



Lecture 3

Data Visualization with ggplot2

Dr. Abbas Maarooof
aimaarooof@gmail.com
AITRS-3 June 2021

Reference: R for Data Science by Garrett Grolemund, Hadley Wickham, Chapter 1. Data Visualization with ggplot2

Introduction to ggplot2

What is ggplot2 ?

ggplot2 is a powerful R package, implemented by Hadley Wickham, for producing nice graphs

One of the strengths of R is that it's more than just a programming language — it also has thousands of packages written and contributed by independent developers. One of these packages, *ggplot2*, is tremendously popular and offers a new way of creating insightful graphics using R.

Much of the ggplot2 philosophy is based on the so-called “*grammar of graphics*,” a theoretically sound way of describing all the components that go into a graphical plot. You don't need to know anything about the grammar of graphics to use ggplot2 effectively, but now you know where its name comes from.

Grammar of graphics, and ggplot and ggplot2 are implementations of that grammar of graphics. The basic idea here is to separate what is graphed. That is, the actual data behind it, from how it is graphed

Structure of ggplot2 Commands

```
ggplot(data = <DATA>) +  
  <GEOM_FUNCTION>(  
    mapping = aes(<MAPPINGS>),  
    stat = <STAT>,  
    position = <POSITION>  
  ) +  
  <COORDINATE_FUNCTION> +  
  <FACET_FUNCTION>
```

Installing and Loading ggplot2

Looking At Layers

The basic concept of a ggplot2 graphic is that you combine different elements into layers. Each layer of a ggplot2 graphic contains information about the following:

- **The data that you want to plot:** For `ggplot()`, this must be a data frame.
- **A mapping from the data to your plot:** This usually is as simple as telling `ggplot()` what goes on the x-axis and what goes on the y-axis. (In the *“Mapping data to plot aesthetics”* section, later in this part, we explain how to use the **`aes()` function** to set up the mapping.)
- **A geometric object, or geom in ggplot terminology:** The geom defines the overall look of the layer (for example, whether the plot is made up of bars, points, or lines).
- **A statistical summary, called a stat in ggplot:** This describes how you want the data to be summarized (for example, binning for histograms, or smoothing to draw regression lines).

Now, `ggplot2` also include something called `qplot`, which stands for `quick plot`. These are commands that are quicker to work with. They're easy, they're fast, but they do have less power and control

- 1.install and load the ggplot2 package and then take a first look at layers, the building blocks of the ggplot2 graphics.
- 2.you define the data, geoms, and stats that make up a layer, and use these to create some plots.
- 3.you take full control over your graphics by adding facets and scales as well as controlling other plot options, such as adding labels and titles.

Now, I want to give you a few other resources for ggplot2.

1. One is the actual ggplot2 page on tidyverse.org, which explains a little bit about how to install it and gives a link to some other information.

<https://ggplot2.tidyverse.org/>

1. One thing you might want to look at is this page, which is ggplot2 extensions.

<https://exts.ggplot2.tidyverse.org/gallery/>

2. One thing you might want to look at is this page, which is ggplot2 extensions.

<https://exts.ggplot2.tidyverse.org/gallery/>

These are other packages that build onto and connect with the functionality of ggplot. They allow you to do some impressive things, like animations or simple things, like modifying where the labels appear. There are so many possibilities, and obviously, this is where you can see the power of ggplot because it lets you specify things at such a micro level. It enables enormous creativity in the exploration and the presentation of your data.

Finally, I want you to be aware of the cheat sheets that are available through our studio because the people who have developed ggplot2, Hadley Wickham in particular, works at studio his is a downloadable PDF, which can give you a list of commands including the over 40 different geometric objects and how you can specify some of the commands for working in ggplot. So these are resources that are available to you

<https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf>

<https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf>

install.packages

Each R package is hosted at <http://cran.r-project.org>, the same website that hosts R.

However, you don't need to visit the website to download an R package; you can download packages straight from R's command line. Here's how:

1. Open RStudio.
2. Make sure you are connected to the Internet.
3. Run `install.packages("ggplot2")` at the command line (console)

Or using the menu bar –Tools-Install Packages
(easy and fast way to install packages)

Installing Packages:

Open R and type the following into the command line:

```
install.packages("package name")
```

Loading Packages:

Installing a package doesn't immediately place its functions at your fingertips. It just places them on your computer. To use an R package, you next have to load it in your R session with the command:

```
library(package name)
```

Updating R Packages:

For example if you already have ggplot2, reshape2, and dplyr on your computer, it'd be a good idea to check for updates before you use them:

```
update.packages(c("ggplot2", "reshape2", "dplyr"))
```

Library

Installing a package doesn't place its functions at your fingertips just yet: it simply places them in your hard drive. To use an R package, you next have to load it in your R session with the command `library("ggplot2")`. If you would like to load a different package, replace ggplot2 with your package name in the code.

If you could not manage to download ggplot2 package . We can use ***Plot()*** instead.

Installation:

The easiest way to get ggplot2 is to install the whole tidyverse:

```
install.packages("tidyverse")
```

Alternatively, install just ggplot2:

```
install.packages("ggplot2")
```

Or the development version from GitHub:

```
# install.packages("devtools")
```

```
devtools::install_github("tidyverse/ggplot2")
```

How to install a package for the first time with the `install.packages()` function and to load the package at the start of each R session with the `library()` function.

To install the `ggplot2` package, use the following:

```
> install.packages("ggplot2")
```

And then to load it, use the following:

```
> library("ggplot2")
```

More Data Visualization References for R

If you want to get started with visualizations in R, take some time to study the [ggplot2](#) package. One of the (if not the) most famous packages in R for creating graphs and plots. ggplot2 makes intensive use of the [grammar of graphics](#), and as a result is very intuitive in usage (you're continuously building part of your graphs so it's a bit like playing with lego). There are tons of resources to get you started such as this

https://www.datacamp.com/courses/data-visualization-with-ggplot2-1?tap_a=5644-dce66f&tap_s=14201-e863d5

Besides ggplot2 there are multiple other packages that allow you to create highly engaging graphics and that have good learning resources to get you up to speed. Some of our favourites are:

- [ggvis](#) for interactive web graphics

<http://ggvis.rstudio.com/>

- [googleVis](#) to interface with google charts.

<https://developers.google.com/chart/interactive/docs/gallery>

- [Plotly for R](#)

<https://plotly.com/r/>

Using Colors in R

R uses color names for 657 different names, there are actually just about 500 unique colors, and they're arranged alphabetically.

?colors

Let's get a list of the color names

colors()

you can see that we've got a lot of different names, and they're just alphabetical.

Or we can use a resource that I've compiled and it's available for free on the web at this address

<https://datalab.cc/rcolors>

Let me show you what that looks like. What this website has is all the different ways that you can identify or call on colors in R, and it's available right here as an embedded spreadsheet where you see the color samples, and the numbers, and so on and so forth, scroll across. You can open this in your web browser in Google Sheets, you can download it as an Excel file or download it as a PDF, and that's just free and available for anyone

Exercise (1): Let's make a bar plot with different colors

```
library("ggplot2")
```

Color names

```
barplot(x, col = "skyblue") # skyblue
```

```
barplot(x, col = "linen") # linen
```

RGB triplets (0.00-1.00)

```
barplot(x, col = rgb(.52, .80, .92)) # skyblue
```

```
barplot(x, col = rgb(.98, .94, .90)) # linen
```

RGB triplets (0-255)

```
barplot(x, col = rgb(135, 206, 235, max = 255)) # skyblue
```

```
barplot(x, col = rgb(250, 240, 230, max = 255)) # linen
```

RGB hexcodes

```
barplot(x, col = "#87CEEB") # skyblue
```

```
barplot(x, col = "#FAF0E6") # linen
```

Index numbers

```
barplot(x, col = colors() [589]) # skyblue
```

```
barplot(x, col = colors() [449]) # linen
```

MULTIPLE COLORS

Can specify several colors in a vector, which will cycle

```
barplot(x, col = c("skyblue", "linen"))
```

```
barplot(x, col = c("#FAF0E6", "#87CEEB"))
```

Using color palettes

#INSTALL AND LOAD PACKAGES

Load base packages manually

library(datasets) # For example datasets

Install pacman ("package manager") if needed

if (!require("pacman")) install.packages("pacman")

pacman must already be installed; then load contributed

packages (including pacman) with pacman

pacman::p_load(datasets, pacman, rio, tidyverse)

datasets: for demonstration purposes

pacman: for loading/unloading packages

rio: for importing data

tidyverse: for so many reasons

Exercise (1): Let's make a bar plot with different color palettes

```
# LOAD DATA
```

```
x <- c(24, 13, 7, 5, 3, 2) # Sample data
```

```
barplot(x) # Default barplot
```

```
# BUILT-IN COLOR PALETTES
```

```
?palette # Info on palettes
```

```
palette() # See current palette
```

```
barplot(x, col = 1:6) # Use current palette
```

```
barplot(x, col = rainbow(6)) # Rainbow colors
```

```
barplot(x, col = heat.colors(6)) # Yellow through red
```

```
barplot(x, col = terrain.colors(6)) # Gray through green
```

```
barplot(x, col = topo.colors(6)) # Purple through tan
```

```
barplot(x, col = cm.colors(6)) # Pinks and blues
```

We have more choices of colors packages

RCOLORBREWER package

```
browseURL("http://colorbrewer.org/")
```

```
p_load(RColorBrewer)
```

```
?RColorBrewer
```

```
display.brewer.all() # Show all palettes
```

```
# SEQUENTIAL PALETTES: Blues, BuGn, BuPu, GnBu, Greens,  
# Greys, Oranges, OrRd, PuBu, PuBuGn, PuRd, Purples, RdPu,  
# Reds, YlGn, YlGnBu, YlOrBr, YlOrRd
```

```
display.brewer.pal(7,"BuPu")
```

```
# DIVERGING PALETTES: BrBG, PiYG, PRGn, PuOr, RdBu, RdGy,  
# RdYlBu, RdYlGn, Spectral
```

```
display.brewer.pal(5,"BrBG")
```

```
# QUALITATIVE PALETTES: Accent (8), Dark2 (8), Paired (12),  
# Pastel1 (9), Pastel2 (8), Set1 (9), Set2 (8), Set3 (12)
```

```
# (All sets require at least three groups)
```

```
display.brewer.pal(4,"Paired")
```

```
barplot(x, col = 1:6) # Default palette
```

```
barplot(x, col = brewer.pal(6,"BuPu")) # Sequential
```

```
barplot(x, col = brewer.pal(6,"PuOr")) # Diverging
```

```
barplot(x, col = brewer.pal(6,"Set3")) # Qualitative
```

WESANDERSON

```
browseURL("https://github.com/karthik/wesanderson")  
p_load(wesanderson)  
?wesanderson  
names(wes_palettes)
```

```
barplot(x, col = wes_palette("BottleRocket1"))  
barplot(x, col = wes_palette("Zissou1"))  
barplot(x, col = wes_palette("GrandBudapest2"))  
barplot(x, col = wes_palette("IsleofDogs1"))
```

OTHER PALETTE PACKAGES

```
# The viridis color palettes  
browseURL("http://bit.ly/2tFEqKe")
```

```
# Scientific Journal and Sci-Fi Themed Color Palettes  
browseURL("http://bit.ly/2NXxlpT")
```

CUSTOM PALETTES - You can have you customise colors

Can specify colors with names

```
palette1 <- c("lightcyan", "orange2", "salmon", "tan")
```

Can specify colors with hex codes (or other methods)

```
palette2 <- c("#D2B48C", "#FA8072", "#EE9A00", "#E0FFFF")
```

```
barplot(x, col = palette1)
```

```
barplot(x, col = palette2)
```

In this lecture will teach you how to visualize your data using **ggplot2**. R has several systems for making graphs, but ggplot2 is one of the most elegant and most versatile. ggplot2 implements the grammar of graphics, a coherent system for describing and building graphs.

With ggplot2, you can do more faster by learning one system and applying it in many places.

If you'd like to learn more about the theoretical underpinnings of ggplot2 before you start, I'd recommend reading "A Layered Grammar of Graphics".

Prerequisites

That one line of code loads the core tidyverse, packages that you will use in almost every data analysis. It also tells you which functions from the tidyverse conflict with functions in base R (or from other packages you might have loaded).

R for Data Science by Garrett Grolemund, Hadley Wickham Search... If you run this code and get the error message “there is no package called ‘tidyverse’,” you’ll need to first install it, then run `library()` once again:

```
install.packages("tidyverse")  
library(tidyverse)
```

You only need to install a package once, but you need to reload it every time you start a new session.

Create plots with {ggplot2}

In the following sections we will show how to draw the following plots:

- scatter plot
- line plot
- histogram
- density plot
- boxplot
- barplot

Using Geoms and Stats

To create a `ggplot2` graphic, you have to explicitly tell the function what's in each of the components of the layer. In other words, you have to tell the `ggplot()` function your data, the mapping between your data and the `geom`, and then either a `geom` or a `stat`.

Data

To illustrate plots with the `{ggplot2}` package we will use the mpg dataset available in the package. The dataset contains observations collected by the US Environmental Protection Agency on fuel economy from 1999 to 2008 for 38 popular models of cars (run `?mpg` for more information about the data):

```
library(ggplot2)
dat <- ggplot2::mpg
```

```
> mpg
# A tibble: 234 x 11
  manufacturer model      displ  year   cyl trans      drv    cty   hwy fl    class
  <chr>         <chr>    <dbl> <int> <int> <chr>    <chr> <int> <int> <chr> <chr>
1 audi         a4          1.8   1999     4 auto(l5)  f       18     29 p     compact
2 audi         a4          1.8   1999     4 manual(m5) f       21     29 p     compact
3 audi         a4          2     2008     4 manual(m6) f       20     31 p     compact
4 audi         a4          2     2008     4 auto(av)   f       21     30 p     compact
5 audi         a4          2.8   1999     6 auto(l5)  f       16     26 p     compact
6 audi         a4          2.8   1999     6 manual(m5) f       18     26 p     compact
7 audi         a4          3.1   2008     6 auto(av)   f       18     27 p     compact
8 audi         a4 quattro  1.8   1999     4 manual(m5) 4       18     26 p     compact
9 audi         a4 quattro  1.8   1999     4 auto(l5)   4       16     25 p     compact
10 audi         a4 quattro  2     2008     4 manual(m6) 4       20     28 p     compact
# ... with 224 more rows
>
```

This dataset contains a subset of the fuel economy data that the EPA makes available on <http://fueleconomy.gov>. It contains only models which had a new release every year between 1999 and 2008 - this was used as a proxy for the popularity of the car.

First Steps

Let's use our first graph to answer a question:

Do cars with big engines use more fuel than cars with small engines?

You probably already have an answer but try to make your answer precise.

What does the relationship between engine size and fuel efficiency look like? Is it positive? Negative? Linear? Nonlinear?

Hint: If we need to be explicit about where a function (or dataset) comes from, we'll use the special form `package::function()`.

The mpg Data Frame

You can test your answer with the mpg data frame found in ggplot2 (`ggplot2::mpg`).

A data frame is a rectangular collection of variables (in the columns) and observations (in the rows).

Mpg contains observations collected by the US Environment Protection Agency on 38 models of cars:

> mpg

A tibble: 234 × 11

	manufacturer	model	displ	year	cyl	trans	drv	cty	hwy
	<chr>	<chr>	<dbl>	<int>	<int>	<chr>	<chr>	<int>	<int>
1	audi	a4	1.8	1999	4	auto(l5)	f	18	29
2	audi	a4	1.8	1999	4	manual(m5)	f	21	29
3	audi	a4	2.0	2008	4	manual(m6)	f	20	31
4	audi	a4	2.0	2008	4	auto(av)	f	21	30
5	audi	a4	2.8	1999	6	auto(l5)	f	16	26
6	audi	a4	2.8	1999	6	manual(m5)	f	18	26
7	audi	a4	3.1	2008	6	auto(av)	f	18	27
8	audi	a4 quattro	1.8	1999	4	manual(m5)	4	18	26
9	audi	a4 quattro	1.8	1999	4	auto(l5)	4	16	25
10	audi	a4 quattro	2.0	2008	4	manual(m6)	4	20	28

... with 224 more rows, and 2 more variables: fl <chr>, class <chr>

>

>

Among the variables in mpg are:

- **displ**, a car's engine size, in liters.
- **hwy**, a car's fuel efficiency on the highway, in miles per gallon (mpg). A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance.

To plot mpg, run this code to put displ on the x-axis and hwy on the y-axis:

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy))
```

Exercises (1):

1. Run `ggplot(data = mpg)`. What do you see?
2. How many rows are in `mtcars`? How many columns?
3. What does the `drv` variable describe? Read the help for `?mpg` to find out.
4. Make a scatterplot of `hwy` versus `cyl`.
5. What happens if you make a scatterplot of `class` versus `drv`? Why is the plot not useful?

The plot shows a negative relationship between engine size (displ) and fuel efficiency (hwy). In other words, cars with big engines use more fuel. Does this confirm or refute your hypothesis about fuel efficiency and engine size?

A Graphing Template

Let's turn this code into a *reusable* template for making graphs with ggplot2. To make a graph, replace the bracketed sections in the following code with a dataset, a geom function, or a collection of mappings:

```
ggplot(data = <DATA>) + <GEOM_FUNCTION>(mapping =  
aes(<MAPPINGS>))
```

We will show you how to complete and extend this template to make different types of graphs. We will begin with the <MAPPINGS> component.

Aesthetic Mappings

Aesthetic Mappings

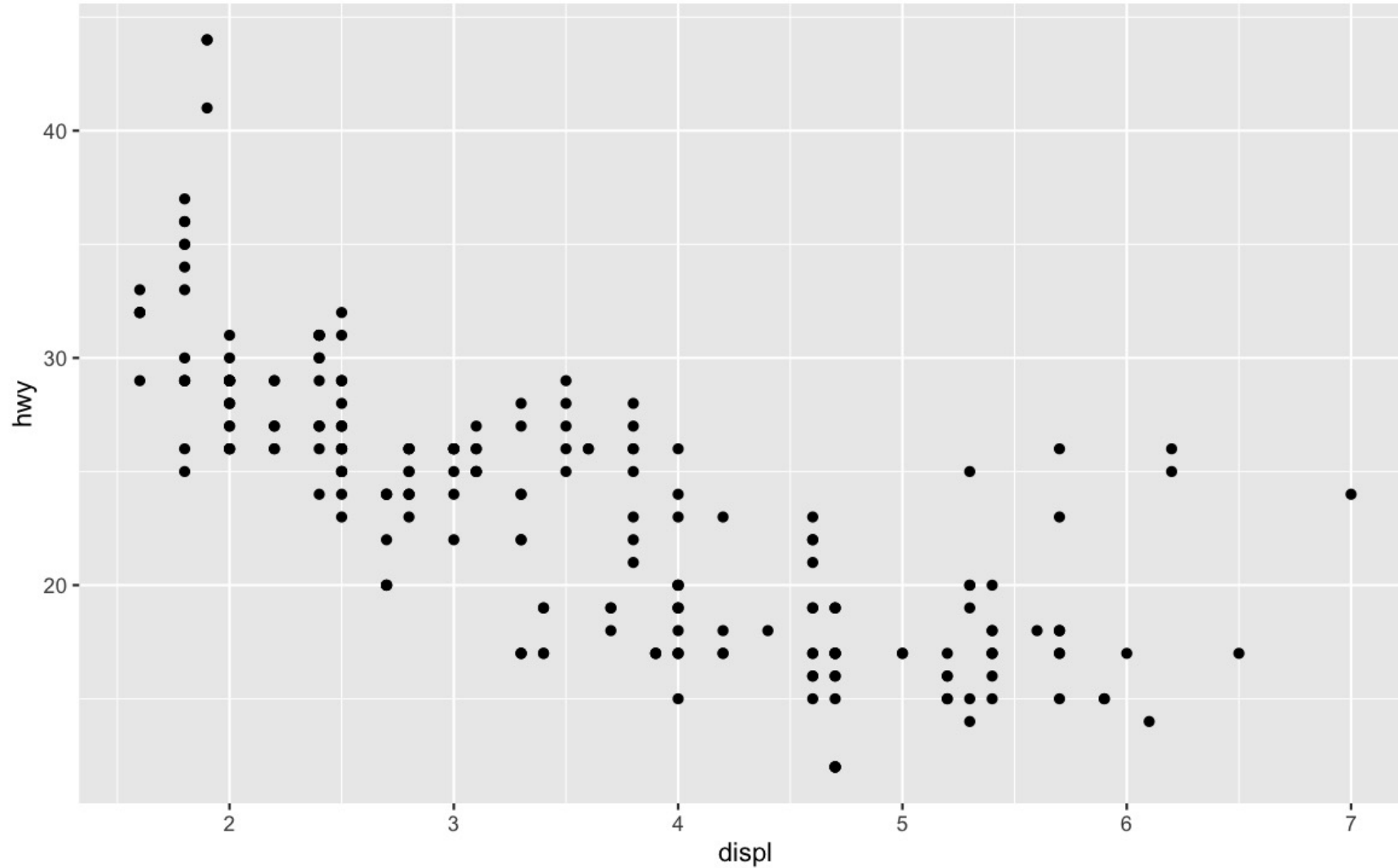
You can add a third variable, like class, to a two-dimensional scatterplot by mapping it to an aesthetic.

You can convey information about your data by mapping the aesthetics in your plot to the variables in your dataset. For example, you can map the colors of your points to the class variable to reveal the class of each car:

```
ggplot(mpg) + # data  
  aes(x = displ, y = hwy) + # variables  
  geom_point() # type of plot
```

Or

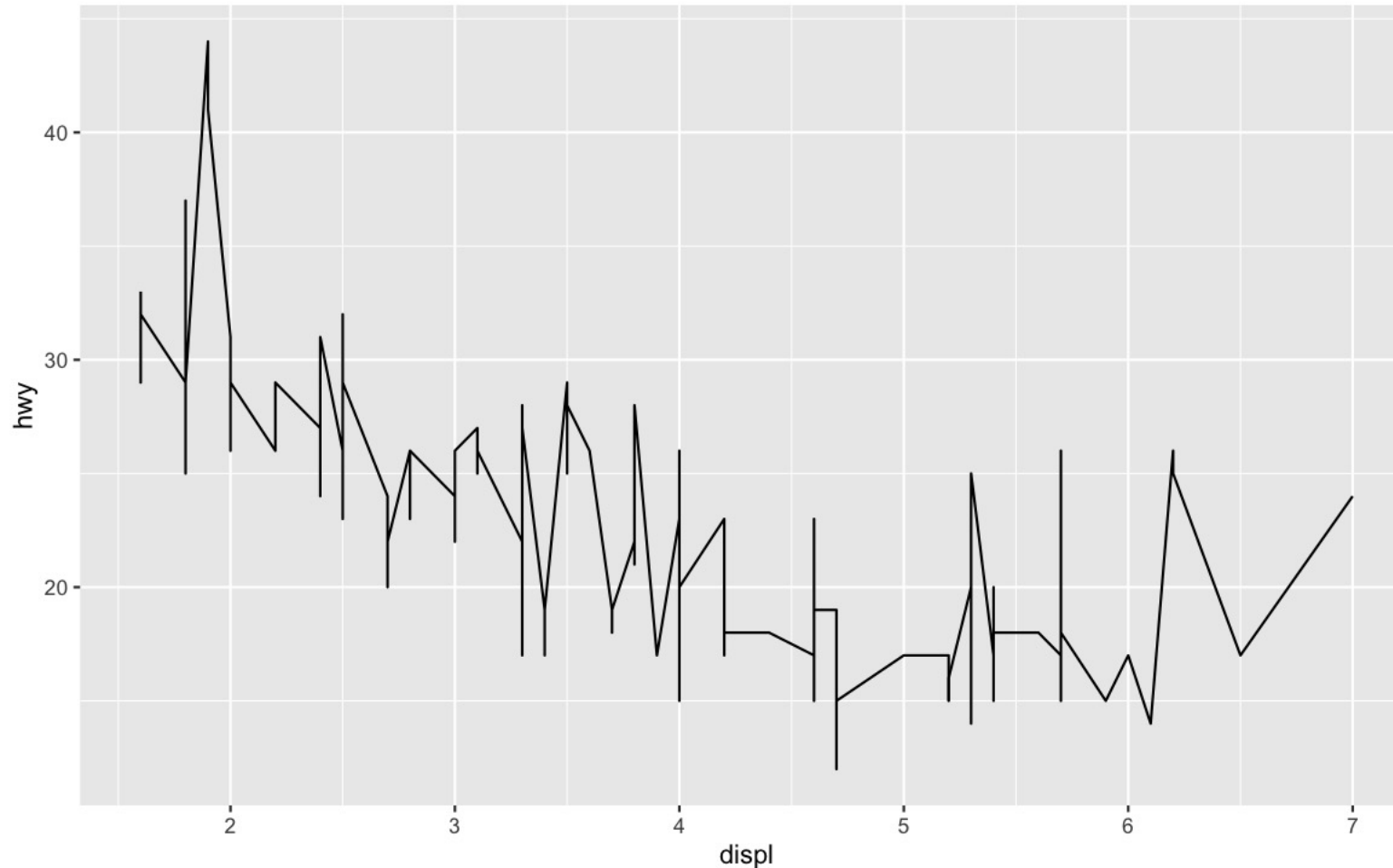
```
ggplot(mpg, aes(x = displ, y = hwy)) +  
  geom_point()
```



Scatter plot

[Line plots](#), particularly useful in time series or finance, can be created similarly but by using `geom_line()`:

```
ggplot(mpg) +  
  aes(x = displ, y = hwy) +  
  geom_line()
```

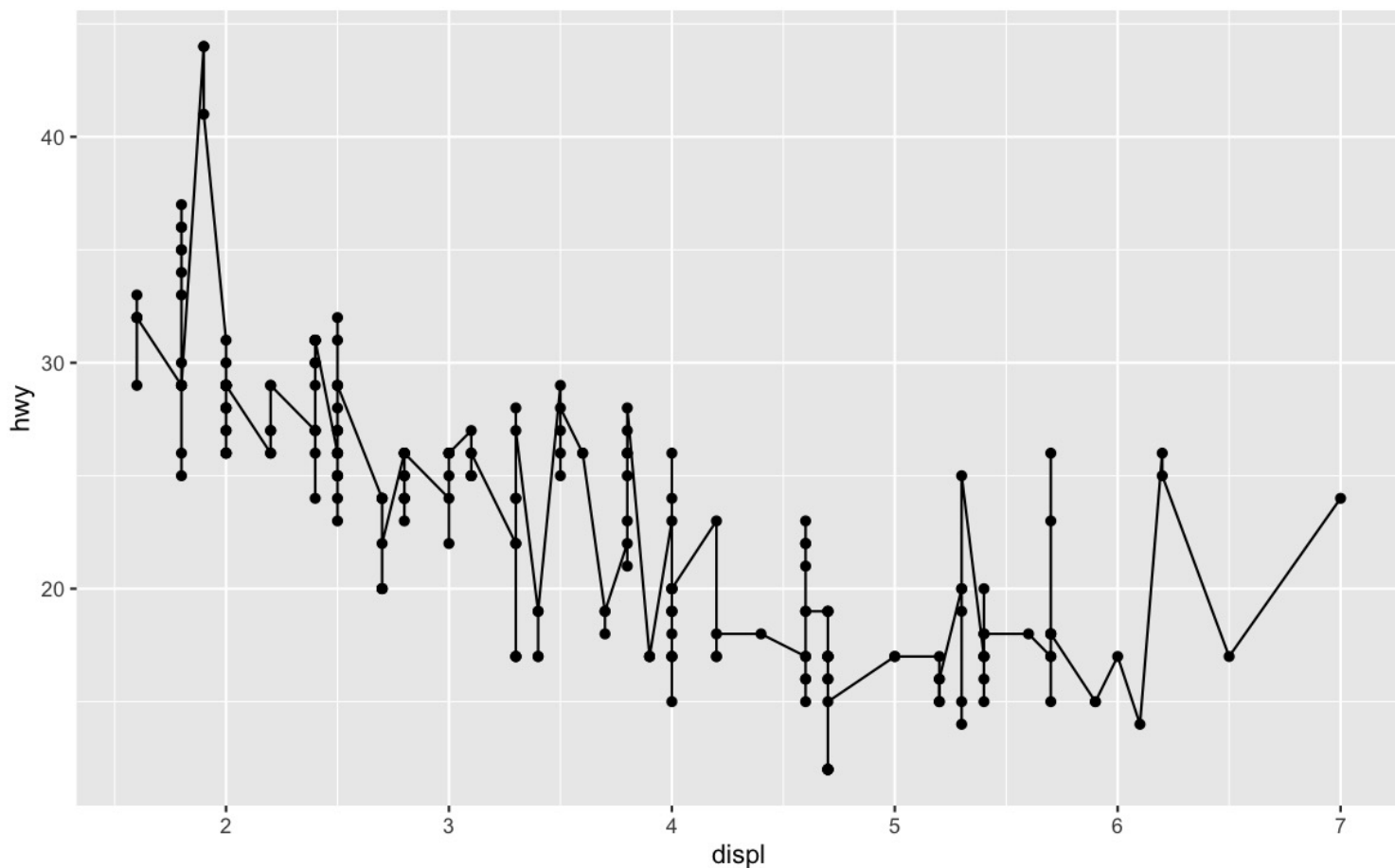


Line plot

Combination of line and points

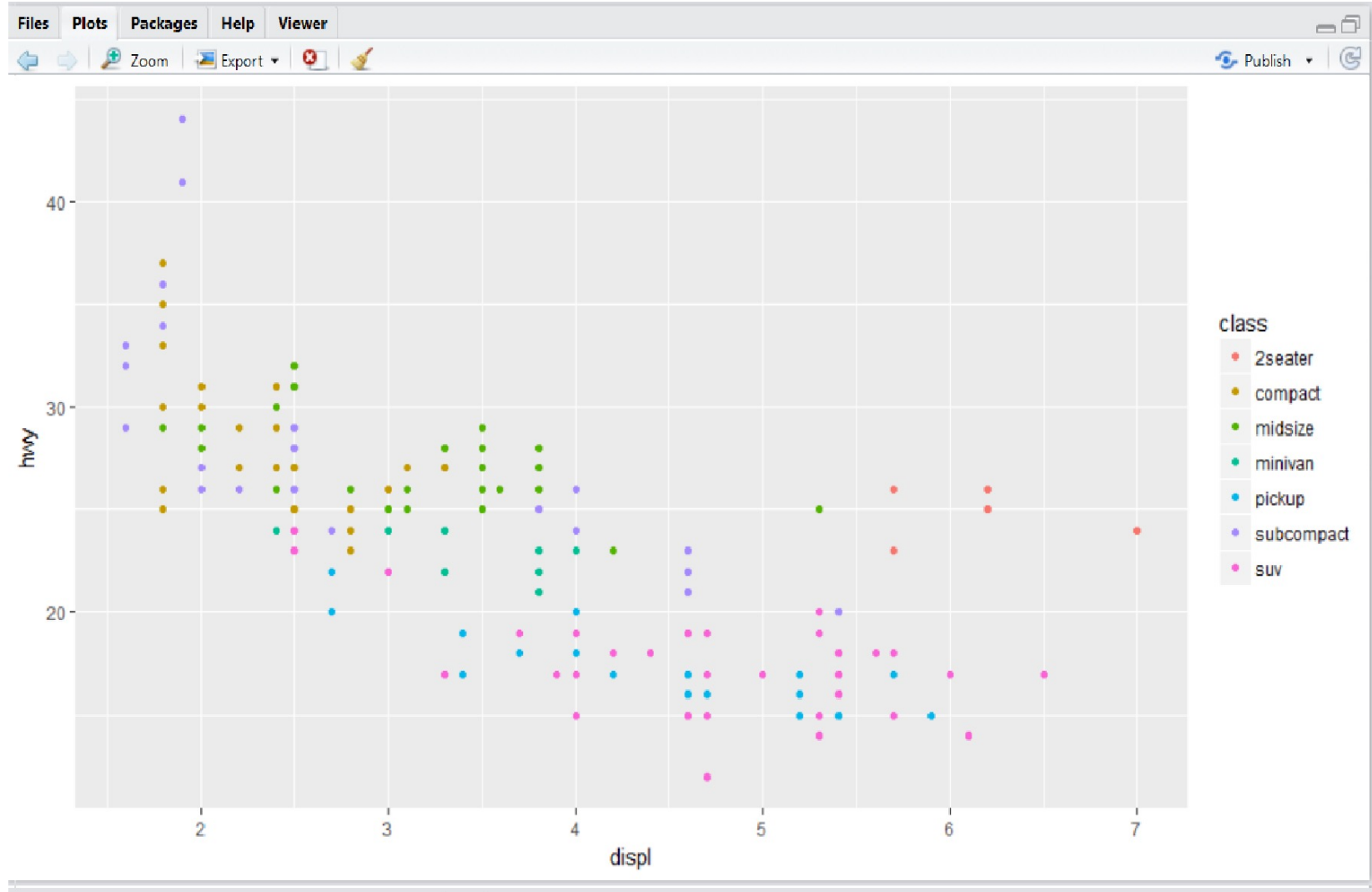
An advantage of {ggplot2} is the ability to combine several types of plots and its flexibility in designing it. For instance, we can add a line to a scatter plot by simply adding a layer to the initial scatter plot:

```
ggplot(mpg) +  
  aes(x = displ, y = hwy) +  
  geom_point() +  
  geom_line() # add line
```



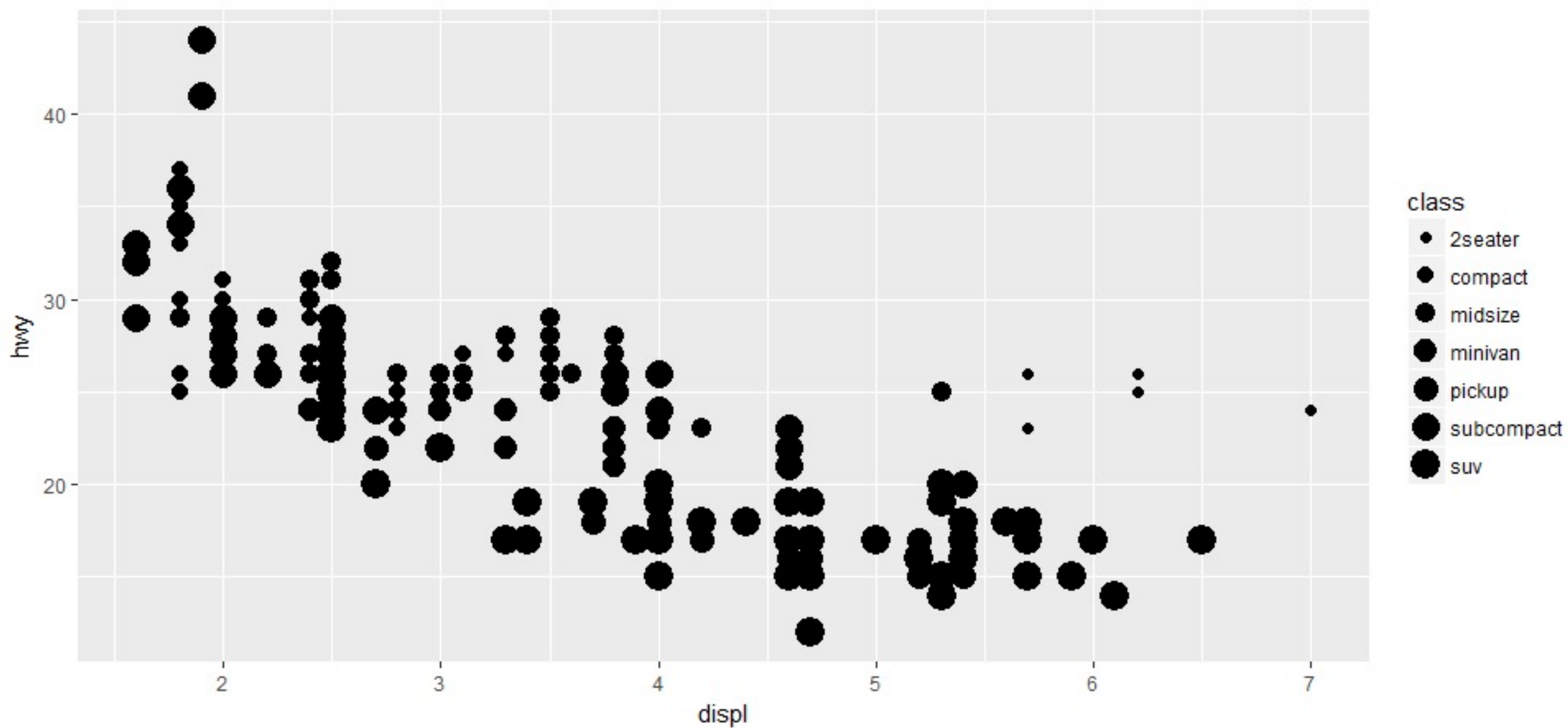
Combination
of line and
points

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



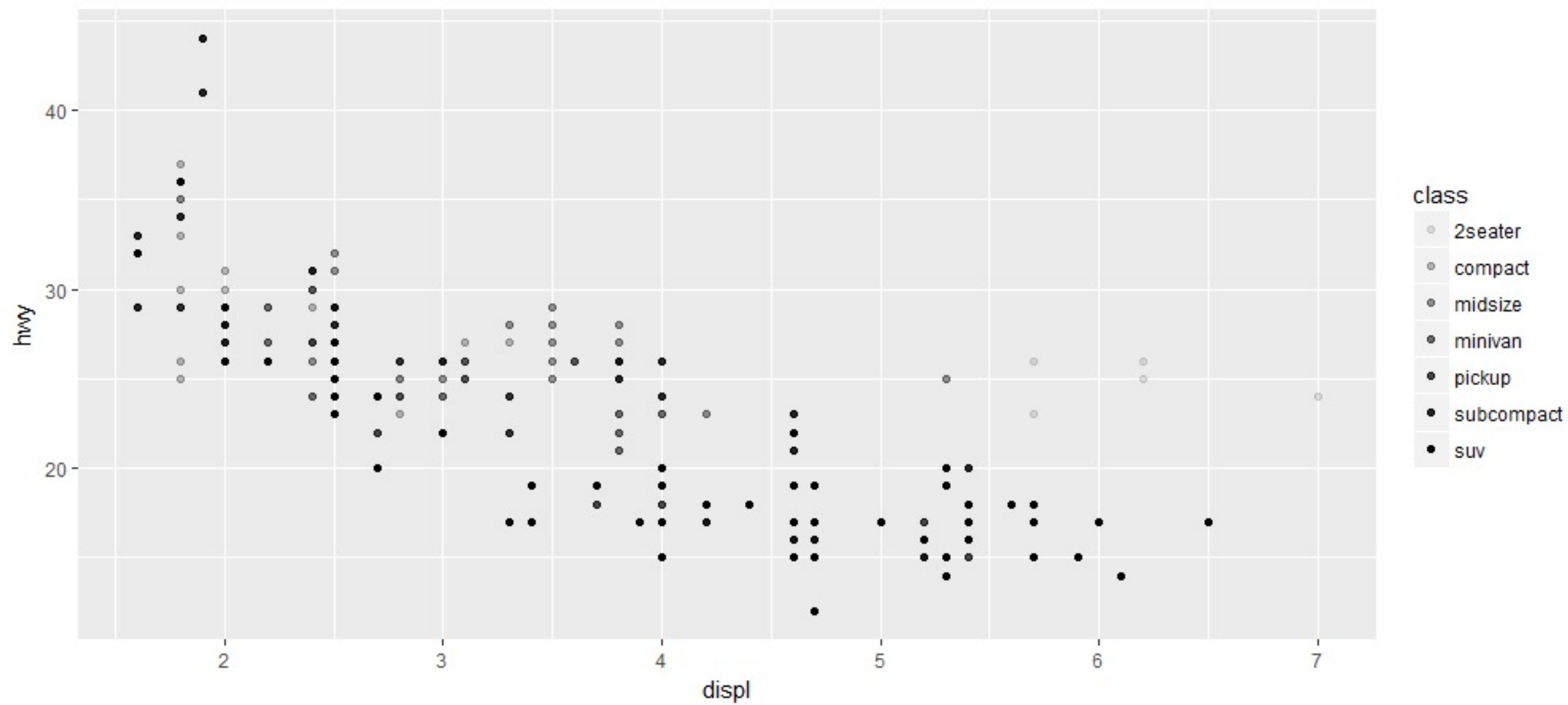
In the preceding example, we mapped class to the color aesthetic, but we could have mapped class to the **size aesthetic** in the same way. In this case, the exact size of each point would reveal its class affiliation. We get a warning here, because mapping an unordered variable (class) to an ordered aesthetic (size) is not a good idea:

```
ggplot(data = mpg) +  
geom_point(mapping = aes(x = displ, y = hwy, size = class))  
#> Warning: Using size for a discrete variable is not advised.
```



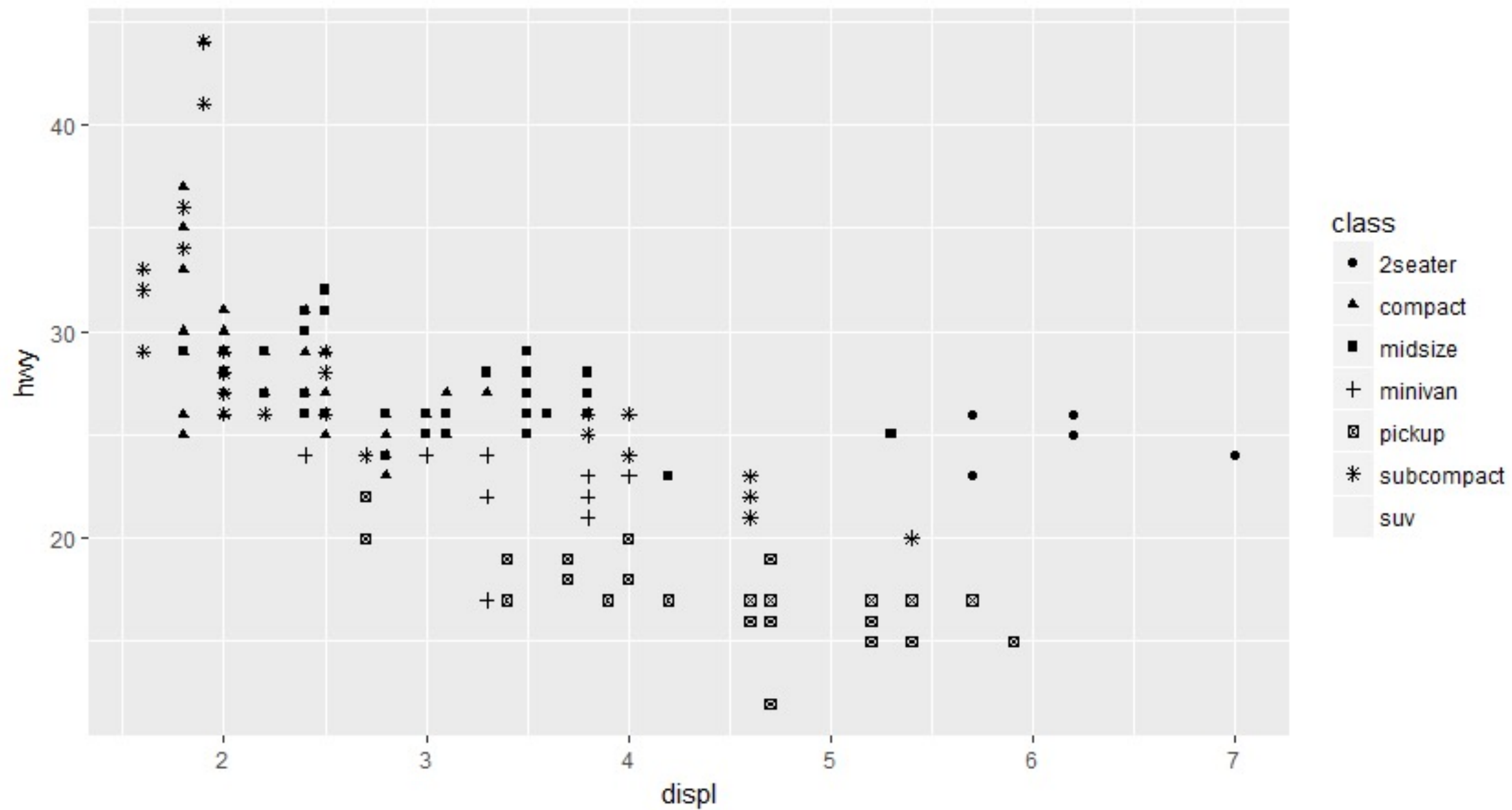
Or we could have mapped class to the alpha aesthetic, which controls the transparency of the points

```
ggplot(data = mpg) +  
geom_point(mapping = aes(x = displ, y = hwy, alpha = class))
```

or the shape of the points:

```
ggplot(data = mpg) +  
geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```



One common problem when creating ggplot2 graphics is to put the + in the wrong place: it must come at the end of the line, not the start. In other words, make sure you haven't accidentally written

```
ggplot(data = mpg)
```

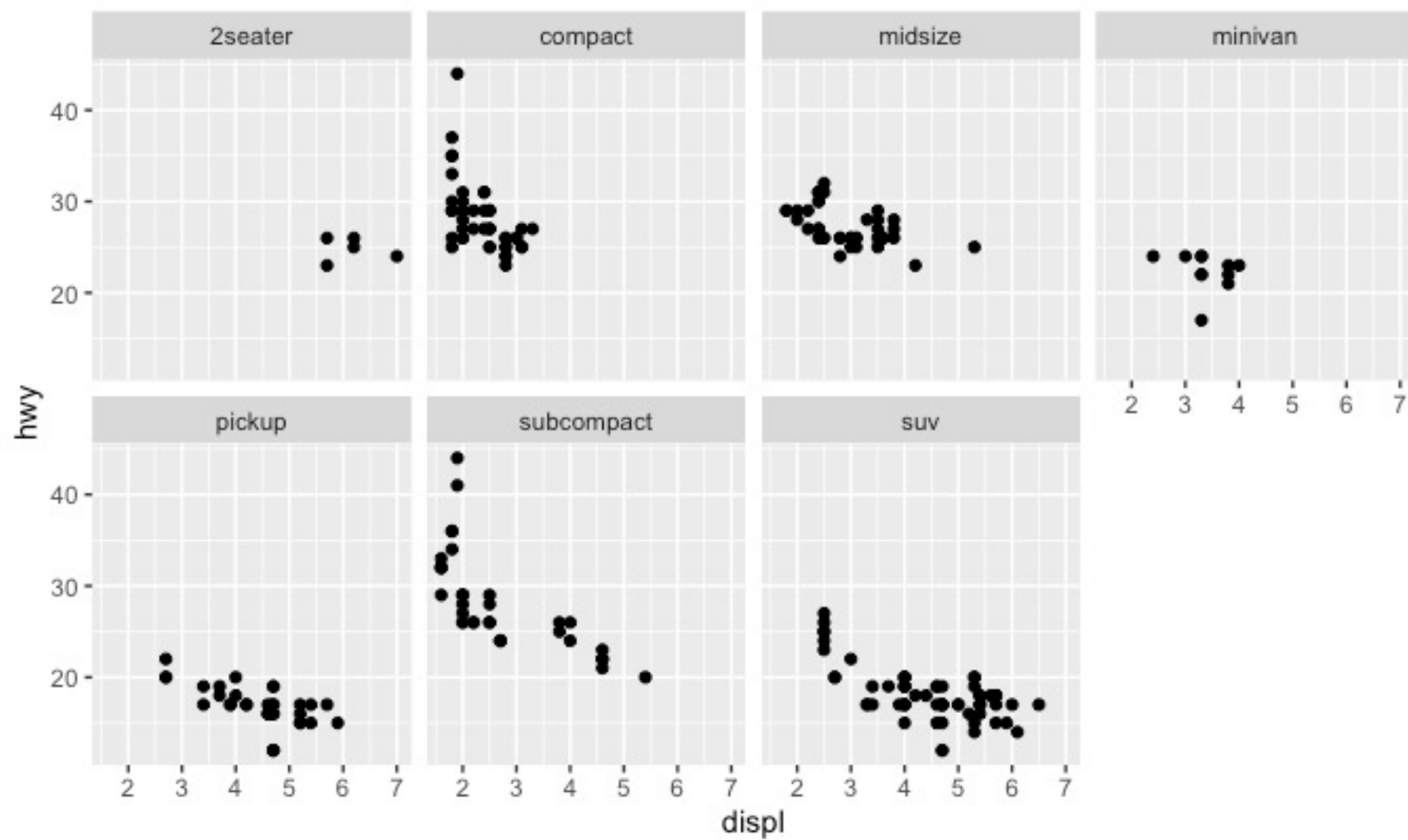
```
+ geom_point(mapping = aes(x = displ, y = hwy))
```

Facets

One way to add additional variables is with aesthetics. Another way, particularly useful for categorical variables, is to split your plot into facets, subplots that each display one subset of the

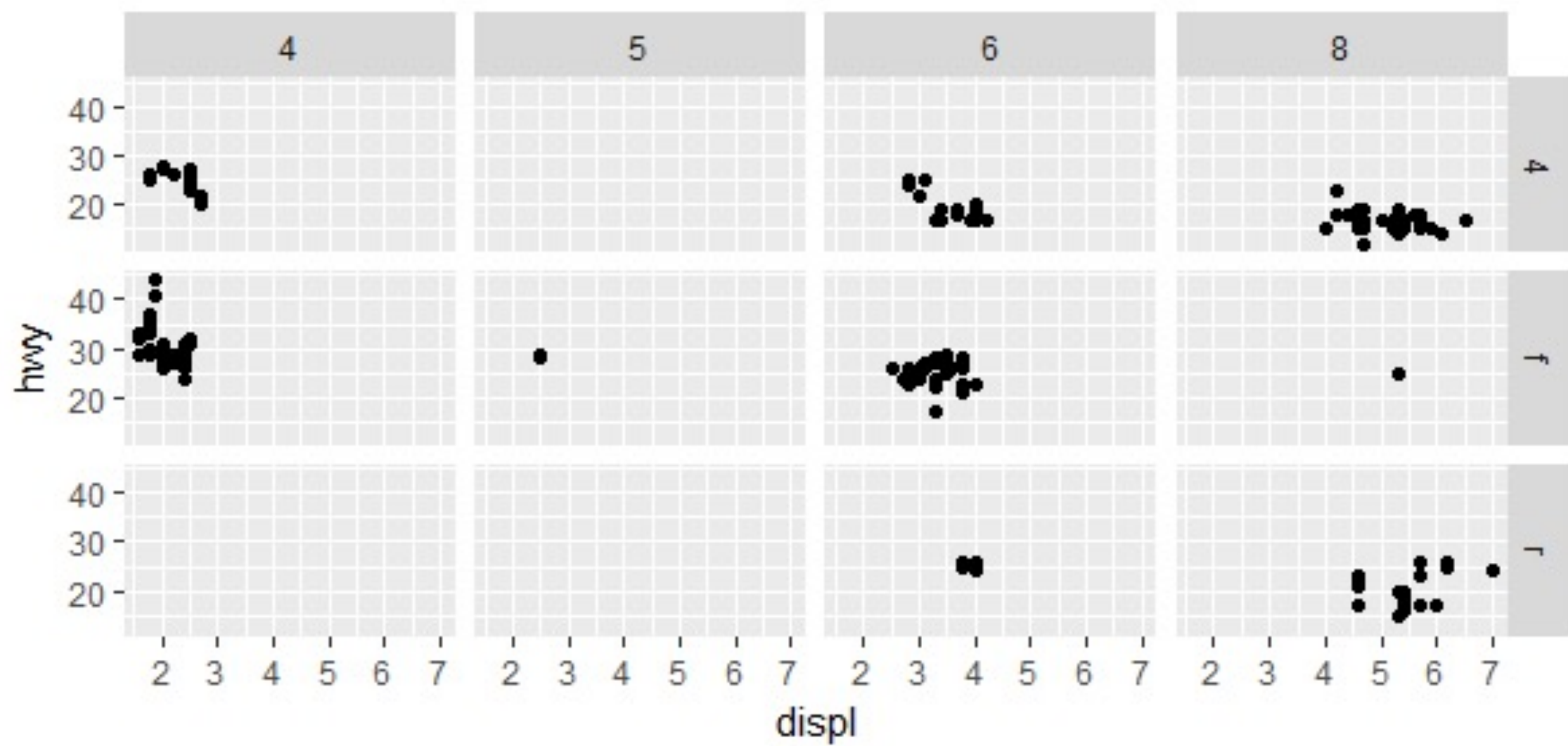
To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` should be a formula, which you create with `~` followed by a variable name (here “formula” is the name of a data structure in R, not a synonym for “equation”). The variable that you pass to `facet_wrap()` should be discrete:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  facet_wrap(~ class, nrow = 2)
```



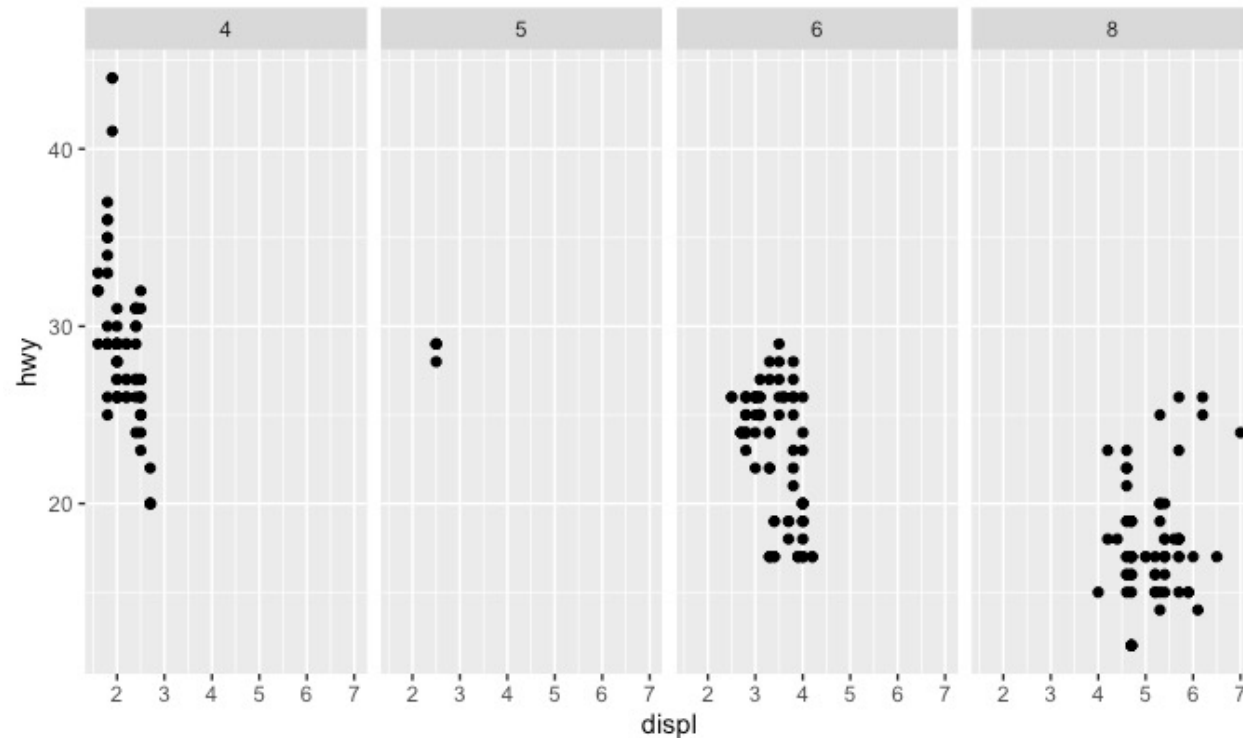
To facet your plot on the combination of two variables, add `facet_grid()` to your plot call. The first argument of `facet_grid()` is also a formula. This time the formula should contain two variable names separated by

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  facet_grid(drv ~ cyl)
```

If you prefer to not facet in the rows or columns dimension, use a . instead of a variable name, e.g.,
`+ facet_grid(. ~ cyl)`.

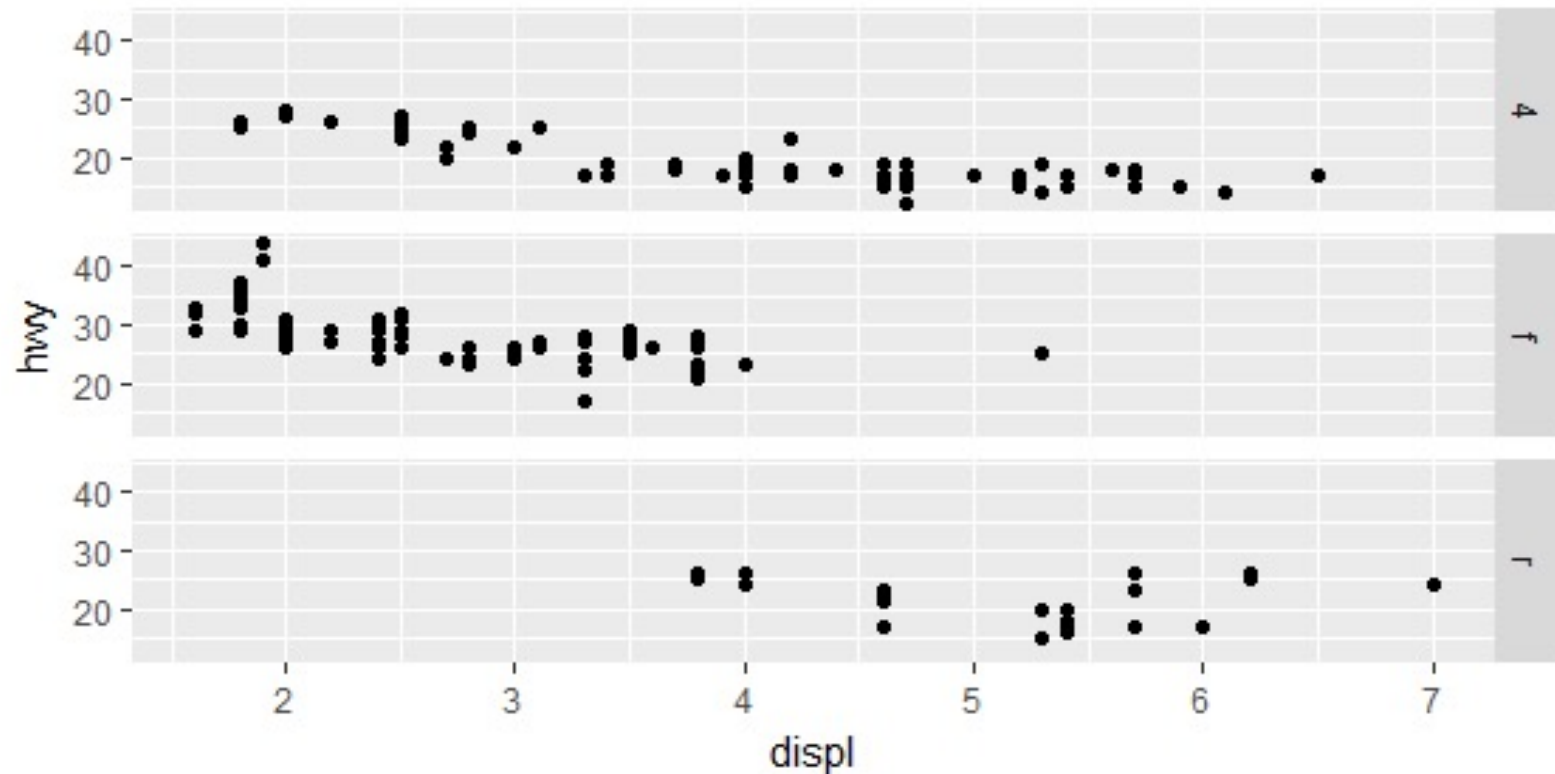
```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  facet_grid(. ~ cyl)
```



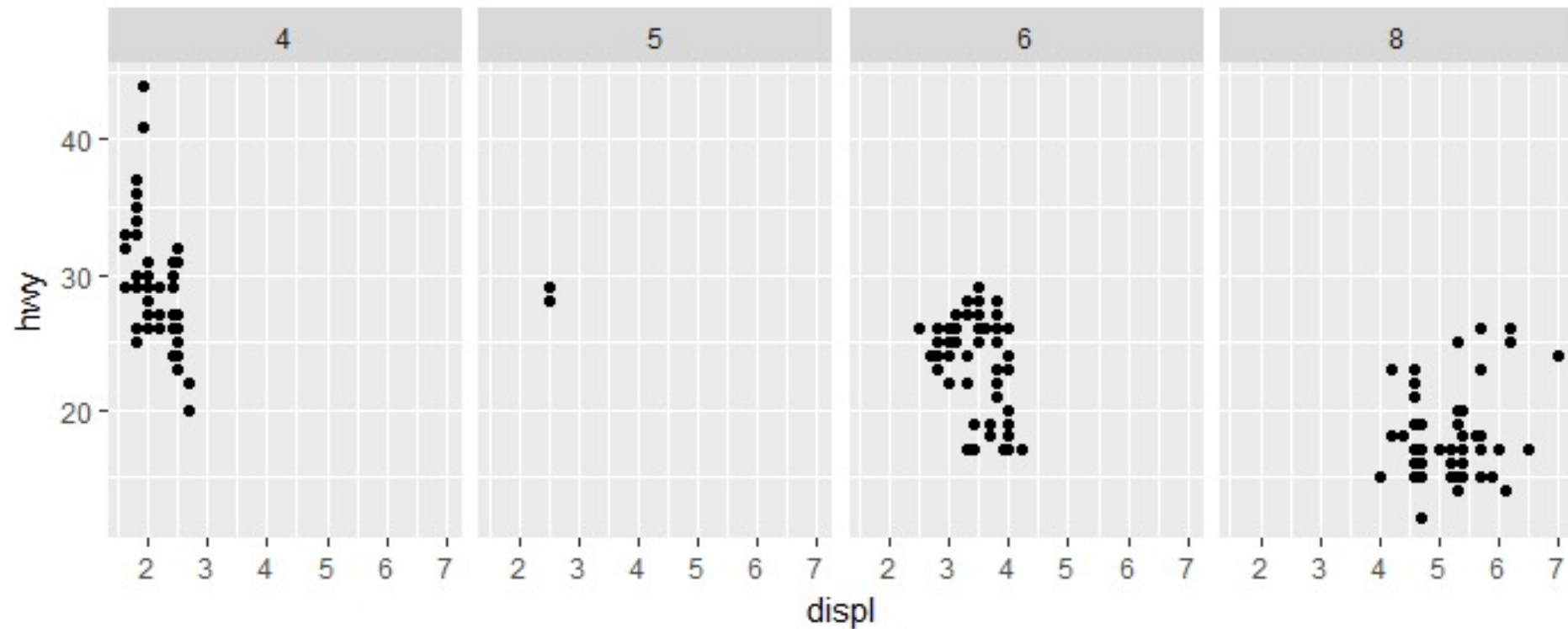
Exercises

1. What plots does the following code make?

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) +  
facet_grid(drv ~ .)
```

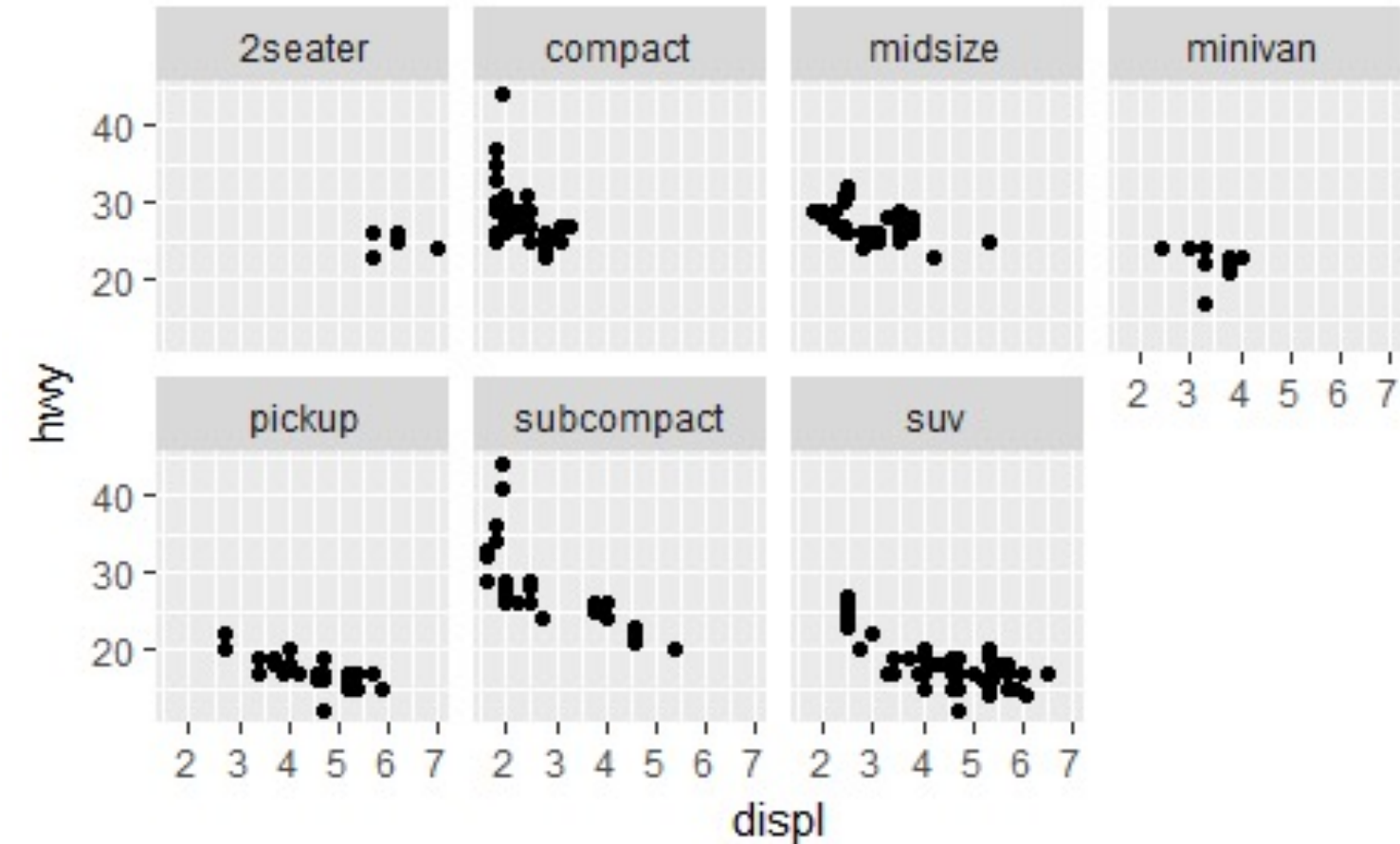


```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) +  
facet_grid(. ~ cyl)
```



2. Take the first faceted plot in this section:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  facet_wrap(~ class, nrow = 2)
```



Geometric Objects

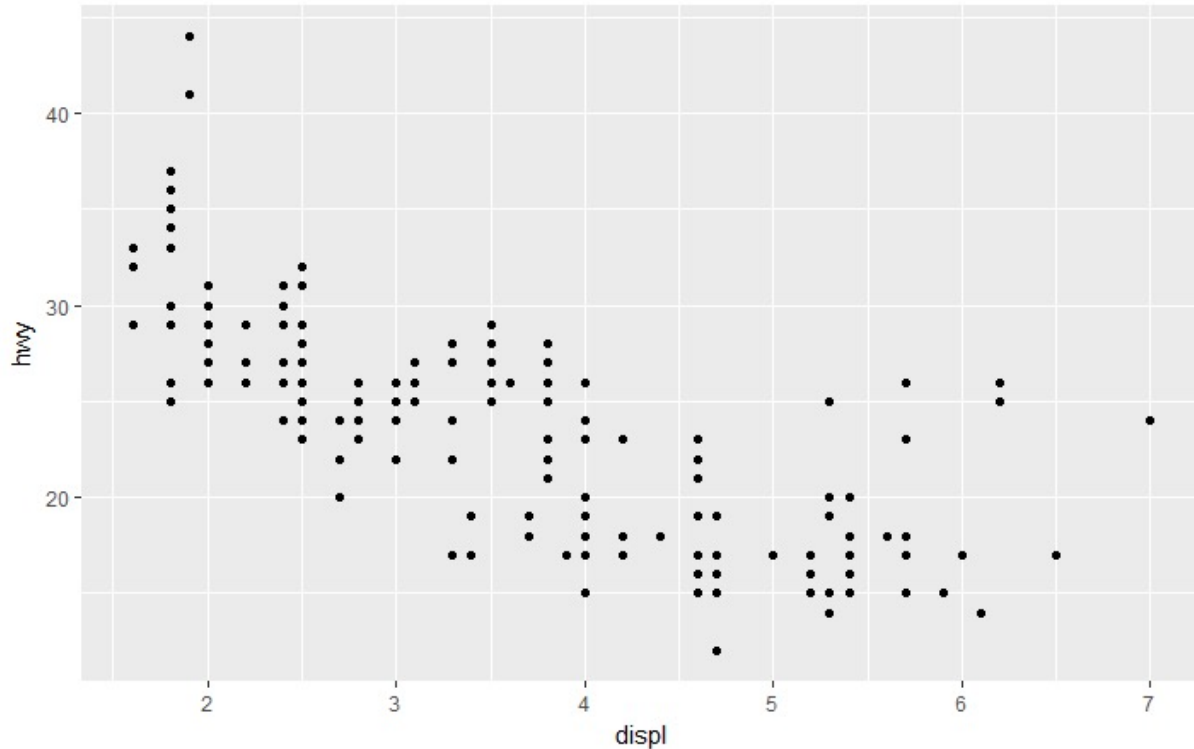
How are these two plots similar?

A **geom** is the geometrical object that a plot uses to represent data. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on.

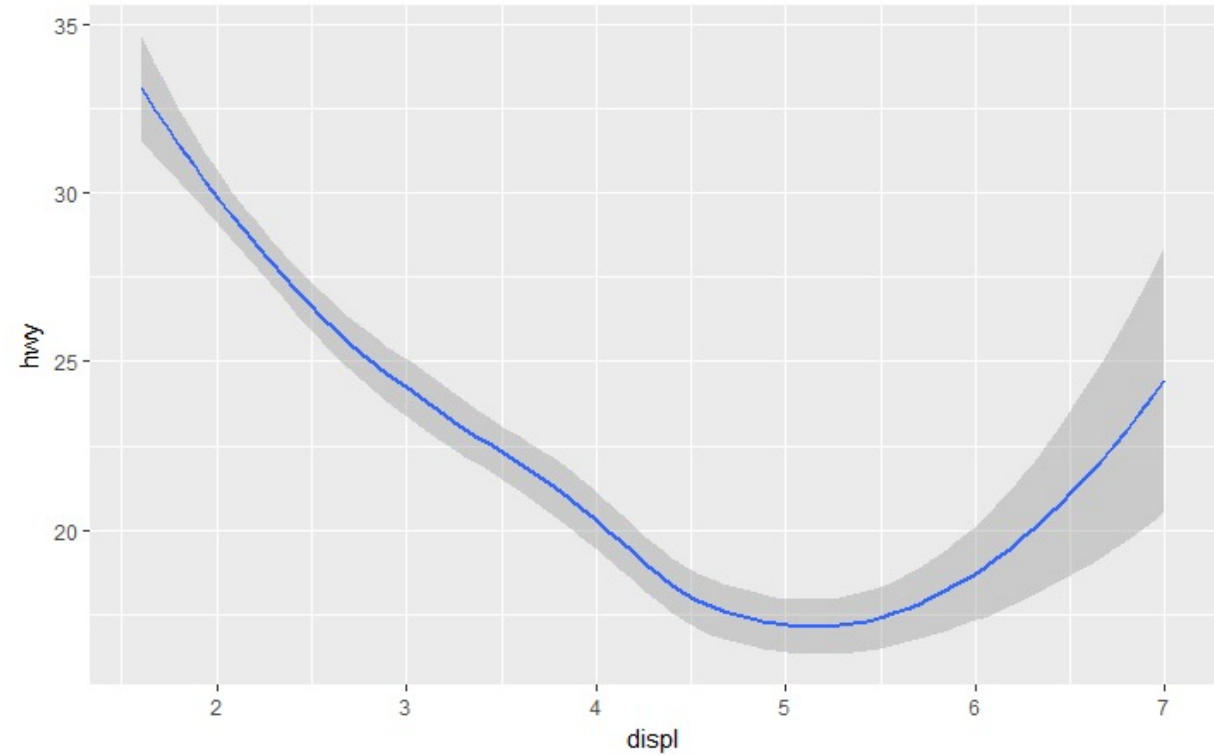
Example: Scatterplots break the trend; they use the point geom. As we see in the preceding plots, you can use different geoms to plot the same data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

To change the geom in your plot, change the geom function that you add to `ggplot()`.

```
ggplot(data = mpg) +  
geom_point(mapping = aes(x = displ, y = hwy))
```

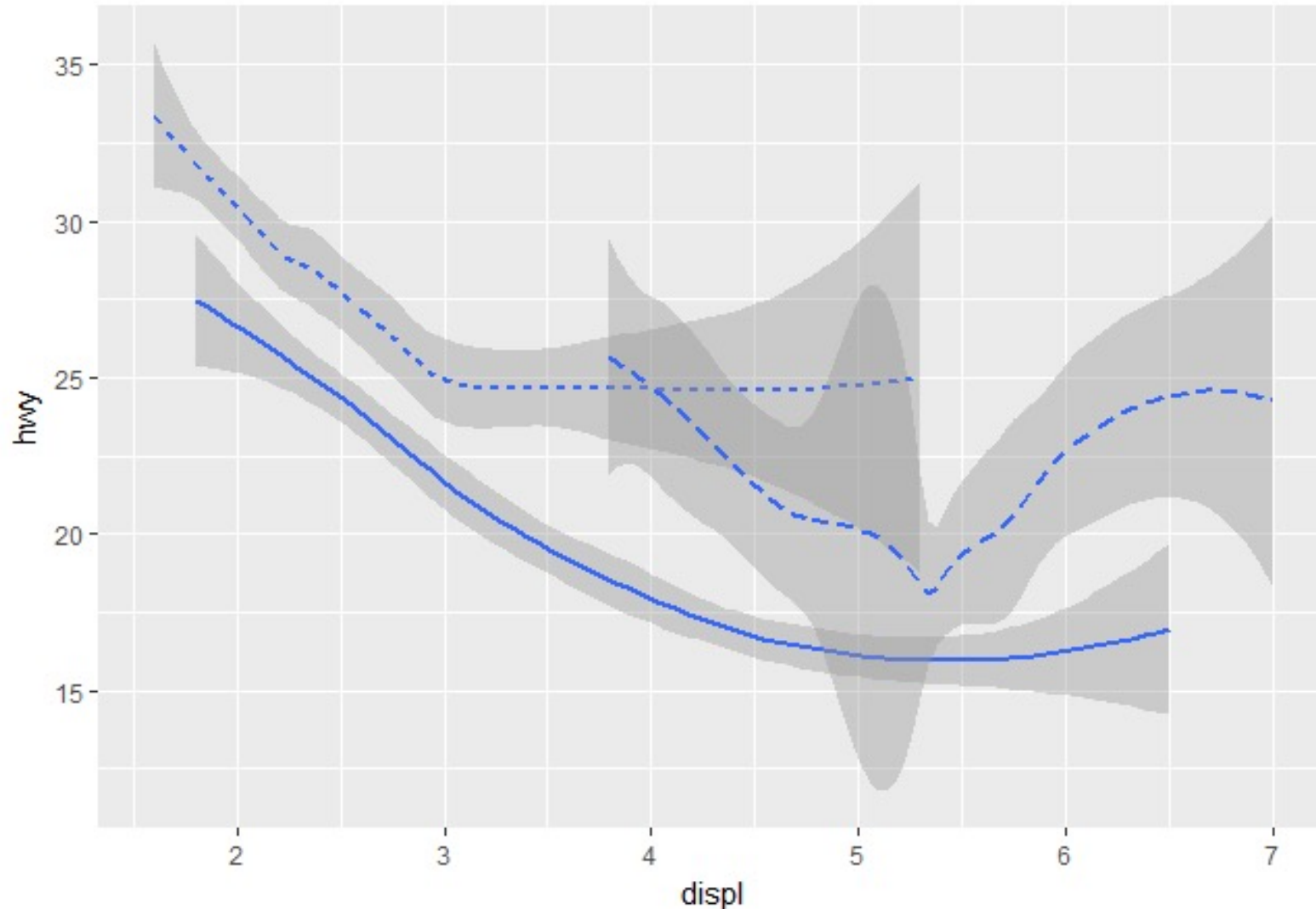


```
ggplot(data = mpg) +  
geom_smooth(mapping = aes(x = displ, y = hwy))
```



you could set the linetype of a line. `geom_smooth()` will draw a different line, with a different linetype, for each unique value of the variable that you map to `linetype`:

```
ggplot(data = mpg) +  
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))
```

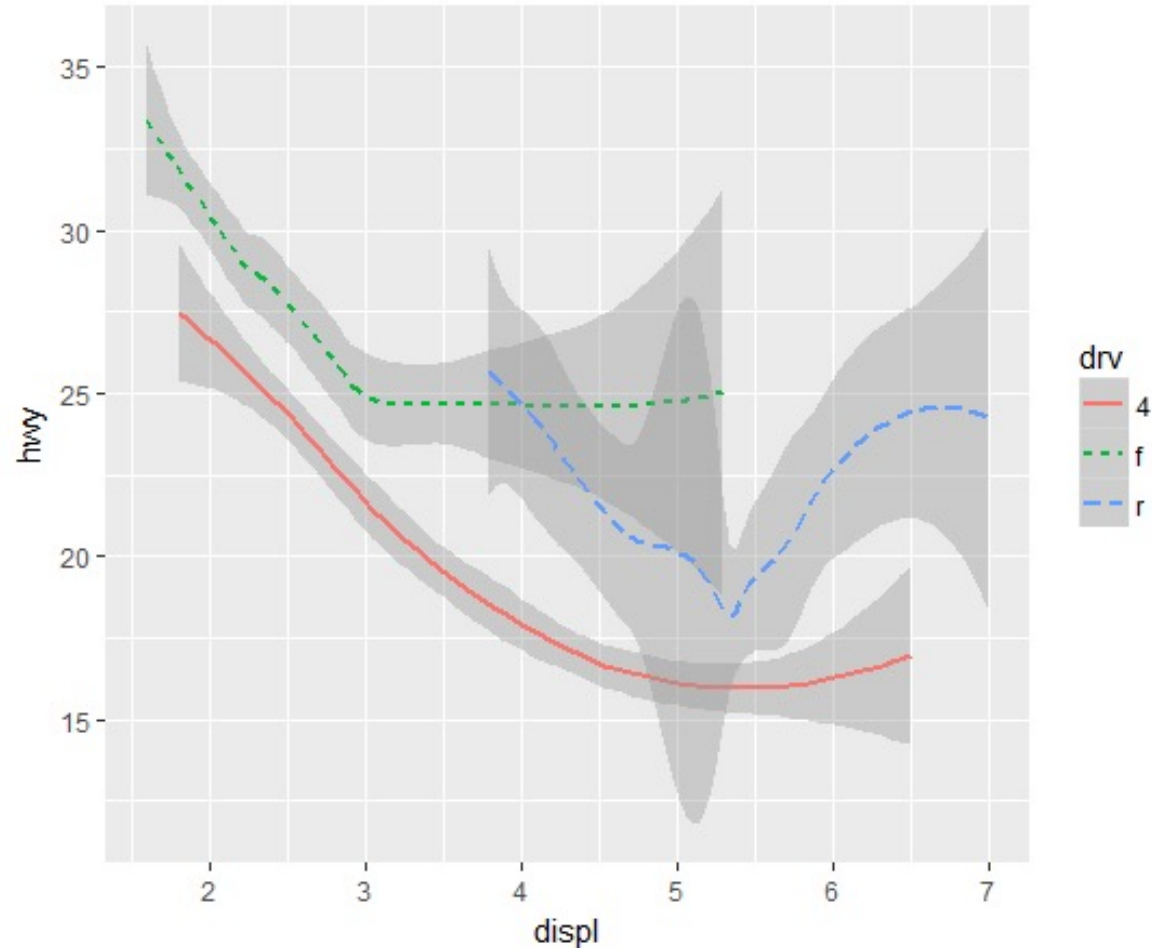


Here `geom_smooth()` separates the cars into three lines based on their `drv` value, which describes a car's drivetrain



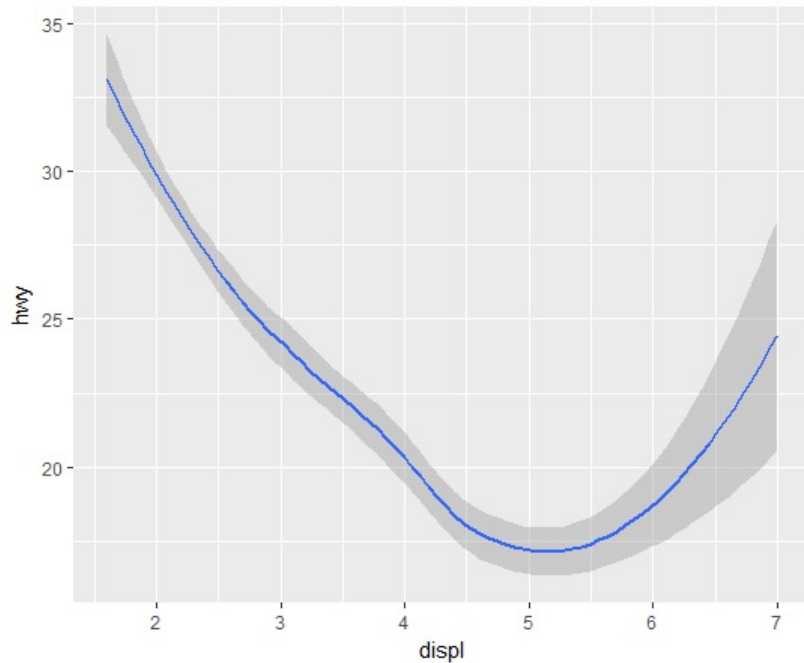
If this sounds strange, we can make it more clear by overlaying the lines on top of the raw data and then **coloring** everything according to drv.

```
ggplot(data = mpg) +  
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype =  
    drv, color = drv))
```

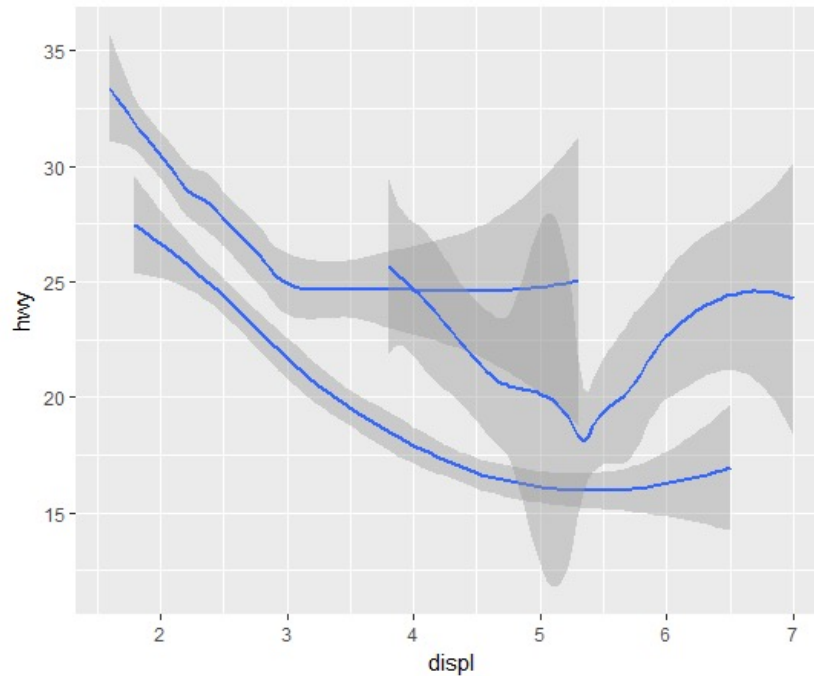


ggplot2 will draw a separate object for each unique value of the grouping variable

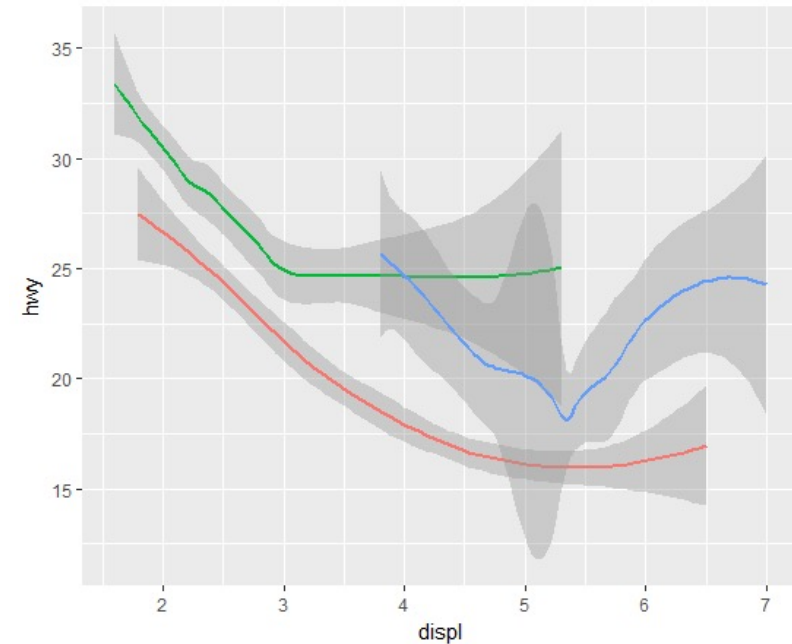
```
ggplot(data = mpg) +  
  geom_smooth(mapping =  
    aes(x = displ, y = hwy))
```



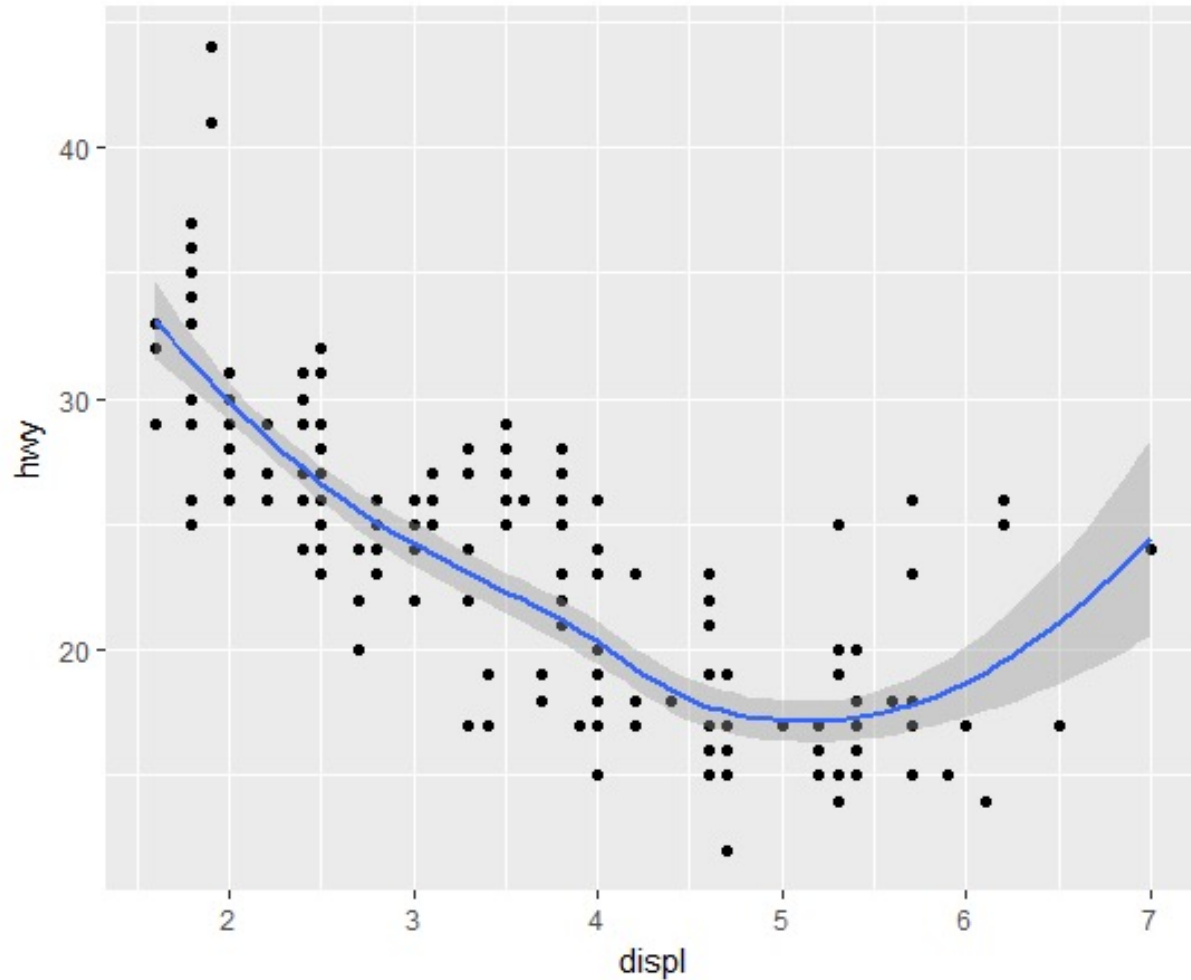
```
ggplot(data = mpg) +  
  geom_smooth(mapping =  
    aes(x = displ, y = hwy, group  
      = drv))
```



```
ggplot(data = mpg) +  
  geom_smooth(  
    mapping = aes(x = displ, y = hwy,  
      color = drv),  
    show.legend = FALSE  
  )
```



To display multiple geoms in the same plot, add multiple geom functions to ggplot():



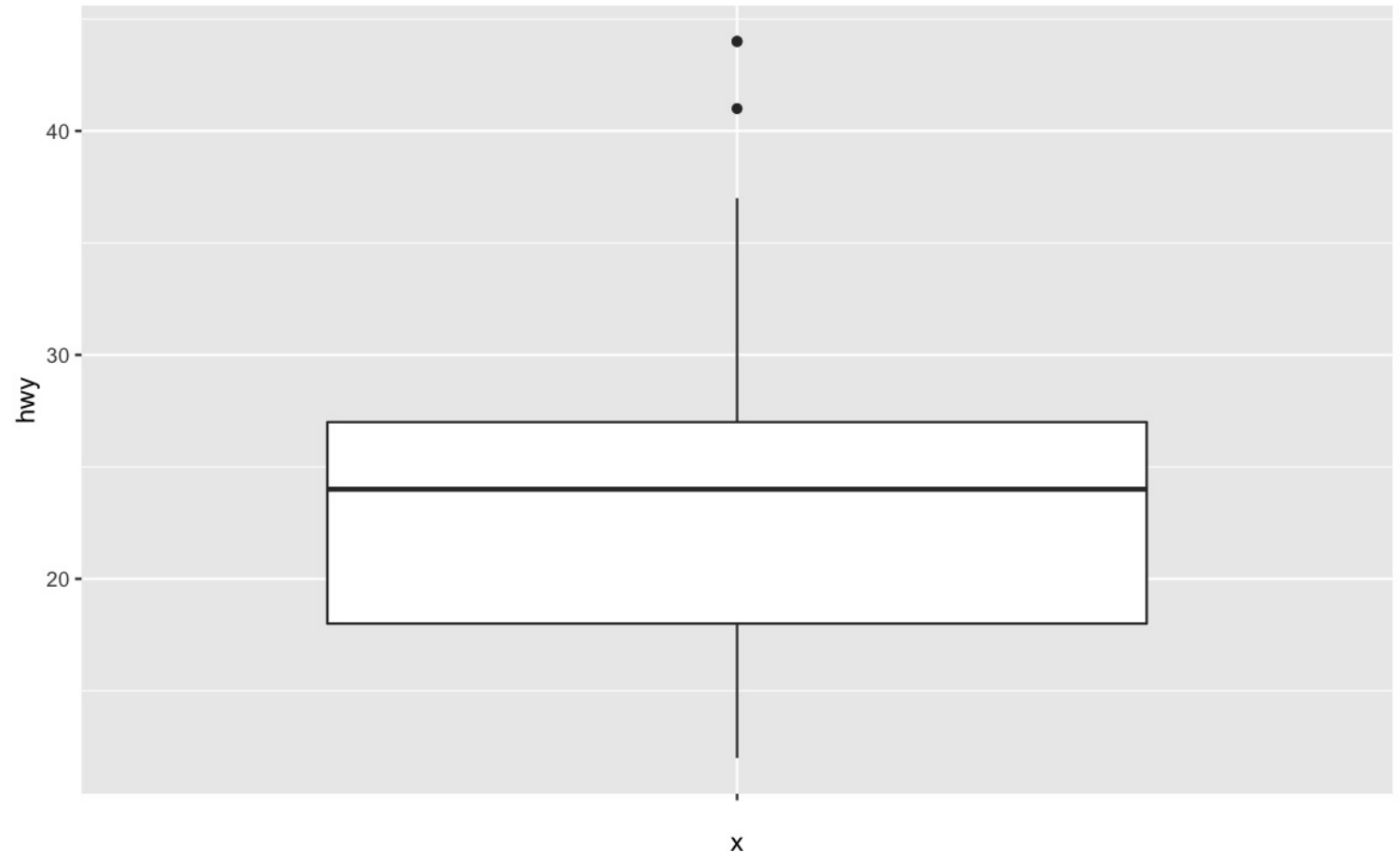
```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

A [boxplot](#) (also very useful to visualize distributions and detect potential [outliers](#)) can be plotted using `geom_boxplot()`:

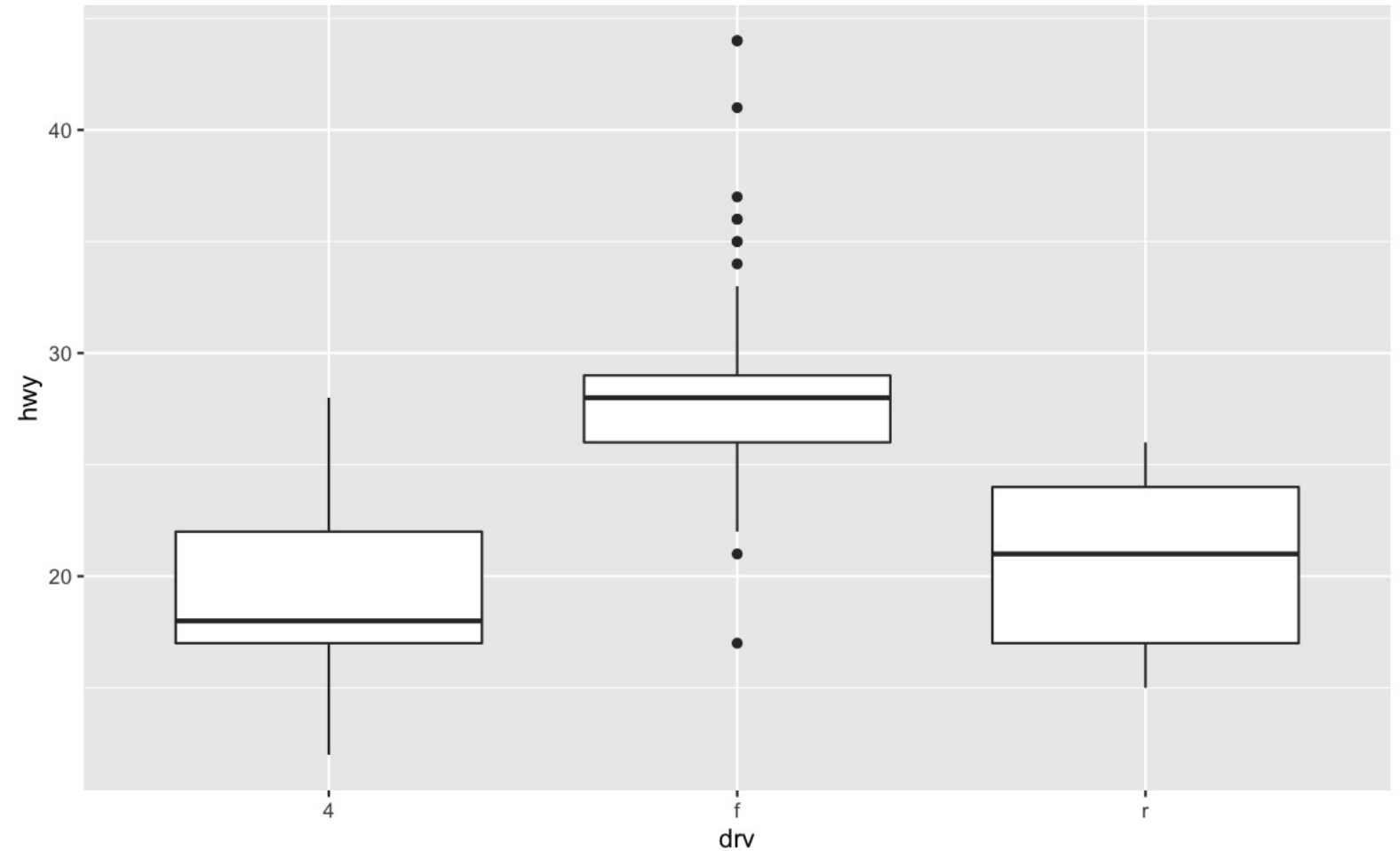
Boxplot for one variable

```
ggplot(mpg) +  
  aes(x = "", y = hwy) +  
  geom_boxplot()
```

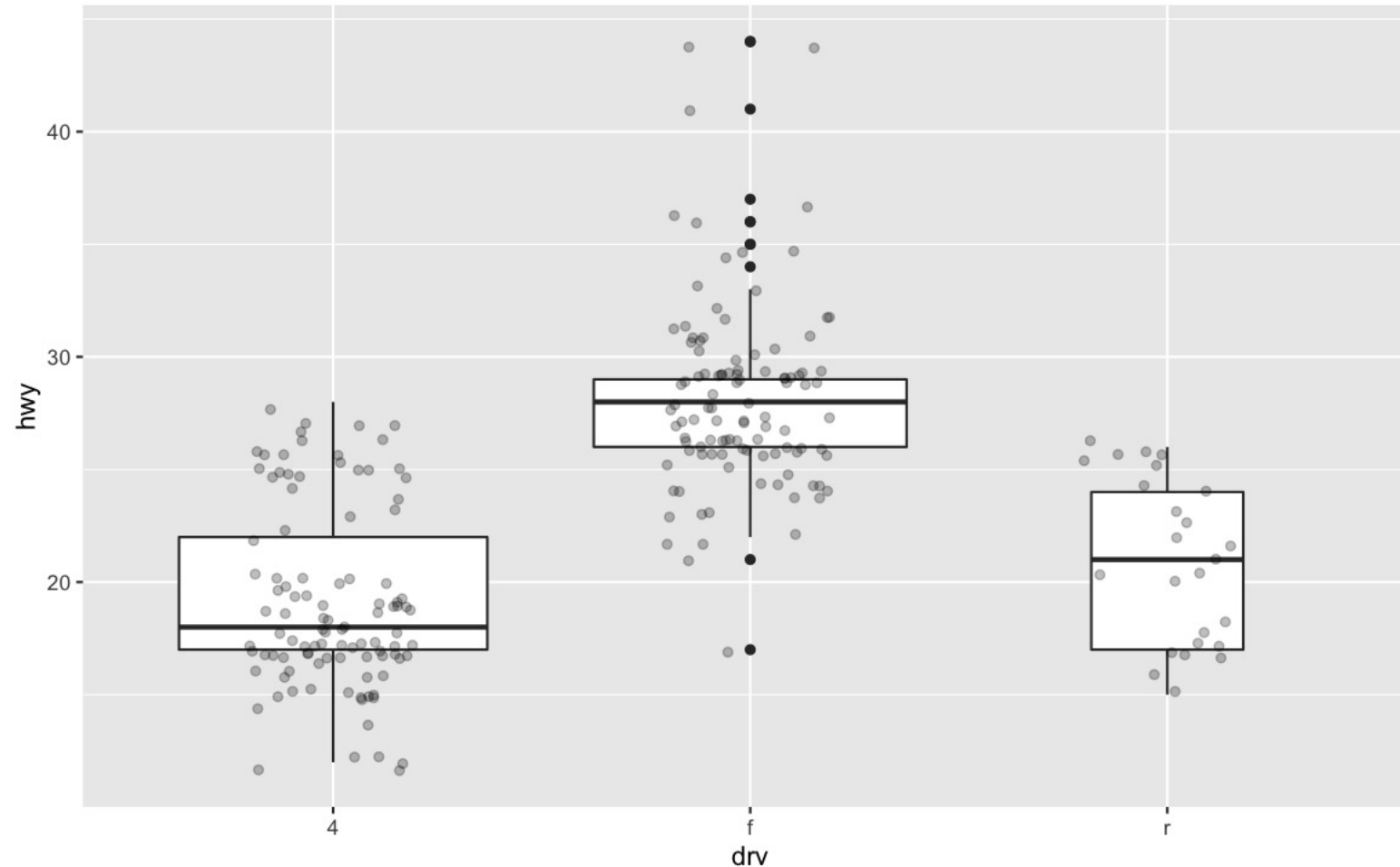
Boxplot



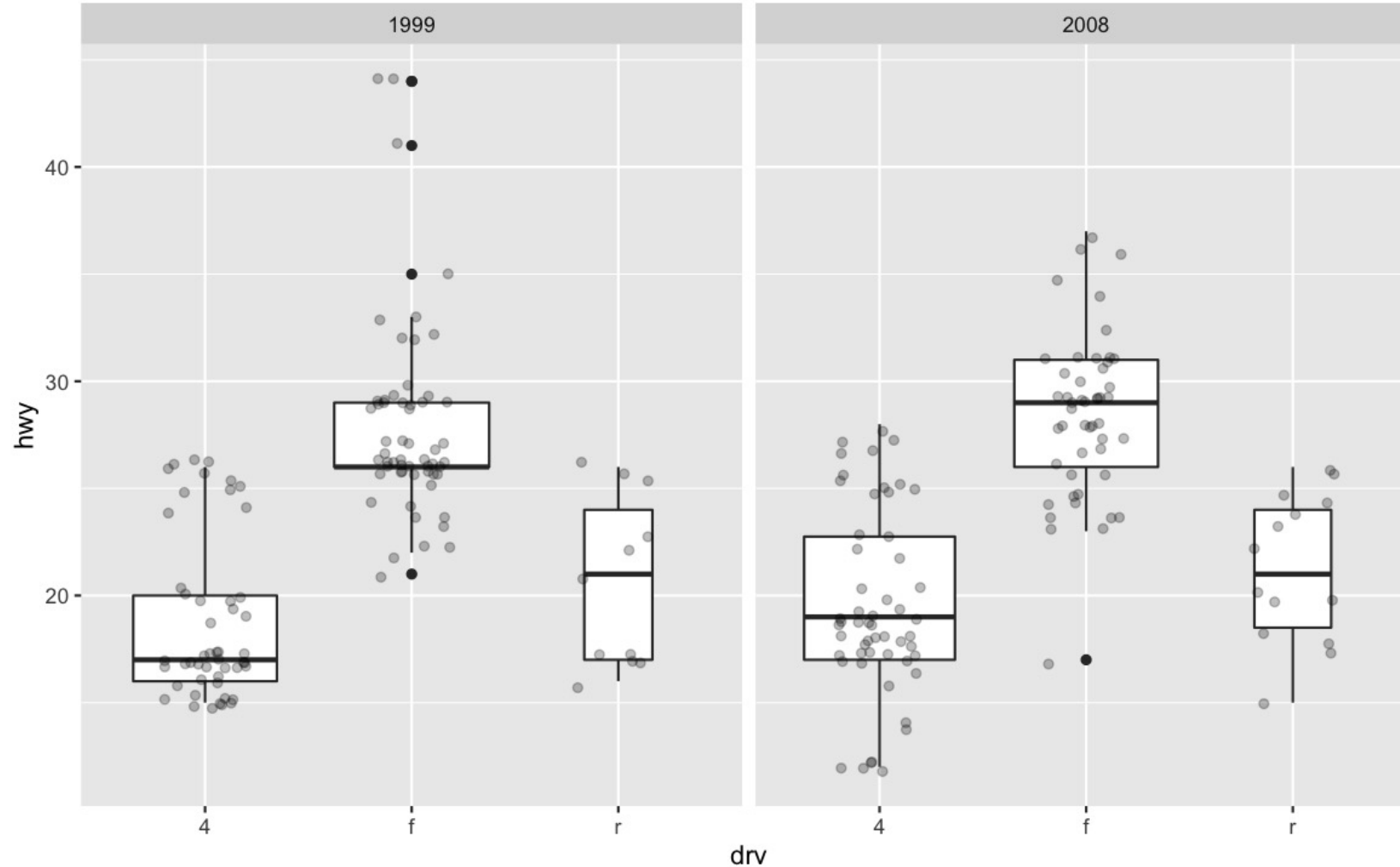
```
# Boxplot by factor  
ggplot(mpg) +  
  aes(x = drv, y = hwy) +  
  geom_boxplot()
```



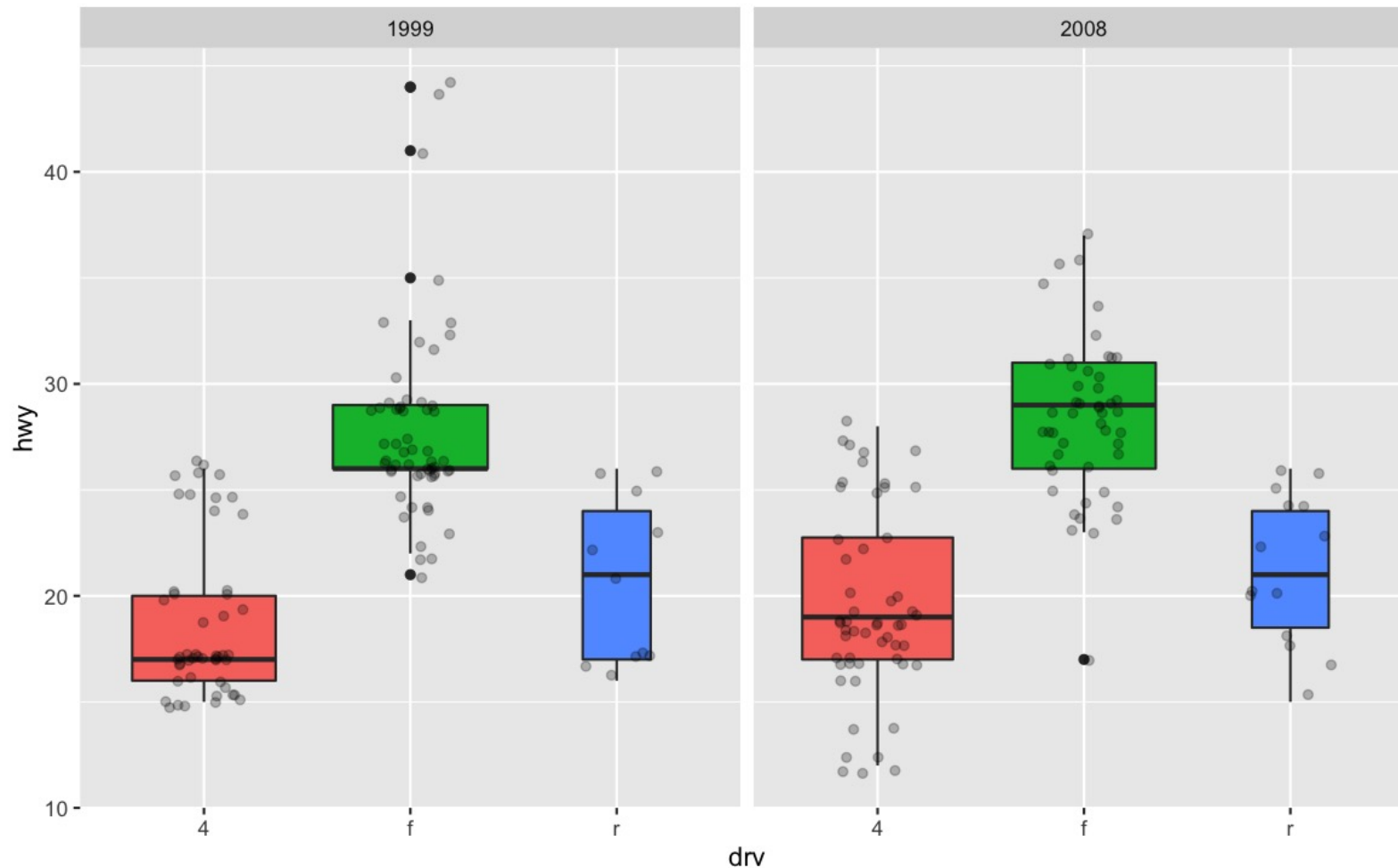
It is also possible to plot the points on the boxplot with `geom_jitter()`, and to vary the width of the boxes according to the size (i.e., the number of observations) of each level with `varwidth = TRUE`:



Finally, it is also possible to divide boxplots into several panels according to the levels of a qualitative variable:




```
ggplot(mpg) +  
  aes(x = drv, y = hwy, fill = drv) + # add color to boxes with fill  
  geom_boxplot(varwidth = TRUE) + # vary boxes width according to n obs.  
  geom_jitter(alpha = 0.25, width = 0.2) + # adds random noise and limit its width  
  facet_wrap(~year) + # divide into 2 panels  
  theme(legend.position = "none") # remove legend
```



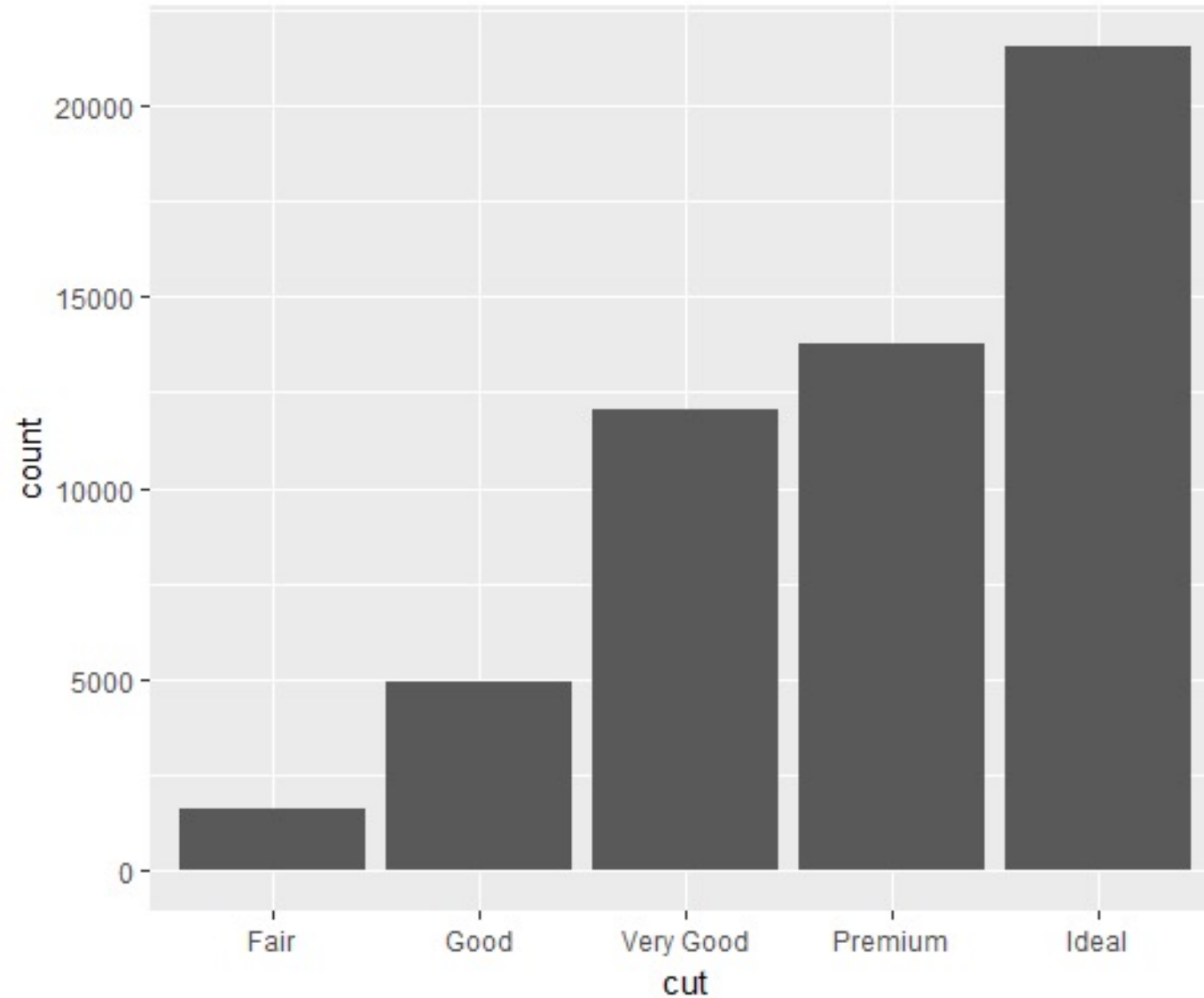
Statistical Transformations

Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with `geom_bar()`.

Example: The following chart displays the total number of diamonds in the diamonds dataset, grouped by cut. The diamonds dataset comes in ggplot2 and contains information about ~54,000 diamonds, including the price, carat, color, clarity, and cut of each diamond. The chart shows that more diamonds are available with high-quality cuts than with low quality cuts:

```
> diamonds
# A tibble: 53,940 × 10
  carat    cut color clarity depth table price      x      y
  <dbl>    <ord> <ord>    <ord> <dbl> <dbl> <int> <dbl> <dbl>
1  0.23    Ideal     E      SI2   61.5    55   326   3.95   3.98
2.43
2  0.21    Premium    E      SI1   59.8    61   326   3.89   3.84
2.31
3  0.23     Good     E      VS1   56.9    65   327   4.05   4.07
2.31
4  0.29    Premium    I      VS2   62.4    58   334   4.20   4.23
2.63
5  0.31     Good     J      SI2   63.3    58   335   4.34   4.35
2.75
6  0.24 Very Good    J     VVS2   62.8    57   336   3.94   3.96
2.48
7  0.24 Very Good    I     VVS1   62.3    57   336   3.95   3.98
2.47
8  0.26 Very Good    H      SI1   61.9    55   337   4.07   4.11
2.53
9  0.22     Fair     E      VS2   65.1    61   337   3.87   3.78
2.49
10 0.23 Very Good    H      VS1   59.4    61   338   4.00   4.05
2.39
# ... with 53,930 more rows
```

```
ggplot(data = diamonds) +  
geom_bar(mapping = aes(x = cut))
```



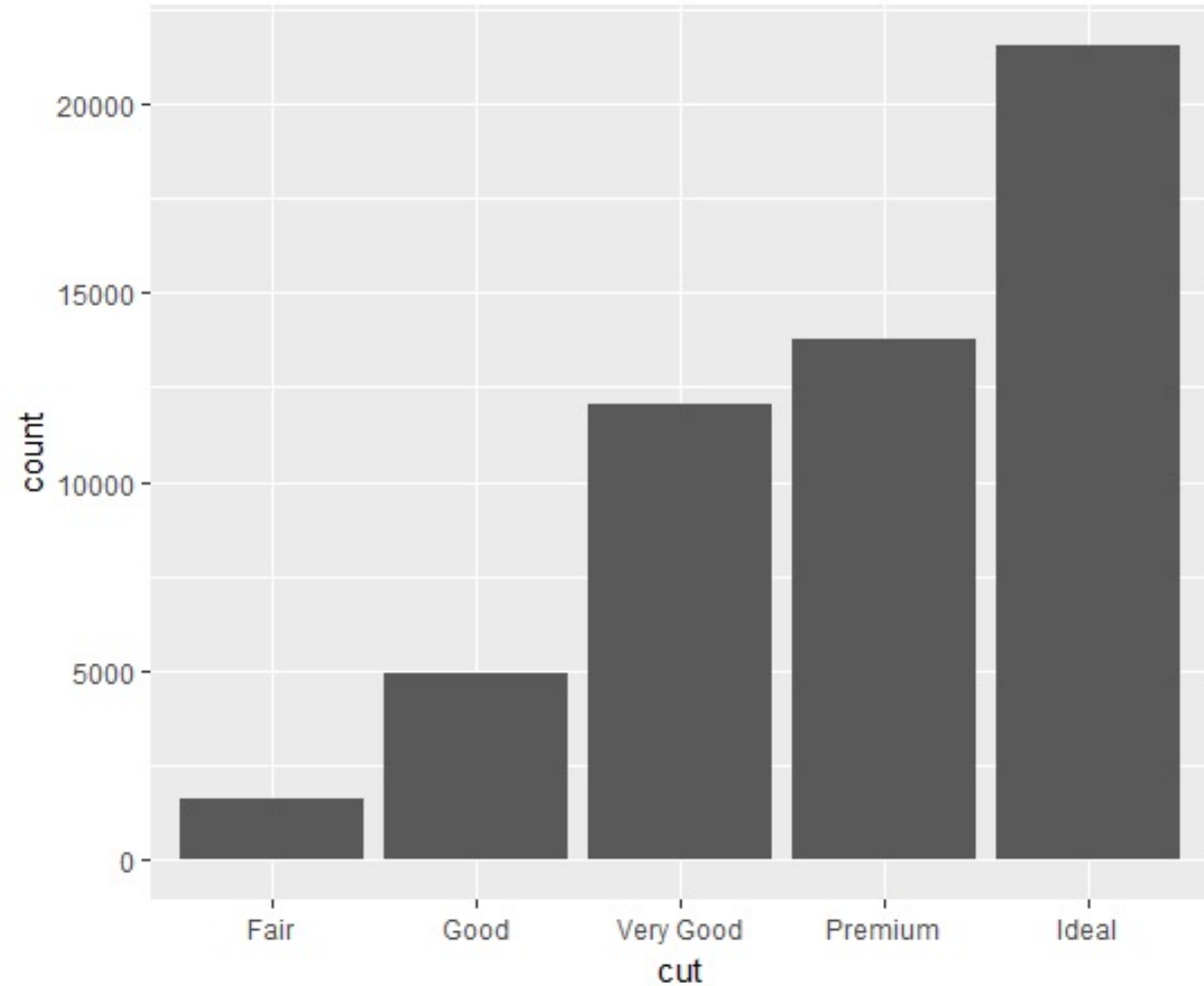
On the x-axis, the chart displays cut, a 'variable from diamonds. On the y-axis, it displays count, but count is not a variable in diamonds! Where does count come from? Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

- Bar charts, **histograms**, and frequency polygons bin your data and then plot bin counts, the number of points that fall in each bin.
- Smoothers fit a model to your data and then plot predictions from the model.
- Boxplots compute a robust summary of the distribution and display a specially formatted box.

The algorithm used to calculate new values for a graph is called a **stat**, short for ***statistical transformation***. The following figure describes how this process works with **geom_bar()**.

You can generally use geoms and stats interchangeably. For example, you can re-create the previous plot using `stat_count()` instead of `geom_bar()`:

```
ggplot(data = diamonds) +  
stat_count(mapping = aes(x =  
cut))
```



ggplot2 provides over 20 stats for you to use. Each stat is a function, so you can get help in the usual way, e.g., `?stat_bin`.

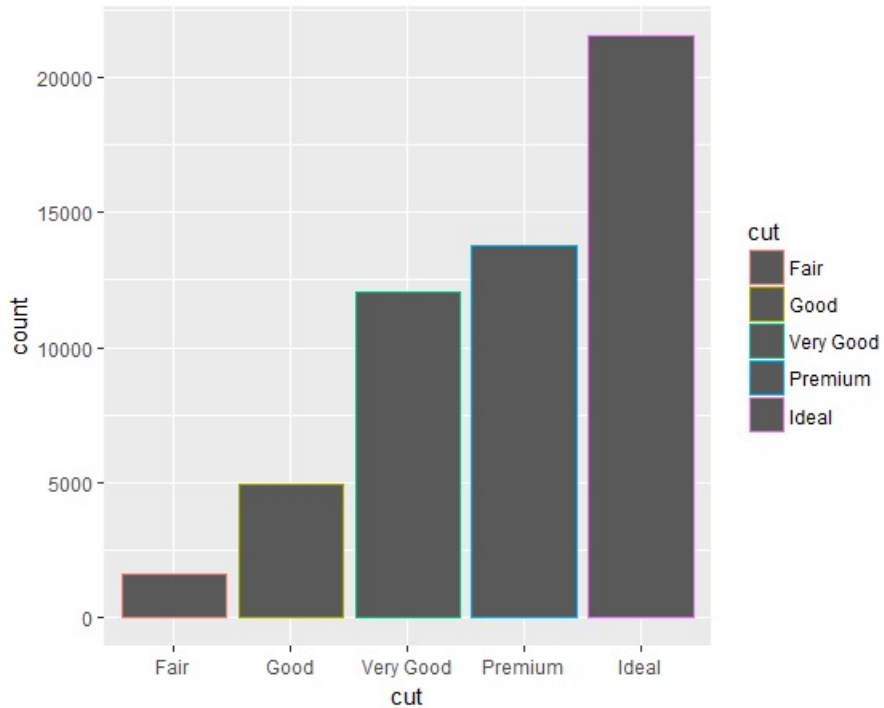
To see a complete list of stats, try the ggplot2 cheatsheet

<https://bda2020.files.wordpress.com/2016/05/ggplot2-cheatsheet-2-0-2.pdf>

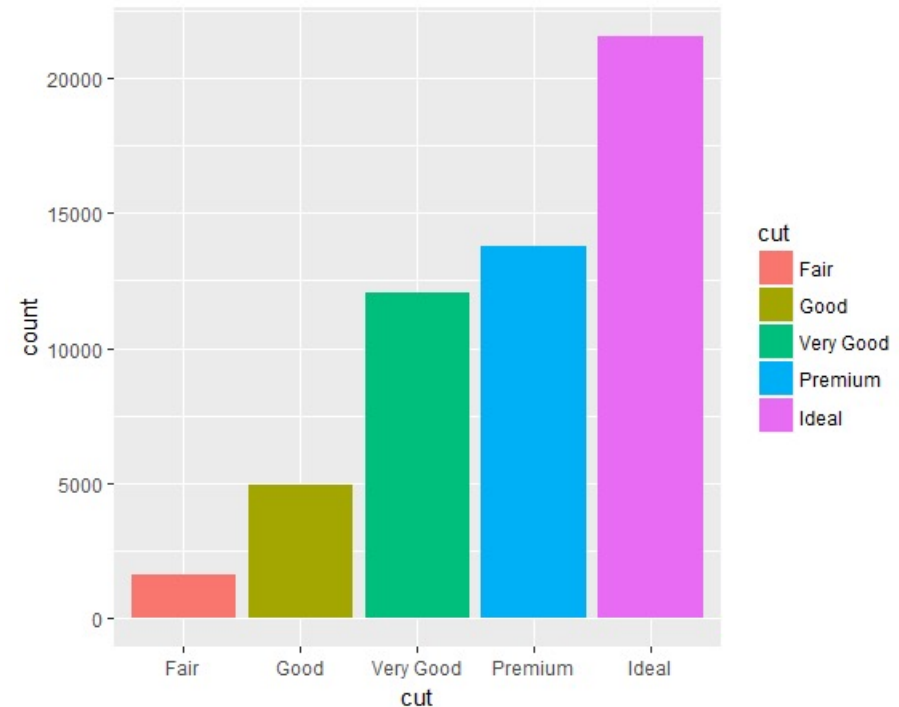
Position Adjustments

There's one more piece of magic associated with bar charts. You can color a bar chart using either the color aesthetic, or more usefully, fill:

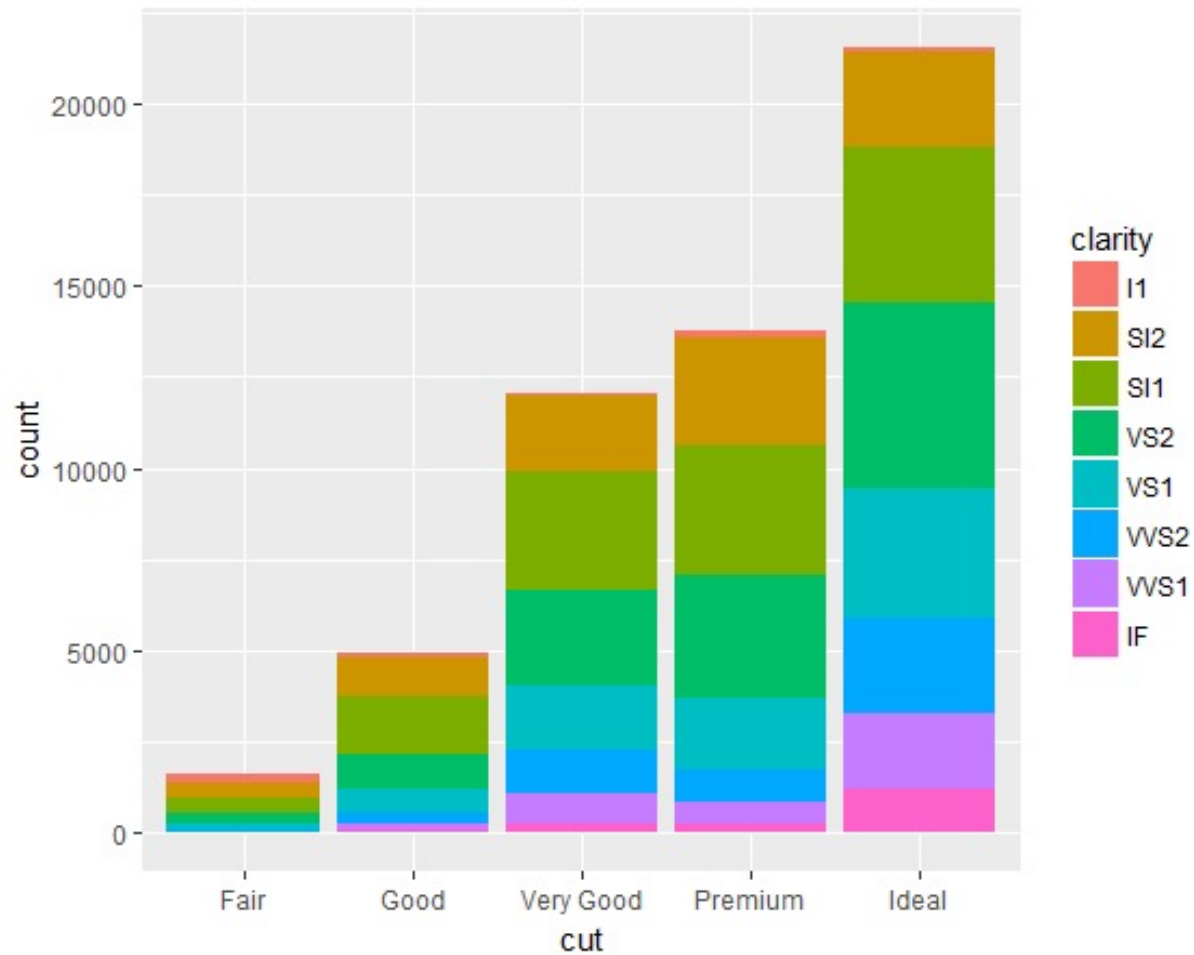
```
ggplot(data = diamonds) +  
geom_bar(mapping = aes(x = cut, color = cut))
```



```
ggplot(data = diamonds) +  
geom_bar(mapping = aes(x = cut, fill = cut))
```

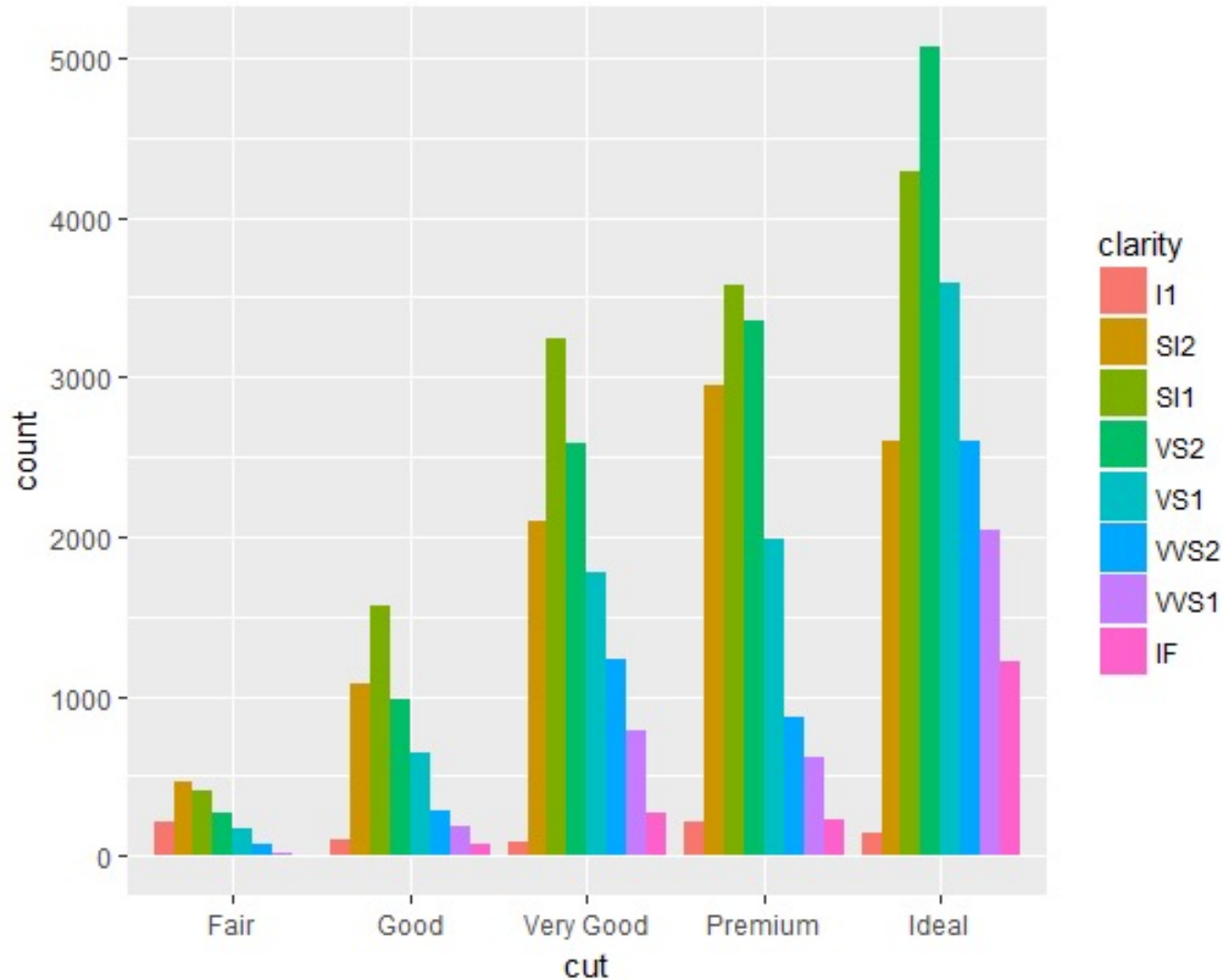


Note what happens if you map the fill aesthetic to another variable, like clarity: the bars are automatically stacked. Each colored rectangle represents a combination of cut and clarity:



```
ggplot(data = diamonds) +  
geom_bar(mapping = aes(x = cut, fill = clarity))
```

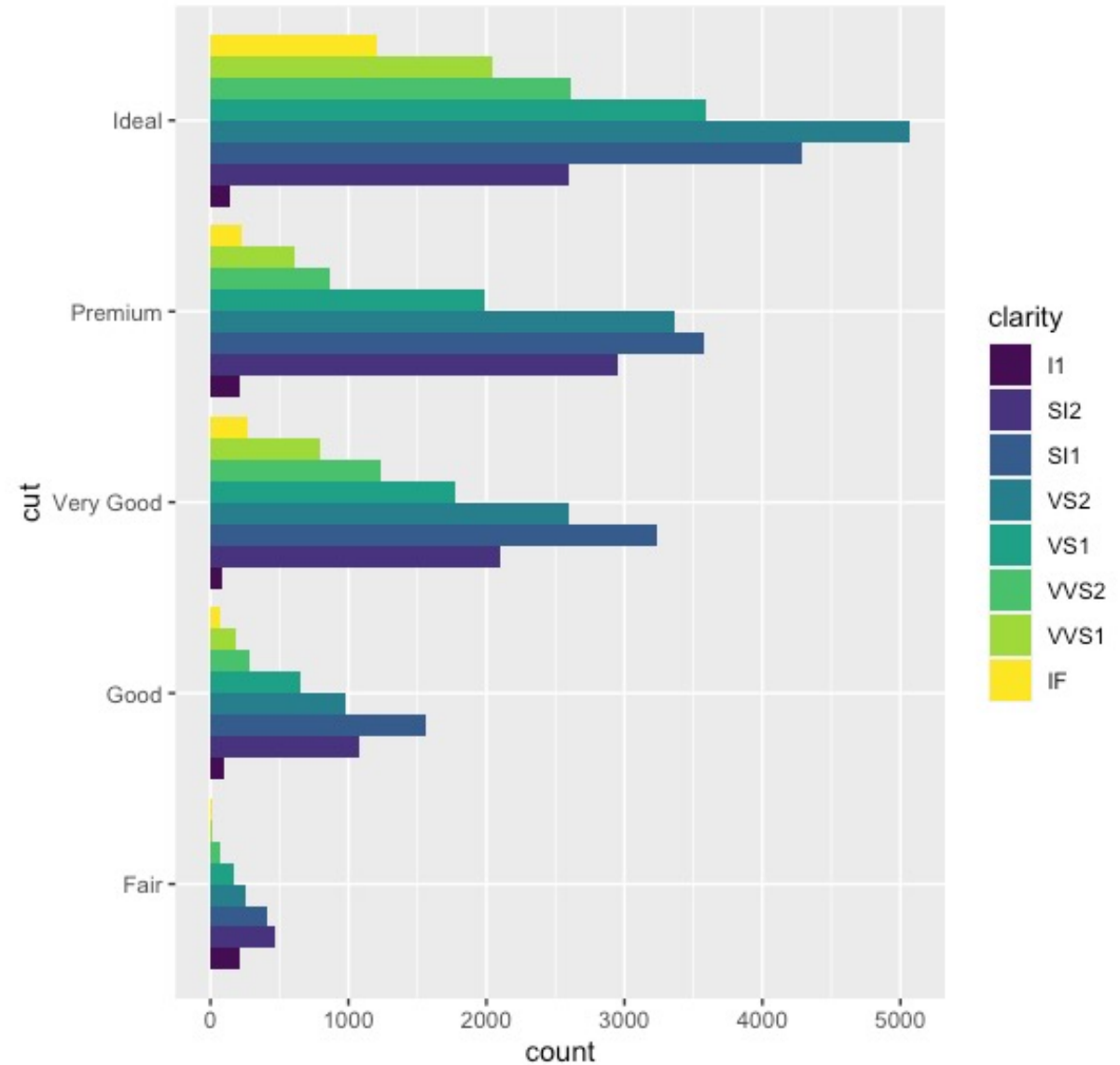
position = "dodge" places overlapping objects directly beside one another. This makes it easier to compare individual values:



```
ggplot(data = diamonds) +  
geom_bar(  
  mapping = aes(x = cut, fill = clarity),  
  position = "dodge")
```

This can be done with many types of plot, not only with boxplots. For instance, if a categorical variable has many levels or the labels are long, it is usually best to flip the coordinates for a better visual:

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity),  
    position = "dodge") +  
  coord_flip()
```



Coordinate Systems

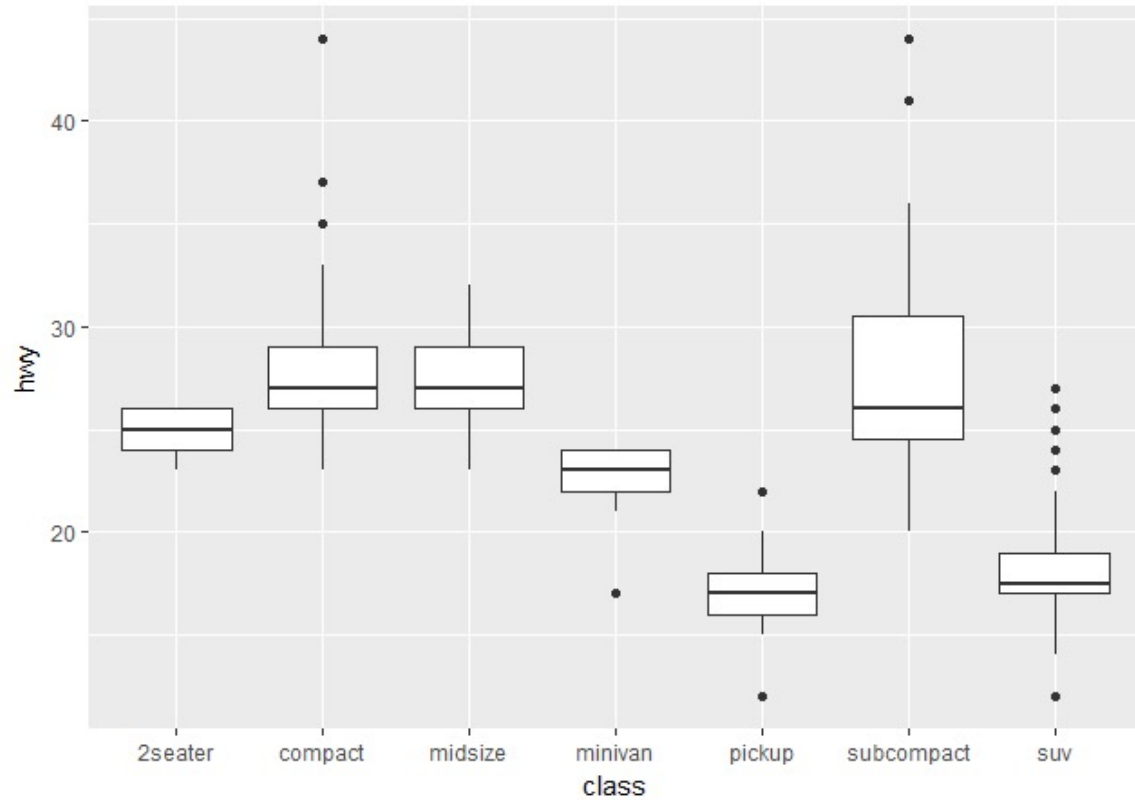
Coordinate Systems

Coordinate systems are probably the most complicated part of ggplot2. The default coordinate system is the Cartesian coordinate system where the x and y position act independently to find the location of each point.

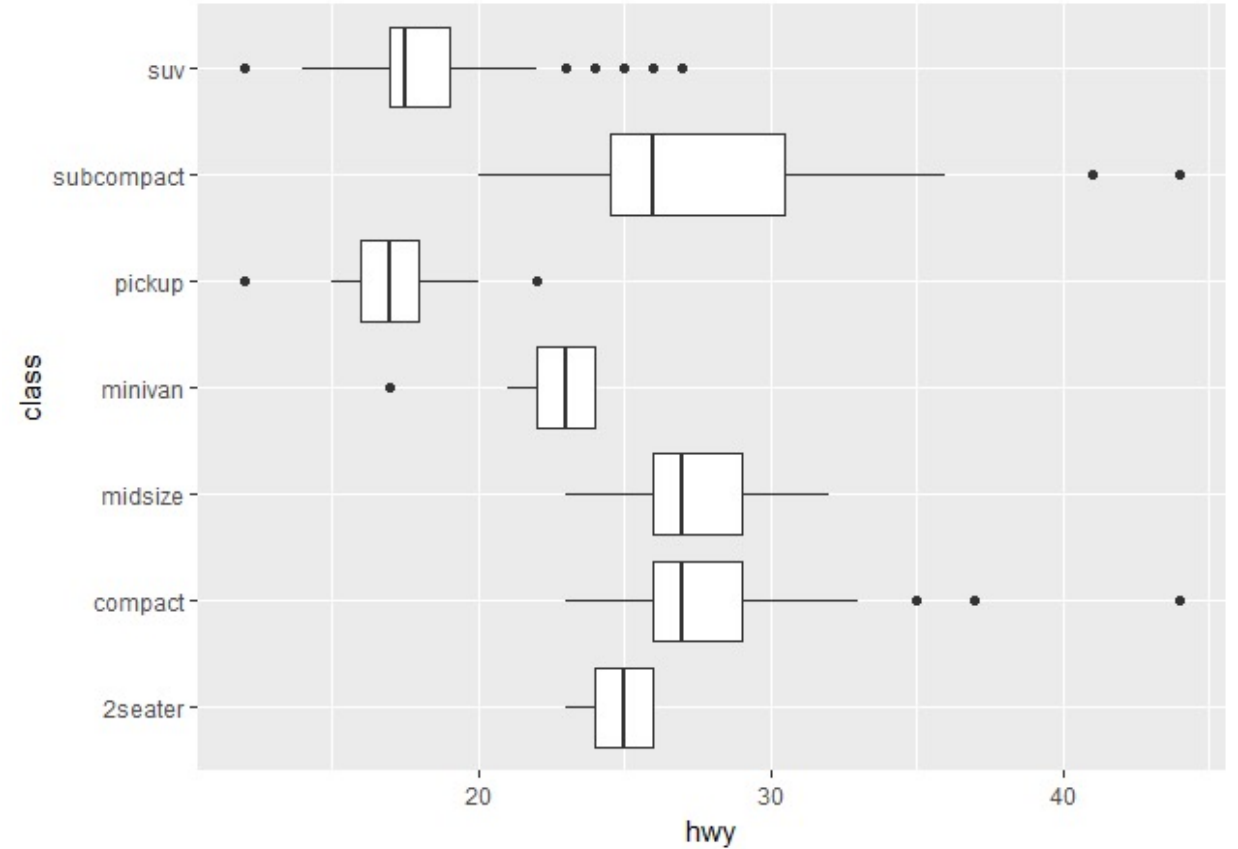
There are a number of other coordinate systems that are occasionally helpful:

- *coord_flip()* switches the x- and y-axes. This is useful (for example) if you want horizontal boxplots. It's also useful for long labels—it's hard to get them to fit without overlapping on the x-axis

```
ggplot(data = mpg,
mapping = aes(x = class, y =
hwy)) +
geom_boxplot()
```

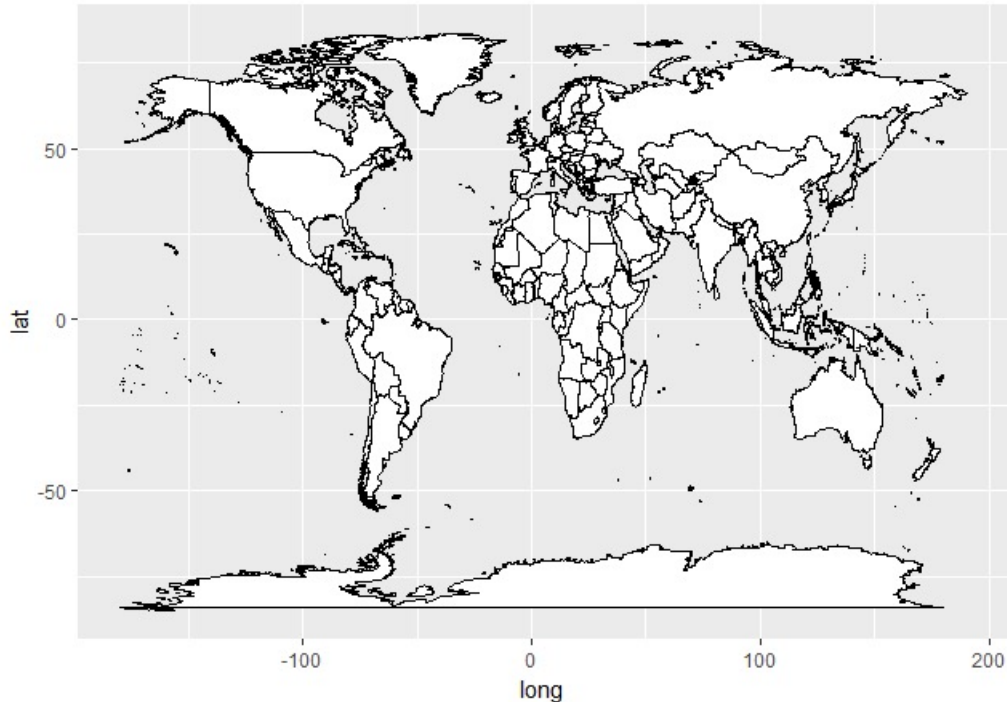


```
ggplot(data = mpg, mapping =
aes(x = class, y = hwy)) +
geom_boxplot() +
coord_flip()
```

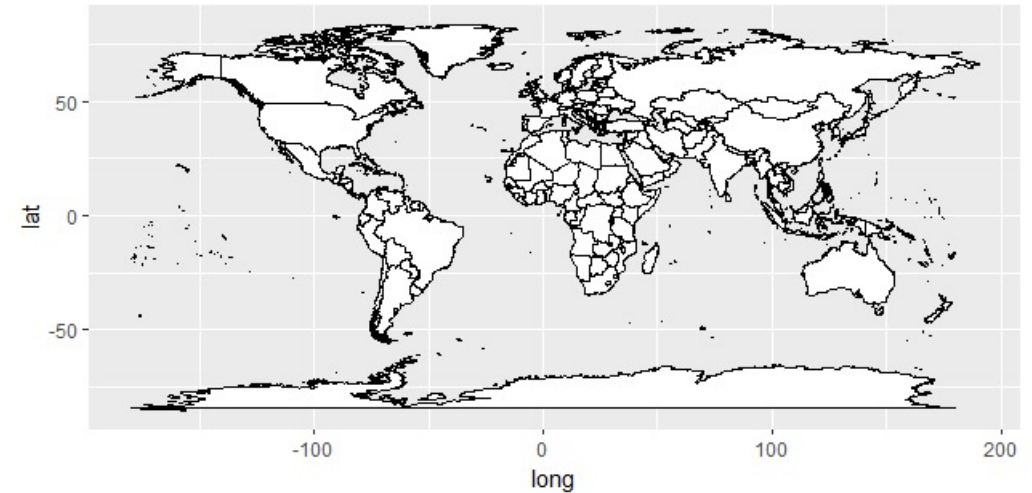


- `coord_quickmap()` sets the aspect ratio correctly for maps. This is very important if you're plotting spatial data with ggplot2:

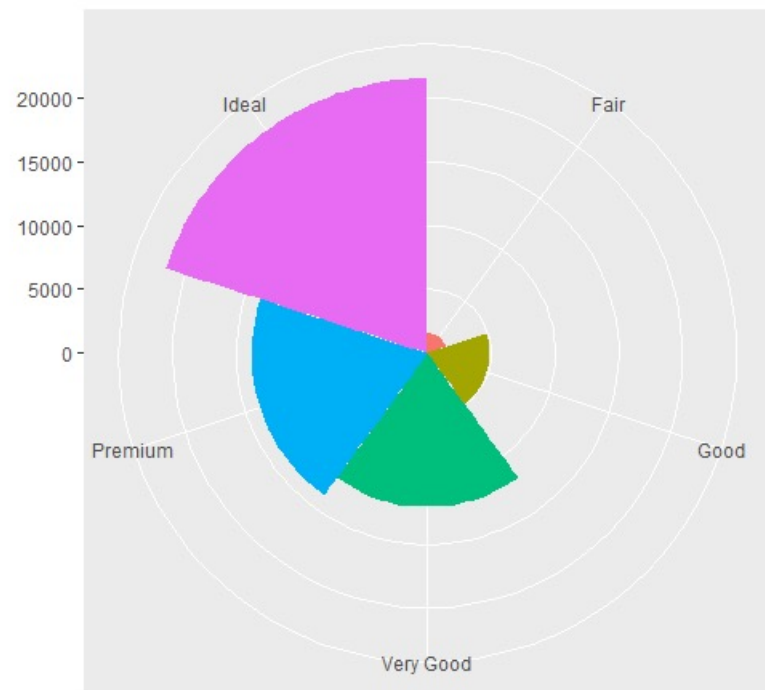
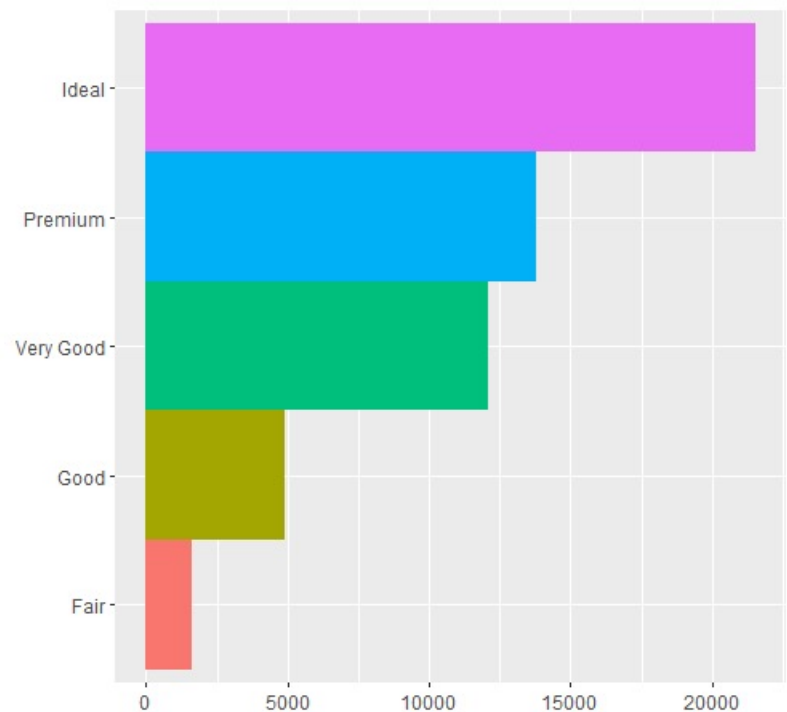
```
world <- map_data("world")  
ggplot(world, aes(long, lat, group = group)) +  
  geom_polygon(fill = "white", color = "black")
```



```
ggplot(world, aes(long, lat, group = group)) +  
  geom_polygon(fill = "white", color = "black") +  
  coord_quickmap()
```



`coord_polar()` uses polar coordinates. Polar coordinates reveal an interesting connection between a bar chart and a Coxcomb chart:



```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)
bar + coord_flip()
bar + coord_polar()
```

Simple pie charts

Create some data :

```
df <- data.frame( group = c("Male", "Female", "Child"), value  
= c(25, 25, 50) )  
head(df)
```

```
> head(df)  
  group value  
1  Male   25  
2 Female   25  
3  Child   50  
>
```

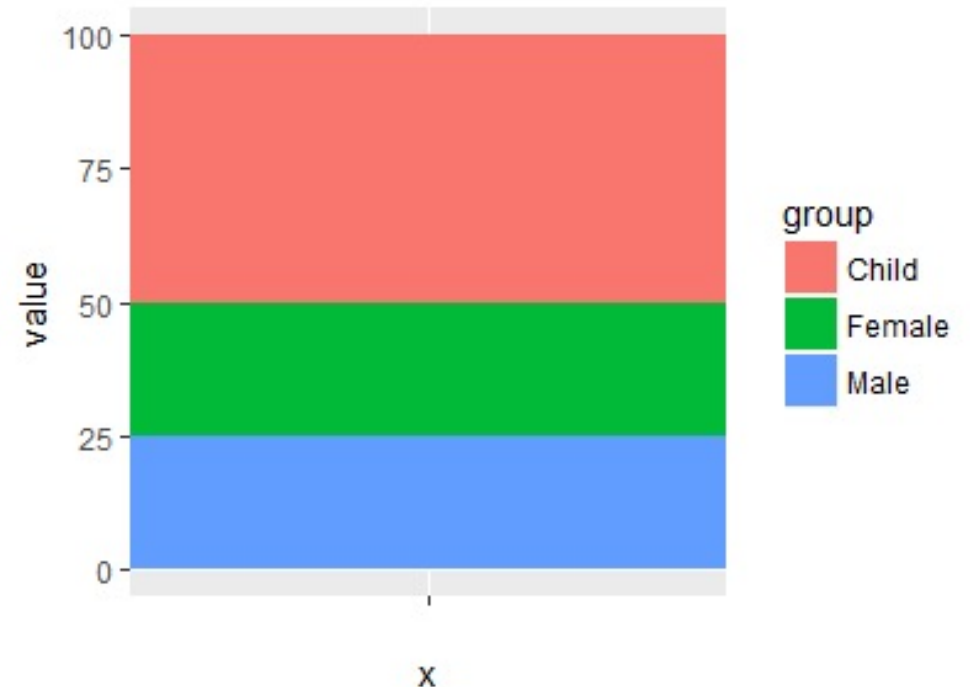
Use a barplot to visualize the data :

```
library(ggplot2)
```

```
# Barplot
```

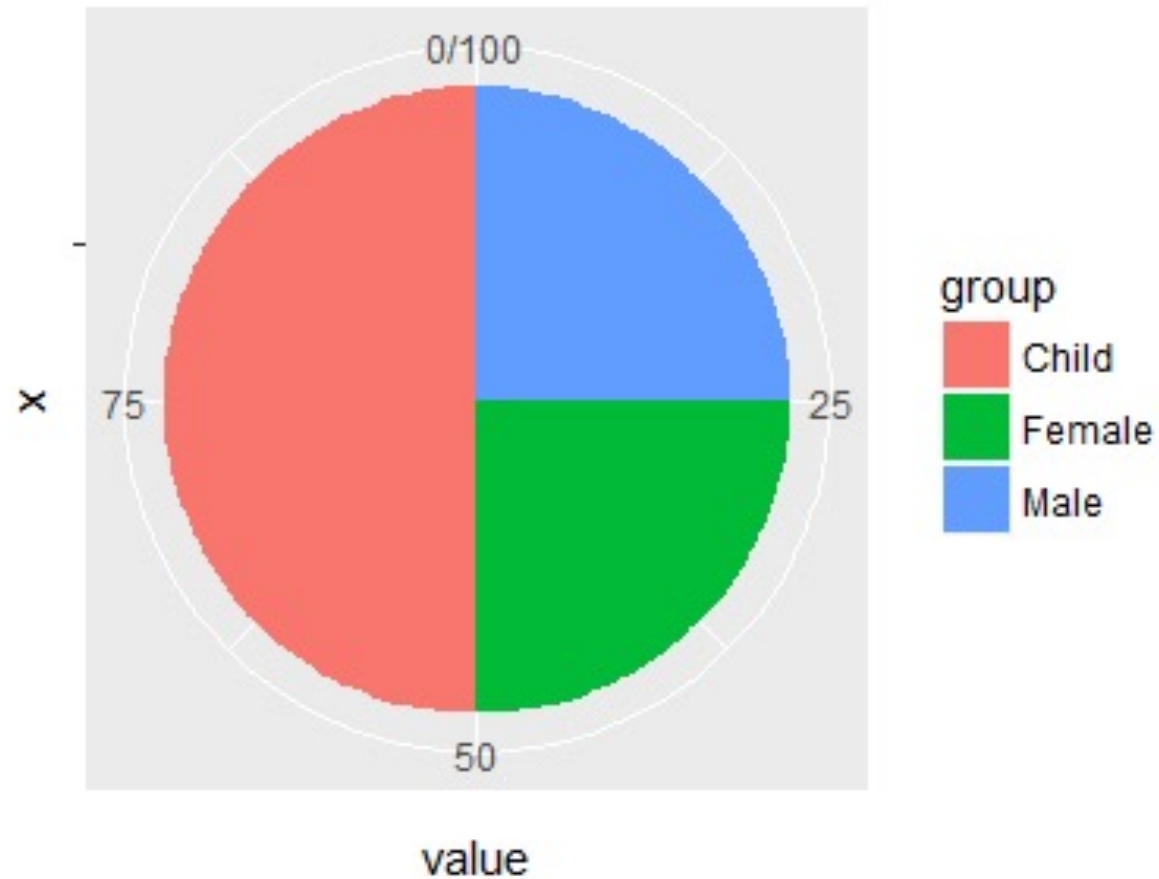
```
bp<- ggplot(df, aes(x="", y=value, fill=group))+  
  geom_bar(width = 1, stat = "identity")
```

```
bp
```



Create a pie chart :

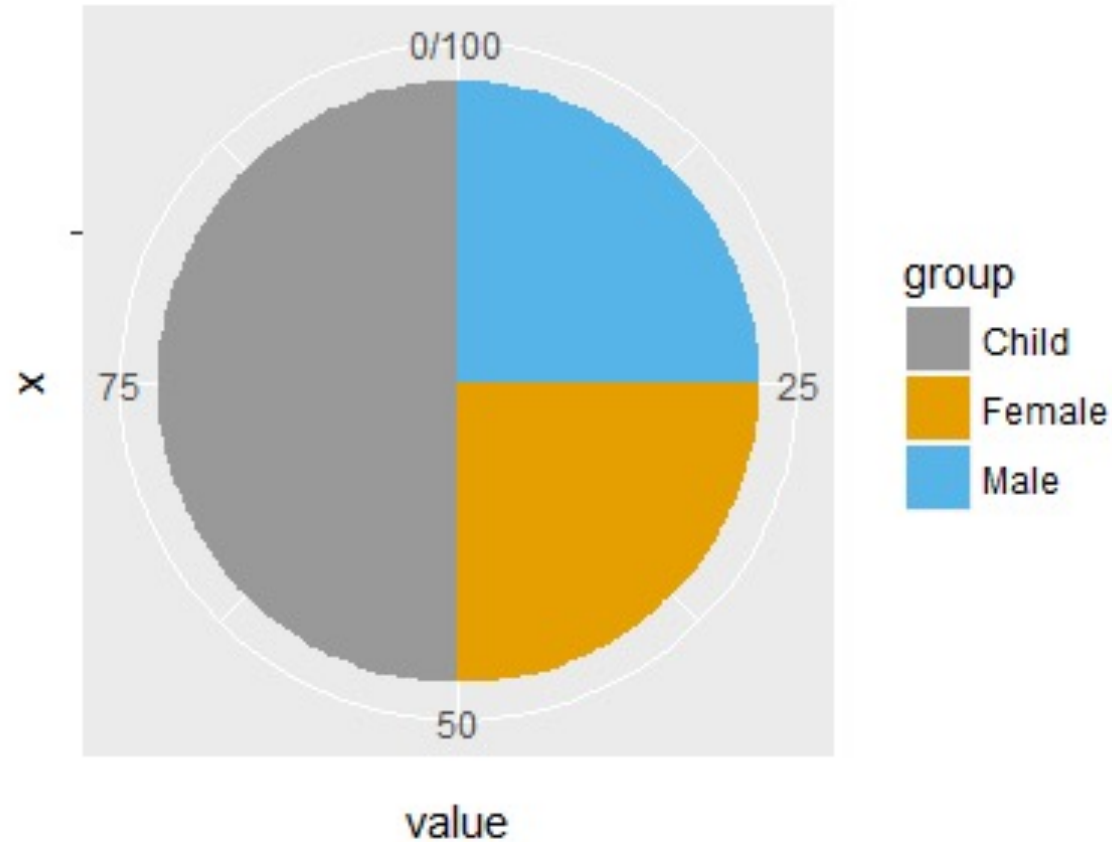
```
pie <- bp + coord_polar("y", start=0)  
pie
```



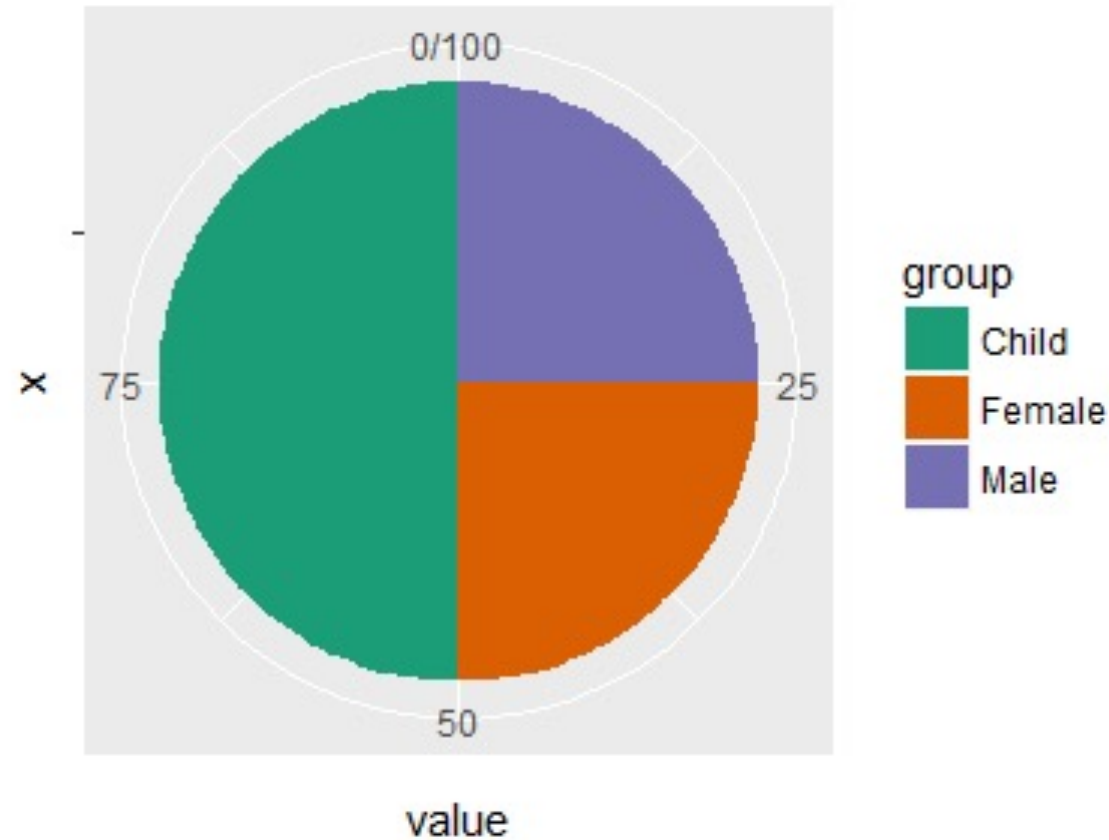
Change the pie chart fill colors

Use custom color palettes

```
pie + scale_fill_manual(values=c("#999999", "#E69F00", "#56B4E9"))
```



```
# use brewer color palettes  
pie + scale_fill_brewer(palette="Dark2")
```



Create a pie chart from a factor variable

PlantGrowth data is used :

```
head(PlantGrowth)
```

```
> head(PlantGrowth)
```

```
  weight group
```

```
1  4.17  ctrl
```

```
2  5.58  ctrl
```

```
3  5.18  ctrl
```

```
4  6.11  ctrl
```

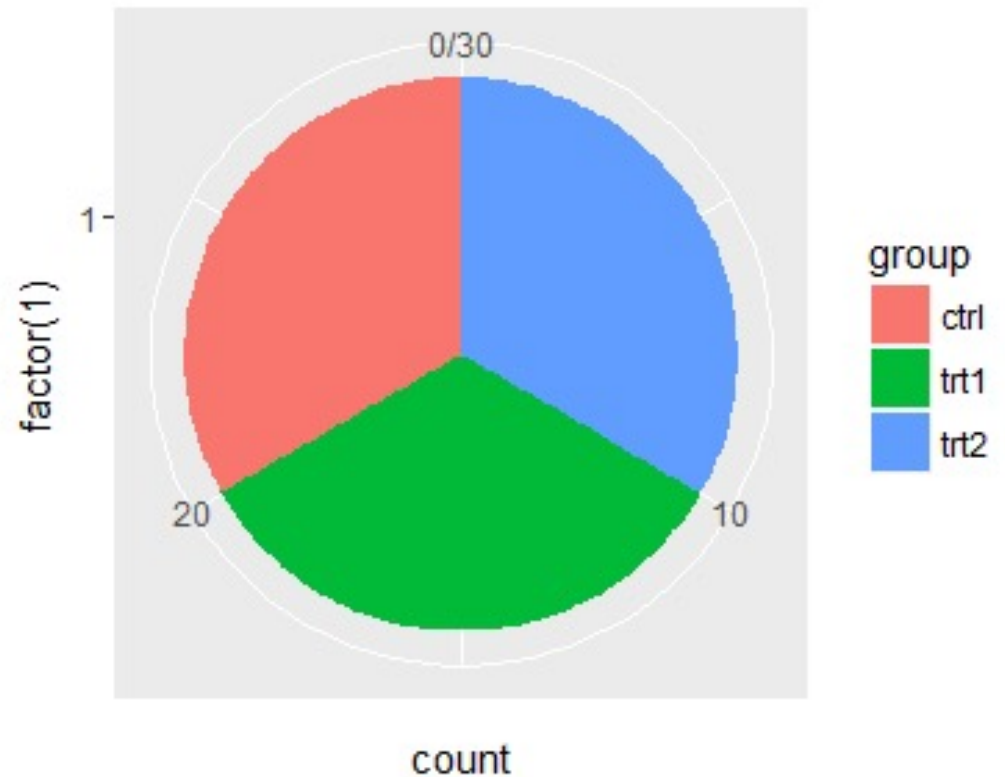
```
5  4.50  ctrl
```

```
6  4.61  ctrl
```

```
>
```

Create the pie chart of the count of observations in each group :

```
ggplot(PlantGrowth, aes(x=factor(1), fill=group))+  
  geom_bar(width = 1)+  
  coord_polar("y")
```



How to create a **histogram plot** using **ggplot2** package

The function **geom_histogram()** is used. You can also add a line for the mean using the function **geom_vline**

Example#1:

```
> chol <-
```

```
read.table(url("http://assets.datacamp.com/blog\_assets/chol.txt"),  
header = TRUE)
```

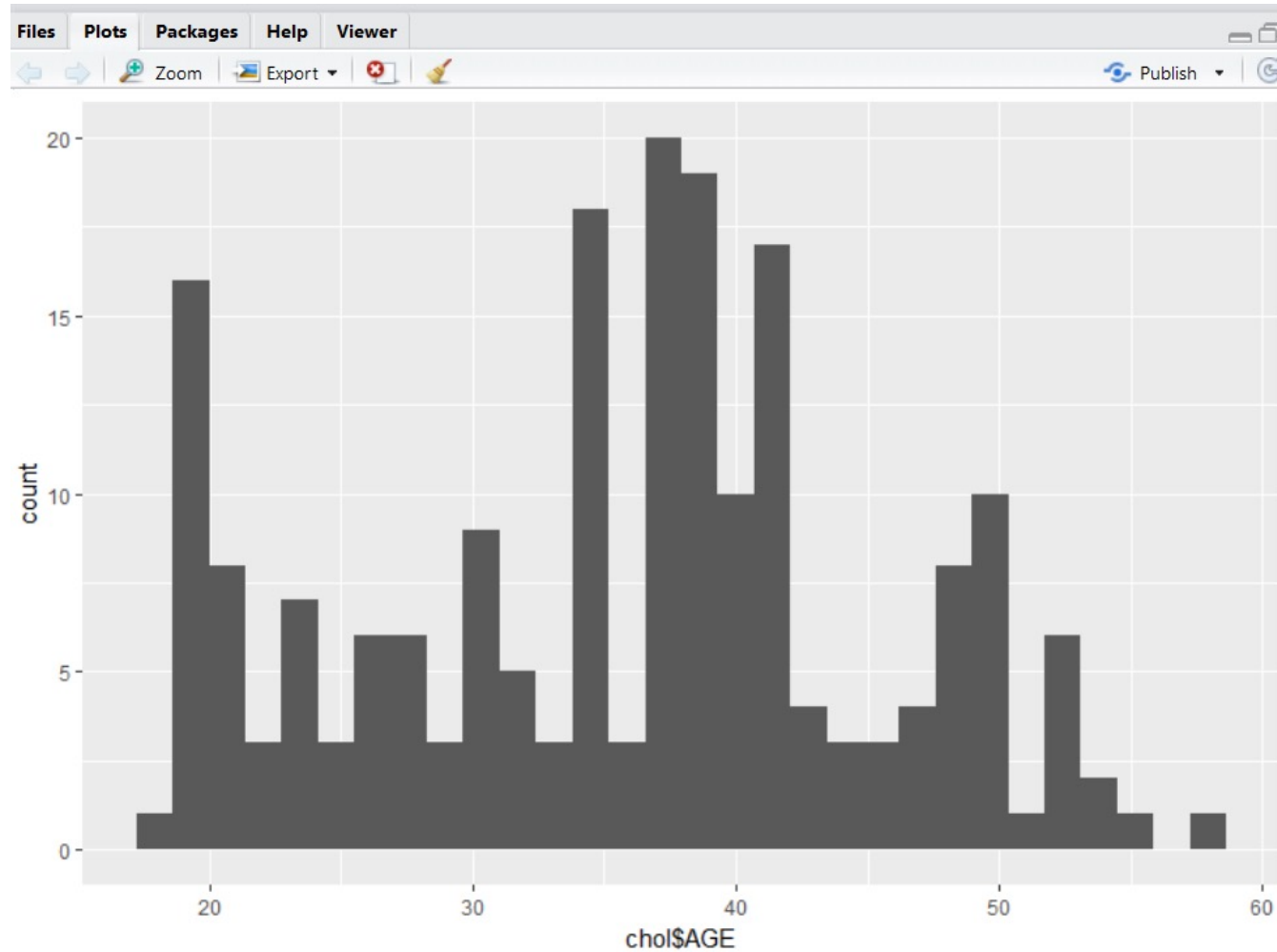
```
> head(chol)
```

	AGE	HEIGHT	WEIGHT	CHOL	SMOKE	BLOOD	MORT
1	20	176	77	195	nonsmo	b	alive
2	53	167	56	250	sigare	o	dead
3	44	170	80	304	sigare	a	dead
4	37	173	89	178	nonsmo	o	alive
5	26	170	71	206	sigare	o	alive
6	41	165	62	284	sigare	o	alive

```
>
```

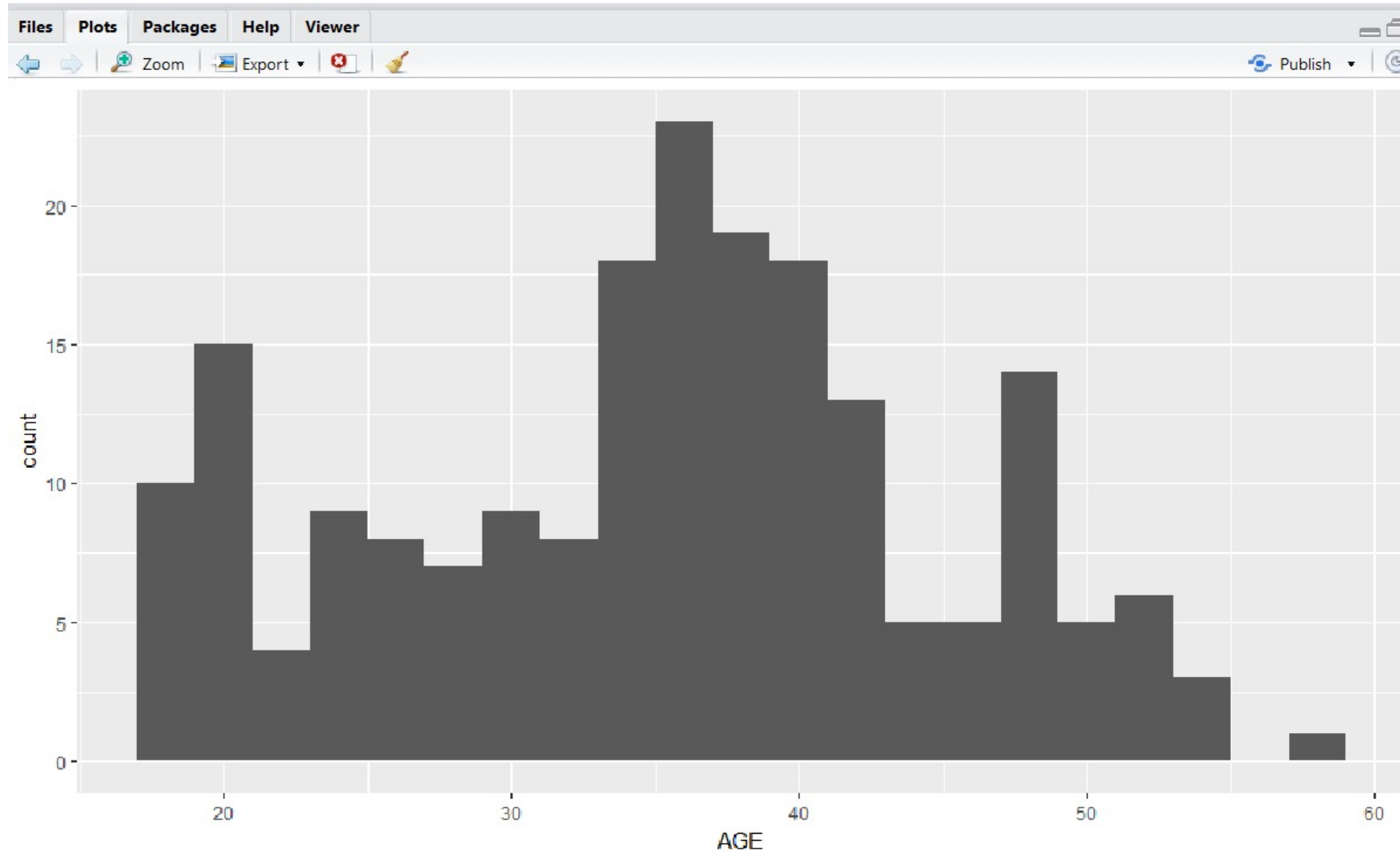
Basic histogram plots

```
> ggplot(chol, aes(x=AGE)) + geom_histogram()
```



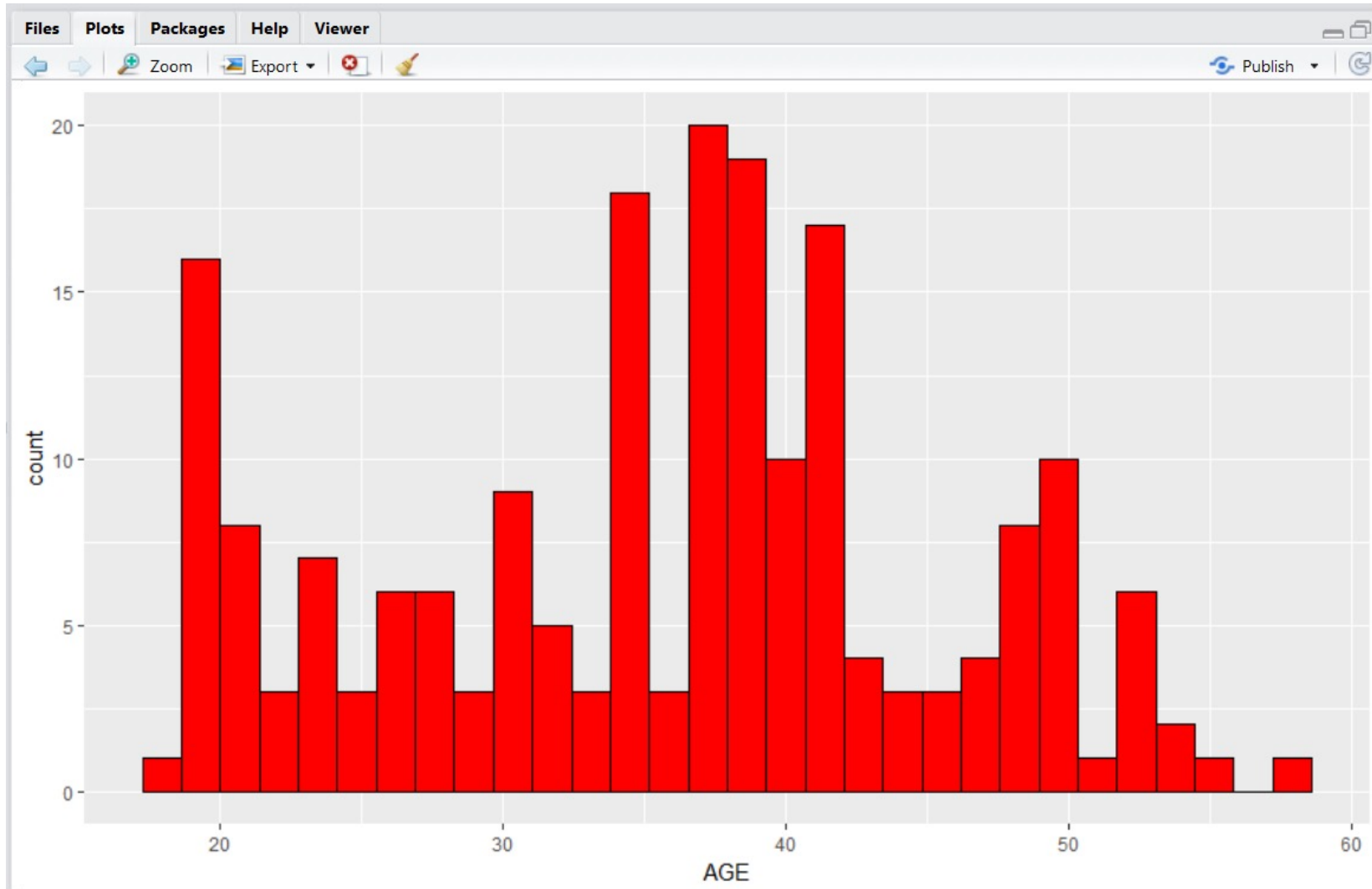
Change the width of bins

```
ggplot(chol, aes(x=AGE)) + geom_histogram(binwidth=2)
```



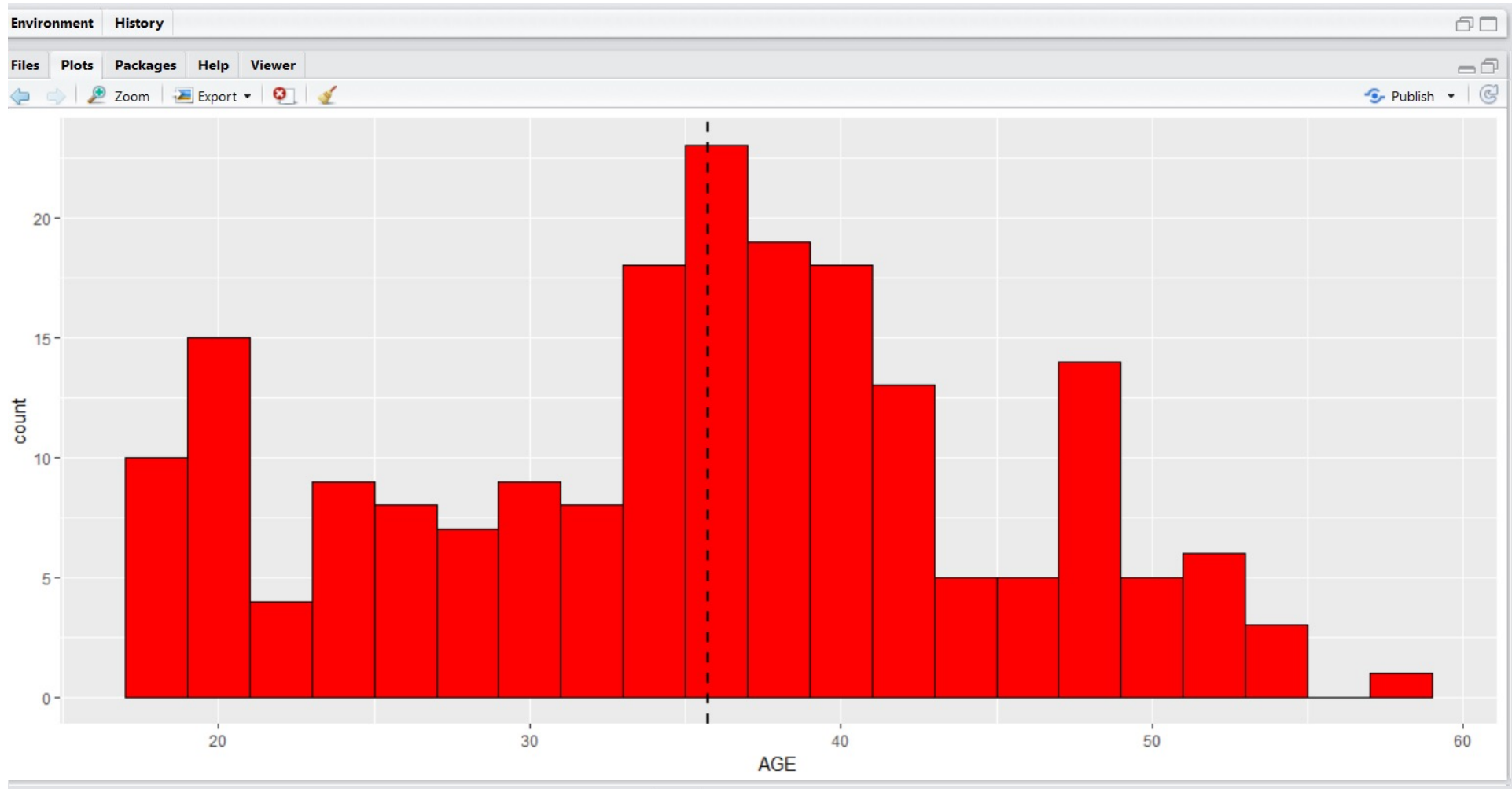
Change colors

```
ggplot(chol, aes(x=AGE)) + geom_histogram(color="black", fill="red")
```



Add mean line on the histogram

```
ggplot(chol, aes(x=AGE)) + geom_histogram(binwidth=2, color="black", fill="red")+  
geom_vline(aes(xintercept=mean(AGE)), color="blue", linetype="dashed", size=1)
```

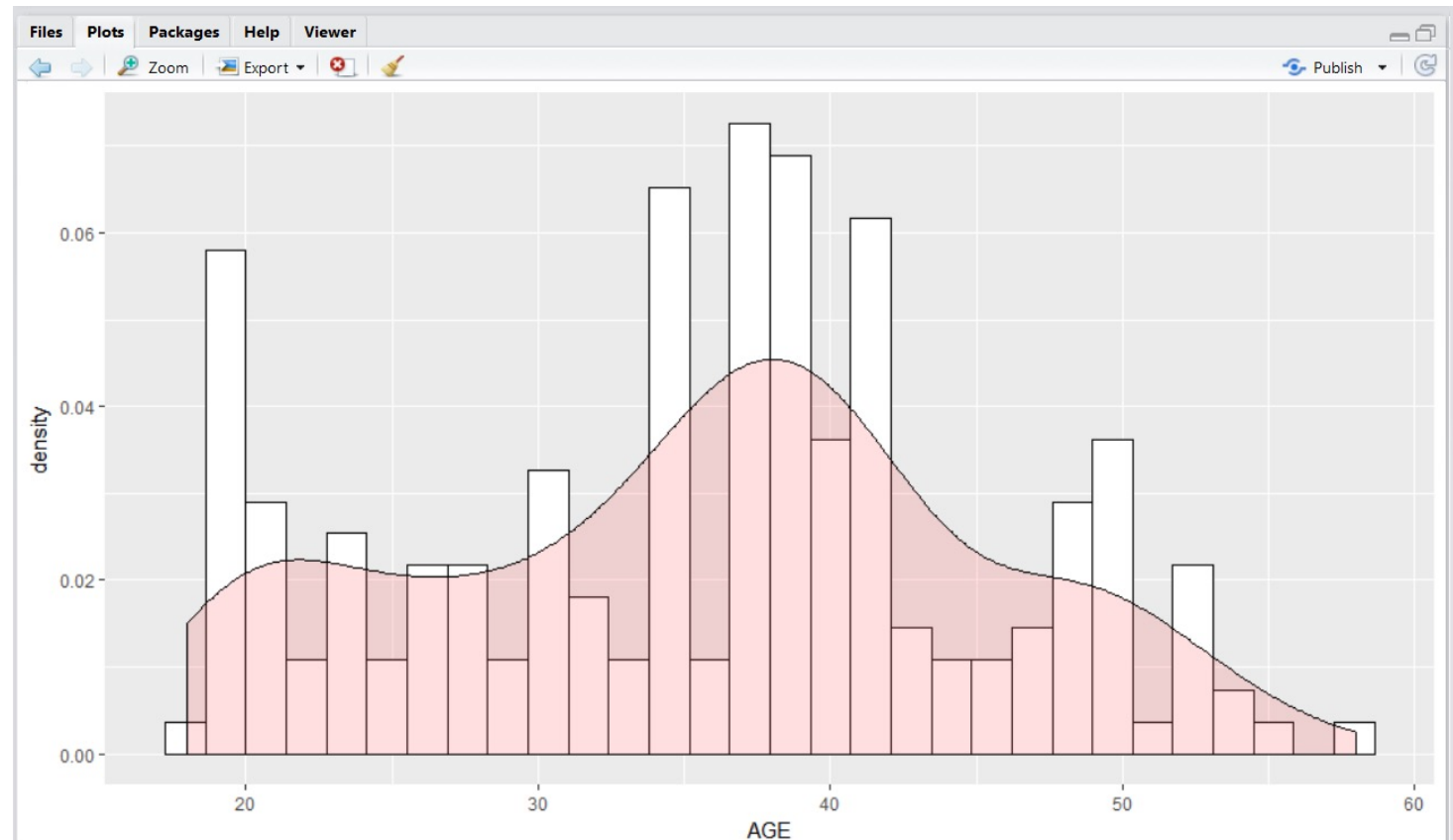


Add Density Plot on the Histogram

- The histogram is plotted with density instead of count on y-axis
- Overlay with transparent density plot. The value of alpha controls the level of transparency

```
ggplot(chol, aes(x=AGE)) + geom_histogram(aes(y=..density..), colour="black", fill="white")+  
geom_density(alpha=.2, fill="#FF6666")
```

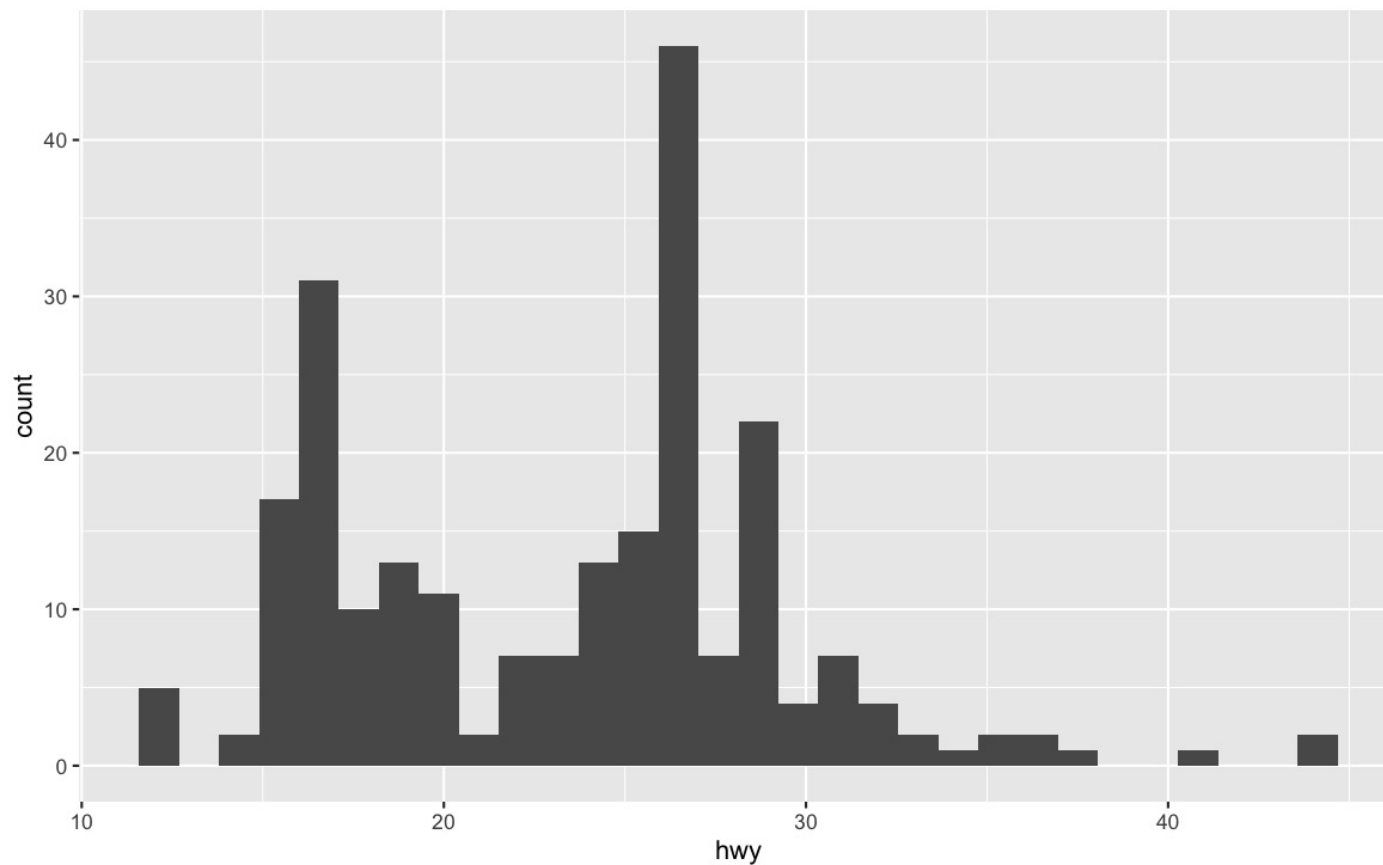
Read more on ggplot2 line
types : [ggplot2 line types](#)



Example#2

A [histogram](#) (useful to visualize distributions and detect potential [outliers](#)) can be plotted using `geom_histogram()`:

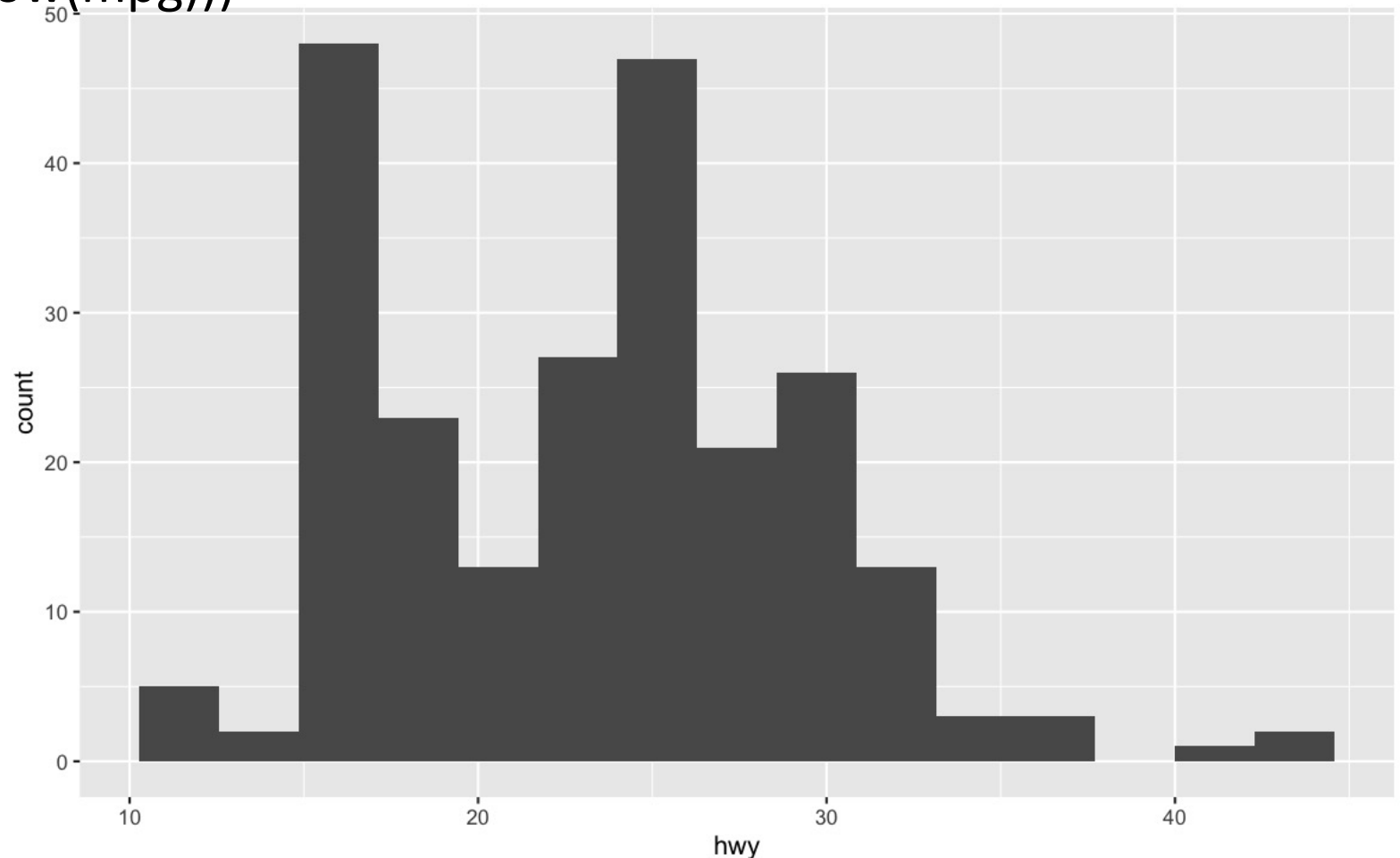
```
ggplot(mpg) +  
  aes(x = hwy) +  
  geom_histogram()
```



By default, the number of bins is equal to 30. You can change this value using the `bins` argument inside the `geom_histogram()` function:

```
ggplot(mpg) +  
  aes(x = hwy) +  
  geom_histogram(bins = sqrt(nrow(mpg)))
```

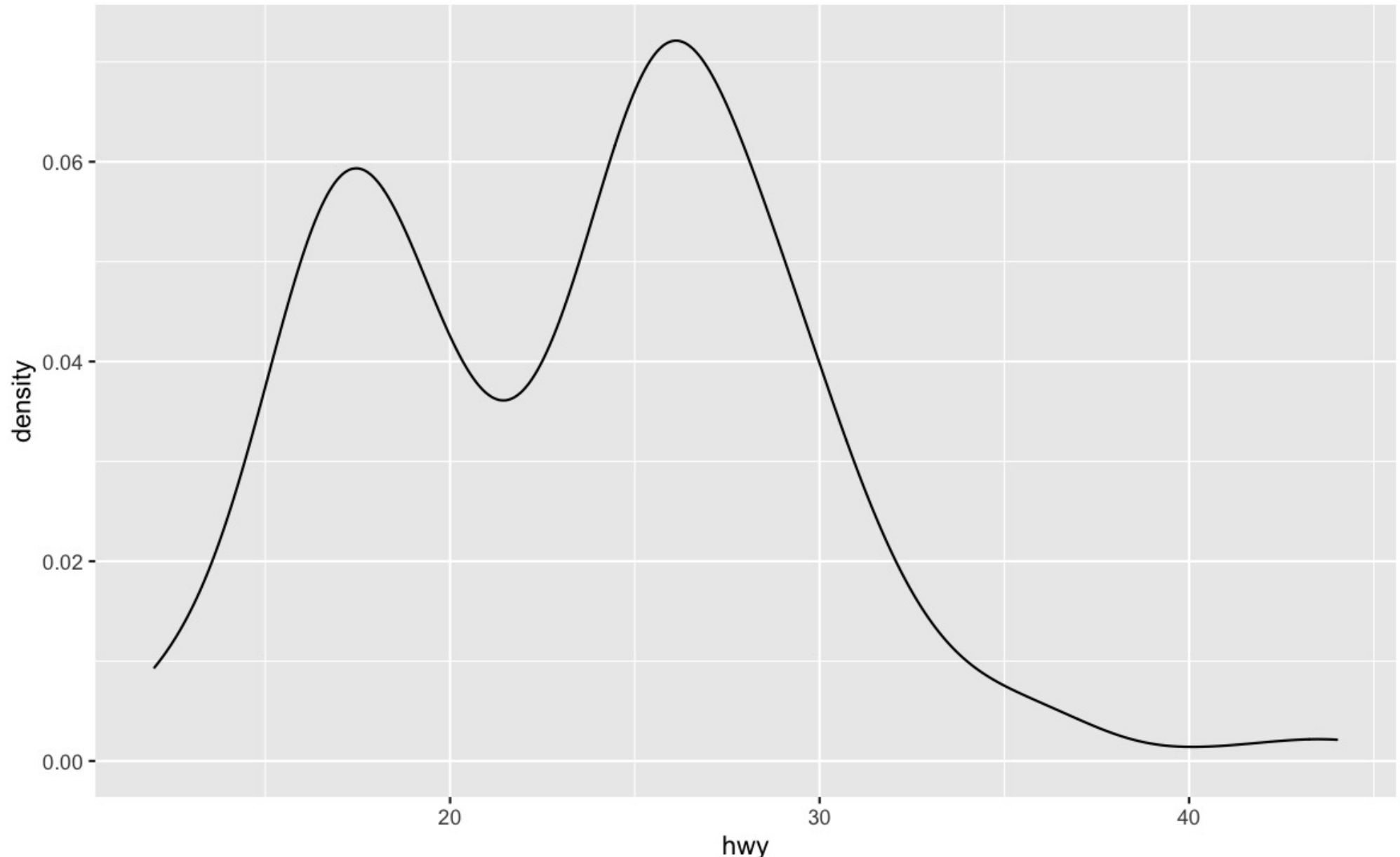
Here I specify the number of bins to be equal to the square root of the number of observations (following Sturge's rule) but you can specify any numeric value.



[Density plots](#) can be created using `geom_density()`

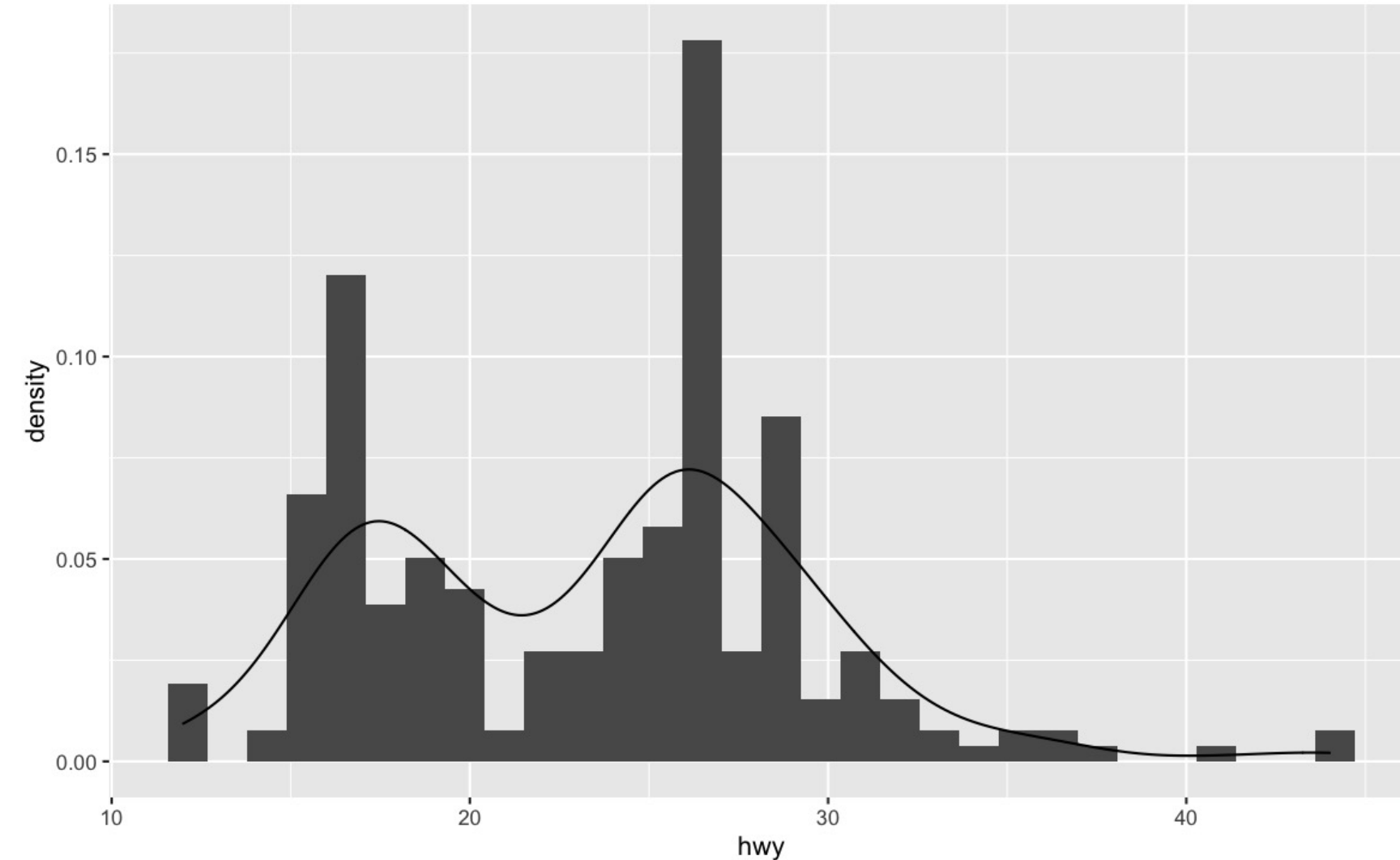
Density plot

```
ggplot(mpg) +  
  aes(x = hwy) +  
  geom_density()
```



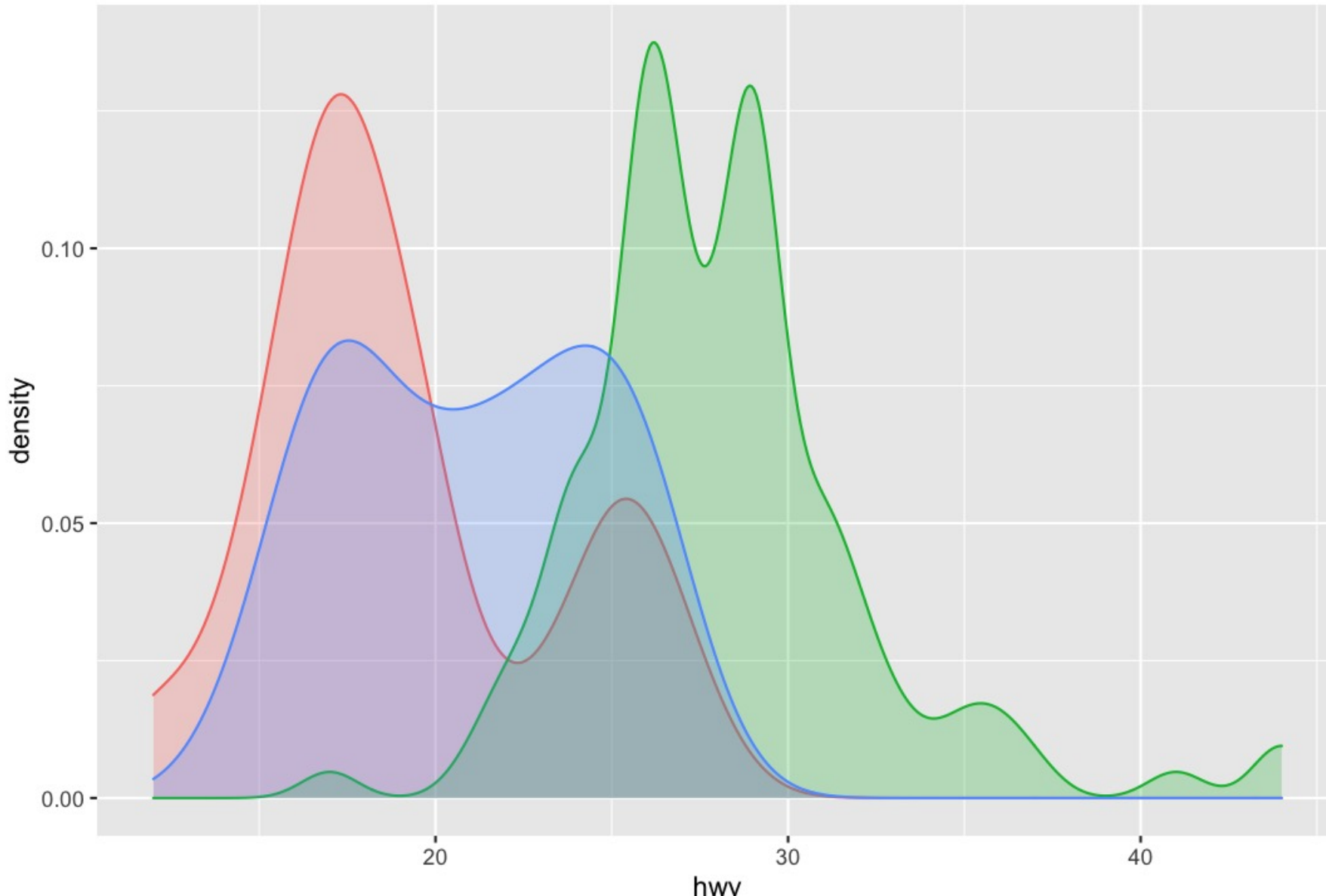
We can also superimpose a histogram and a density curve on the same plot:

```
ggplot(mpg) +  
  aes(x = hwy, y = ..density..) +  
  geom_histogram() +  
  geom_density()
```

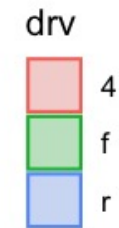


**Combination
of histogram
and densities**

```
ggplot(mpg) +  
  aes(x = hwy, color = drv, fill = drv) +  
  geom_density(alpha = 0.25) # add transparency
```



**Or
superimpose
several
densities:**



How to put Multiple graphs on one page (ggplot2)

Reference: Cookbook for R

The easy way is to use the ***multiplot function***

Exercise 1:

Plot the binomial distribution for $p = 0.3$, $p = 0.5$ and $p = 0.8$ and the total number of trials $n = 60$ as a function of k the number of successful trials. For each value of p , determine 1st Quartile, median, mean, standard deviation and the 3rd Quartile. Present those values as a vertical box plot with the probability p on the horizontal axis.

We begin by calculating the value of k

```
> k <- c(0:60); k  
[1] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26  
    27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53  
    54 55 56 57 58 59 60
```

Then, we calculate the distribution for each values of p by using

```
dbP0.3 <- dbinom(k, 60, 0.3); dbP0.3  
dbP0.5 <- dbinom(k, 60, 0.5); dbP0.5  
dbP0.8 <- dbinom(k, 60, 0.8); dbP0.8
```

After calculating the value of binomial distributions for each p, we can create the plots.

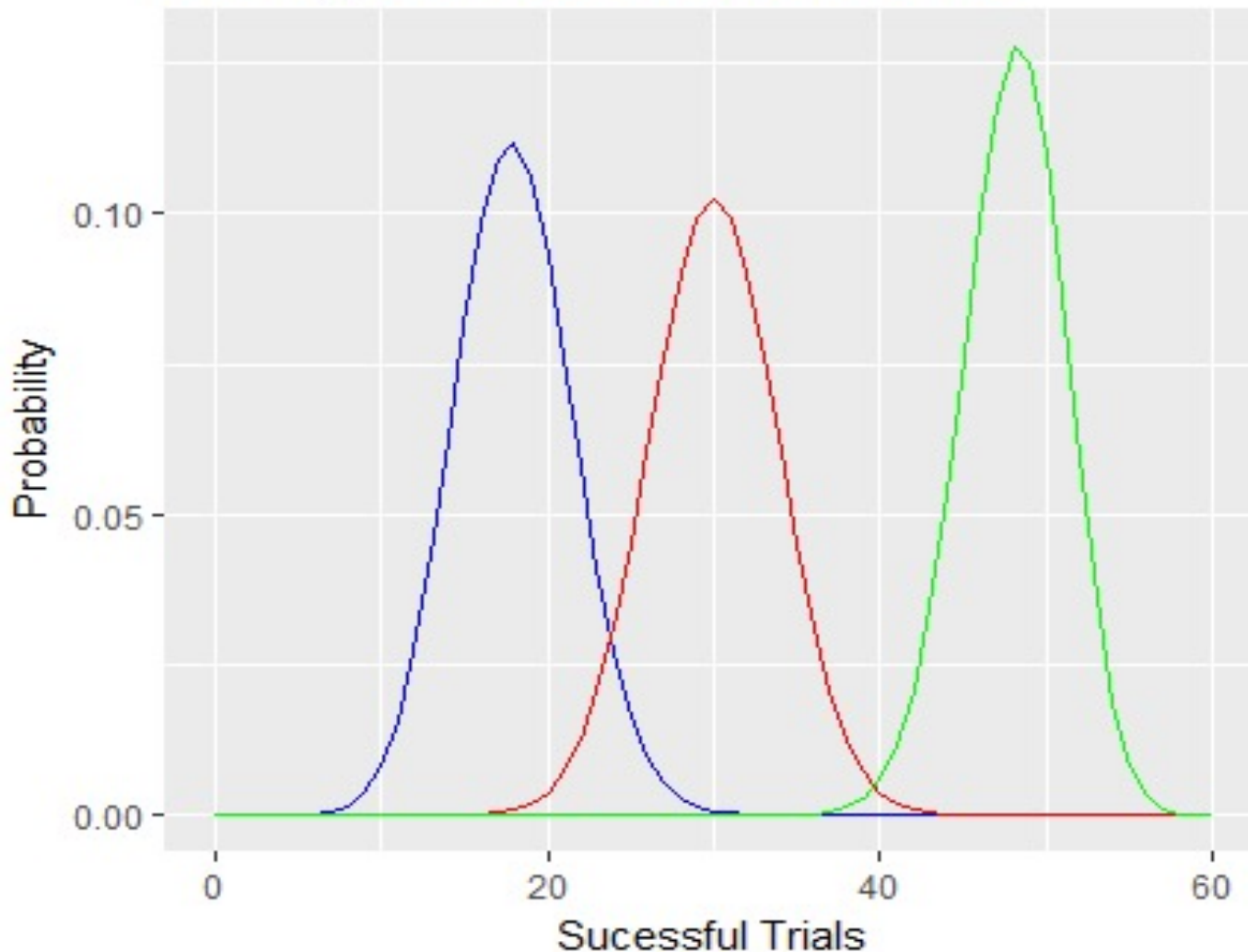
```
library(tidyverse)
k <- c(0:60); k
dbP0.3 <- dbinom(k, 60, 0.3); dbP0.3
dbP0.5 <- dbinom(k, 60, 0.5); dbP0.5
dbP0.8 <- dbinom(k, 60, 0.8); dbP0.8
df<- data.frame (k, dbP0.3,dbP0.5,dbP0.8);
df
```

After calculating the value of binomial distributions for each p , we can create the plots.

```
ggplot(df, aes(x = k))+
  geom_line(aes(y = dbP0.3), colour="blue")+
  geom_line(aes(y = dbP0.5), colour = "red")+
  geom_line(aes(y = dbP0.8), colour = "green")+
  ylab(label="Probability")+
  xlab("Sucessful Trials")+
  ggtitle("Density of Binomial Distributions")+
  theme(plot.title = element_text(lineheight=.8, face="bold"))
```

For each value of p , determine 1st Quartile, median, mean, standard deviation and the 3rd Quartile.

Density of Binomial Distributions



```
quantile(dbP0.3)  
quantile(dbP0.5)  
quantile(dbP0.8)
```

```
median(dbP0.3)  
median(dbP0.5)  
median(dbP0.8)
```

```
mean(dbP0.3)  
mean(dbP0.5)  
mean(dbP0.8)
```

```
sd(dbP0.3)  
sd(dbP0.5)  
sd(dbP0.8)
```

```
> quantile(dbP0.3)
      0%      25%      50%      75%     100%
4.239116e-32 7.460887e-13 8.357380e-06 9.613404e-03
1.118036e-01
```

The 1st Quartile is 7.460887e-13

```
> quantile(dbP0.5)
      0%      25%      50%      75%     100%
8.673617e-19 3.349811e-10 4.613852e-05 1.227688e-02
1.025782e-01
```

The 1st Quartile is 3.349811e-10

```
> quantile(dbP0.8)
      0%      25%      50%      75%     100%
1.152922e-42 6.585109e-20 1.572006e-07 5.842579e-03
1.278228e-01
```

The 1st Quartile is 6.585109e-20

median

```
> median(dbP0.3)
```

```
[1] 8.35738e-06
```

```
> median(dbP0.5)
```

```
[1] 4.613852e-05
```

```
> median(dbP0.8)
```

```
[1] 1.572006e-07
```

mean

```
> mean(dbP0.3)
```

```
[1] 0.01639344
```

```
> mean(dbP0.5)
```

```
[1] 0.01639344
```

```
> mean(dbP0.8)
```

```
[1] 0.01639344
```

standard deviation

```
> sd(dbP0.3)
```

```
[1] 0.03239755
```

```
> sd(dbP0.5)
```

```
[1] 0.03062992
```

```
> sd(dbP0.8)
```

```
[1] 0.03527981
```

3rd Quartile

> quantile(dbP0.3)

0%	25%	50%	75%	100%
4.239116e-32	7.460887e-13	8.357380e-06	9.613404e-03	1.118036e-01

The 3st Quartile is 9.613404e-03

> quantile(dbP0.5)

0%	25%	50%	75%	100%
8.673617e-19	3.349811e-10	4.613852e-05	1.227688e-02	1.025782e-01

The 3st Quartile is 1.227688e-02

> quantile(dbP0.8)

0%	25%	50%	75%	100%
1.152922e-42	6.585109e-20	1.572006e-07	5.842579e-03	1.278228e-01

The 1st Quartile is 5.842579e-03

Exercise 2: Present those values as a vertical box plot with the probability p on the horizontal axis.

References

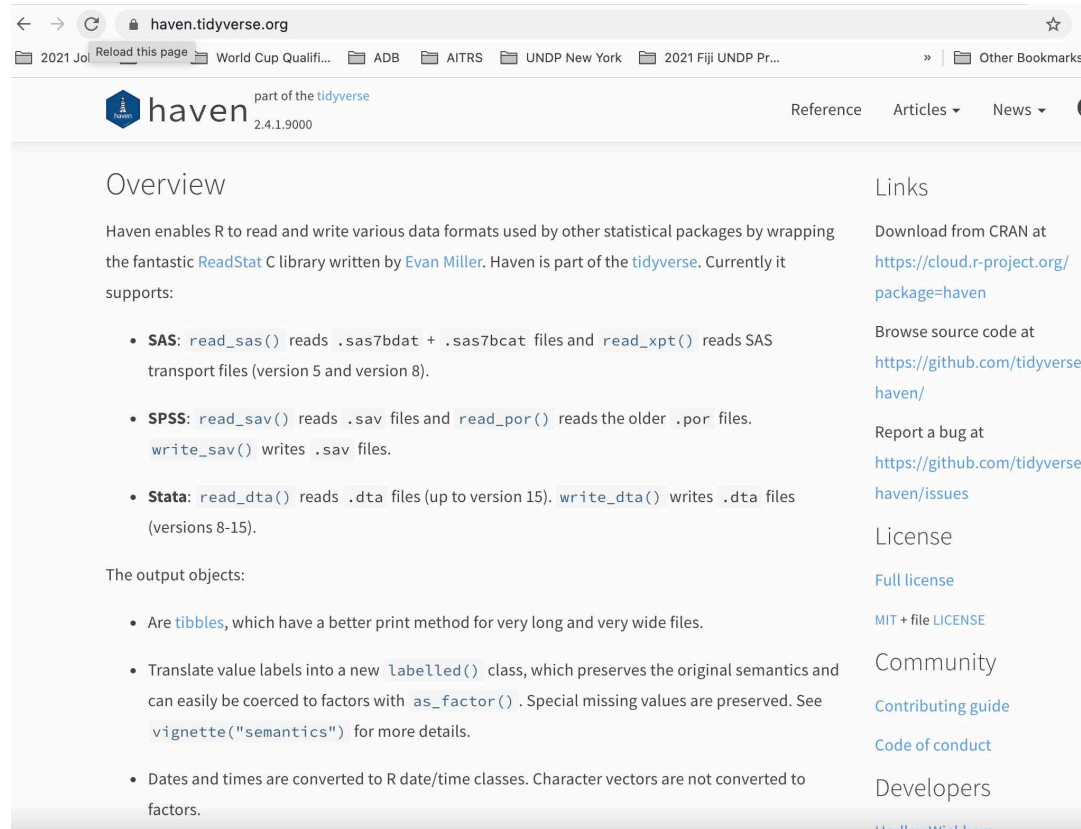
By now you have seen that `{ggplot2}` is a very powerful and complete package to create plots in R. This article illustrated only the tip of the iceberg, and you will find many tutorials on how to create more advanced plots and visualizations with `{ggplot2}` online. If you want to learn more than what is described in the present article, I highly recommend starting with:

- the chapters [Data visualisation](#) and [Graphics for communication](#) from the book [R for Data Science](#) from Garrett Grolemund and Hadley Wickham
- the book [ggplot2: Elegant Graphics for Data Analysis](#) from Hadley Wickham
- the book [R Graphics Cookbook](#) from Winston Chang
- the [ggplot2 extensions guide](#) which lists many of the packages that extend `{ggplot2}`
- the [{ggplot2} cheat sheet](#)

Import Data from Statistical Softwares SAS/SPSS/STATA into R

Haven Package part of tidyverse

<https://haven.tidyverse.org/>



The screenshot shows the web browser interface for the haven.tidyverse.org website. The browser's address bar displays the URL. The website header includes the 'haven' logo, the text 'part of the tidyverse', and the version number '2.4.1.9000'. Navigation links for 'Reference', 'Articles', and 'News' are present. The main content area is titled 'Overview' and describes the package's purpose: enabling R to read and write various data formats by wrapping the ReadStat C library. It lists supported formats: SAS (read_sas(), read_xpt()), SPSS (read_sav(), write_sav(), read_por()), and Stata (read_dta(), write_dta()). It also mentions the output objects, specifically 'tibbles'. A sidebar on the right contains links for downloading from CRAN, browsing source code, reporting bugs, license information, and community resources.

haven part of the tidyverse 2.4.1.9000

Reference Articles News

Overview

Haven enables R to read and write various data formats used by other statistical packages by wrapping the fantastic [ReadStat C](#) library written by [Evan Miller](#). Haven is part of the [tidyverse](#). Currently it supports:

- **SAS:** `read_sas()` reads `.sas7bdat` + `.sas7bcat` files and `read_xpt()` reads SAS transport files (version 5 and version 8).
- **SPSS:** `read_sav()` reads `.sav` files and `read_por()` reads the older `.por` files. `write_sav()` writes `.sav` files.
- **Stata:** `read_dta()` reads `.dta` files (up to version 15). `write_dta()` writes `.dta` files (versions 8-15).

The output objects:

- Are [tibbles](#), which have a better print method for very long and very wide files.
- Translate value labels into a new `labelled()` class, which preserves the original semantics and can easily be coerced to factors with `as_factor()`. Special missing values are preserved. See `vignette("semantics")` for more details.
- Dates and times are converted to R date/time classes. Character vectors are not converted to factors.

Links

Download from CRAN at <https://cloud.r-project.org/package=haven>

Browse source code at <https://github.com/tidyverse/haven/>

Report a bug at <https://github.com/tidyverse/haven/issues>

License

[Full license](#)

[MIT + file LICENSE](#)

Community

[Contributing guide](#)

[Code of conduct](#)

Developers

[Hadley Wickham](#)

The easiest way to get haven is to install the whole tidyverse:
`install.packages("tidyverse")`

Alternatively, install just haven:
`install.packages("haven")`

SAS

```
library(haven)
```

```
read_sas("iris.sas7bdat")
```

```
write_sas(iris, "iris.sas7bdat")
```

```
> read_sas("iris.sas7bdat")  
# A tibble: 150 x 5  
  Sepal_Length Sepal_Width Petal_Length Petal_Width Species  
    <dbl>      <dbl>      <dbl>      <dbl> <chr>  
1      5.1      3.5      1.4      0.2 setosa  
2      4.9      3      1.4      0.2 setosa  
3      4.7      3.2      1.3      0.2 setosa  
4      4.6      3.1      1.5      0.2 setosa  
5      5      3.6      1.4      0.2 setosa  
6      5.4      3.9      1.7      0.4 setosa  
7      4.6      3.4      1.4      0.3 setosa  
8      5      3.4      1.5      0.2 setosa  
9      4.4      2.9      1.4      0.2 setosa  
10     4.9      3.1      1.5      0.1 setosa  
# ... with 140 more rows  
> |
```

SPSS

```
read_sav("iris.sav")
```

```
write_sav(iris, "iris.sav")
```

```
> read_sav("iris.sav")
```

```
# A tibble: 150 x 5
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl+lbl>
1	5.1	3.5	1.4	0.2	1 [setosa]
2	4.9	3	1.4	0.2	1 [setosa]
3	4.7	3.2	1.3	0.2	1 [setosa]
4	4.6	3.1	1.5	0.2	1 [setosa]
5	5	3.6	1.4	0.2	1 [setosa]
6	5.4	3.9	1.7	0.4	1 [setosa]
7	4.6	3.4	1.4	0.3	1 [setosa]
8	5	3.4	1.5	0.2	1 [setosa]
9	4.4	2.9	1.4	0.2	1 [setosa]
10	4.9	3.1	1.5	0.1	1 [setosa]

```
# ... with 140 more rows
```

```
> |
```

Stata

```
read_dta("iris.dta")  
write_dta(iris, "iris.dta")
```

```
> read_dta("iris.dta")  
# A tibble: 150 x 5  
  sepallength sepalwidth petallength petalwidth species  
    <dbl>      <dbl>      <dbl>      <dbl> <chr>  
1      5.10      3.5        1.40      0.200 setosa  
2      4.90      3          1.40      0.200 setosa  
3      4.70      3.20        1.30      0.200 setosa  
4      4.60      3.10        1.5        0.200 setosa  
5      5          3.60        1.40      0.200 setosa  
6      5.40      3.90        1.70      0.400 setosa  
7      4.60      3.40        1.40      0.300 setosa  
8      5          3.40        1.5        0.200 setosa  
9      4.40      2.90        1.40      0.200 setosa  
10     4.90      3.10        1.5        0.100 setosa  
# ... with 140 more rows  
> |
```

```
library(haven)
```

```
# SAS
```

```
read_sas("iris.sas7bdat")
```

```
write_sas(iris, "iris.sas7bdat")
```

```
# SPSS
```

```
read_sav("iris.sav")
```

```
write_sav(iris, "iris.sav")
```

```
# Stata
```

```
read_dta("iris.dta")
```

```
write_dta(iris, "iris.dta")
```

Exercise (1): Read and write **mtcars data** in all statistical softwares SAS/SPSS/STATA data file formats

Thanks you