Graphs Part 2

Joshua F. Wiley

2019-05-27

Contents

Download the raw R markdown code here `https://jwiley.github.io/MonashHonoursStatistics/Graphs_Pt2.rmd`.

Two useful websites:

- Official documentation: `https://ggplot2.tidyverse.org/index.html`
- Cookbook by Winston Chang: `http://www.cookbook-r.com/`
- Question and Answer site: `https://stackoverflow.com/questions/tagged/ggplot2?sort=newest&pageSize=50`

```
options(digits = 2)

## now new packages this week
library(data.table)
library(ggplot2)
library(cowplot)
library(ggthemes)

## read in the dataset
d <- readRDS("aces_daily_sim_processed.RDS")

## small sample dataset
dt <- data.table(Age = c(20, 30, 40, 50, 60),
    Memory = c(7, 5, 6, 4.5, 4))

## convert a few variables in mtcars dataset
```

```
## into discrete factor variables
rm(mtcars)
mtcars$cyl <- factor(mtcars$cyl)
mtcars$vs <- factor(mtcars$vs)
mtcars$am <- factor(mtcars$am)
```

1 Customising Graphs

Often the default options in graphs are not quite what we want for a final, polished presentation. Here we will look at ways different aspects of graphs can be customised.

To start with, we will setup a simple scatter plot, using the `mtcars` dataset in R. We will re-use this base and add on different customisations.

```
p <- ggplot(mtcars, aes(qsec, disp, colour = am)) +
    geom_point()
print(p)
```

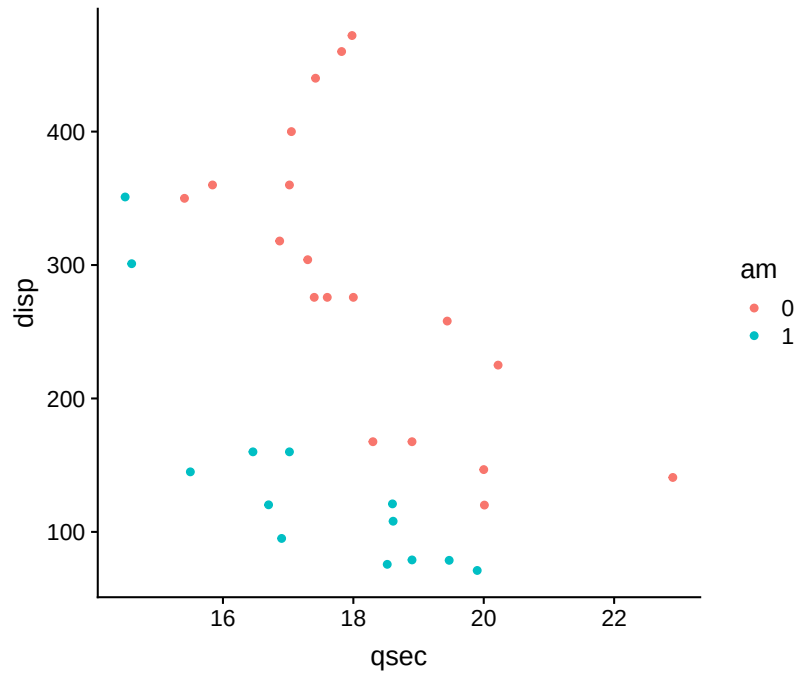

Figure 1: Simple scatter plot.

1.1 Labels

One thing that often needs to be changed are labels as variable names often are not informative. These can be set using the `xlab()`

and `ylab()` functions, shown as follows. You can include grek or math as needed. For a full list of possibilities, see: `?plotmath` or `help("plotmath")`.

```
p + xlab("My X Axis Label") + ylab(expression(alpha ==
    frac(kg, m^2)))
```

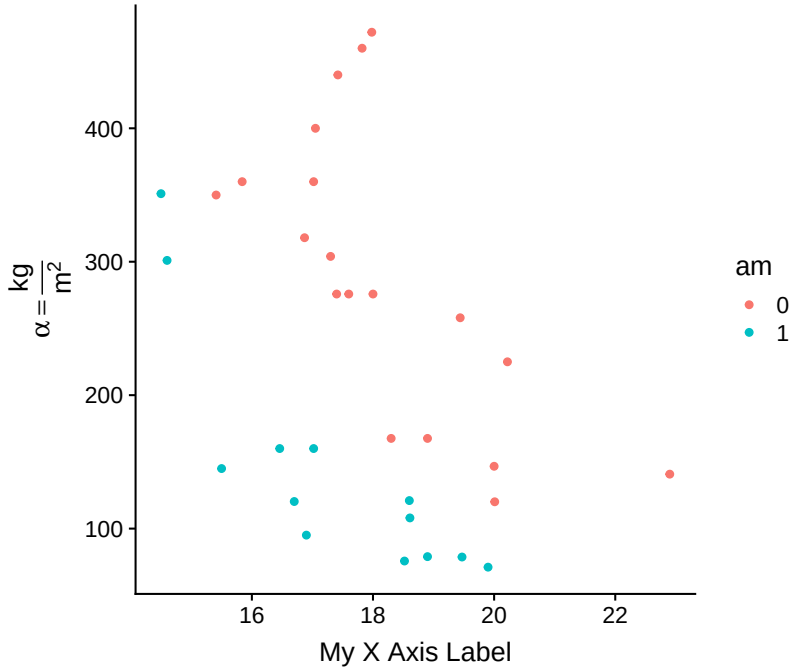

Figure 2: Labeled scatter plot.

The `ggtitle()` function can be used to add a bold title and (optionally) a subtitle. The subtitle can be left blank. As with axis labels, you can use math expressions or Greek letters in titles by using the `expression()` function, if desired.

```
p + ggtitle("Impact of qsec on Displacement",
    subtitle = "something extra here!")
```

1.2 *Changing Limits*

We can **zoom** in or out on a specific region of the plot by changing the limits of the default coordinate system.

```
p + coord_cartesian(xlim = c(10, 30), ylim = c(100,
    300), expand = FALSE)
```

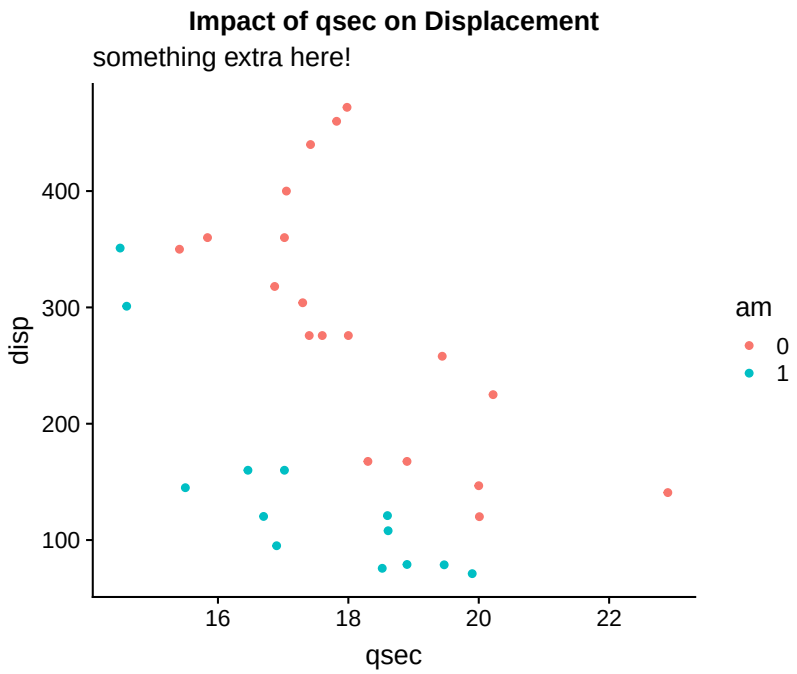

Figure 3: Titled scatter plot.

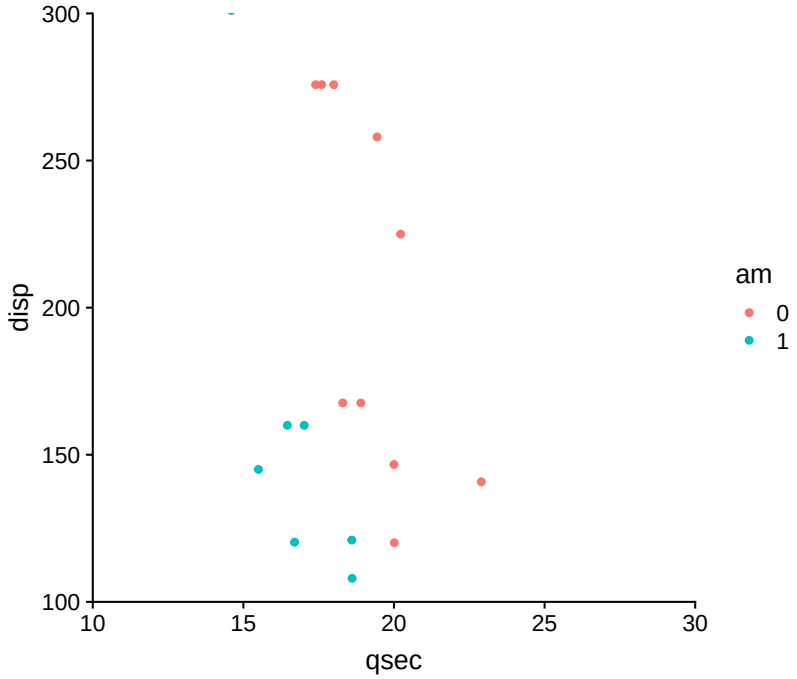

Figure 4: Zoomed scatter plot.

There are also functions `xlim()` and `ylim()` but these do not zoom the plot, they actually exclude any data outside the limits, which changes the underlying data available for plotting. This distinction is particularly important when there are summary statistics calculated on the data automatically. If there are and you use `xlim()` or `ylim()`, you may exclude some data and thus get different summary statistics. If this is the desired behaviour, that is appropriate. If you, however, wanted only to zoom in, not to change the data, then use `coord_cartesian()` as above. By default, it generates a warning message that it is excluding some rows of data when using `xlim()` or `ylim()`.

```
p + xlim(10, 30) + ylim(100, 300)
```

```
## Warning: Removed 16 rows containing missing values
## (geom_point).
```

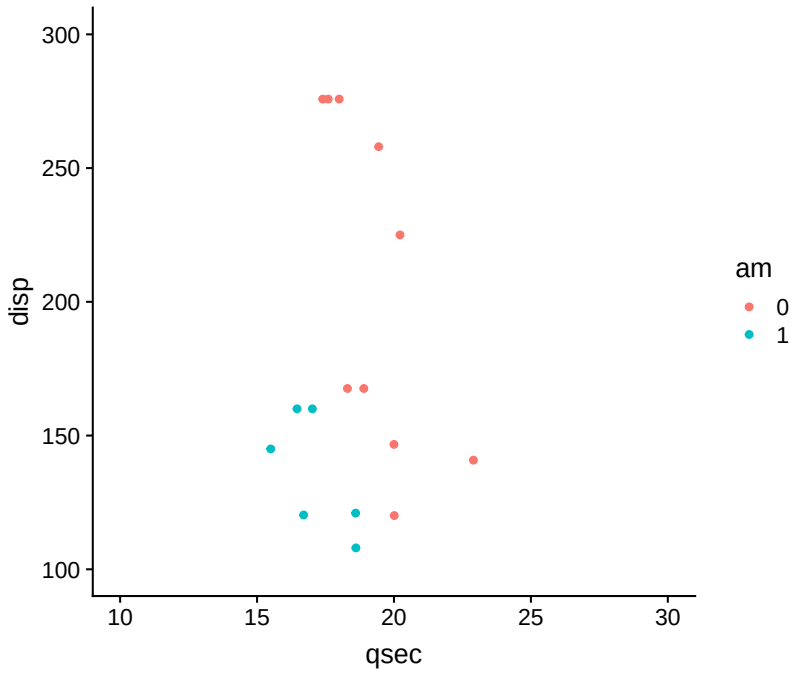

Figure 5: Zoomed scatter plot.

1.3 Breaks

We have already seen how to adjust breaks in the previous week while discussing how to improve plots. Now we will look specifically at the code. The axes are called "scales" in `ggplot2` terminology, and we use corresponding functions to control them. If they are

continuous, we use continuous scales, if categorical, we use `discrete` scales. By default, breaks are labelled whatever their number or value is. However, this can be changed by specifying separate labels. If specified, there should be as many labels as breaks and they should be in the same order. This allows arbitrary text to be included.

```
p + scale_x_continuous(breaks = c(16, 20, 22)) +
    scale_y_continuous(breaks = c(100, 300, 400),
        labels = c("Low", "Middle", "400\n(i.e., High)"))
```

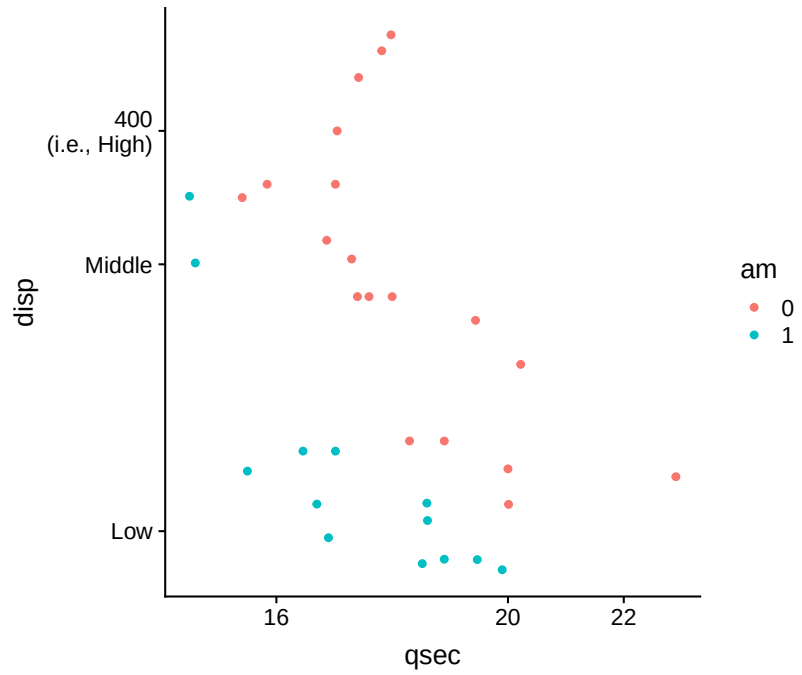

Figure 6: Custom breaks scatter plot.

1.4 Colours

You can control the colours used for different groups, either by switching pallettes or setting them manually.

```
p + scale_color_viridis_d()
```

If other colour pallettes do not have the colours you want, you can manually set the colours. We can also over ride the default legend title, from the variable name of the colouring variable to something else. This same approach works with shapes and lines, as these are all "scales" in `ggplot2` terminology.

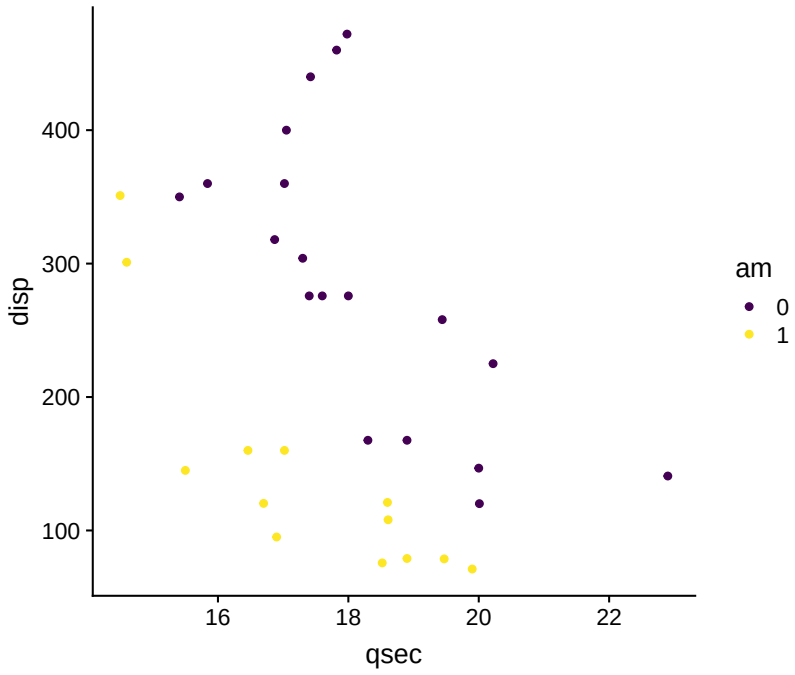

Figure 7: New colours scatter plot.

```
p + scale_color_manual(name = "Transmission\n  0 = automatic\n  1 = manual",
    values = c('0' = "black", '1' = rgb(250, 200,
        50, maxColorValue = 255)))
```

1.5 *Shapes*

You can control the colours used for different groups manually as well. To see more shapes available, see: `http://www.cookbook-r.com/Graphs/Shapes_and_line_types/`.

```
ggplot(mtcars, aes(qsec, disp, shape = am)) +
    geom_point() + scale_shape_manual(name = "Transmission\n  0 = automatic\n  1 = manual",
    values = c('0' = 1, '1' = 16))
```

2 *Summaries in Graphs*

Graphs often show summaries of data, such as means, or model results in addition to being used to show the raw data.

There are two general ways to present summaries in graphs.

- Calculate any summaries you want and store these in a dataset which can be plotted using any of the plotting methods we learned earlier.

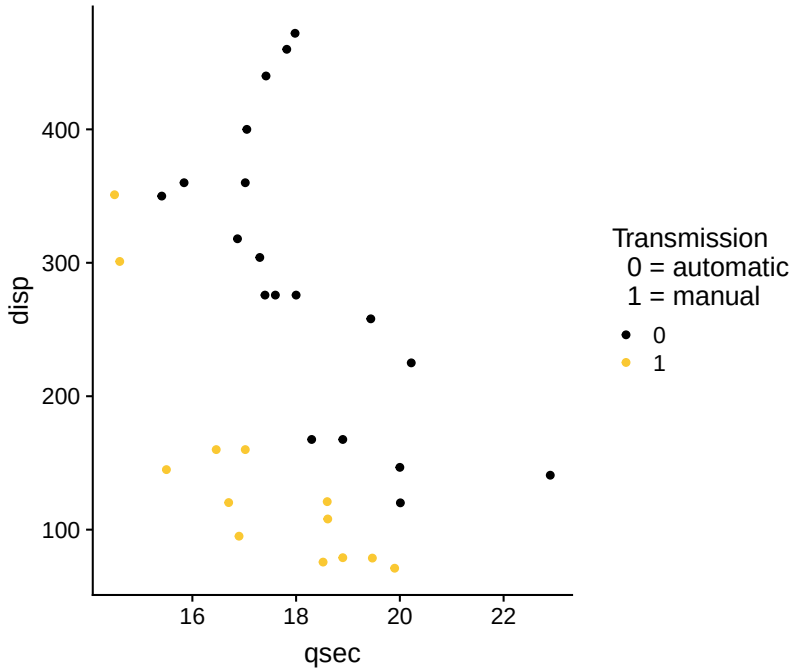

Figure 8: Custom colours scatter plot.

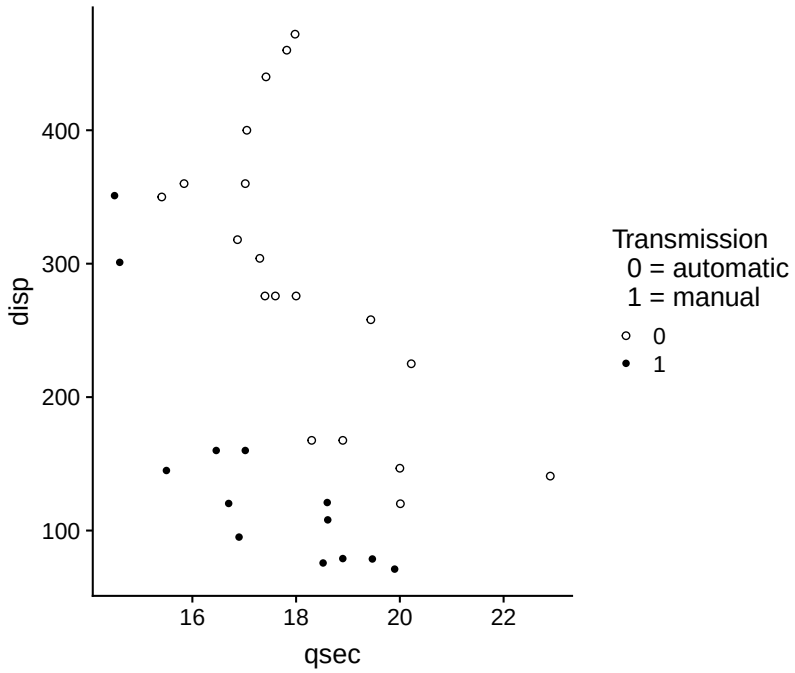

Figure 9: Custom shapes scatter plot.

- Have R calculate the summaries and plot in one step

For simple cases, having R calculate and plot simultaneously is the easiest. This is primarily accomplished using one of two functions:

- `stat_summary()` which summarizes y values at specific x values
- `stat_smooth()` which fits smoothing lines, such as linear regression, or more flexibility regression models

The following example shows a "pointrange" representation of mean and standard error bars for the weight of chicks being given different feeds. We specify how the data should be summarized by passing the `mean_se` function to the `fun.data =` argument.

```
ggplot(chickwts, aes(feed, weight)) + stat_summary(fun.data = mean_se,
    geom = "pointrange")
```

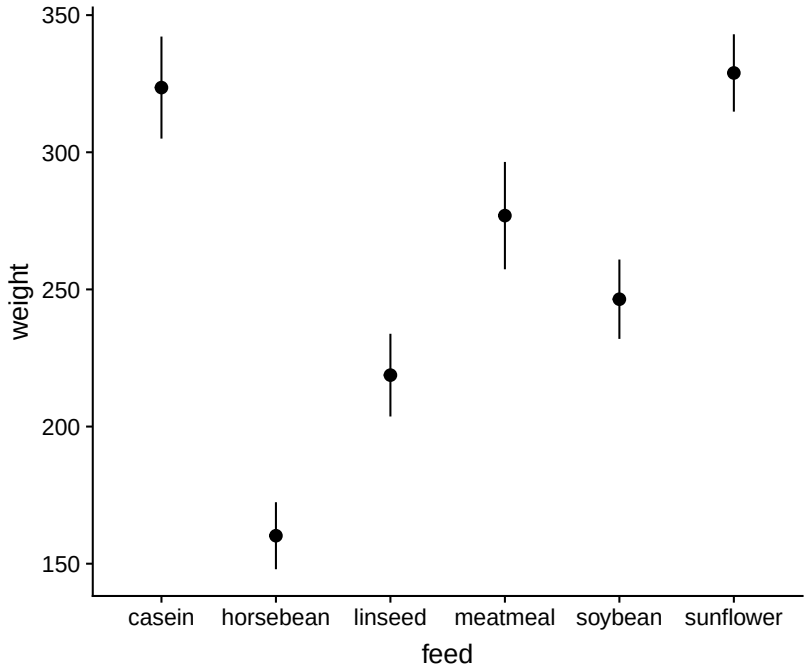

Figure 10: Mean and standard error bars for chick weights by feed.

By default, `feed` is a factor but its levels are ordered alphabetically. One way to improve the graph is to sort the data. In this case, its not clear that alphabetical sorting is very useful. Sorting by the mean is a more helpful way to present the data. We can accomplish this by using the `factor()` function and specifying the levels exactly as they are spelled in the data, but in the order we want them shown. This lets readers easily see the range, for example, in the means.

In addition, rather than standard error bars, we can get 95% confidence intervals assuming the data are normally distributed using the `mean_cl_normal` function with the same point range geometic representation. In practice, we would probably also clean up the x axis label using `xlab()`.

```
ggplot(chickwts, aes(factor(feed, levels = c("horsebean",
    "linseed", "soybean", "meatmeal", "casein",
    "sunflower")), weight)) + stat_summary(fun.data = mean_cl_normal,
    geom = "pointrange")
```

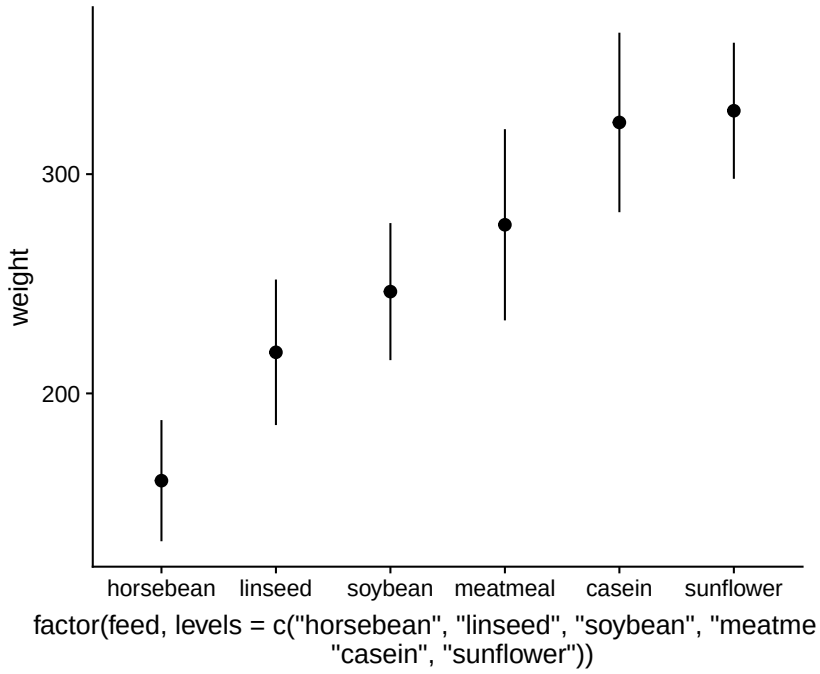

Figure 11: Mean confidence interval bars for chick weights by feed.

If desired, we can calculate summaries outside of `ggplot2` and then just graph them. For example, we could use dummy coding in regression to get estimated means and confidence intervals for plotting.

```
m <- lm(weight ~ 0 + feed, data = chickwts)
summary(m)
```

```
##
## Call:
## lm(formula = weight ~ 0 + feed, data = chickwts)
##
## Residuals:
```

```
##     Min      1Q  Median      3Q     Max
## -123.91  -34.41    1.57   38.17  103.09
##
## Coefficients:
##               Estimate Std. Error t value
## feedcasein       323.6       15.8   20.44
## feedhorsebean    160.2       17.3    9.24
## feedlinseed      218.8       15.8   13.82
## feedmeatmeal     276.9       16.5   16.74
## feedsoybean      246.4       14.7   16.81
## feedsunflower    328.9       15.8   20.77
##               Pr(>|t|)
## feedcasein     < 2e-16 ***
## feedhorsebean  1.9e-13 ***
## feedlinseed    < 2e-16 ***
## feedmeatmeal   < 2e-16 ***
## feedsoybean    < 2e-16 ***
## feedsunflower  < 2e-16 ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Residual standard error: 55 on 65 degrees of freedom
## Multiple R-squared:  0.963,  Adjusted R-squared:  0.96
## F-statistic:  281 on 6 and 65 DF,  p-value: <2e-16
```

```
plotdat <- data.table(feed = gsub("feed", "",
    names(coef(m))), M = coef(m), LL = confint(m)[,
    1], UL = confint(m)[, 2])

## sort and factor to get nice ordering
plotdat <- plotdat[order(M)][, `:=`(feed, factor(feed,
    levels = feed))]

ggplot(plotdat, aes(feed, y = M, ymin = LL, ymax = UL)) +
    geom_pointrange()
```

Error bars can be used in place of the point range. Error bars do not have a middle point by default, but we can easily add points.

```
ggplot(plotdat, aes(feed, y = M, ymin = LL, ymax = UL)) +
    geom_errorbar() + geom_point()
```

Although not a very efficient way to convey data relative to the ink, barplots can be easily made. Shrinking the width of error bars is conventional.

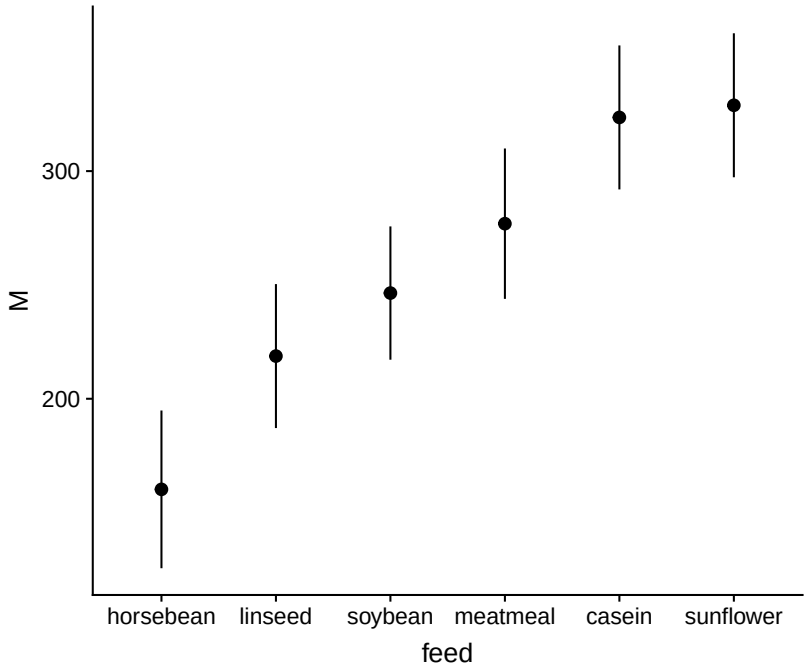

Figure 12: Model-based confidence intervals for chick weights by feed.

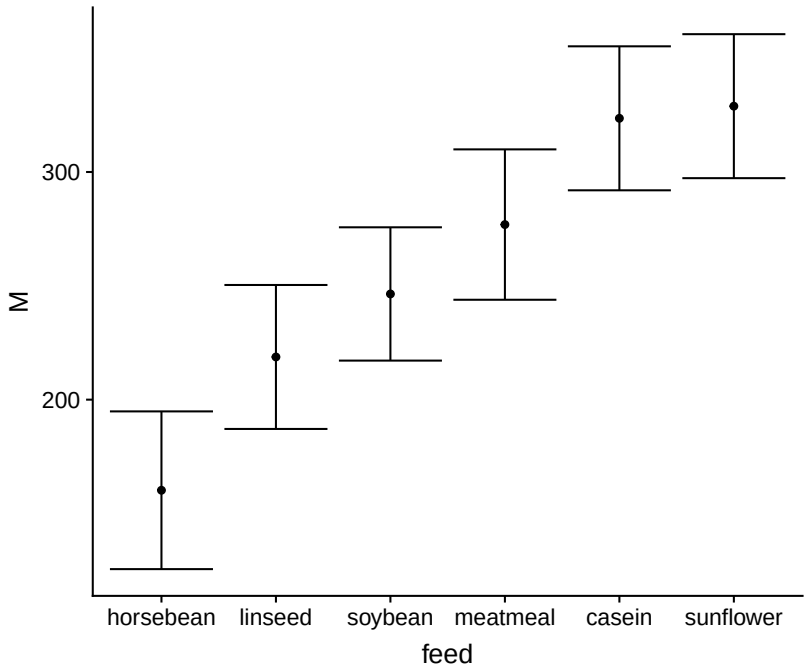

Figure 13: Model-based confidence intervals on error bars for chick weights by feed.

```
ggplot(plotdat, aes(feed, y = M, ymin = LL, ymax = UL)) +
    geom_bar(stat = "identity") + geom_errorbar(width = 0.5)
```

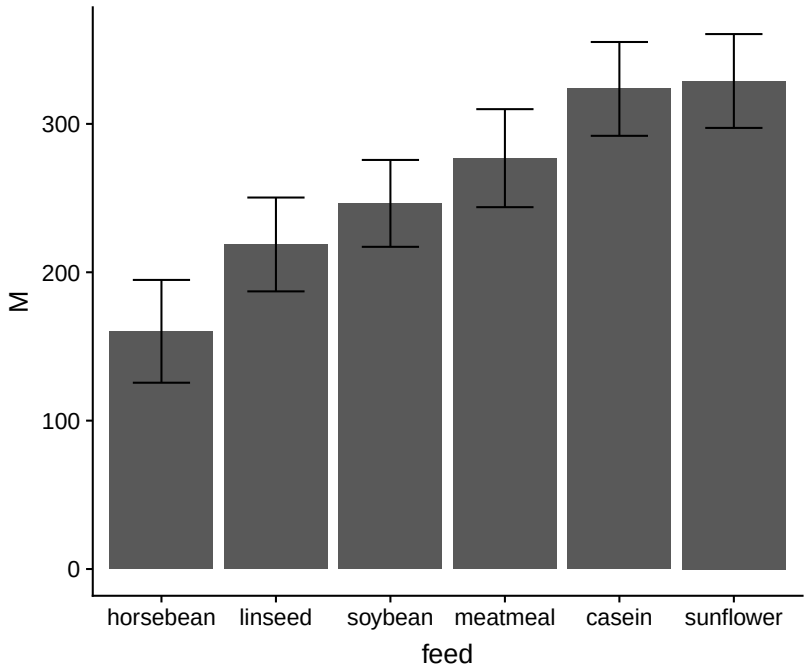

Figure 14: Model-based confidence intervals barplot with error bars for chick weights by feed.

By zooming the plot in, we can make it look somewhat cleaner and add better labels.

```
ggplot(plotdat, aes(feed, y = M, ymin = LL, ymax = UL)) +
    geom_bar(stat = "identity", fill = "grey80") +
    geom_errorbar(width = 0.5) + coord_cartesian(ylim = c(0,
    370), expand = FALSE) + xlab("Type of Feed") +
    ylab("Chick Weight") + ggtitle("Means and 95% CIs for Chick Weights by Feed")
```

A more minimal approach for presenting the data is shown below.

```
ggplot(plotdat, aes(feed, y = M, ymin = LL, ymax = UL)) +
    geom_errorbar(width = 0.5) + geom_point(size = 3,
    shape = 18) + geom_text(aes(x = as.integer(feed) +
    0.25, label = round(M))) + xlab("Type of Feed") +
    ylab("Chick Weight") + ggtitle("Means and 95% CIs for Chick Weights by Feed") +
    theme(axis.line = element_blank(), axis.ticks = element_blank(),
        axis.text.y = element_blank(), axis.title.y = element_blank()) +
    coord_cartesian(xlim = c(0.5, 6.5), expand = FALSE)
```

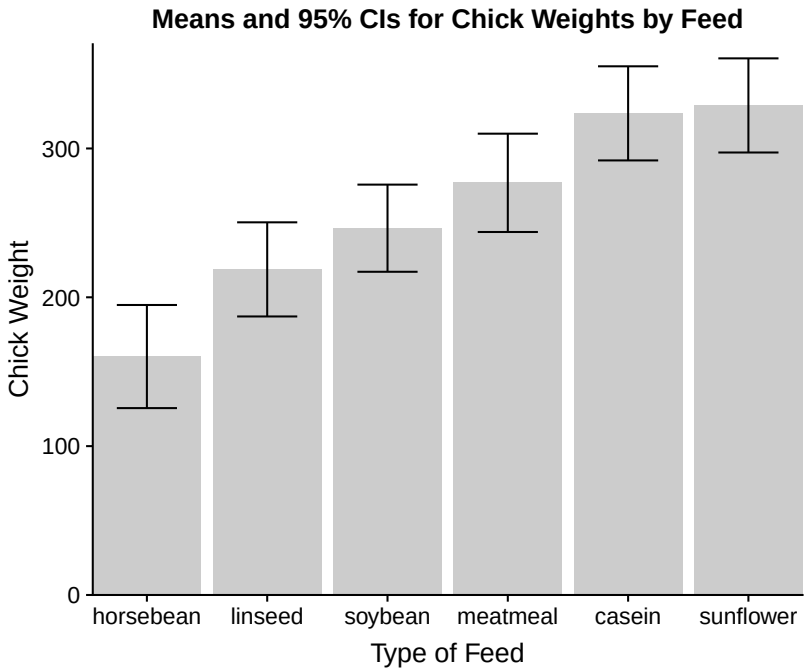

Figure 15: Model-based confidence intervals barplot with error bars for chick weights by feed.

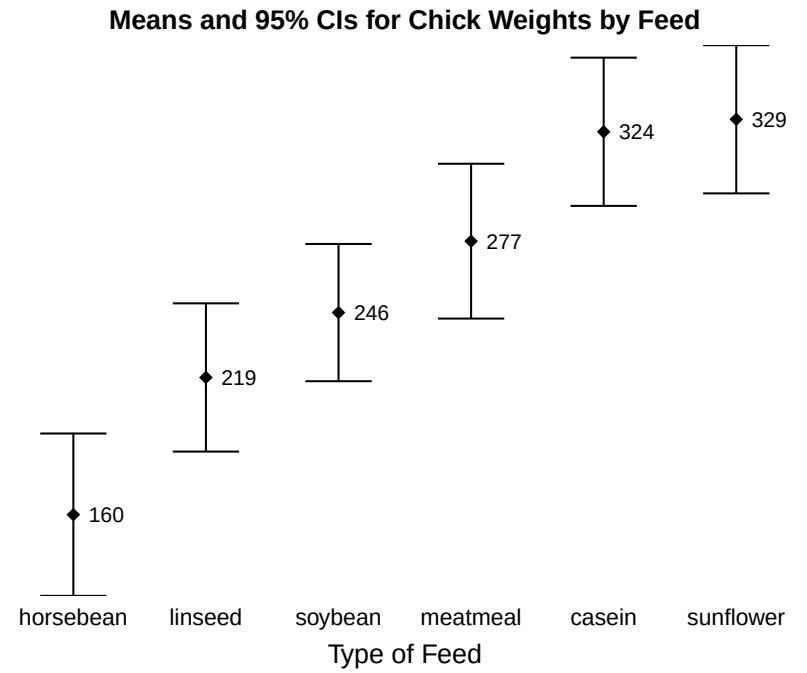

Figure 16: Model-based confidence intervals on error bars for chick weights by feed.

3 *Panels of Graphs*

The last topic we will look at is combining multiple plots together into a panel of plots. This functionality is supported by the `cowplot` package which provides the `plot_grid()` function. First, we can start by making two density plots from the `iris` dataset and combining these.

```
p1a <- ggplot(iris, aes(Sepal.Length, colour = Species)) +
    geom_density() + scale_x_continuous(breaks = as.numeric(quantile(iris$Sepal.Length)))

p1b <- ggplot(iris, aes(Sepal.Width, colour = Species)) +
    geom_density() + scale_x_continuous(breaks = as.numeric(quantile(iris$Sepal.Width)))


p1grid <- plot_grid(p1a, p1b, nrow = 1)
print(p1grid)
```

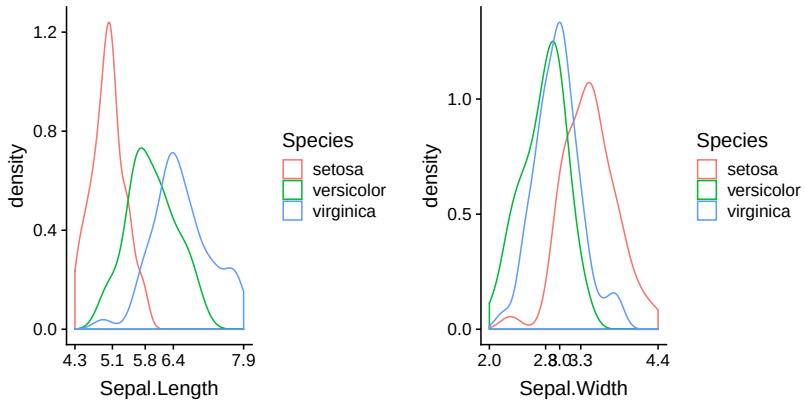

Figure 17: Panel density plots by species.

In this case, both graphs share the same legend. We could clean it up by combining these. It would also help to add labels so we can distinguish each plot when writing or talking about them (e.g., Figure 1A, etc.). First, we copy out the legend, then we remove it from each individual plot and give it a space at the end of the grid. We save and then print (plot) the final product.

```
legend <- get_legend(p1a)

p1grid <- plot_grid(p1a + theme(legend.position = "none"),
    p1b + theme(legend.position = "none"), legend,
    nrow = 1, rel_widths = c(1, 1, 0.4), labels = c("A",
        "B"))
```

```
print(p1grid)
```

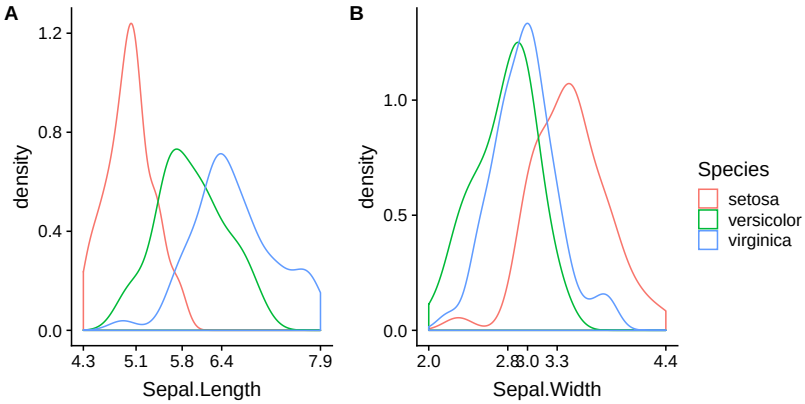

Figure 18: Panel density plots by species, common legend.

We could show the number of observations from each species and a bivariate distribution using contours together.

```
p2a <- ggplot(iris, aes(Species)) + geom_bar() +
    coord_cartesian(xlim = c(0.5, 3.5), ylim = c(0,
        55), expand = FALSE)

p2b <- ggplot(iris, aes(Sepal.Width, Sepal.Length)) +
    stat_density_2d() + scale_x_continuous(breaks = as.numeric(quantile(iris$Sepal.Width))) +
    scale_y_continuous(breaks = as.numeric(quantile(iris$Sepal.Length))) +
    theme(axis.line = element_blank()) + geom_rangeframe()

p2grid <- plot_grid(p2a, p2b, nrow = 1, rel_widths = c(1,
    1.4), labels = c("C", "D"))

print(p2grid)
```

Although we often, make up grids ourselves, if the variables exist in the dataset, `ggplot2` can make grids for us to a degree. In the following plot, we get a panel of graphs showing scatter plots and linear trends (linear smooth lines) for sepal width and length by species. Making "small multiples" of graphs with related but distinct information is one way to break down large figures and help people read them more easily, when, for example, multiple coloured points in the same graph may get harder to read.

```
p3 <- ggplot(iris, aes(Sepal.Width, Sepal.Length)) +
    geom_point() + stat_smooth(method = "lm",
```

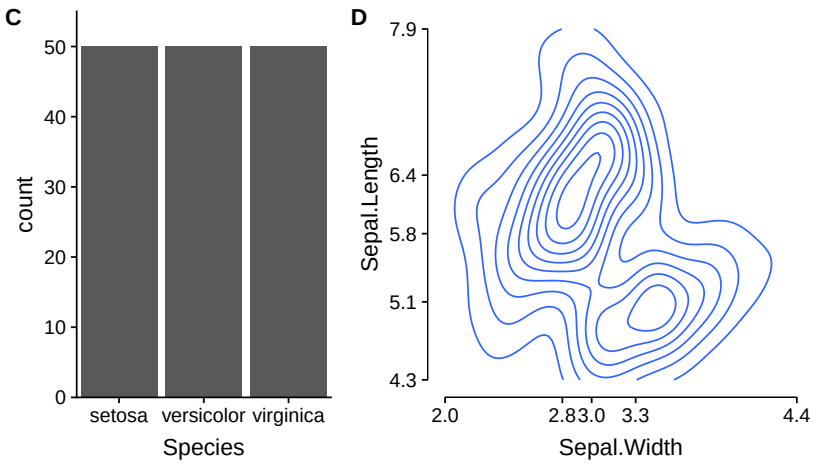

Figure 19: Count and bivariate contours.

```
    se = FALSE) + scale_x_continuous(breaks = as.numeric(quantile(iris$Sepal.Width))) +
    scale_y_continuous(breaks = as.numeric(quantile(iris$Sepal.Length))) +
    facet_wrap(~Species)

print(p3)
```

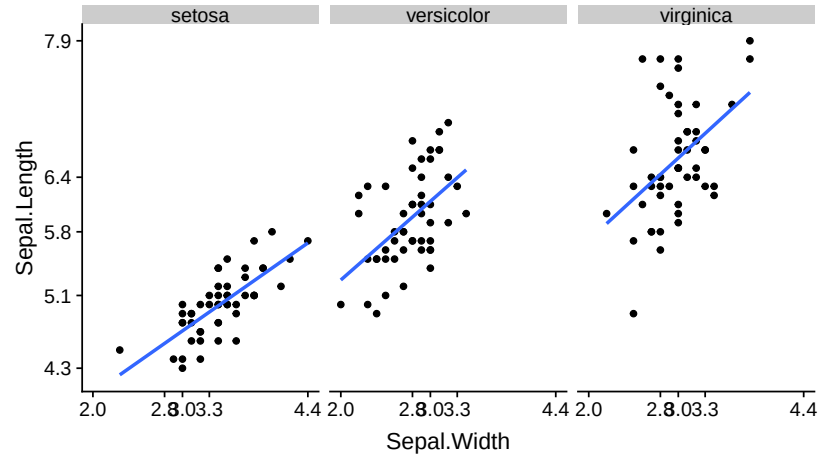

Figure 20: Small multiples plot.

This shows an issue that can arrise, though, with labels being masked by overlapping. One way to address this is to "dodge" them by adding linebreaks, in R captured by slash n. We use the `paste0()` function

```
xbreaks <- as.numeric(quantile(iris$Sepal.Width))
xlabels <- paste0(c("", "\n", "", "\n", ""), xbreaks)
```

```
p3 <- ggplot(iris, aes(Sepal.Width, Sepal.Length)) +
    geom_point() + stat_smooth(method = "lm",
    se = FALSE) + scale_x_continuous(breaks = xbreaks,
    labels = xlabels) + scale_y_continuous(breaks = as.numeric(quantile(iris$Sepal.Length))) +
    facet_wrap(~Species)

print(p3)
```

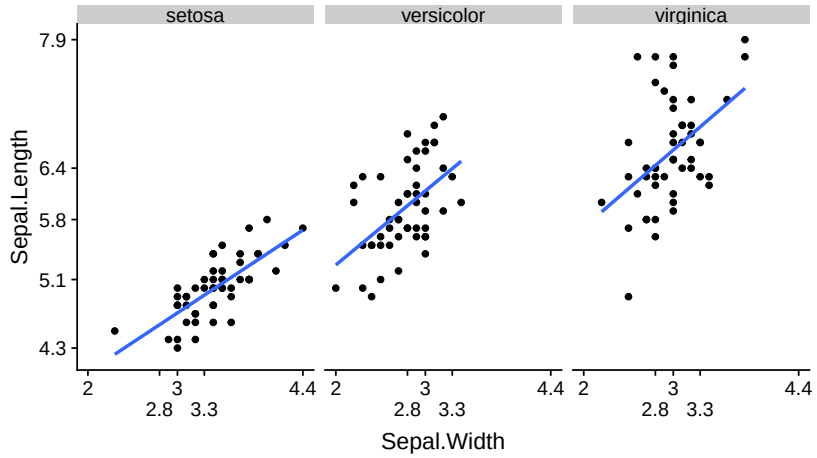

Figure 21: Cleaner small multiples plot.

We might get the average sepal length and width by species. Instead of normal based confidence intervals, if we are worried about non-normal data, we can base confidence intervals off bootstrapping.

```
p4a <- ggplot(iris, aes(Species, Sepal.Length)) +
    stat_summary(fun.data = mean_cl_boot)
p4b <- ggplot(iris, aes(Species, Sepal.Width)) +
    stat_summary(fun.data = mean_cl_boot)

p4grid <- plot_grid(p4a, p4b, nrow = 1, labels = c("F",
    "G"))

print(p4grid)
```

Finally, we could put all of these together into one large panel plot. You can nest `plot_grid()` calls within each other and if they are saved, as we have, just re-use the pieces. To see it well, you might need to "zoom" in in RStudio on the plot.

```
plot_grid(p1grid, p2grid, plot_grid(p3, labels = "E"),
    p4grid, nrow = 4)
```

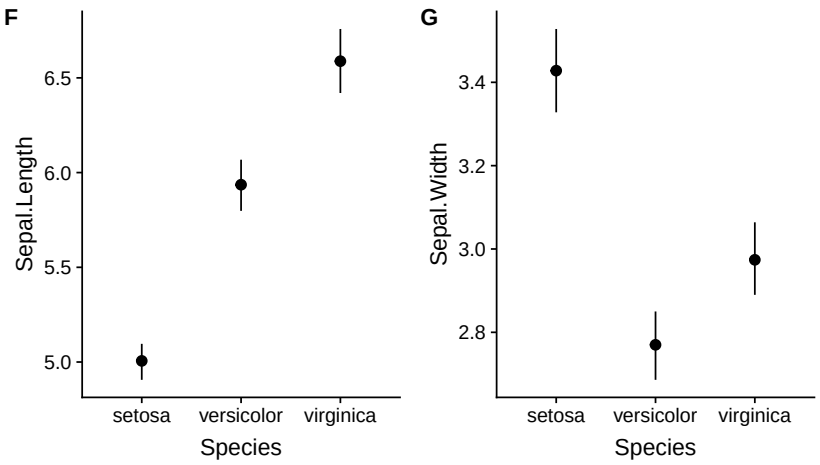

Figure 22: Summary plots.

4 *You Try It*

Use the `plot_grid()` function to combine all the graphs into one large panel. You should have seven graphs including each of the following:

- Distribution of "hp" (cont) broken down by "vs" (binary)
- Distribution of "mpg" (cont) broken down by "vs" (binary)
- Univariate distribution of "vs" (binary; frequency count)
- Bivariate distribution of "mpg" (cont) and "hp" (cont)
- Scatter plot of "mpg" (cont) and "hp" (cont) broken down by "vs" (binary) with linear trend (smooth) line
- Mean and confidence interval for "mpg" (cont) at each level of "vs" (binary)
- Mean and confidence interval for "hp" (cont) at each level of "vs" (binary)

Your overall large panel plot should be similar to the large panel plot we made in lecture.

```
## build up your individual plots and save them
## here


## combine them all using one or more
## plot_grid() calls don't forget to label
## them!
```

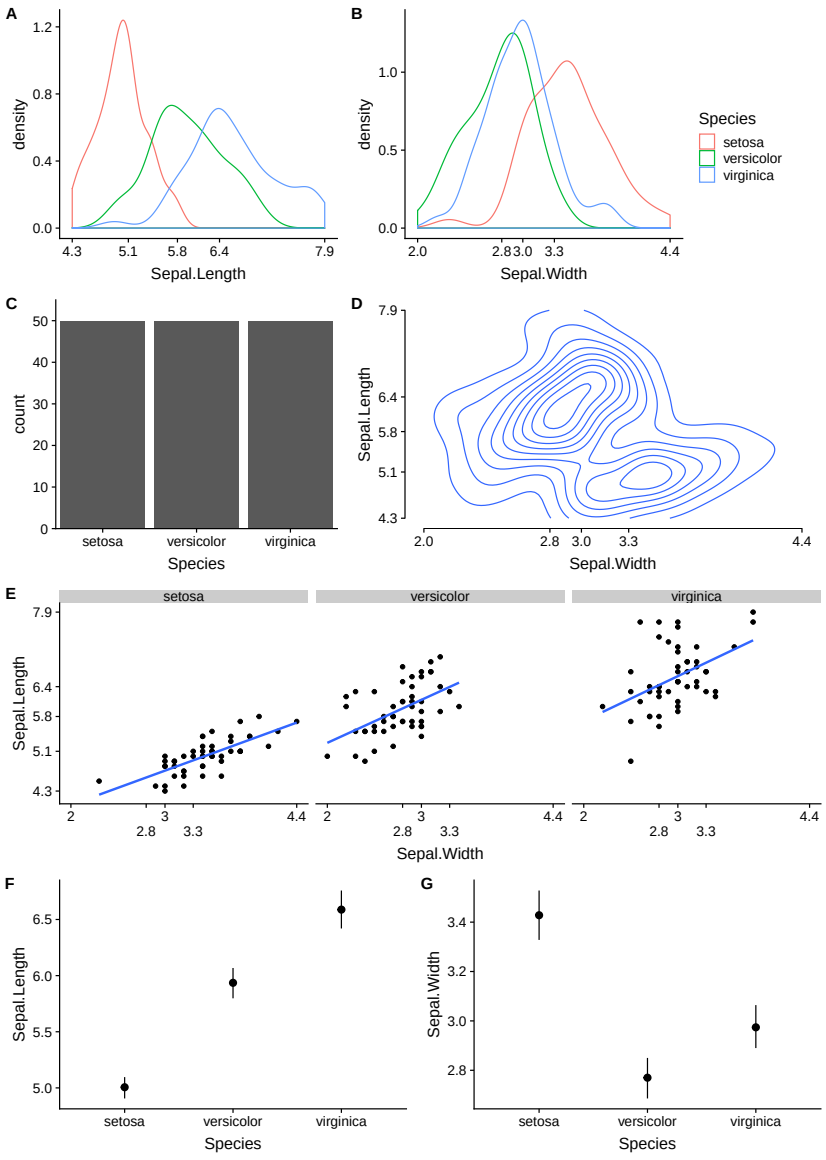

Figure 23: Large panel plot.