Graphics for Communication

Introduction

- In previous sections, we have learned about Exploratory Data Analysis and Data Visualization
- Now that you understand your data, you need to *communicate* your understanding to others.
- When you make exploratory plots, you have to know which variables the plot will display.
- Graphics for communication include
  - ✓Label
  - ✓Annotations
  - ✓Scales
  - ✓Zooming
  - ✓Themes
  - ✓Saving your plots

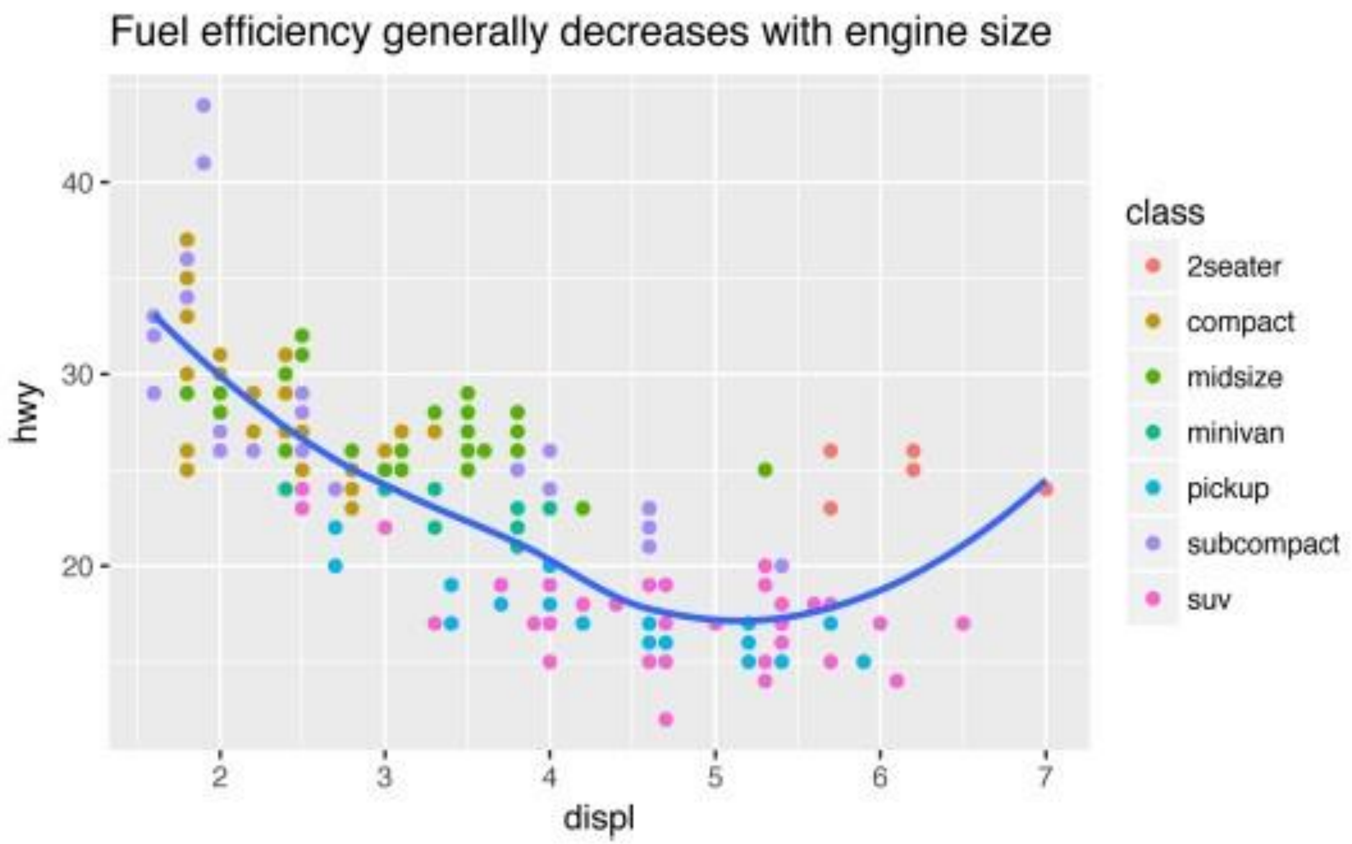
Label

Label

- The easiest place to start when turning an exploratory data analysis into an expository graphic is with good labels.
- You add labels with the **labs()** function.

Example: adds a plot title

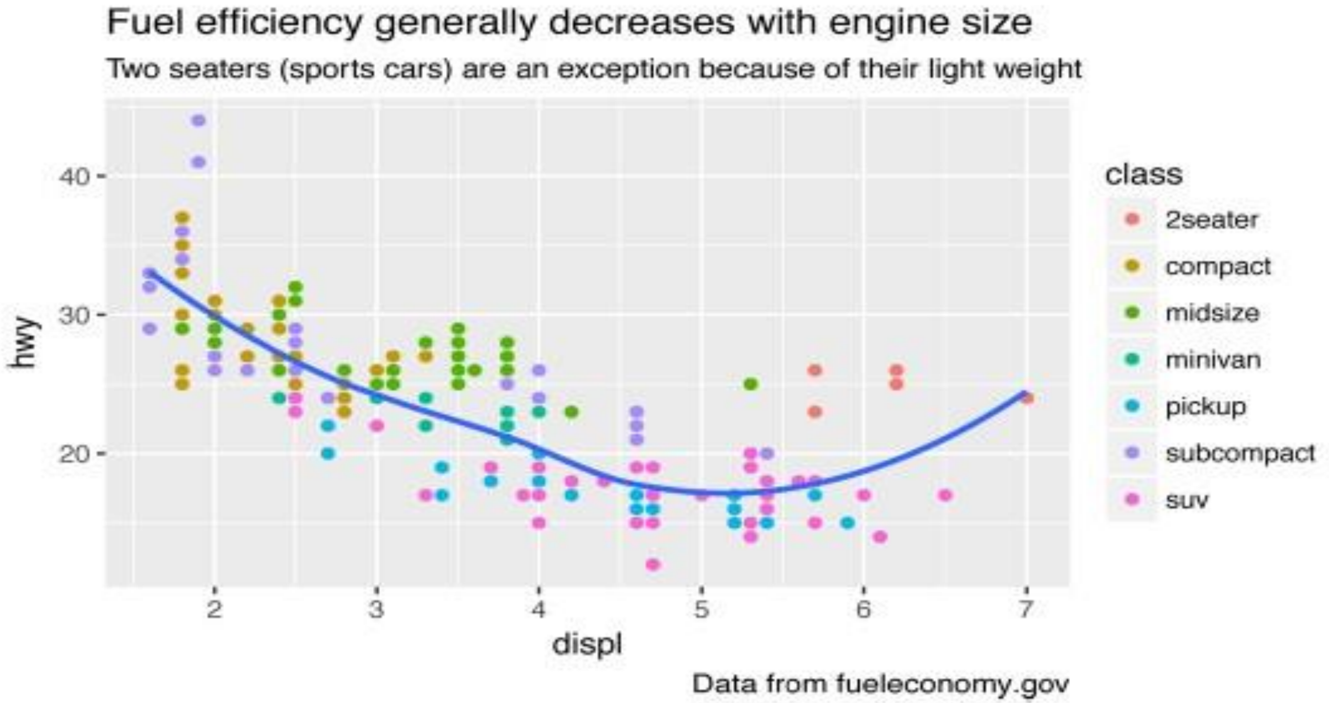
```
ggplot(mpg, aes(displ, hwy)) +
 geom_point(aes(color = class)) +
 geom_smooth(se = FALSE) +
 labs(title = paste("Fuel efficiency generally decreases with engine size"))
```



- The purpose of a plot title is to summarize the main finding.
- Avoid titles that just describe what the plot is, e.g., “A scatterplot of engine displacement vs. fuel economy.”

- If you need to add more text, there are two other useful labels that you can use in **ggplot2**
  - ✓**subtitle** adds additional detail in a smaller font beneath the title.
  - ✓**caption** adds text at the bottom right of the plot, often used to describe the source of the data:

Example:
```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(color = class)) +
 geom_smooth(se = FALSE) +
 labs(title = paste("Fuel efficiency generally decreases with engine
 size“),
subtitle = paste("Two seaters (sports cars) are an exception because
of their light weight"),
caption = "Data from fueleconomy.gov")
```



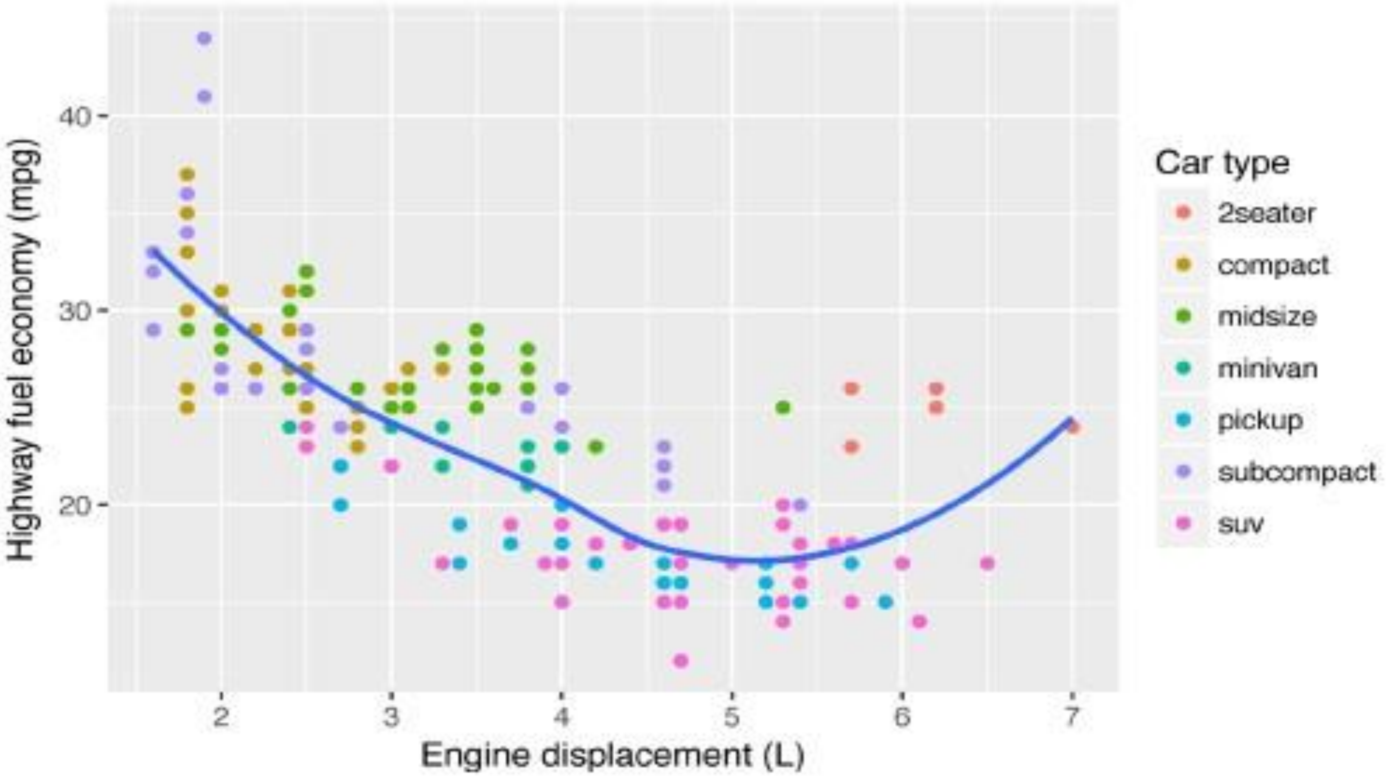
Fuel efficiency generally decreases with engine size

Two seaters (sports cars) are an exception because of their light weight

Data from fueleconomy.gov

- You can also use labs() to replace the axis and legend titles. It’s usually a good idea to replace short variable names with more detailed descriptions.

```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(title = " ", x = "Engine displacement (L)", y = "Highway fuel economy(mpg)",
  colour = "Car type")
```



- It’s possible to use mathematical equations instead of text strings.
- Just switch "" out for quote() and read about the available options in ?plotmath:

Example: df <- tibble(

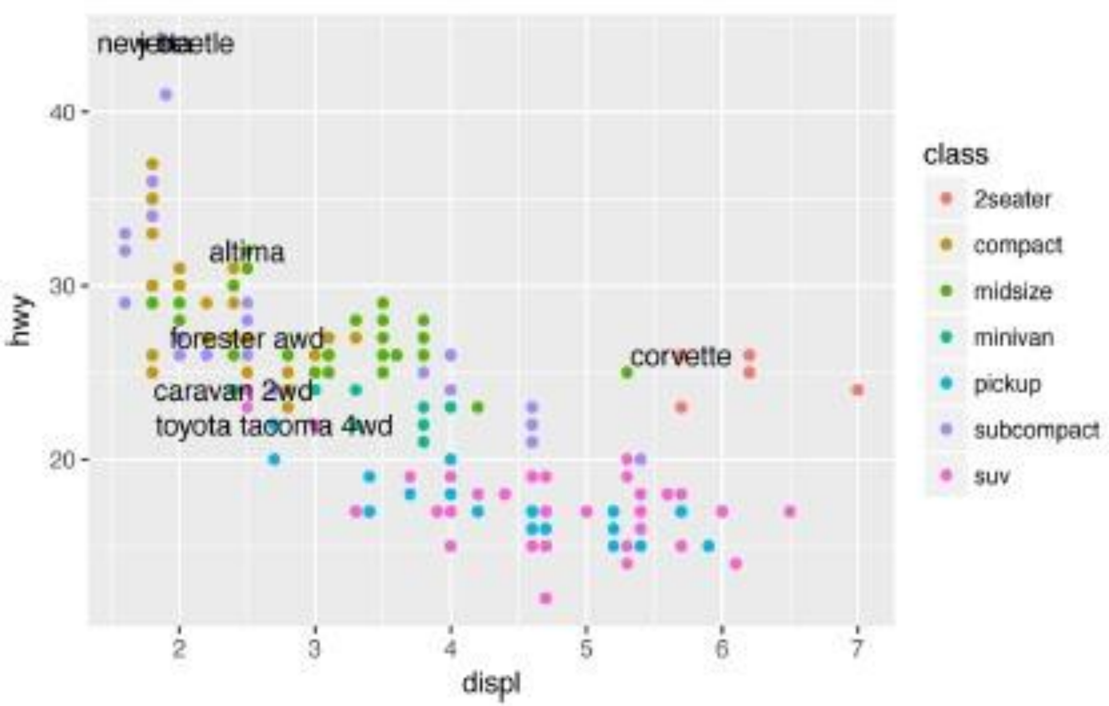
```
                    x = runif(10),
                    y = runif(10)
                   )
          ggplot(df, aes(x, y)) +
            geom_point() +
              labs(
                  x = quote(sum(x[i] ^ 2, i == 1, n)),
                  y = quote(alpha + beta + frac(delta, theta))
                 )
```

Annotations

Annotations

- In addition to labeling major components of your plot, it’s often useful to label individual observations or groups of observations.
- The first tool you have is geom_text().
- geom_text() is similar to geom_point(), but it has an additional aesthetic: label.
- This makes it possible to add textual labels to your plots.
- There are two possible sources of labels.
- First, you might have a tibble that provides labels.

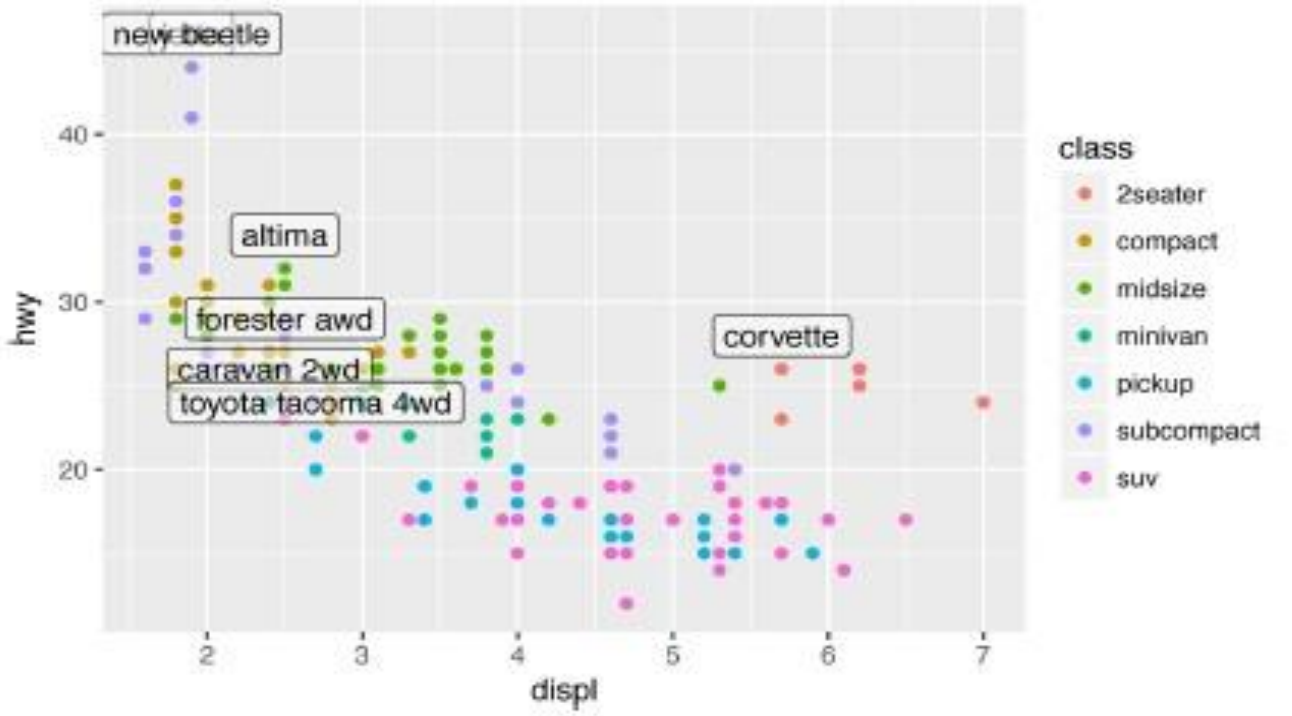
```
best_in_class <- mpg %>%
group_by(class) %>%
filter(row_number(desc(hwy)) == 1)

ggplot(mpg, aes(displ, hwy)) +
geom_point(aes(color = class)) +
geom_text(aes(label = model),
                    data = best_in_class)
```



- This is hard to read because the labels overlap with each other, and with the points.
- We can make things a little better by switching to **geom_label(),** which draws a rectangle behind the text.
- We also use the **nudge_y** parameter to move the labels slightly above the corresponding points.

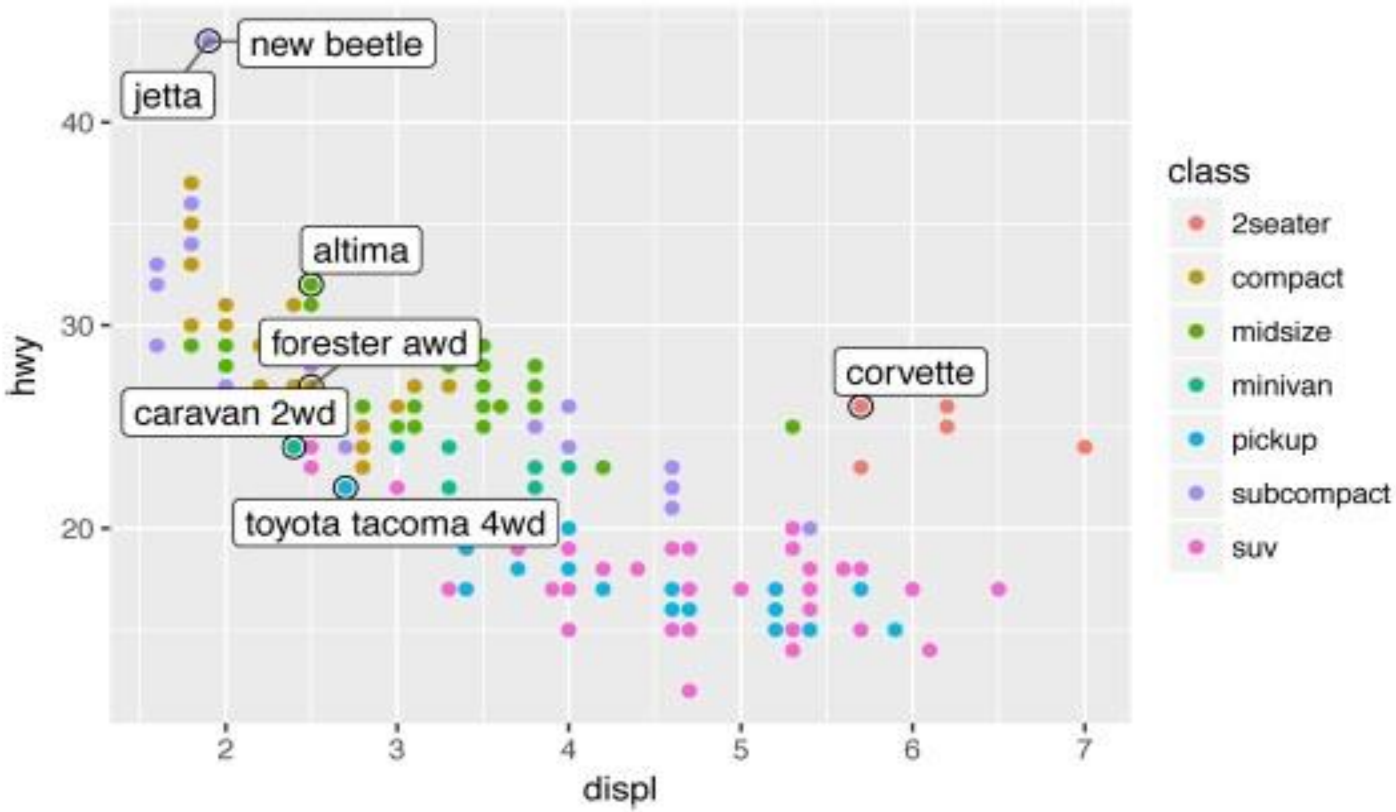
Example:

```
ggplot(mpg, aes(displ, hwy)) +
    geom_point(aes(color = class)) +
    geom_label( aes(label = model),
                data = best_in_class,
                nudge_y = 2,
                alpha = 0.5)
```



- But if you look closely in the top lefthand corner, you'll notice that there are two labels practically on top of each other.
- Instead, we can fix these by using **ggrepel** package by Kamil Slowikowski. This package will automatically adjust labels so that they don't overlap.

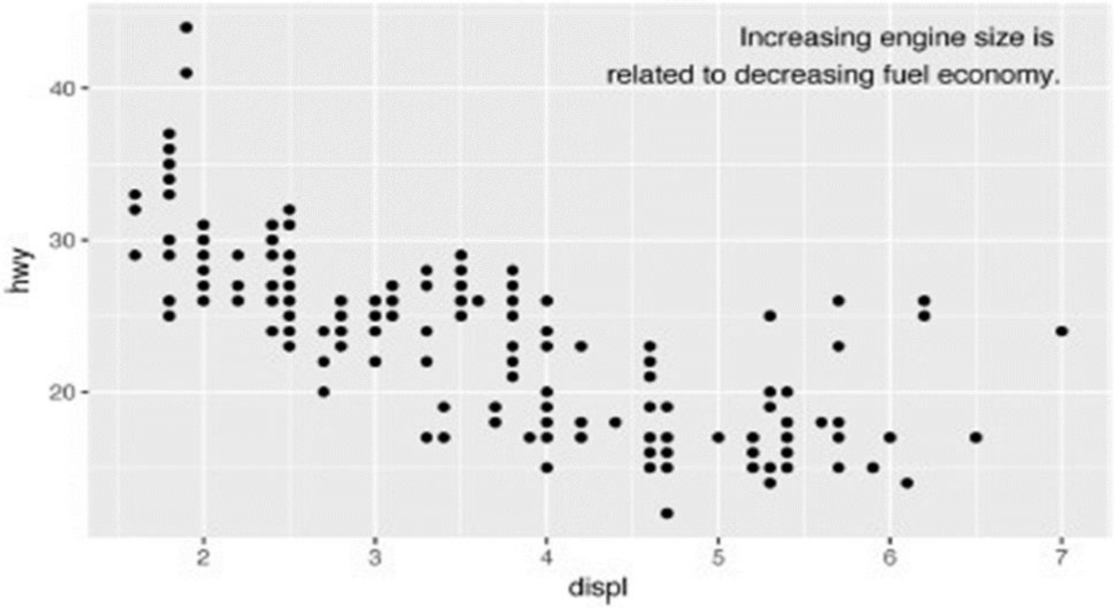
```
Example: ggplot(mpg, aes(displ, hwy)) +
              geom_point(aes(color = class)) +
              geom_point(size = 3, shape = 1, data = best_in_class) +
              ggrepel::geom_label_repel(aes(label = model),
                                                   data = best_in_class)
```



- If you want to add a single label to the plot, but you want the label in the corner of the plot, so it's convenient to create a new data frame using **summarize()** to compute the maximum values of x and y.

Example:

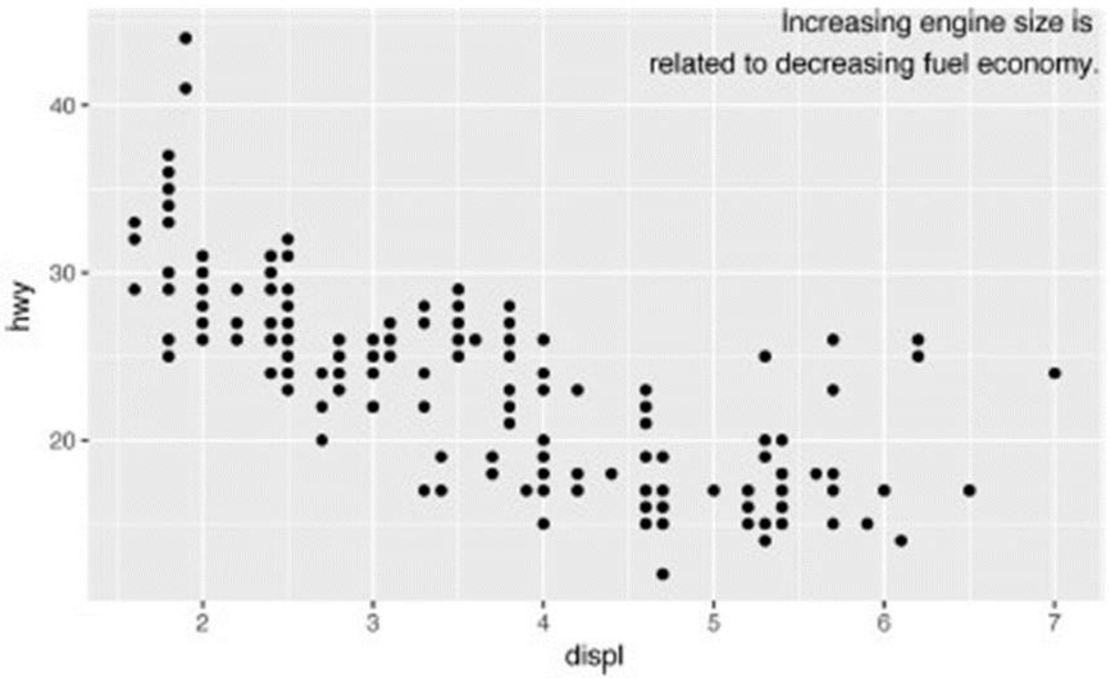
```
label <- mpg %>%
         summarize(
           displ = max(displ),
           hwy = max(hwy),
           label = paste( "Increasing engine size is \nrelated to"
                          "decreasing fuel economy."))
       ggplot(mpg, aes(displ, hwy)) +
         geom_point() +
         geom_text(
               aes(label = label),
               data = label,
               vjust = "top",
               hjust = "right")
```



- If you want to place the text exactly on the borders of the plot, you can use **+Inf** and **-Inf.**

Example:

```
label <- mpg %>%
        summarize(
          displ = Inf,
          hwy = Inf,
          label = paste("Increasing engine size is \nrelated to"
                        "decreasing fuel economy."))
      ggplot(mpg, aes(displ, hwy))
        geom_point() +
        geom_text(
              aes(label = label),
              data = label,
              vjust = "top",
              hjust = "right")
```



- Another approach is to use stringr::str_wrap() to automatically add line breaks, given the number of characters you want per line:

```
"Increasing engine size related to decreasing fuel economy." %>%
stringr::str_wrap(width = 40) %>%
writeLines()
```

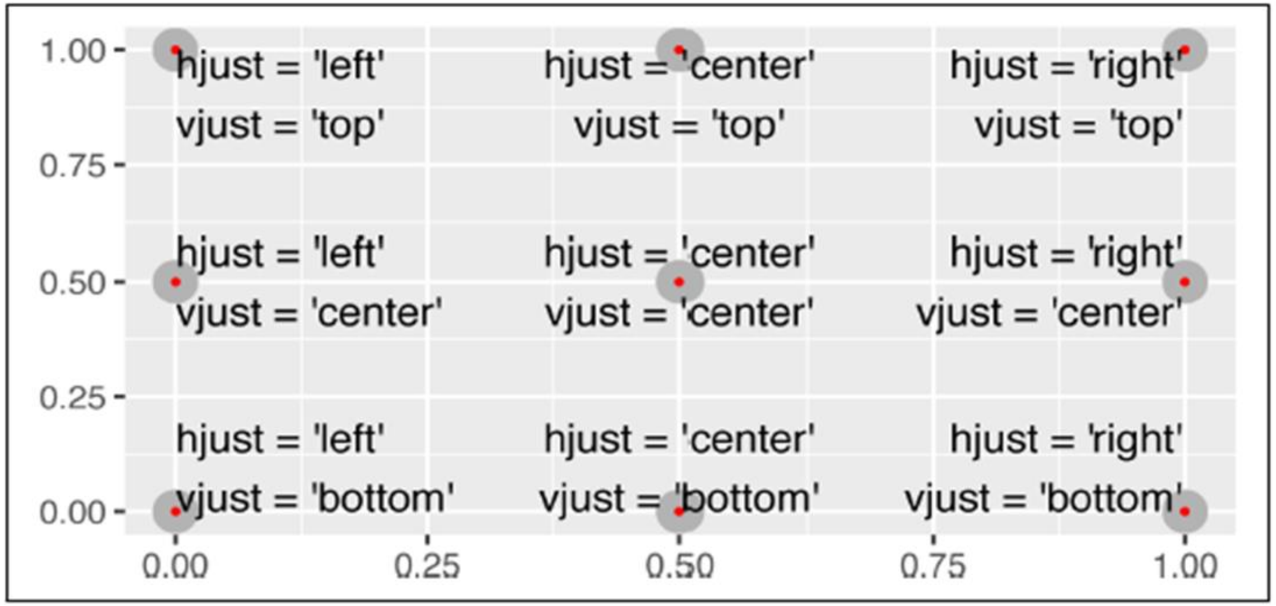


Figure 22-1. All nine combinations of hjust and vjust

- In addition to geom_text(), you have many other geoms in **ggplot2** available to help annotate your plot. A few ideas:

- Use `geom_hline()` and `geom_vline()` to add reference lines. I often make them thick (`size = 2`) and white (`color = white`), and draw them underneath the primary data layer. That makes them easy to see, without drawing attention away from the data.
- Use `geom_rect()` to draw a rectangle around points of interest. The boundaries of the rectangle are defined by the `xmin`, `xmax`, `ymin`, and `ymax` aesthetics.
- Use `geom_segment()` with the `arrow` argument to draw attention to a point with an arrow. Use the `x` and `y` aesthetics to define the starting location, and `xend` and `yend` to define the end location.

Scales

Scales

- The third way you can make your plot better for communication is to adjust the scales.
- Scales control the mapping from data values to things that you can perceive. **ggplot2** automatically adds scales.

Example:
```
ggplot(mpg, aes(displ, hwy)) +
            geom_point(aes(color = class))
```

- **ggplot2** automatically adds default scales behind the scenes:

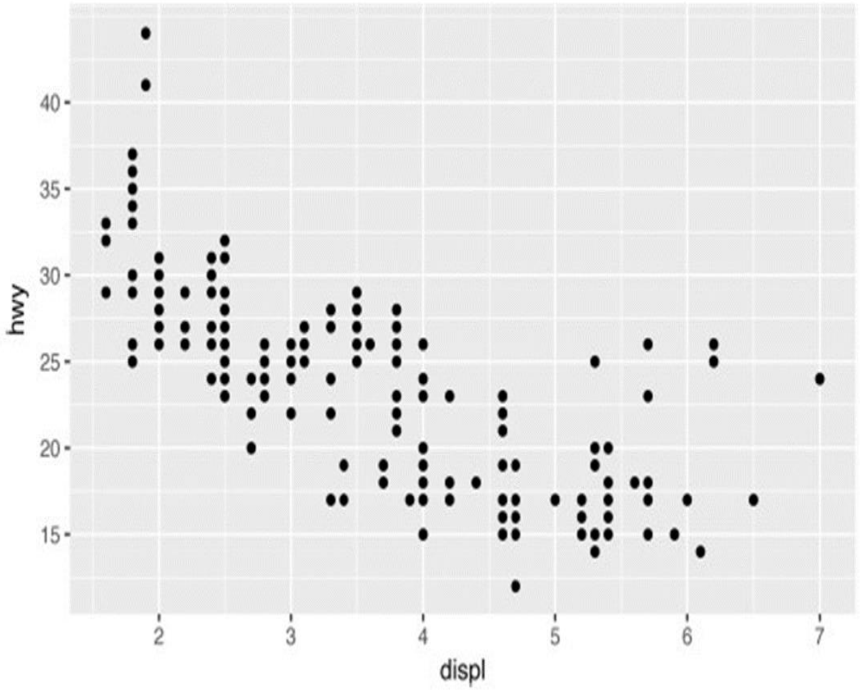
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
     scale_x_continuous() +
     scale_y_continuous() +
     scale_color_discrete()
```

- The naming scheme for scales: **scale_** followed by the name of the aesthetic, **then _**, then the name of the scale.
- The default scales are named according to the type of variable they align with: continuous, discrete, datetime, or date.

Axis Ticks and Legend Keys

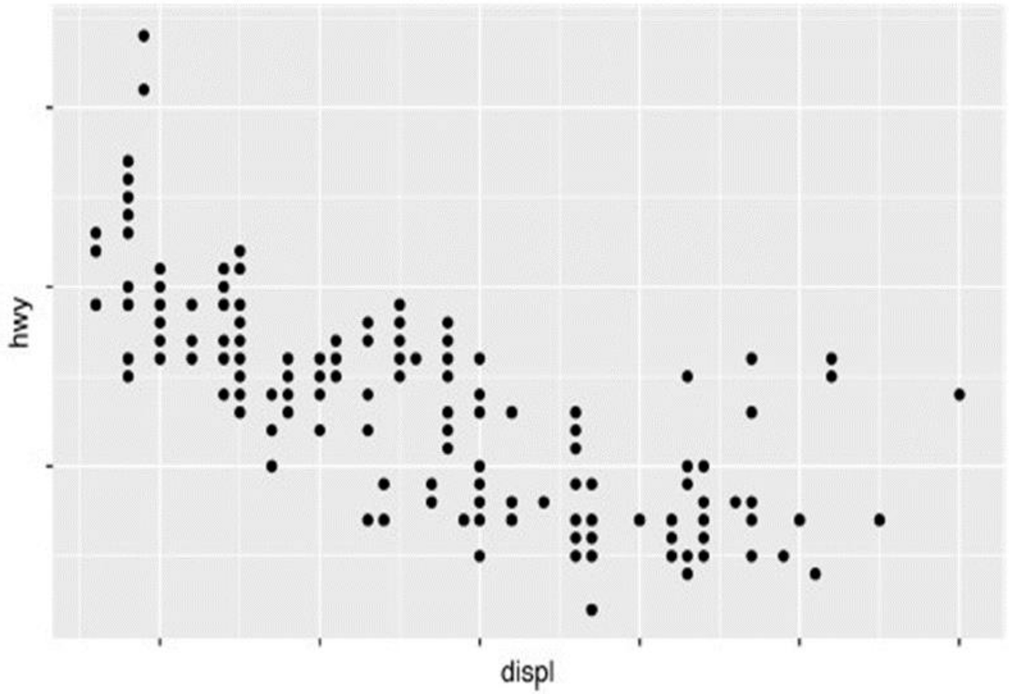
- There are two primary arguments that affect the appearance of the ticks on the axes and the keys on the legend:
  - ✓**Breaks** - controls the position of the ticks, or the values associated with the keys. and
  - ✓**Labels** – controls the text label associated with each tick/key.
- The most common use of breaks is to override the default choice:

```
ggplot(mpg, aes(displ, hwy)) + geom_point() +
scale_y_continuous(breaks = seq(15, 40, by = 5))
```



- You can use labels in the same way, but you can also set it to NULL to suppress the labels altogether.

```
ggplot(mpg, aes(displ, hwy)) +
geom_point() +
scale_x_continuous(labels = NULL) +
scale_y_continuous(labels = NULL)
```



- You can also use **breaks** and **labels** to control the appearance of **legends**.
- Collectively **axes** and **legends** are called ***guides***. Axes are used for the x and y aesthetics; legends are used for everything else.
- Another use of breaks is when you have relatively few data points and want to highlight exactly where the observations occur.

Legend Layout

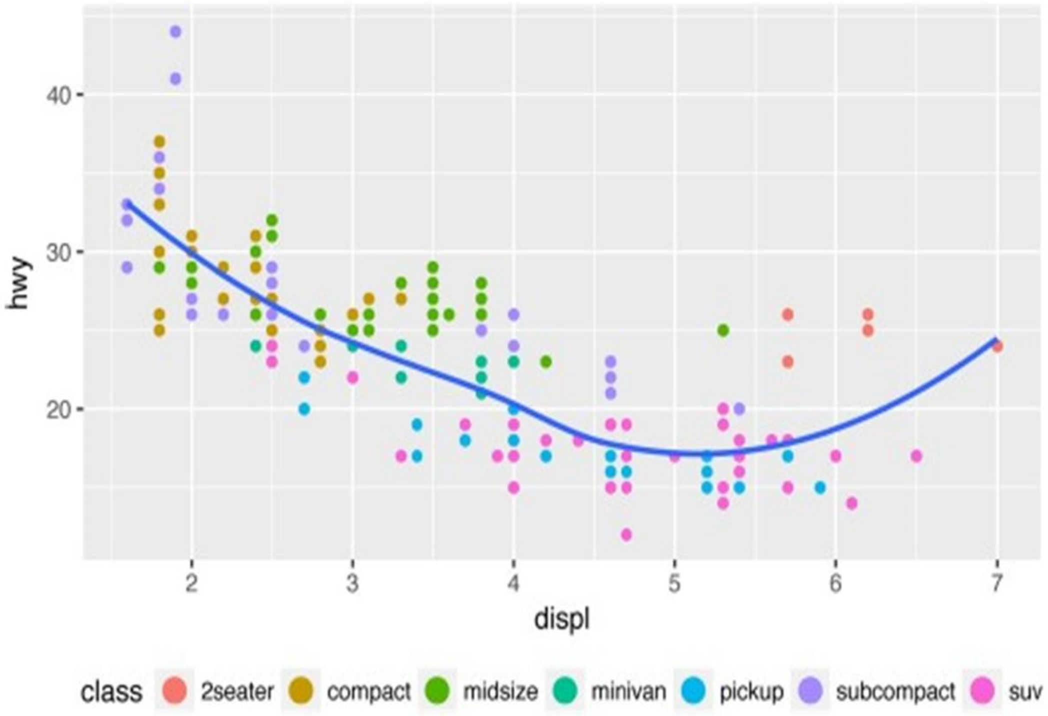
- To control the overall position of the legend, you need to use a **theme()** setting.
- The theme setting legend.position controls where the legend is drawn:

```
base <- ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class))
    base + theme(legend.position = "left")
    base + theme(legend.position = "top")
    base + theme(legend.position = "bottom")
    base + theme(legend.position = "right") # the default
```

- You can also use **legend.position = "none"** to suppress the display of the legend altogether.
- To control the display of individual legends, use **guides()** along with **guide_legend()** or **guide_colorbar().**

- controlling the number of rows the legend uses with **nrow,** and overriding one of the aesthetics to make the points bigger.
- This is particularly useful if you have used a low alpha to display many points on a plot. "se=FALSE", where "s.e." stands for "standard error."

```
ggplot(mpg, aes(displ, hwy)) +
geom_point(aes(color = class)) +
geom_smooth(se = FALSE) +
theme(legend.position = "bottom") +
guides(color = guide_legend(
             nrow = 1,
            override.aes = list(size = 4)))
```



Replacing a Scale

- There are two types of scales that you want to switch out:

✓continuous position scales and

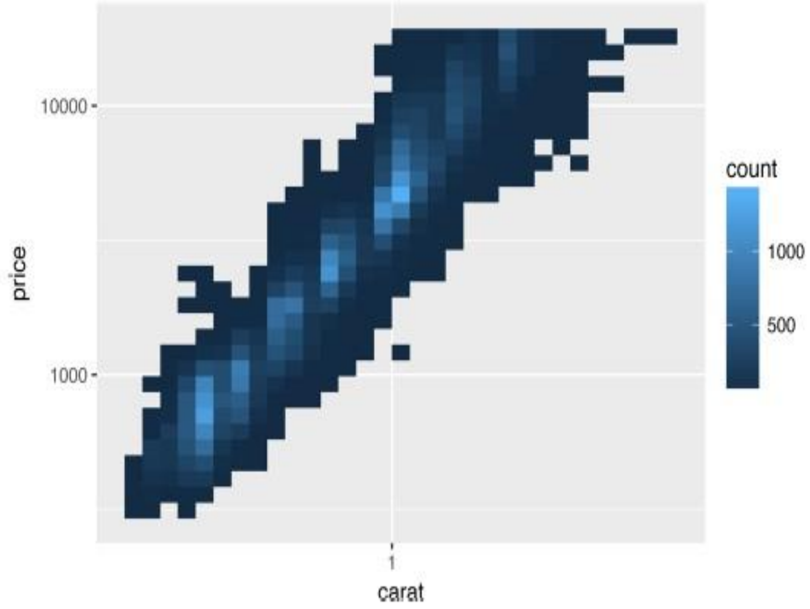
✓color scales.

- The same principles apply to all the other aesthetics, so once you’ve mastered position and color, you’ll be able to quickly pick up other scale replacements.
- It is very useful to plot transformations of your variable

```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d()
ggplot(diamonds, aes(log10(carat), log10(price))) +
  geom_bin2d()
```

- The disadvantage of this transformation is that the axes are now labeled with the transformed values, making it hard to interpret the plot.
- Instead of doing the transformation in the aesthetic mapping, we can do it with the **scale**.
- This is visually identical, except the axes are labeled on the original data scale

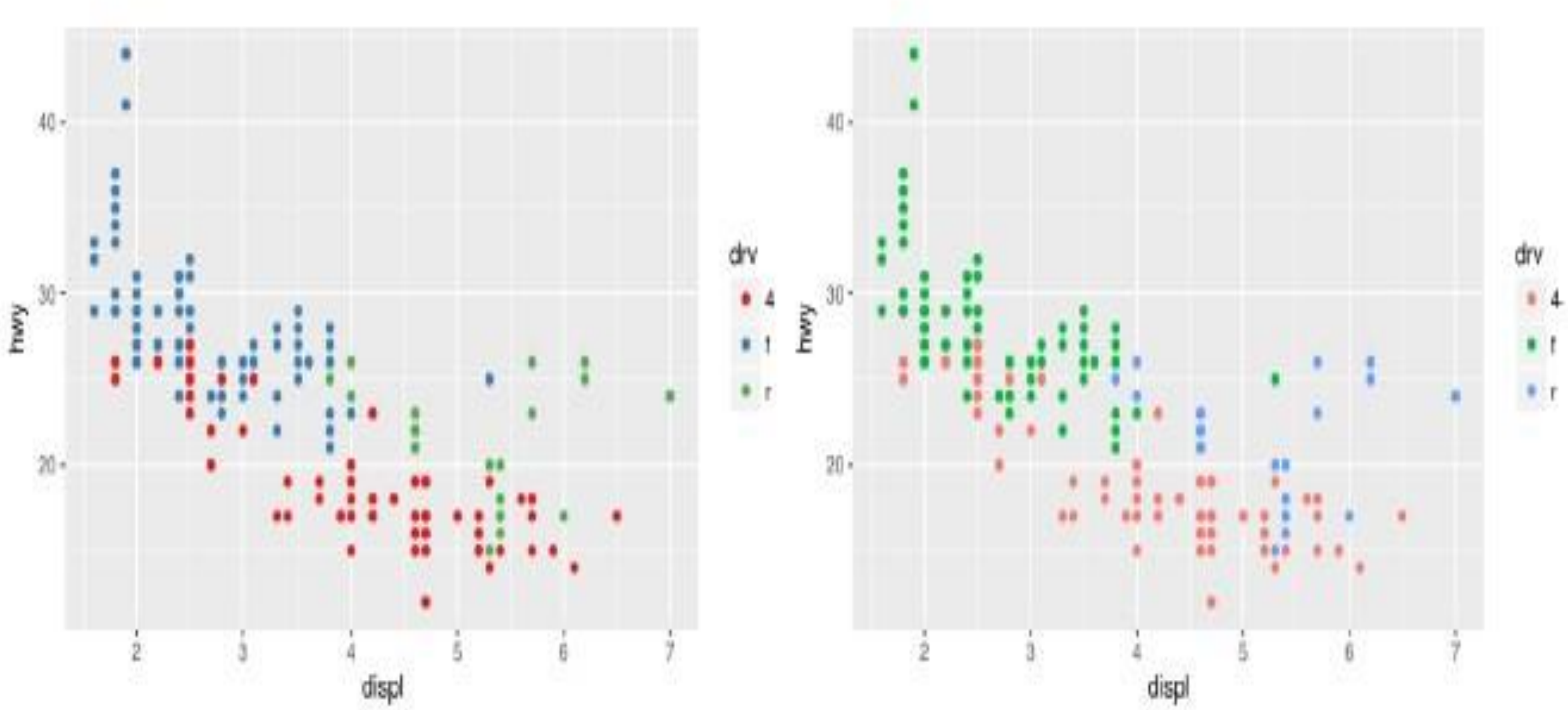
```
Example: ggplot(diamonds, aes(carat, price)) +
                    geom_bin2d() +
                    scale_x_log10() +
                    scale_y_log10()
```



- Another/ alternative useful scale are the ColorBrewer scales that is frequently customized is color. .
- There is enough difference in the shades of red and green that the dots on the right can be distinguished even by people with red-green color blindness

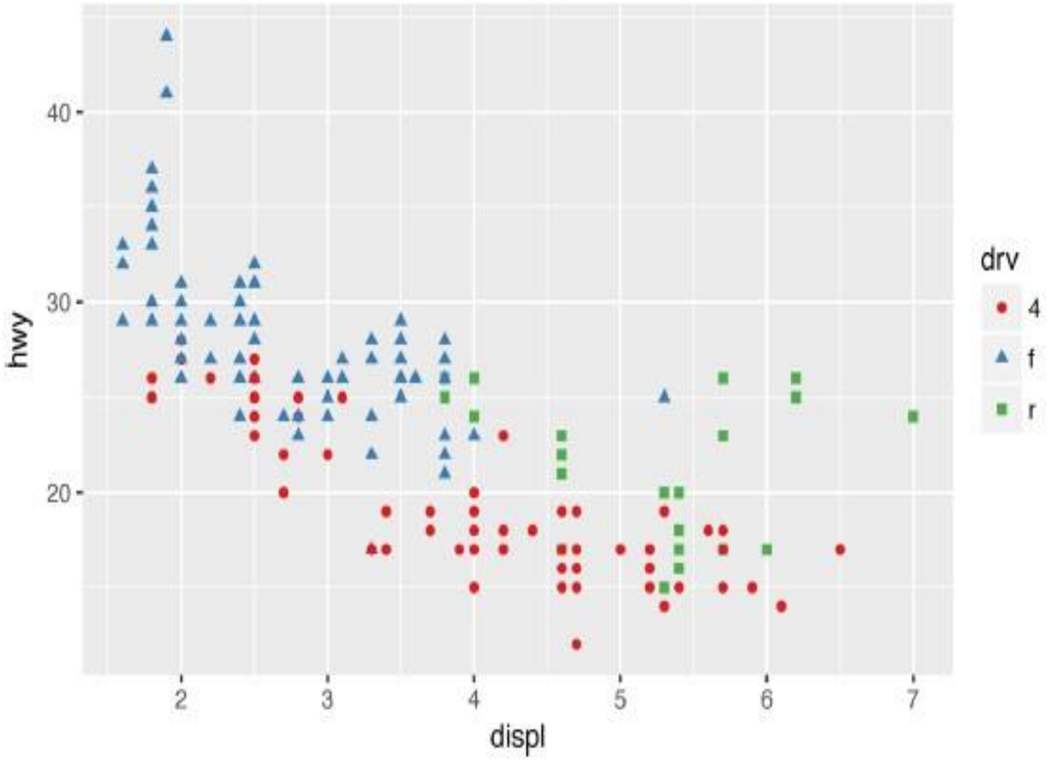
Example: ggplot(mpg, aes(displ, hwy)) +

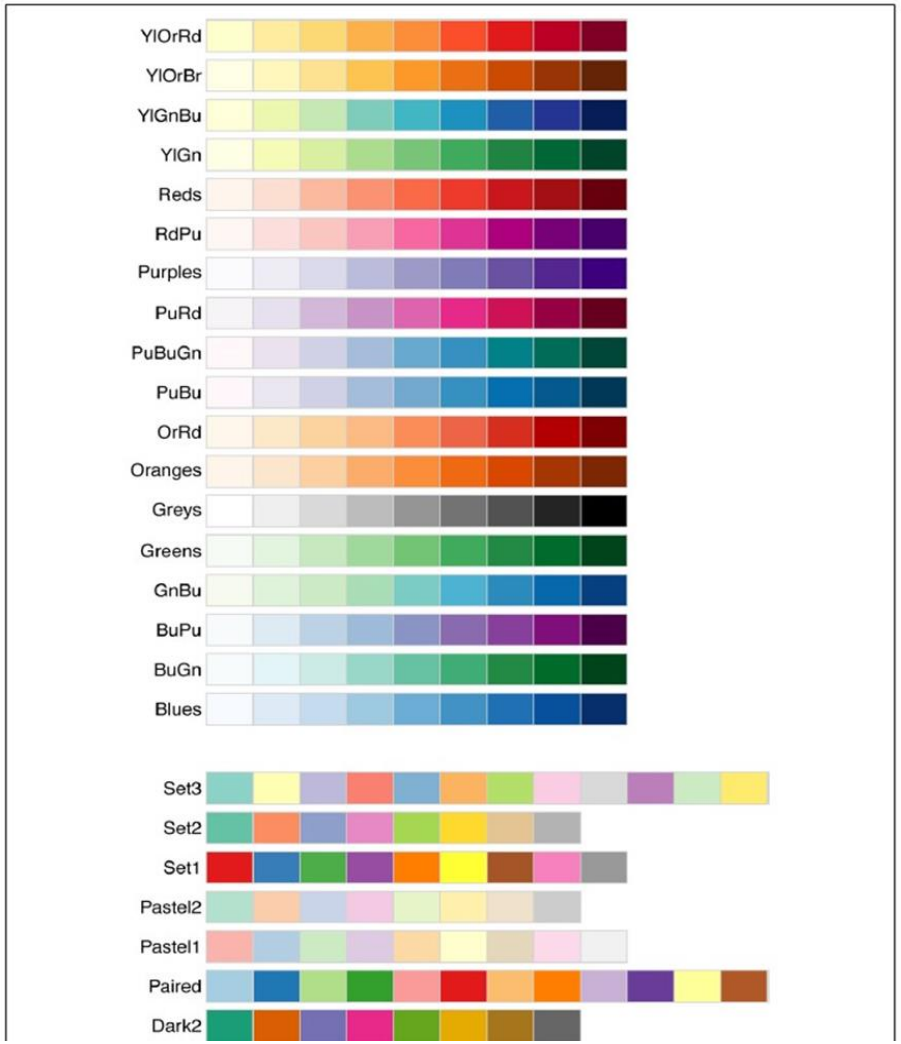
```
                  geom_point(aes(color = drv))
               ggplot(mpg, aes(displ, hwy)) +
                   geom_point(aes(color = drv)) +
                   scale_color_brewer(palette = "Set1")
```



- If there are just a few colors, you can add a redundant shape mapping.

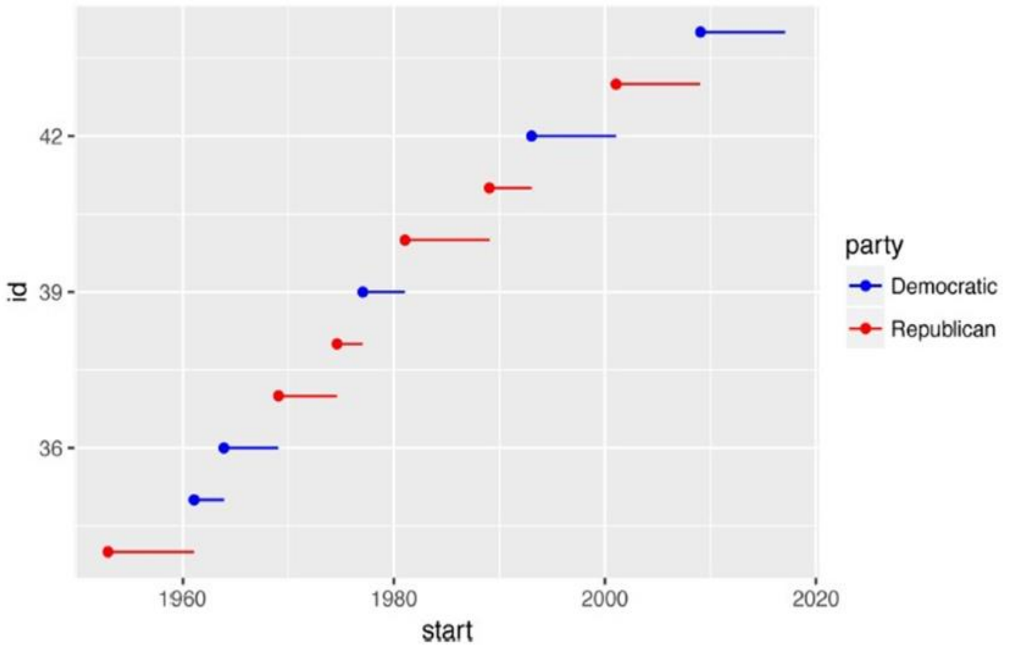
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv, shape = drv)) +
  scale_color_brewer(palette = "Set1")
```





- When you have a predefined mapping between values and colors, use **scale_color_manual().**

```
presidential %>%
mutate(id = 33 + row_number()) %>%
ggplot(aes(start, id, color = party)) +
geom_point() +
geom_segment(aes(xend = end, yend = id)) +
scale_colour_manual(
values = c(Republican = "red",
                Democratic = "blue"))
```



- For continuous color, you can use the built-in **scale_color_gradient()** or **scale_fill_gradient().**

- Another option is **scale_color_viridis()** provided by the **viridis** package.
- It's a continuous analog of the categorical ColorBrewer scales.

```
df <- tibble( x = rnorm(10000),
                  y = rnorm(10000)
                )
  ggplot(df, aes(x, y)) + geom_hex() +
    coord_fixed()
#> Loading required package: methods
  ggplot(df, aes(x, y)) + geom_hex() +
     viridis::scale_fill_viridis() +
     coord_fixed()
```

- all color scales come in two varieties: **scale_color_x()** and **scale_fill_x()** for the color and fill aesthetics.

Zooming

Zooming

- There are three ways to control the plot limits:
    - ➢Adjusting what data is plotted
    - ➢Setting the limits in each scale
    - ➢Setting xlim and ylim in coord_cartesian()
- To zoom in on a region of the plot, the best to use is coord_cartesian().

```
ggplot(mpg, mapping = aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  coord_cartesian(xlim = c(5, 7), ylim = c(10, 30))
```

```
mpg %>%
  filter(displ >= 5, displ <= 7, hwy >= 10, hwy <= 30) %>%
  ggplot(aes(displ, hwy)) +
    geom_point(aes(color = class)) +
    geom_smooth()
```

- You can also set the limits on individual scales.
- Reducing the limits is basically equivalent to subsetting the data.
- If we extract two classes of cars and plot them separately, it’s difficult to compare the plots because all three scales (the x-axis, the y-axis, and the color aesthetic) have different ranges.

```
suv <- mpg %>% filter(class == "suv")
compact <- mpg %>% filter(class == "compact")
ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point()
ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point()
```

- One way to overcome this problem is to share scales across multiple plots, training the scales with the limits of the full data:

```
x_scale <- scale_x_continuous(limits = range(mpg$displ))
y_scale <- scale_y_continuous(limits = range(mpg$hwy))
col_scale <- scale_color_discrete(limits = unique(mpg$drv))
```

```
ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```

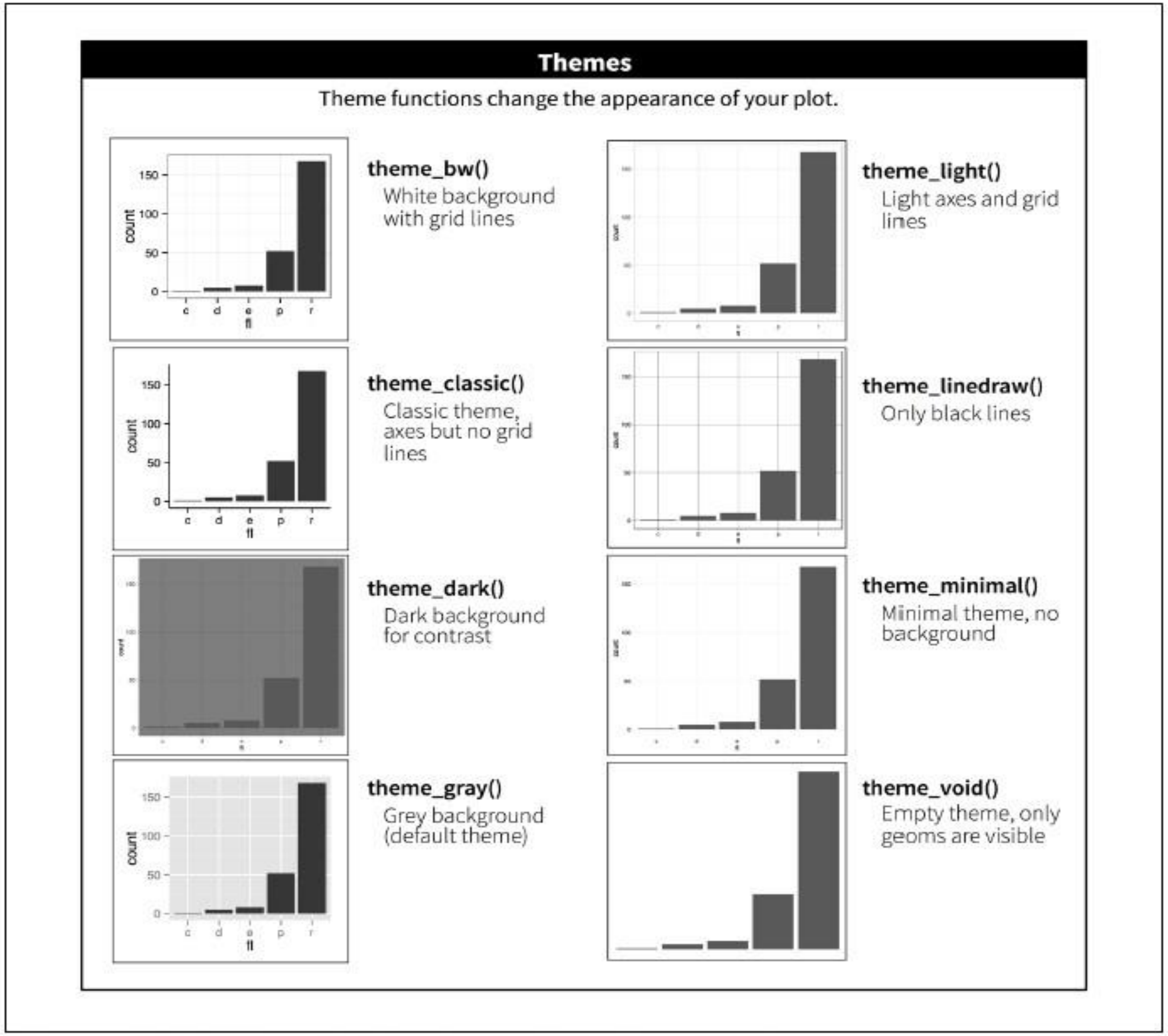
Themes

- you can customize the non-data elements of your plot with a theme:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme_bw()
```

- **ggplot2** includes eight themes by default, many more are included in add-on packages like **ggthemes**, by Jeffrey Arnold.

- The default theme has a gray background, because it puts the data forward while still making the grid lines visible.



Themes

Theme functions change the appearance of your plot.

theme_bw()
White background with grid lines

theme_light()
Light axes and grid lines

theme_classic()
Classic theme, axes but no grid lines

theme_linedraw()
Only black lines

theme_dark()
Dark background for contrast

theme_minimal()
Minimal theme, no background

theme_gray()
Grey background (default theme)

theme_void()
Empty theme, only geoms are visible

Figure 22-3. The eight themes built into ggplot2

- The **white grid lines** are, but they have little visual impact and we can easily tune them out.

- The **gray background** gives the plot a similar typographic color to the text, ensuring that the graphics fit in with the flow of a document without jumping out with a bright white background.

- Finally, the gray background creates a continuous field of color, which ensures that the plot is perceived as a single visual entity.

Saving Your Plots

- There are two main ways to get your plots out of R and into your final write-up:
    1. ggsave() and
    2. **knitr**. ggsave()

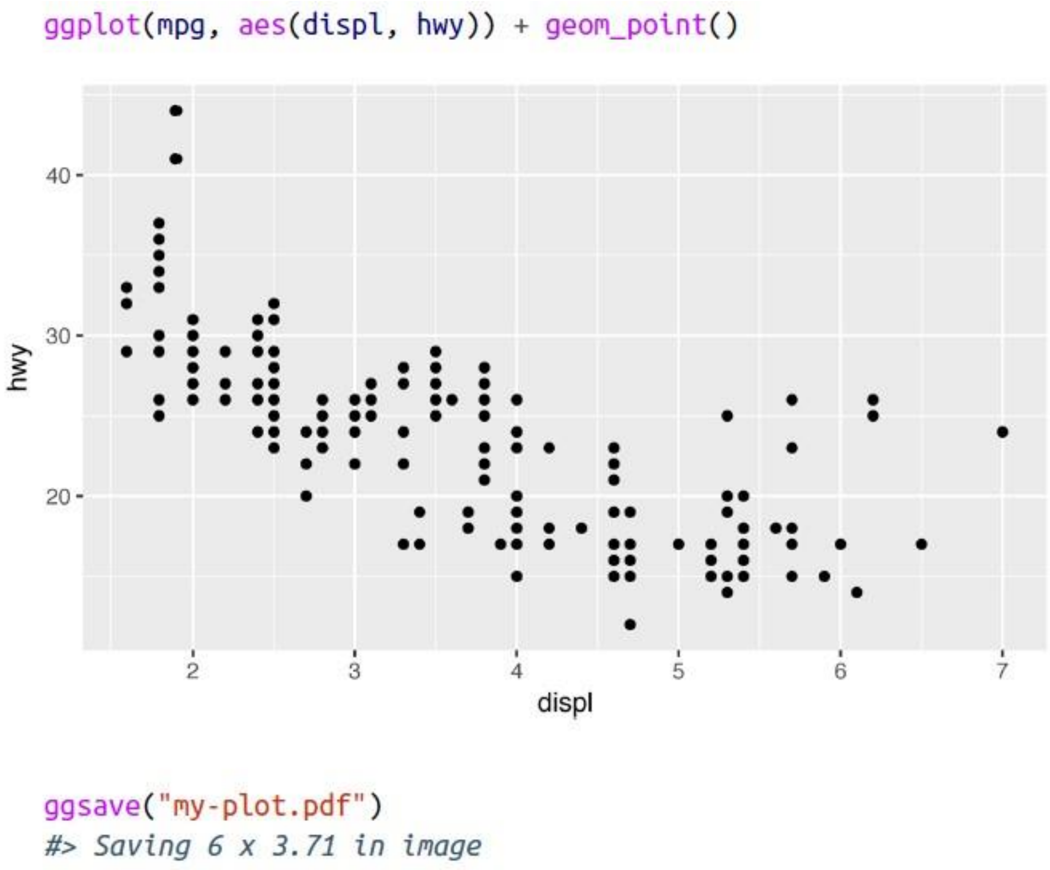
will save the most recent plot to disk

- If you don't specify the **width**

and **height** they will be taken

From the dimensions of the

current plotting device.



```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```

```
ggsave("my-plot.pdf")
#> Saving 6 x 3.71 in image
```