir = "center")
x, y, alpha, color, fill, group

f + geom_violin(scale = "area")
x, y, alpha, color, fill, group, linetype, size, weight

both discrete
g <- ggplot(diamonds, aes(cut, color))

g + geom_count()
x, y, alpha, color, fill, shape, size, stroke

e + geom_jitter(height = 2, width = 2)
x, y, alpha, color, fill, shape, size

continuous bivariate distribution
h <- ggplot(diamonds, aes(carat, price))

h + geom_bin2d(binwidth = c(0.25, 500))
x, y, alpha, color, fill, linetype, size, weight

h + geom_density_2d()
x, y, alpha, color, group, linetype, size

h + geom_hex()
x, y, alpha, color, fill, size

continuous function
i <- ggplot(economics, aes(date, unemploy))

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

i + geom_step(direction = "hv")
x, y, alpha, color, group, linetype, size

visualizing error
df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))

j + geom_crossbar(fatten = 2) - x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width
Also **geom_errorbarh()**.

j + geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size

j + geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

maps
Draw the appropriate geometric object depending on the simple features present in the data. aes() arguments: map_id, alpha, color, fill, linetype, linewidth.

nc <- **sf::st_read(**system.file("shape/nc.shp", package = "sf"))

ggplot(nc) +
geom_sf(aes(fill = AREA))

THREE VARIABLES
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))

l + geom_contour(aes(z = z))
x, y, z, alpha, color, group, linetype, size, weight

l + geom_contour_filled(aes(fill = z))
x, y, alpha, color, fill, group, linetype, size, subgroup

l + geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE)
x, y, alpha, fill

l + geom_tile(aes(fill = z))
x, y, alpha, color, fill, linetype, size, width

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **ggplot2.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • ggplot2 3.5.1 • Updated: 2024-05

Figure 1: Geometry of ggplot2

Example time!

We're going to be using the `iris` dataset for the first part of this walkthrough, which is built-in to R, and so we **don't** need to call `library` on it. What I **will** do, though, is save it into its own object so that it's in our global environment:

```
iris <- datasets::iris
```

I used `datasets` explicitly here to show you all where it's coming from, but `datasets` is one of the many packages loaded in whenever you open RStudio.

One Variable Visualizations, Continuous

We're first going to focus on visualizing one variable. Some things we might be interested in are the *distribution* or *shape* of the variable, or how often some values occur more than others. These are often called **graphical summaries** because we're producing graphs!

The graphs at our disposal depend on what *type* of variable we're working with. If we are interested in a *continuous* or *numeric* variable, then some common plots include:

- **density plot:** a *density plot* attempts to plot the density function of the data. Sorry for the word vomit. It is a smooth line that attempts to capture the relative shape and spread of the data. It's often used in conjunction with…
- **histogram:** …the histogram! One plot you can't leave home without. A histogram *partitions* (separates into nonoverlapping groups) the numeric data into *groups* or *bins*, then plots the relative frequency of data values in those groups. It's sorta like a bar graph, but for numeric variables. We can do a *lot* with histograms.

There are two ways to apply `ggplot` to data. Let's check them out. First, let's check out the `ggplot` book from our bookshelf of packages by using `library`:

```
library(ggplot2)
```

If we don't want to save our "canvas" (our `ggplot`) object, we can use the pipe operator directly. I illustrate using a histogram:

```
iris |> ggplot(aes(x = Sepal.Length)) |> geom_histogram()
```

```
## Error in `geom_histogram()`:
## ! `mapping` must be created by `aes()`.
## i Did you use `%>%` or `|>` instead of `+`?
```

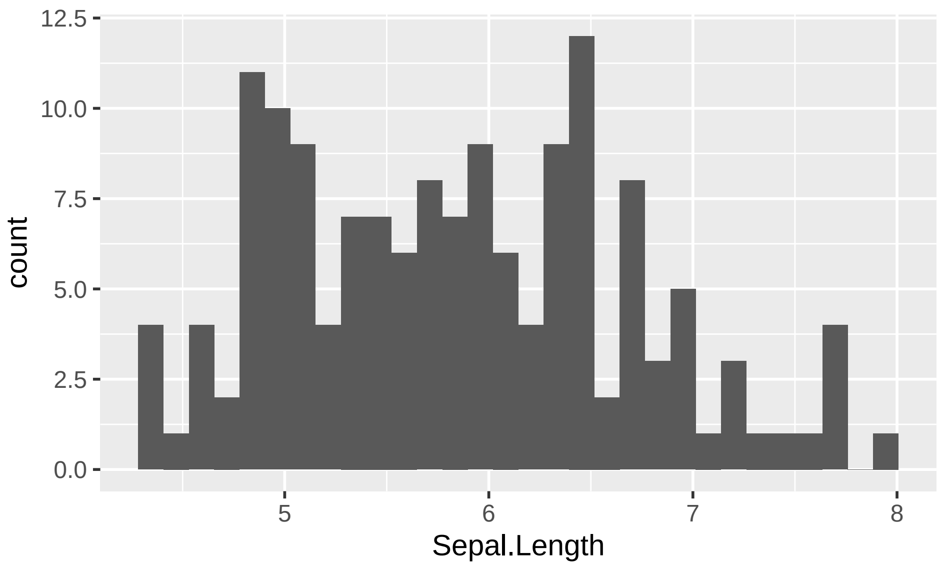
What's this? An *error*?

This is what I referred to earlier: `ggplot`'s pipe operator is `+`, not `%>%`, and R recognizes this, and even tells us! I love informative errors

As for the rest of the code, I would read it like this:

Take the 'iris' dataset and then use it as a canvas or base for our graph, with 'Sepal.Length' on the x-axis. Then create a histogram using as many bins as you see fit.

```r
iris |> ggplot(aes(x = Sepal.Length)) + geom_histogram()
```

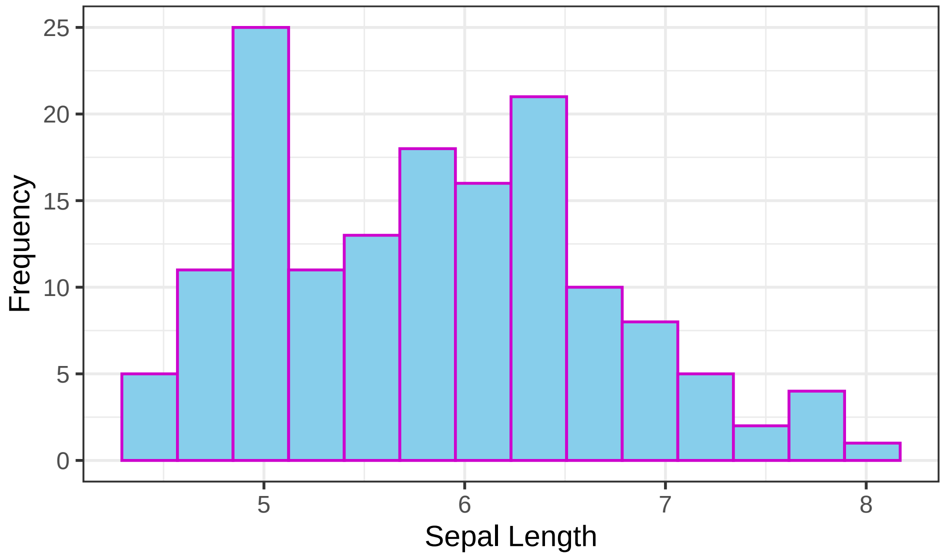


Hooray! We have a histogram! But it's a bit problematic:

- there's far too any bins. Some of them are empty! That's because the default is 30 bins. We can change this with the `bins =` argument.
- the background *sucks*, and our axes don't have labels. We can change the axes with the `labs()` statement, and change the background with `theme_` (there are a few).
- I'm not sure about you, but I personally am not a fan of the dark gray histogram color. We can change this with the `fill =` option, and the we can actually change the outline with `color =` as well!

Additionally, we can save our "canvas" to an R object we can play with:

```r
a <- ggplot(data = iris, mapping = aes(x = Sepal.Length))
a + geom_histogram(bins = 14, fill = "skyblue", color = "magenta3") +
  labs(x = "Sepal Length", y = "Frequency") + theme_bw()
```



For the first line, instead of `iris |> ggplot(...)`, we threw `iris` *into* `ggplot`! I stated the `data =` and `mapping = aes(...)` explicitly for the sake of demonstration. Whether you do `ggplot(iris, aes())` or `iris |> ggplot(aes())`, they will work exactly the same.

Then we assigned our `ggplot` object to a new variable `a`, which is just a placeholder, essentially. Then, much like the chaining with `dplyr`, we use `+` to add new layers! You might notice `theme_bw()`; this stands for "theme **black and white**". There are a few others we'll play with–use your favorite (as long as it's not the `ggplot` default).

As mentioned, we used `fill` and `color` arguments to change the color of the bars on our histogram:

```
geom_histogram(bins = 14, fill = "skyblue", color = "magenta3")
```

`fill` colors the *inside* of the histograms, while `color` actually colors the *border* of the histograms. This can be confusing, as with other visualizations, `color`, well, colors our plots!

You mentioned `geom_density`, and we haven't even used it yet! What the heck, Kris?

You're right! I *did* say density plots are often used in conjunction with histograms. Let's see it in action:

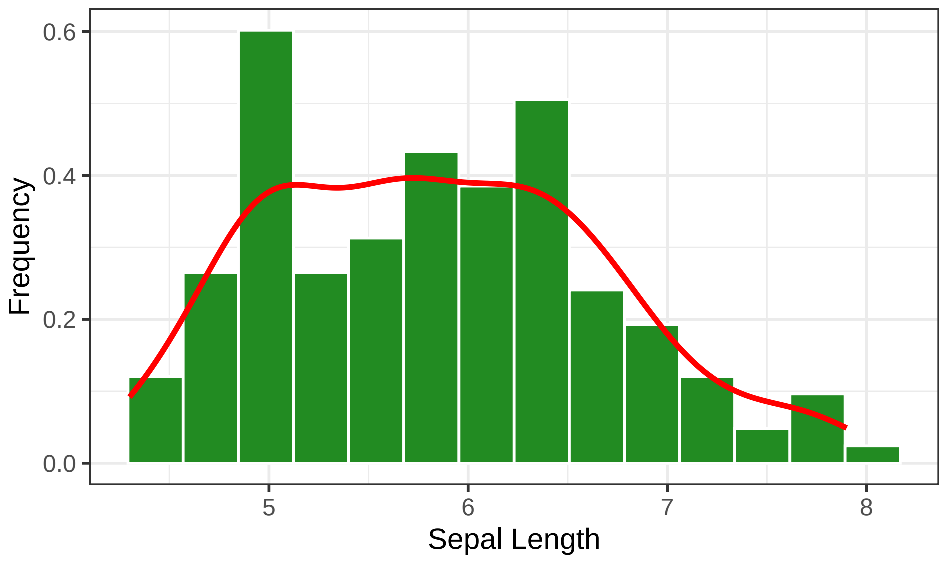
Since we already defined `a` as our `ggplot` object:

```
class(a)
```

```
## [1] "gg"     "ggplot"
```

We don't have to recreate it every time–we can just use it.

```
a +
  geom_histogram(aes(y = stat(density)),
                 bins = 14, fill = "forestgreen", color = "white") +
  geom_density(color = "red", size = 1) +
  labs(x = "Sepal Length", y = "Frequency") + theme_bw()
```



A few things:

- We changed the `geom_histogram()` to now include and `aes()` argument where we specify what we want to be on the y-axis:

```
geom_histogram(aes(y = stat(density)), bins = 14,
               fill = "forestgreen", color = "white")
```

- This is an example of a **statistical transformation** we talked about earlier. Instead of `stat_density()`, we wrote `stat(density)` inside of our mapping.
- Specifically, `aes(y = stat(density))` changes the y-axis from the *count* (the quantity) to the *relative frequency* that the category appears. As a result, when we call `geom_density(color = "red", size = 1)`, the curve will be plotted on a scale appropriate to the histogram. You can remove `aes(y = stat(density))`, keep the rest of the code the same, and see what happens! It's not a pretty sight.

One Variable Visualizations, Discrete

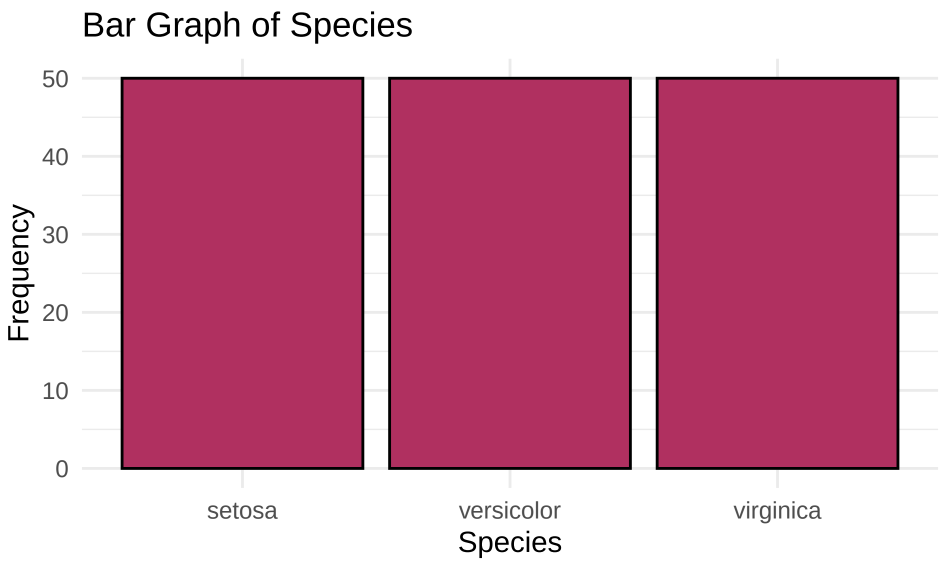
Next, we're going to make a bar graph! That is the only useful one variable discrete visualization. By default, `geom_bar` produces the count for each level of our categorical variable. Unfortunately, for this particular dataset, this isn't super useful, since no `Species` appears more than the others:

```
table(iris$Species)
```

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

That won't stop us though! It just means if we want to summarize `Species` graphically, we'll probably have to do so in conjunction with another variable.

```r
ggplot(iris, aes(x = Species)) +
  geom_bar(fill = "maroon", color = "black") +
  labs(x = "Species", y = "Frequency", title = "Bar Graph of Species") +
  theme_minimal()
```



As alluded to, this is pretty much useless. But it sure looks cool! One thing I want to point out is just like the histograms, we used `fill` and `color` arguments to change the color of the bars

Two Variable Visualizations: One Discrete, One Continuous

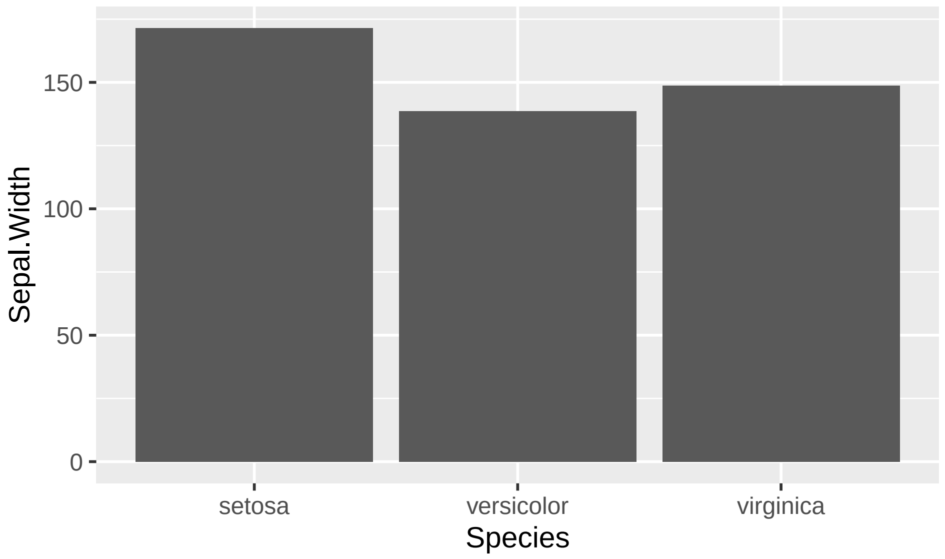
The purpose of two variable visualizations is to investigate things about one variable relative to another. When one of the variables is discrete or categorical, we can investigate at the continuous variable *at each level of the categorical variable*. This is a graphical analog to the combination of `group_by() %>% summarise()` we utilize in order to get summary statistics for each level of a categorical variable. In this case, we have two main layers at our disposal:

- `geom_col()`: This computes the count of the continuous variable at each level of the discrete variable and plots it. This is our common bar graph or bar plot. The count (total) is the default action–we could report a different summary statistic!
- `geom_boxplot()`: This produces a boxplot, which reports the minimum, first quartile, median, third quartile, and maximum values (aka the “five number summary”: min, Q1, median, Q3, max). This is extremely useful for visualizing the shape or distribution of a (numeric) variable at each level of our discrete variable.

But first, the bar graph!

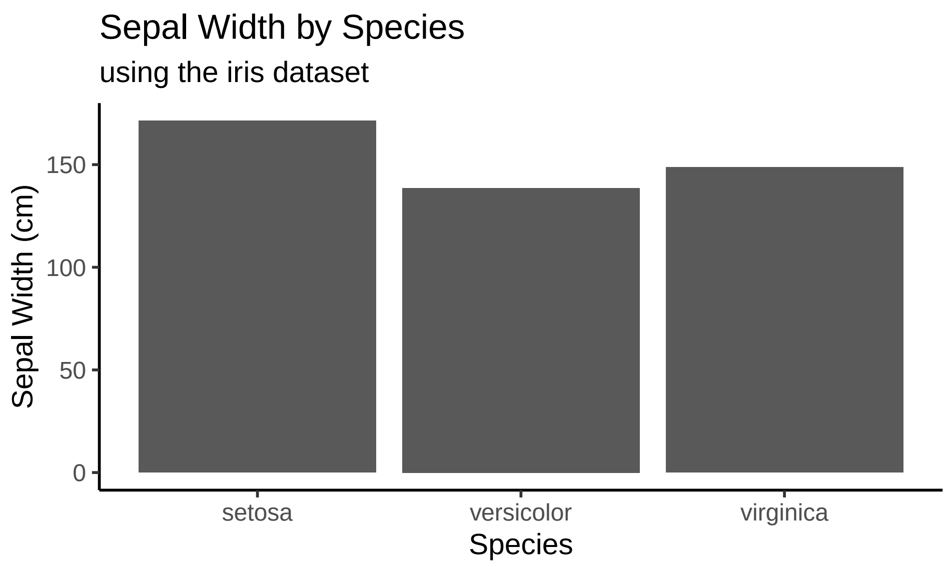
We can look at the values of `Sepal.Width` for each `Species`. Note that I can’t use `a` that we created earlier, since its aesthetics, its canvas, is only for `Sepal.Length`.

```r
iris |> ggplot(aes(x = Species, y = Sepal.Width)) + geom_col()
```



Here we have plotted `Species` on the x-axis, and `Sepal.Width` on the y-axis. It's pretty clear that the `virginica` species (*Iris virginica*) has the greatest Sepal Width. followed by *Iris versicolor*, then lastly by *Iris setosa*.That said, Just like our first histogram, this bar graph is ugly, but it works! Let's add some *flair*, much like how we did before:

```r
iris |> ggplot(aes(x = Species, y = Sepal.Width)) +
  geom_col() + labs(
    y = "Sepal Width (cm)",
    x = "Species",
    title = "Sepal Width by Species",
    subtitle = "using the iris dataset"
  ) + theme_classic()
```

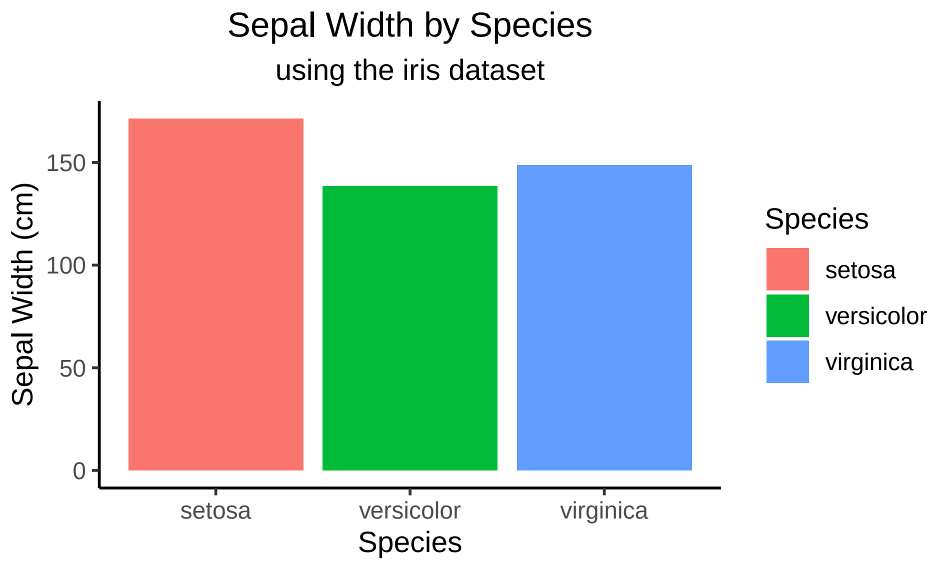


Now:

- `theme_classic()` just makes the background blank. I prefer the grid lines, that other themes have, though!
- We set out axis labels and titles. Note that the `subtitle =` argument adds a subtitle!

I don't know about you, but I *hate* it seeing the titles left-aligned. I'd rather they be in the center! Let's make that happen! Also, we're going to color the bar graphs, each according to their species:

```
iris |> ggplot(aes(x = Species, y = Sepal.Width, fill = Species)) +
  geom_col() + labs(
    y = "Sepal Width (cm)",
    x = "Species",
    title = "Sepal Width by Species",
    subtitle = "using the iris dataset"
  ) + theme_classic() +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5))
```



```
theme(plot.title = element_text(hjust = 0.5),
      plot.subtitle = element_text(hjust = 0.5))
```

- These lines allow us to center the title with `hjust` (horizontal adjustment). There's far, far more customizations you can make than just this–I encourage you to run `help(theme)` or type `?theme` in your R console to learn more!

```
ggplot(aes(x = Species, y = Sepal.Width, fill = Species))
```

- using `fill = Species` colors the bar by the corresponding species. We can change the position of the legend if we want! This is found in `theme()` as well.

If I was describing what I made to a friend, I'd say, "I was interested in seeing if the Sepal Width of iris flowers differed by species, so I made a bar graph where I plotted the total Sepal Width for each species."

(This is one aspect of *exploratory data analysis*, or EDA, where we investigate the data, looking for patterns or trends. This allows us to gain an understanding of the data in order to formulate statistical questions of interest. If you've taken ST 312, 350, 370, or 372, you'll see this is a question we could answer using ANOVA!)

As mentioned, bar graphs, by default, sum the values of the categories to produce the bars. That is, if we wanted to find the number at which those above bars stopped, we could run:

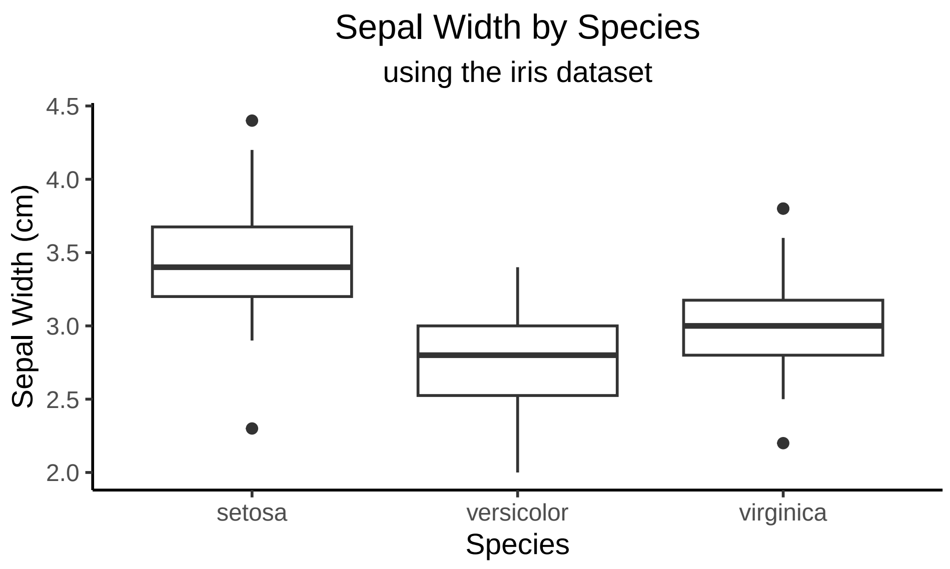
```r
iris |>
  dplyr::group_by(Species) |>
  dplyr::summarise(total_petal_length = sum(Sepal.Width))
```

```
## # A tibble: 3 x 2
##   Species    total_petal_length
##   <fct>                   <dbl>
## 1 setosa                   171.
## 2 versicolor               138.
## 3 virginica                149.
```

You'll notice I specified `dplyr` directly; that's because I never actually loaded dplyr with `library()`!

We have no idea what the distribution of the variable of interest actually looks like! What is the most common value? How spread out is the data? Are there any outliers? In order to get this, we can use a **boxplot**:

```r
iris |> ggplot(aes(x = Species, y = Sepal.Width)) +
  geom_boxplot() + labs(
    y = "Sepal Width (cm)",
    x = "Species",
    title = "Sepal Width by Species",
    subtitle = "using the iris dataset"
  ) + theme_classic() +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5))
```

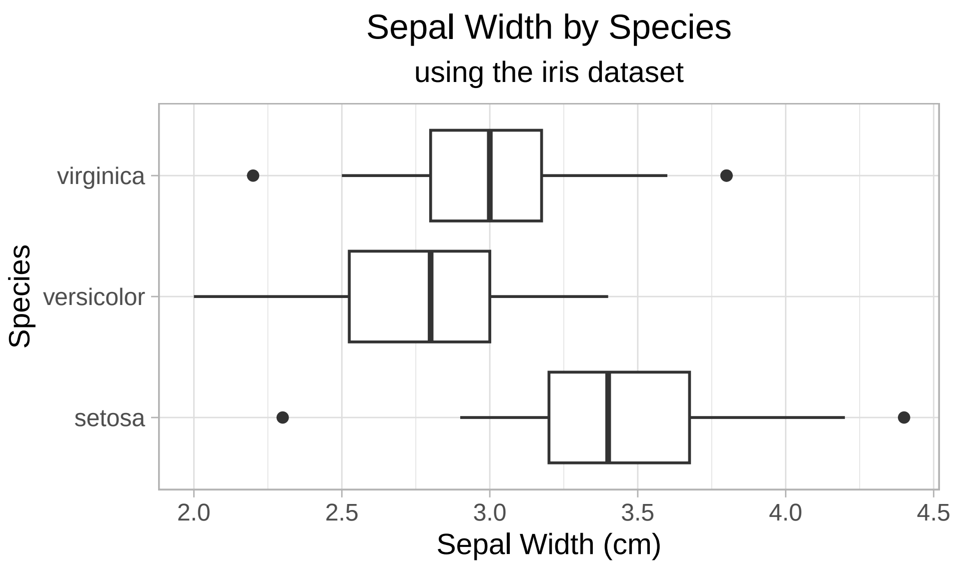


This code looks *eerily similar* to the bar graph code; I just took off the `fill = Species` option from the `aes()` argument inside of the `ggplot` statement. Also, `geom_col()` became `geom_boxplot()`.

The bold, horizontal bar in the middle of the box is the median, while the two boxes on either side of the median represent the quarter (25%) of the data that lies below and above the median, respectively. The whiskers, or the lines jetting out from the boxes, represent the other 25% of data that lie below the first and third quartiles, respectively. This allows us to make statements about the shape of the data, letting us talk about symmetry, skewness, or outliers (the dots).

Alternatively can swap the axes, and it might read a bit better (or worse!):

```
iris |> ggplot(aes(y = Species, x = Sepal.Width)) +
  geom_boxplot() + labs(
    x = "Sepal Width (cm)",
    y = "Species",
    title = "Sepal Width by Species",
    subtitle = "using the iris dataset"
  ) + theme_light() +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5))
```



Personally, I like this better, but that will depend on the number of categories in your categorical variable, and the spread of your data!

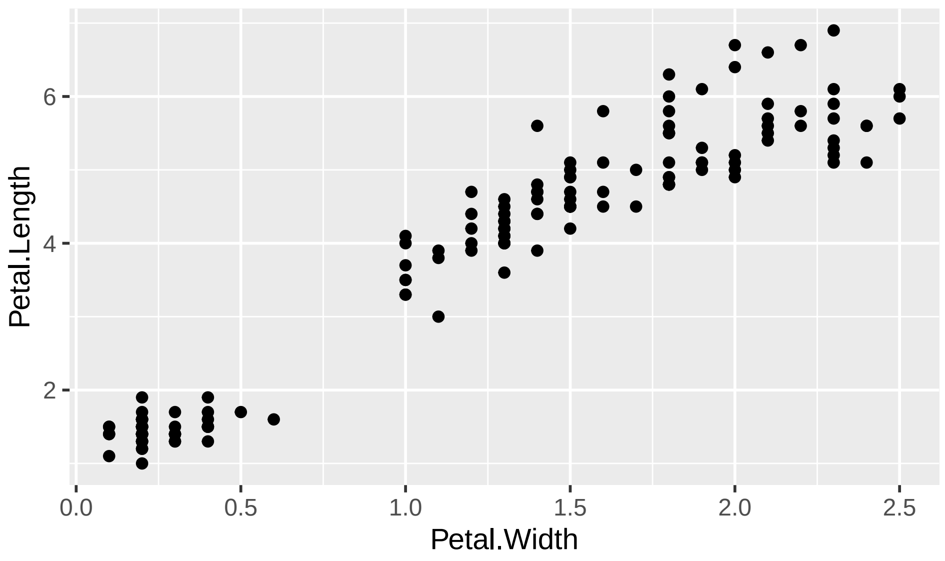
Two Variable Visualizations, Both Continuous

You might be wondering what happened to "two variable visualizations, both discrete"… for that, I refer you to the cheat sheet. We don't use it *super often*, but they are useful!

For the two variable case when they're both continuous, the main layer we'll use is `geom_point()`. This is for creating scatterplots. We'll often use this alongside `geom_smooth()` to plot our regression line over the points. Without going into too much detail, this is effectively plotting $y = mx + b$ from your grade-school math days on top of plotting and x and y against each other.

I don't know about you, but I know *very little* about flowers. That said, I *feel* like petal width and petal length should be related. Let's see if this is true!

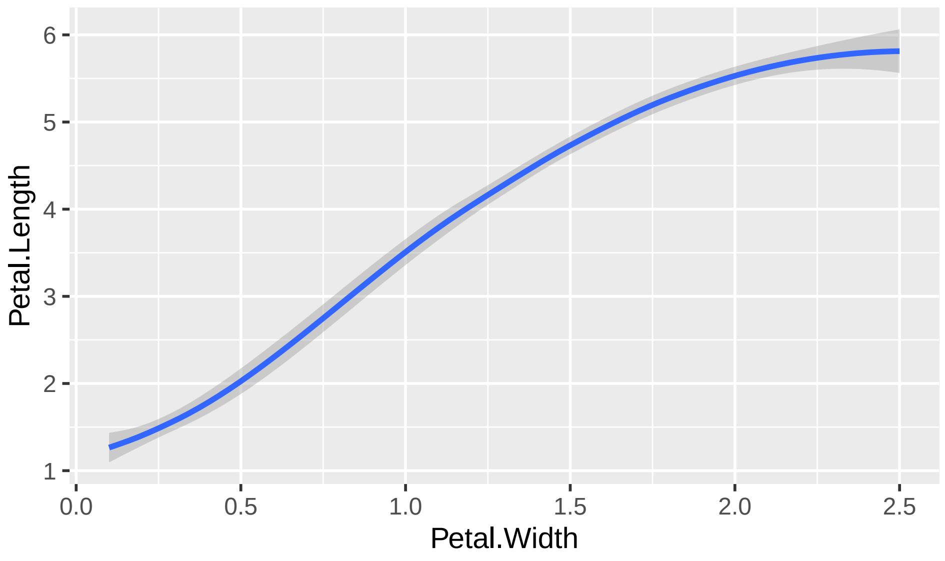
```r
canvas <- ggplot(iris, aes(y = Petal.Length, x = Petal.Width))
canvas + geom_point()
```



We see there is a **strong, positive linear relationship** between petal length and petal width (remember that phrase! It'll show up again! Just not in this class). Not sure what that gap is about, though.

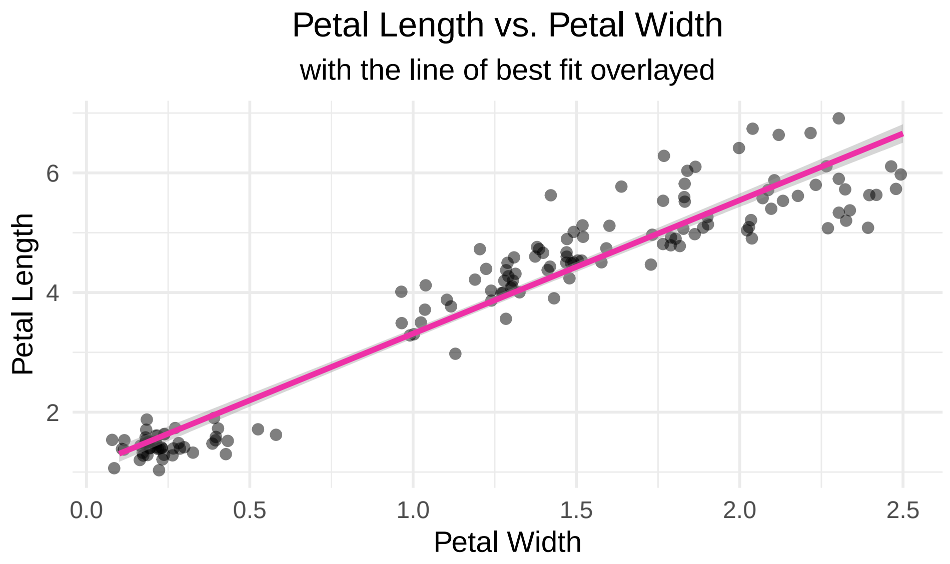
If we used `geom_smooth()` without `geom_point()`, the points will go away, and we'll just have the line:

```
canvas + geom_smooth()
```



Boring! Let's put them together! (Also, I don't know about you, but ~~I'm feeling 22~~ I don't like all those points vertically stacked on top of each other. Let's fix that too)

```
canvas + geom_point(position = "jitter", alpha = 0.5) +
  geom_smooth(method = "lm", color = "maroon2") +
  labs(
    x = "Petal Width", y = "Petal Length",
    title = "Petal Length vs. Petal Width",
    subtitle = "with the line of best fit overlayed"
  ) + theme_minimal() + theme(
    plot.title = element_text(hjust = 0.5),
    plot.subtitle = element_text(hjust = 0.5)
  )
```



I know, I know. I just threw a *lot* into that code chunk; Let's go through it line-by-line:

- First, let's recall our `canvas` object, which if you recall:

```
canvas <- ggplot(iris, aes(y = Petal.Length, x = Petal.Width))
```

- This creates an object called `canvas`, which is blank plot with petal length on the y-axis and petal width on the x-axis.

```
canvas + geom_point(position = "jitter", alpha = 0.5) +
```

- `geom_point()` creates a scatterplot of petal length and petal width. That is, this line creates a layer that puts the actual dots on the background.
-   - The `position = "jitter"` argument is a *position* argument that moves the points so that they're not literally right on top of each other. This is purely a visual thing–it doesn't change the values themselves. We could also use `geom_jitter()` instead of `geom_point(position = "jitter")` as a shorthand.
-   - `alpha` controls the shading of the points. The default is `alpha = 1`, so `alpha = 0.5` makes the points about half as dark as they normally are. This, together with the `position = "jitter"`, makes it so that we can actually see our points!

```
geom_smooth(method = "lm", color = "maroon2") +
```

- `geom_smooth()` puts the line on top of the points. Order matters! If `geom_smooth` came before `geom_point`, then the points would be on top of the line

- - `method = "lm"` specifies the method to fit the line. I won't go into too much detail about it here, except that **lm** produces a straight line, and other methods produce curved lines! There are benefits and drawbacks to both.
- - The gray outline you see is called the *standard error bands*, which is a fancy name for the confidence interval for the line of best fit.

```
labs(
    x = "Petal Width", y = "Petal Length",
    title = "Petal Length vs. Petal Width",
    subtitle = "with the line of best fit overlayed"
) +
```

- As we've seen before, `labs()` creates custom labels for our x-axis, y-axis, title, and subtitle.

```
theme_minimal() + theme(
    plot.title = element_text(hjust = 0.5),
    plot.subtitle = element_text(hjust = 0.5)
    )
```

- `theme_minimal()` is my favorite theme :D
- the `plot.title` and `plot.subtitle` allow us to center our title and subtitle like we did earlier.

And, scene! That's all there is for the Week 12 Walkthrough. See you in class and/or office hours!