DATA SCIENCE AND MACHINE LEARNING

Introduction to GGPLOT

Dimitris Fouskakis
Professor in Applied Statistics,
Department of Mathematics,
School of Applied Mathematical & Physical Sciences,
National Technical University of Athens
Email: fouskakis@math.ntua.gr

Visualization

- Creating ***visualizations*** (graphical representations) of data is a key step in being able to communicate information and findings to others.
- Intro to ggplot2.
- Preeminent plotting library in R.
- This gets you started with ggplot2; however, there is a lot more to learn.

GGplot2

- Install and load ggplot2 library.
- ggplot2 comes with a number of built-in datasets. Here we will use the mpg dataset, which is a data frame that contains information about fuel economy for different cars.

Mpg Dataset

```
library(ggplot2)

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>
```

Mpg Dataset

- **A data frame with 234 rows and 11 variables.**
  - manufacturer
  - model (model name)
  - displ (engine displacement, in litres)
  - year (year of manufacture)
  - cyl (number of cylinders)
  - Trans (type of transmission)
  - drv (f = front-wheel drive, r = rear wheel drive, 4 = 4wd)
  - cty (city miles per gallon)
  - hwy (highway miles per gallon)
  - fl (fuel type)
  - class ("type" of car)

Grammar of Graphics

- the data being plotted
- the geometric objects (circles, lines, etc.) that appear on the plot
- a set of mappings from variables in the data to the aesthetics (appearance) of the geometric objects
- a statistical transformation used to calculate the data values used in the plot
- a position adjustment for locating each geometric object on the plot
- a scale (e.g., range of values) for each aesthetic mapping used
- a coordinate system used to organize the geometric objects
- the facets or groups of data shown in different plots

The Basics

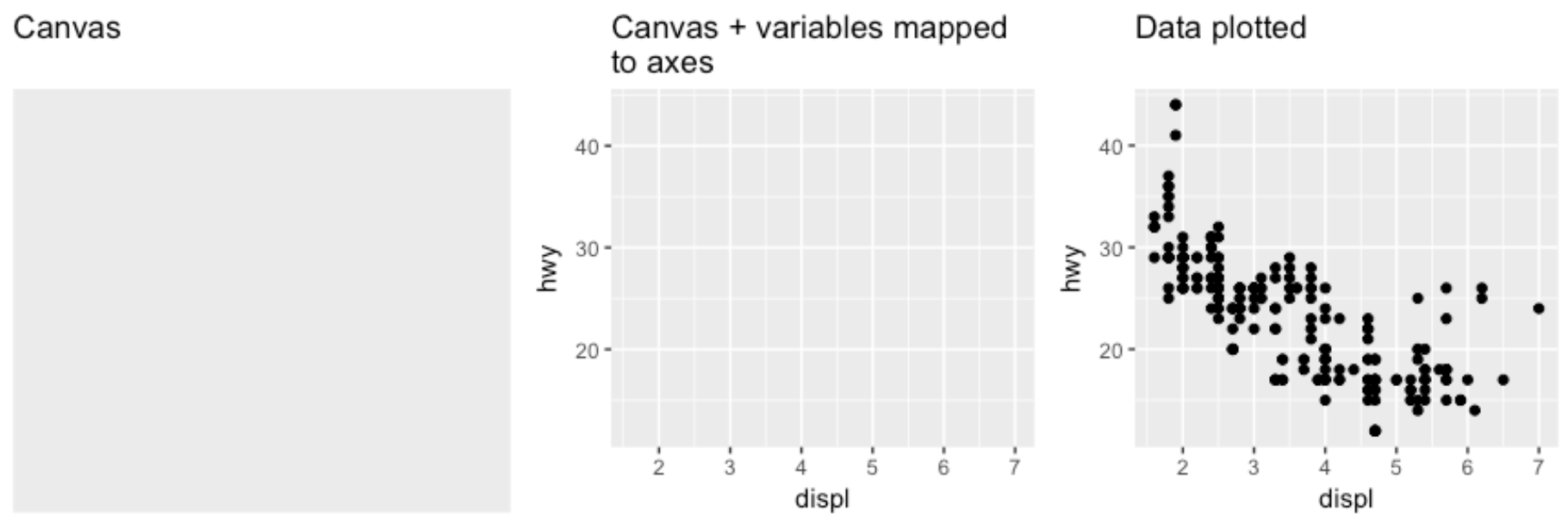
- Call the ggplot() function which creates a blank canvas.
- Specify aesthetic mappings, i.e. how you want to map variables to visual aspects. In the next slide we are simply mapping the *displ* and *hwy* variables to the x- and y-axes.
- You then add new layers that are geometric objects which will show up on the plot. In the next slide we add geom_point to add a layer with points (dot) elements as the geometric shapes to represent the data.

The Basics

```
# create canvas
ggplot(mpg)
# variables of interest mapped
ggplot(mpg, aes(x = displ, y = hwy))
# data plotted
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point()
```

Note that when you added the geom layer you used the addition (+) operator. As you add new layers you will always use + to add onto your visualization.

The Basics

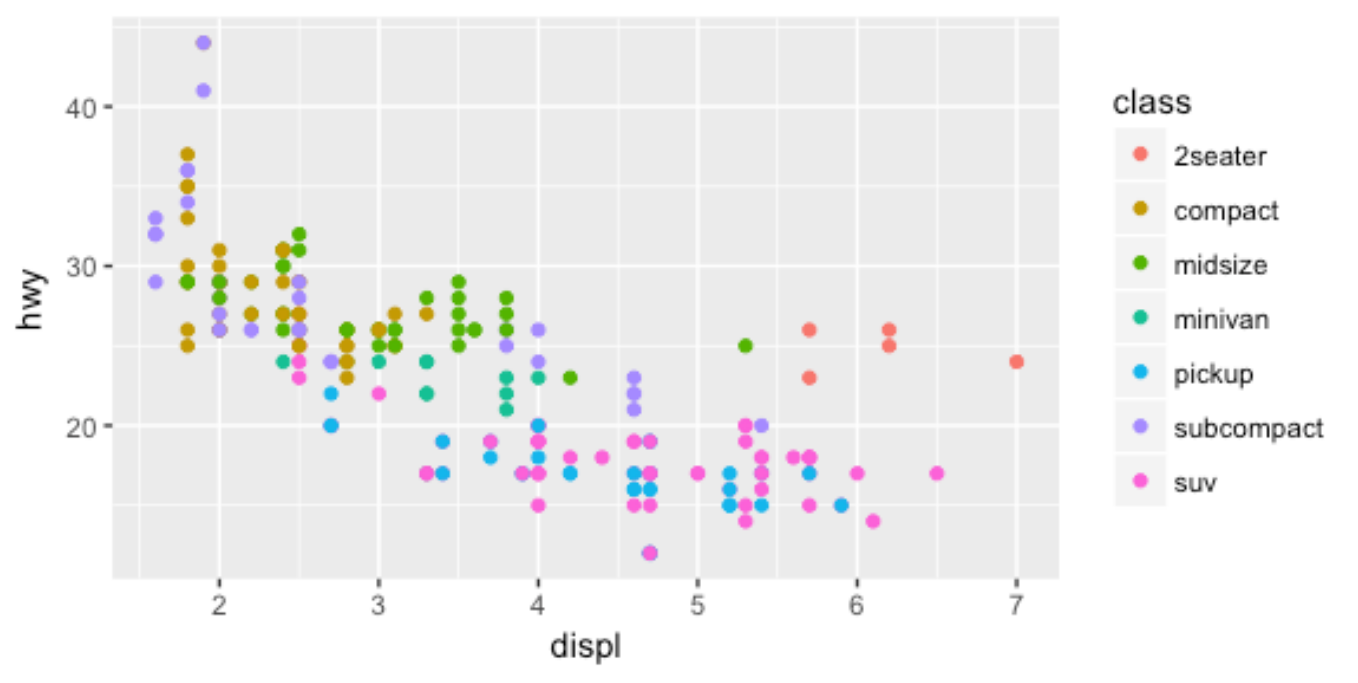

Canvas
Canvas + variables mapped to axes
Data plotted
hwy
40
30
20
2
3
4
5
6
7
displ

Aesthetic Mappings

- The aesthetic mappings take properties of the data and use them to influence visual characteristics, such as position, *color, size, shape, or transparency*. Each visual characteristic can thus encode an aspect of the data and be used to convey information.
- All aesthetics for a plot are specified in the aes() function call. For example, we can add a mapping from the class of the cars to a color characteristic:

Aesthetic Mappings

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

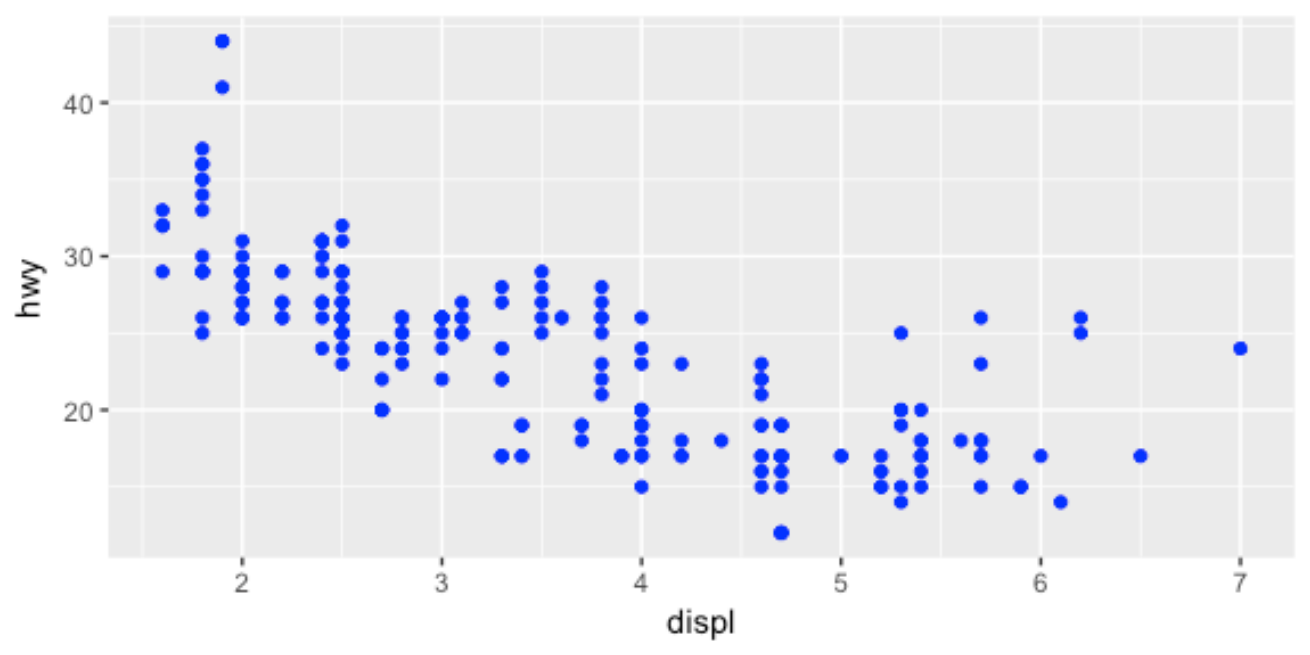


Aesthetic Mappings

- Note that using the aes() function will cause the visual channel to be based on the data specified in the argument. For example, using aes(color = "blue") won't cause the geometry's color to be "blue", but will instead cause the visual channel to be mapped from the vector c("blue") — as if we only had a single type of engine that happened to be called "blue". If you wish to apply an aesthetic property to an entire geometry, you can set that property as an argument to the geom method, outside of the aes() call:

Aesthetic Mappings

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



Geometric Shapes

- geom_point for drawing individual points (e.g., a scatter plot)
- geom_line for drawing lines (e.g., for a line charts)
- geom_smooth for drawing smoothed lines (e.g., for simple trends or approximations)
- geom_bar for drawing bars (e.g., for bar charts)
- geom_histogram for drawing binned values (e.g. a histogram)
- geom_polygon for drawing arbitrary shapes
- geom_map for drawing polygons in the shape of a map! (You can access the data to use for these maps by using the map_data() function).

Geometric Shapes

- Each of these geometries will leverage the aesthetic mappings supplied although the specific visual properties that the data will map to will vary. For example, you can map data to the shape of a geom_point (e.g., if they should be circles or squares), or you can map data to the linetype of a geom_line (e.g., if it is solid or dotted), but not vice versa.
- Almost all geoms require an x and y mapping at the bare minimum.

Geometric Shapes

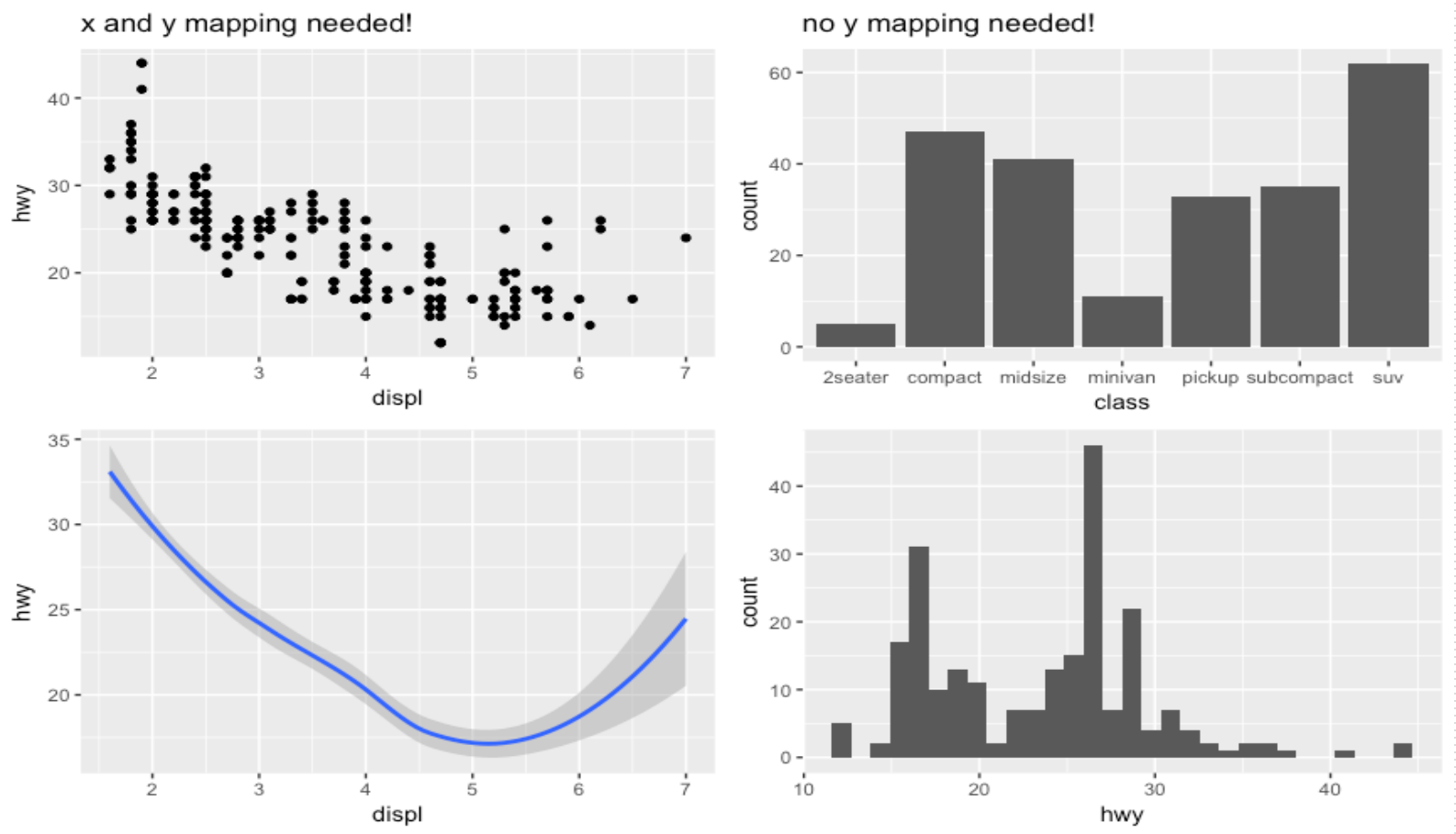
```
# Left column: x and y mapping needed!
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point()

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

# Right column: no y mapping needed!
ggplot(data = mpg, aes(x = class)) +
  geom_bar()

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

Geometric Shapes

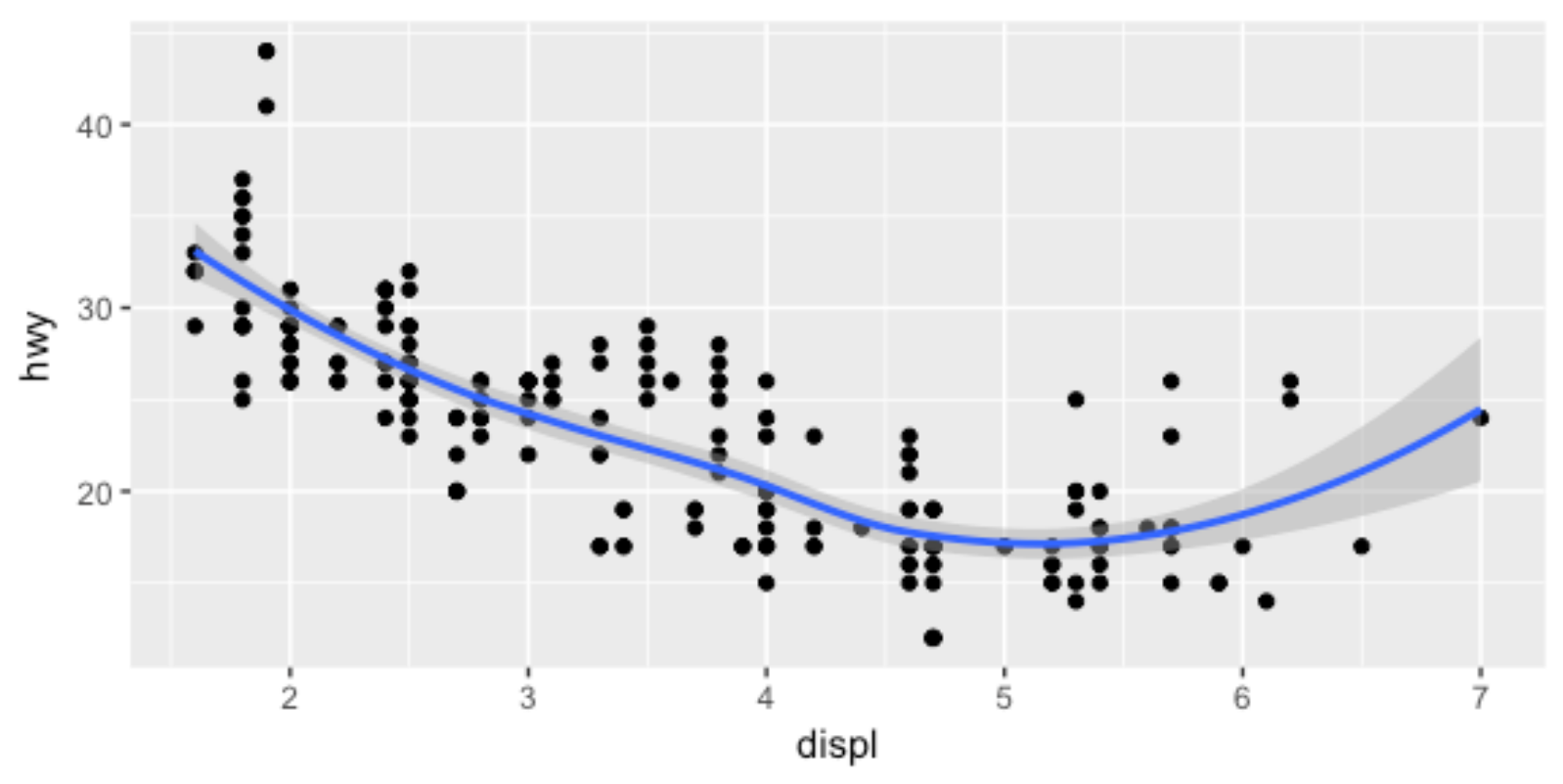


Geometric Shapes

- What makes this really powerful is that you can add multiple geometries to a plot, thus allowing you to create complex graphics showing multiple aspects of your data.

```
# plot with both points and smoothed line
ggplot(mpg, aes(x = displ, y = hwy)) +
geom_point() +
geom_smooth()
```

Geometric Shapes

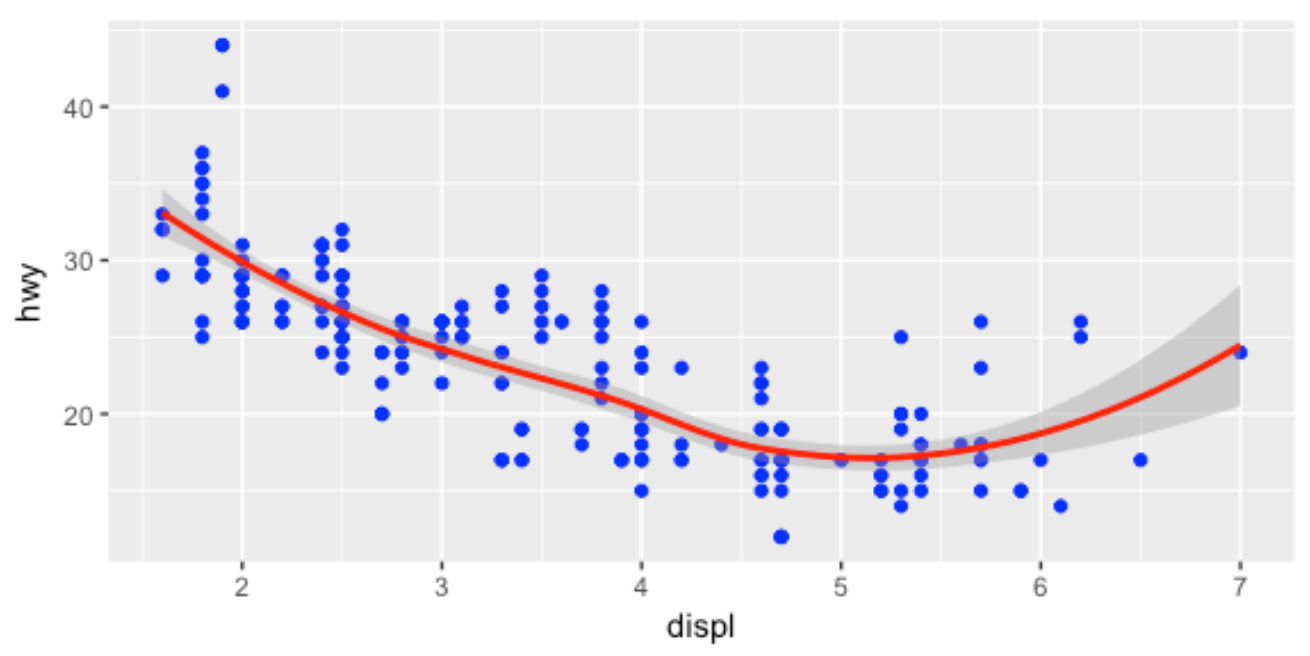


Geometric Shapes

- Of course the aesthetics for each geom can be different, so you could show multiple lines on the same plot (or with different colors, styles, etc). It's also possible to give each geom a different data argument, so that you can show multiple data sets in the same plot.
- For example, we can plot both points and a smoothed line for the same x and y variable but specify unique colors within each geom:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
geom_point(color = "blue") +
geom_smooth(color = "red")
```

Geometric Shapes



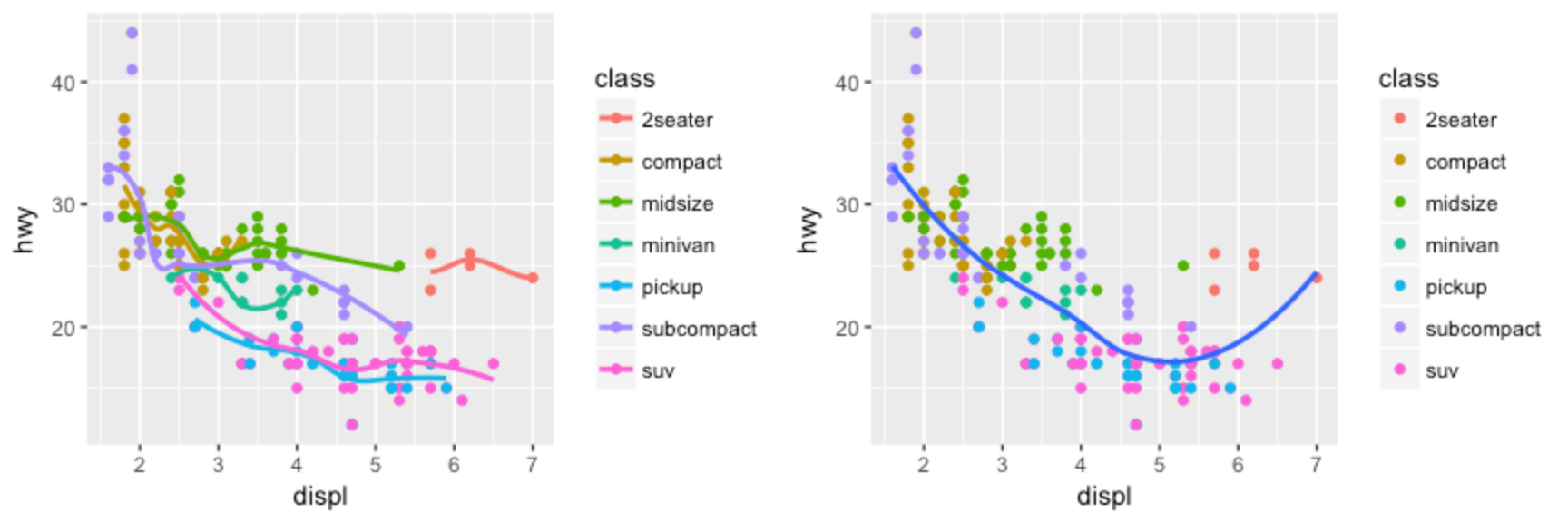
Geometric Shapes

- So as you can see if we specify an aesthetic within ggplot it will be passed on to each geom that follows. Or we can specify certain aes within each geom, which allows us to only show certain characteristics for that specific layer (i.e. geom_point).

```
# color aesthetic passed to each geom layer
  ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
  geom_point() +
  geom_smooth(se = FALSE)

# color aesthetic specified for only the geom_point layer
  ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE)
```

Geometric Shapes

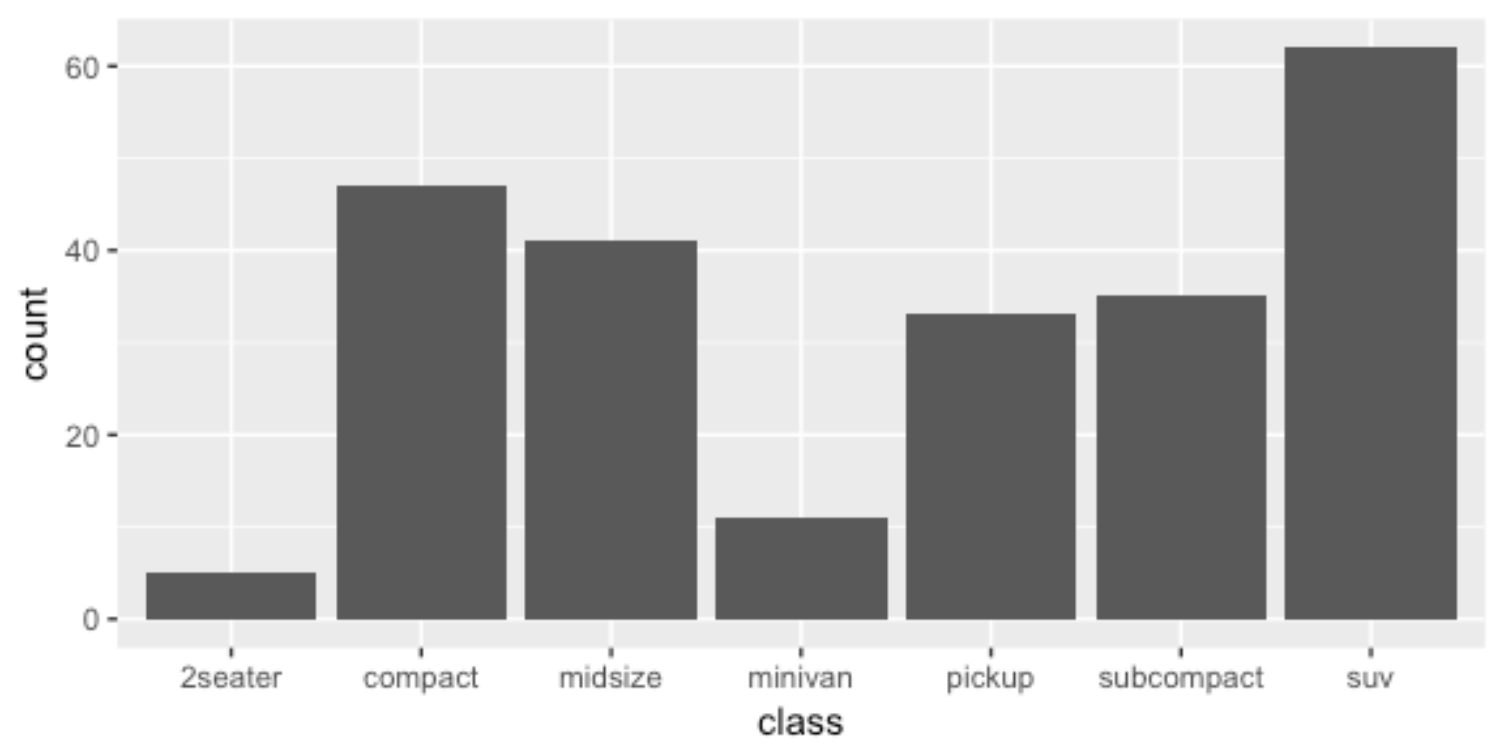


Statistical Transformations

- If you look at the bar chart in the next slide, you'll notice that the y axis was defined for us as the count of elements that have the particular type. This count isn't part of the data set (it's not a column in mpg), but is instead a **statistical transformation** that the geom_bar automatically applies to the data. In particular, it applies the stat_count transformation.

```
ggplot(mpg, aes(x = class)) +
geom_bar()
```

Statistical Transformations



Statistical Transformations

- ggplot2 supports many different statistical transformations. For example, the "identity" transformation will leave the data "as is". You can specify which statistical transformation a geom uses by passing it as the stat argument. For example, consider our data already had the count as a variable:

Statistical Transformations

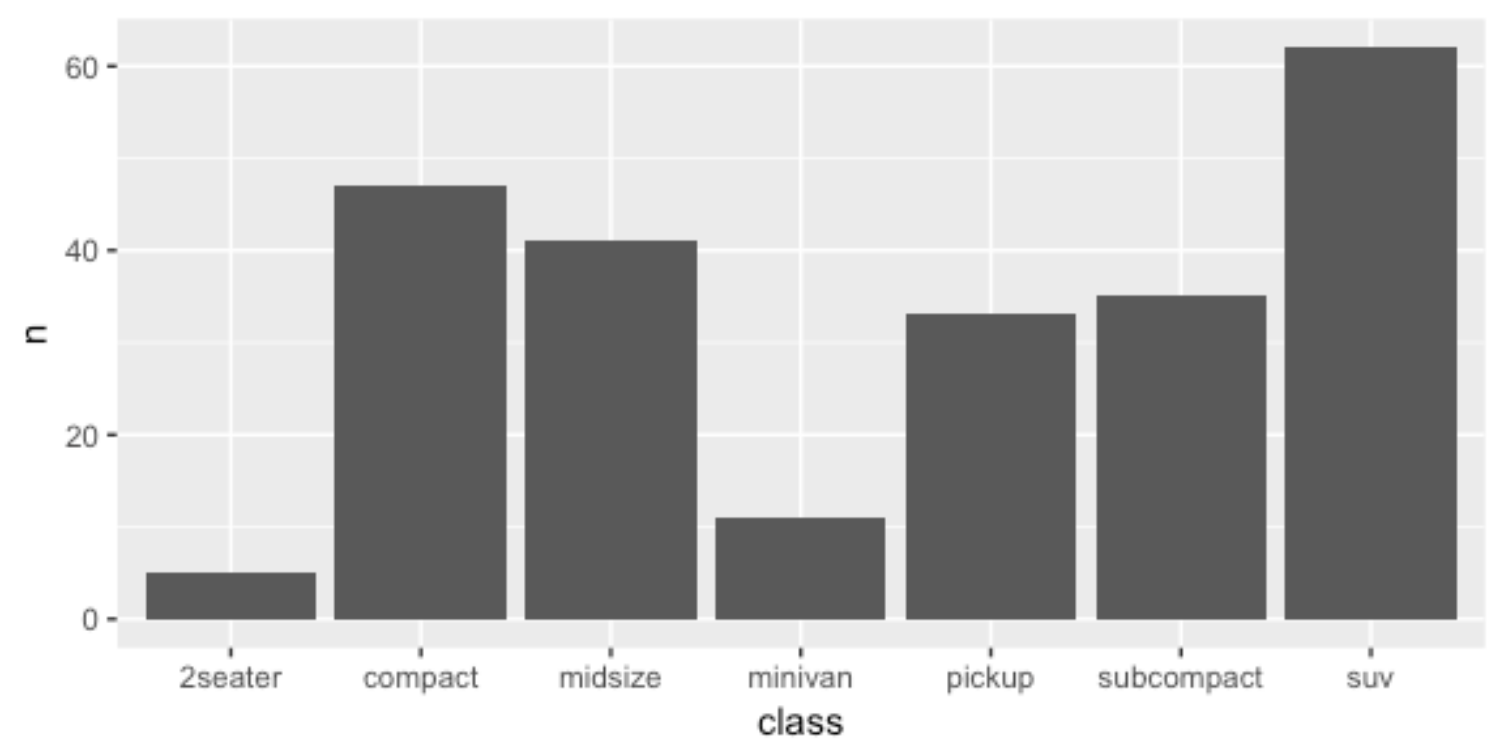
```
class_count <- dplyr::count(mpg, class)
class_count
## # A tibble: 7 × 2
##        class     n
##        <chr> <int>
## 1    2seater     5
## 2    compact    47
## 3    midsize    41
## 4    minivan    11
## 5     pickup    33
## 6 subcompact    35
## 7        suv    62
```

Statistical Transformations

- We can use stat = "identity" within geom_bar to plot our bar height values to this variable. Also, note that we now include n for our y variable:

```
ggplot(class_count, aes(x = class, y = n)) +
  geom_bar(stat = "identity")
```

Statistical Transformations

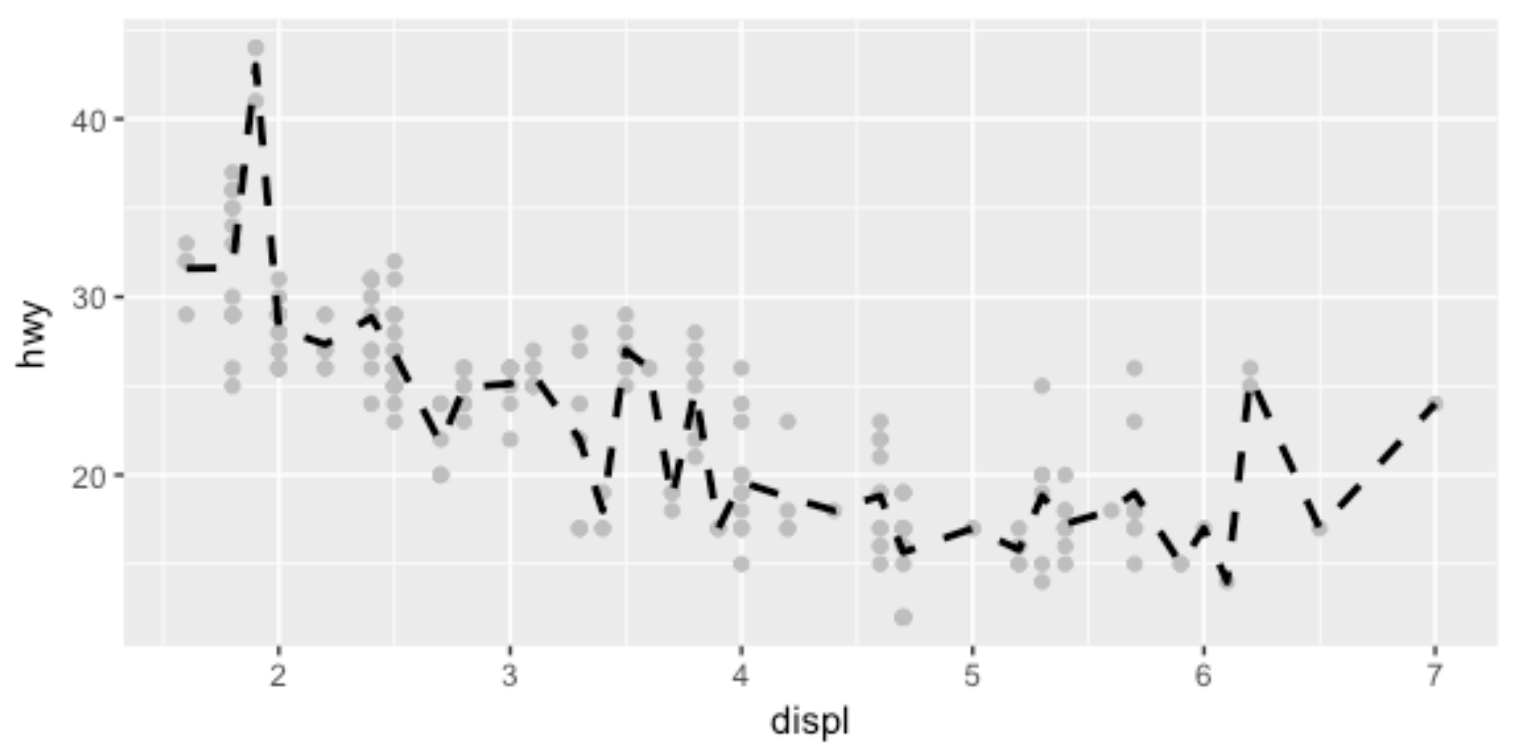


Statistical Transformations

- We can also call stat_ functions directly to add additional layers. For example, here we create a scatter plot of highway miles for each displacement value and then use stat_summary to plot the mean highway miles at each displacement value.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(color = "grey") +
  stat_summary(fun.y = "mean", geom = "line", size = 1,
linetype = "dashed")
```

Statistical Transformations

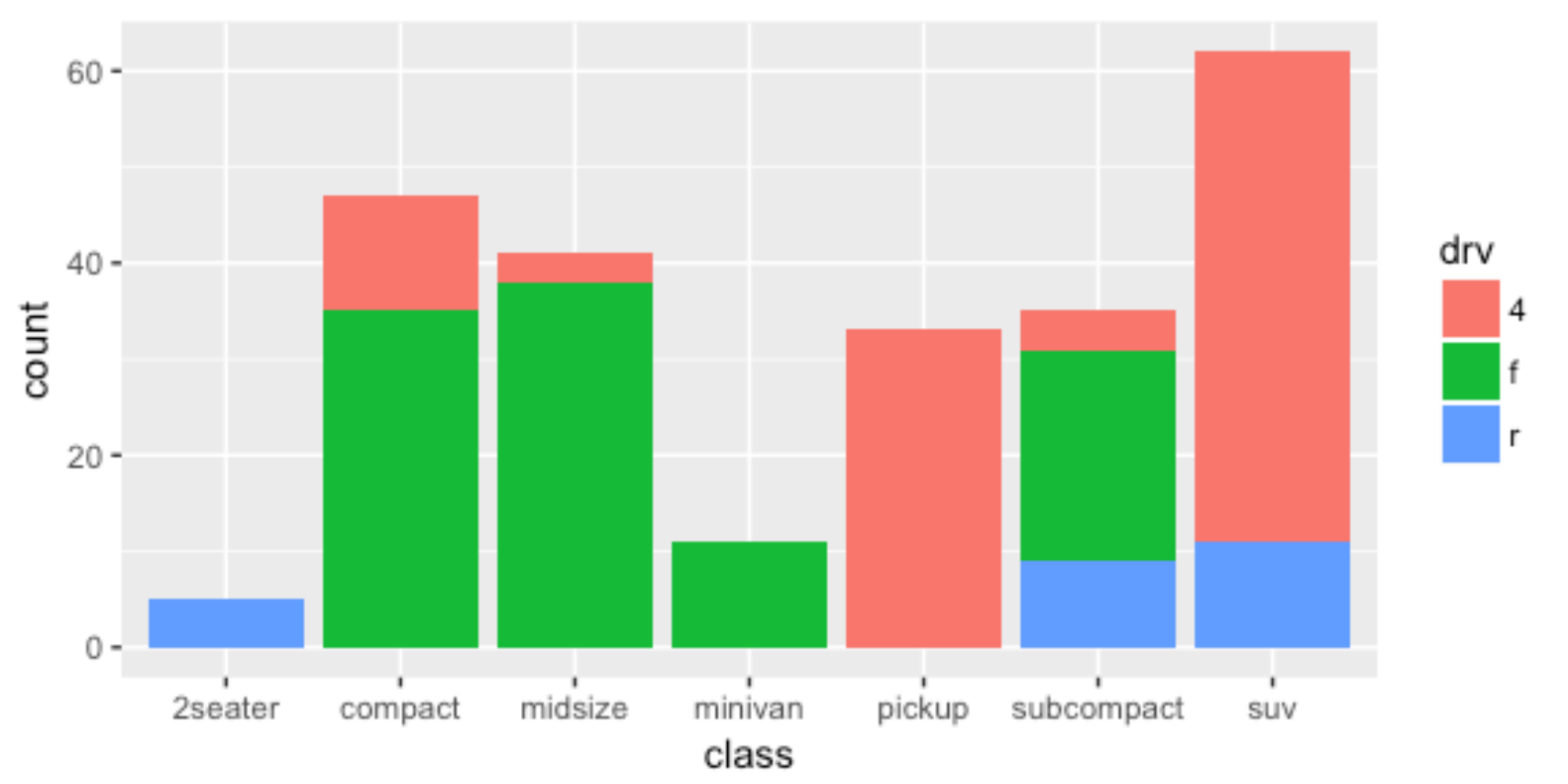


Position Adjustment

- In addition to a default statistical transformation, each geom also has a default **position adjustment** which specifies a set of "rules" as to how different components should be positioned relative to each other. This position is noticeable in a geom_bar if you map a different variable to the color visual characteristic (**stacked barplot**):

```
# bar chart of class, colored by drive (front, rear, 4-wheel)
  ggplot(mpg, aes(x = class, fill = drv)) +
  geom_bar()
```

Position Adjustment



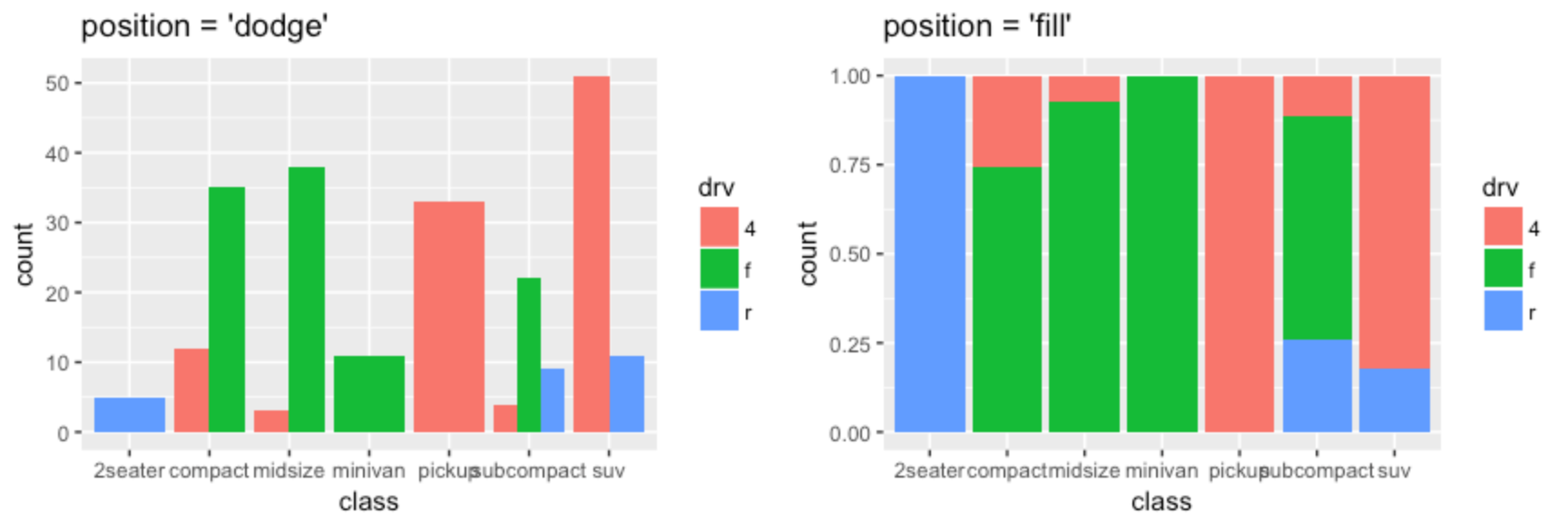
Position Adjustment

- The geom_bar by default uses a position adjustment of "stack", which makes each rectangle's height proportional to its value and stacks them on top of each other. We can use the position argument to specify what position adjustment rules to follow:

```
# position = "dodge": values next to each other (grouped barplot)
  ggplot(mpg, aes(x = class, fill = drv)) +
  geom_bar(position = "dodge")

# position = "fill": percentage chart (stacked barplot with % in y-axis)
  ggplot(mpg, aes(x = class, fill = drv)) +
  geom_bar(position = "fill")
```

Position Adjustment



Managing Scales

- Whenever you specify an aesthetic mapping, ggplot uses a particular **scale** to determine the range of values that the data should map to. Thus when you specify

```
# color the data by engine type
 ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
 geom_point()
```

- ggplot automatically adds a scale for each mapping to the plot:

```
# same as above, with explicit scales
 ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
 geom_point() +
 scale_x_continuous() +
 scale_y_continuous() +
 scale_colour_discrete()
```

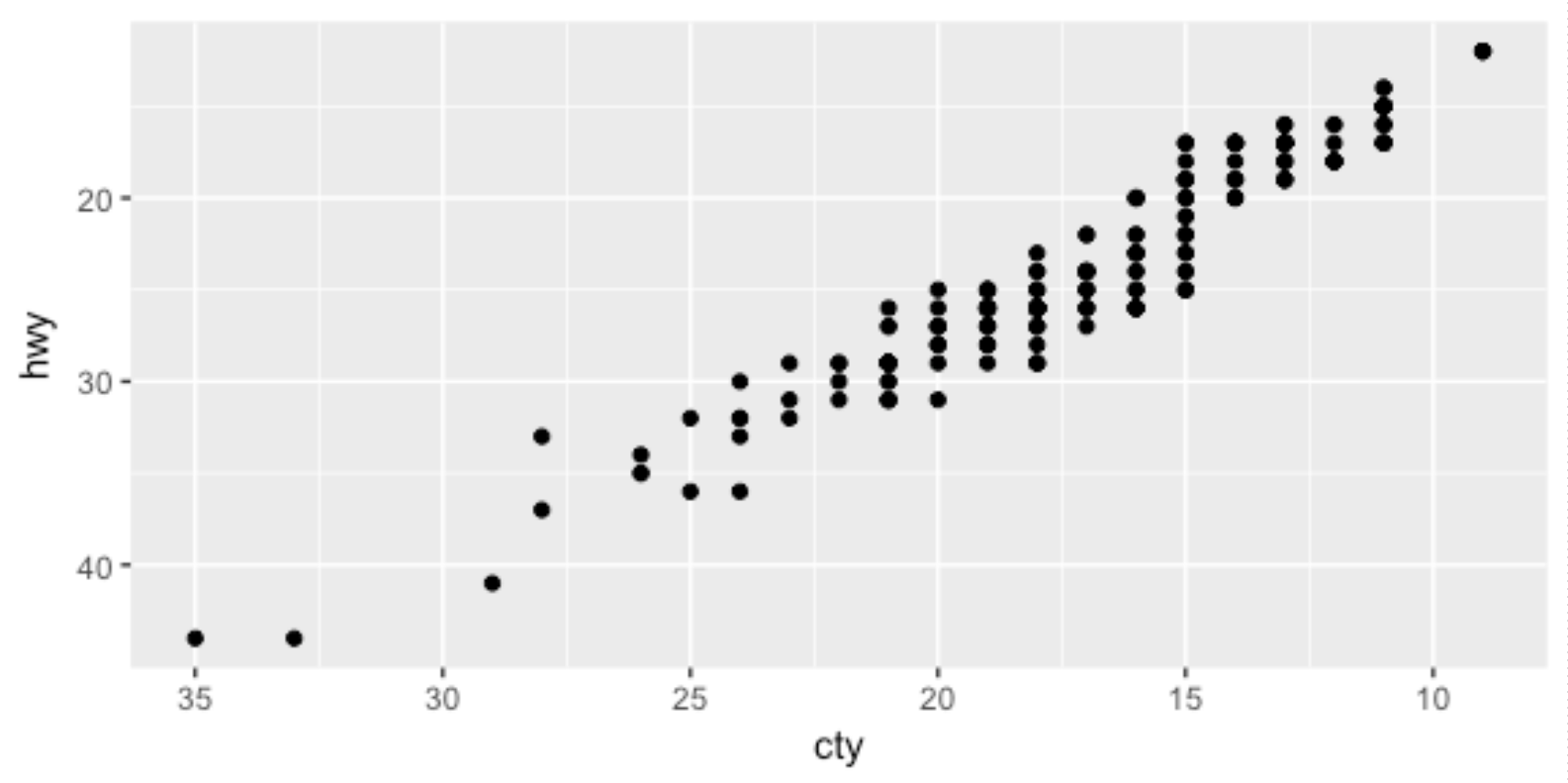
Managing Scales

- Each scale can be represented by a function with the following name: scale_, followed by the name of the aesthetic property, followed by an _ and the name of the scale. A continuous scale will handle things like numeric data (where there is a continuous set of numbers), whereas a discrete scale will handle things like colors (since there is a small list of distinct colors).

- While the default scales will work fine, it is possible to explicitly add different scales to replace the defaults. For example, you can use a scale to change the direction of an axis:

```
# milage relationship, ordered in reverse
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point() +
  scale_x_reverse() +
  scale_y_reverse()
```

Managing Scales

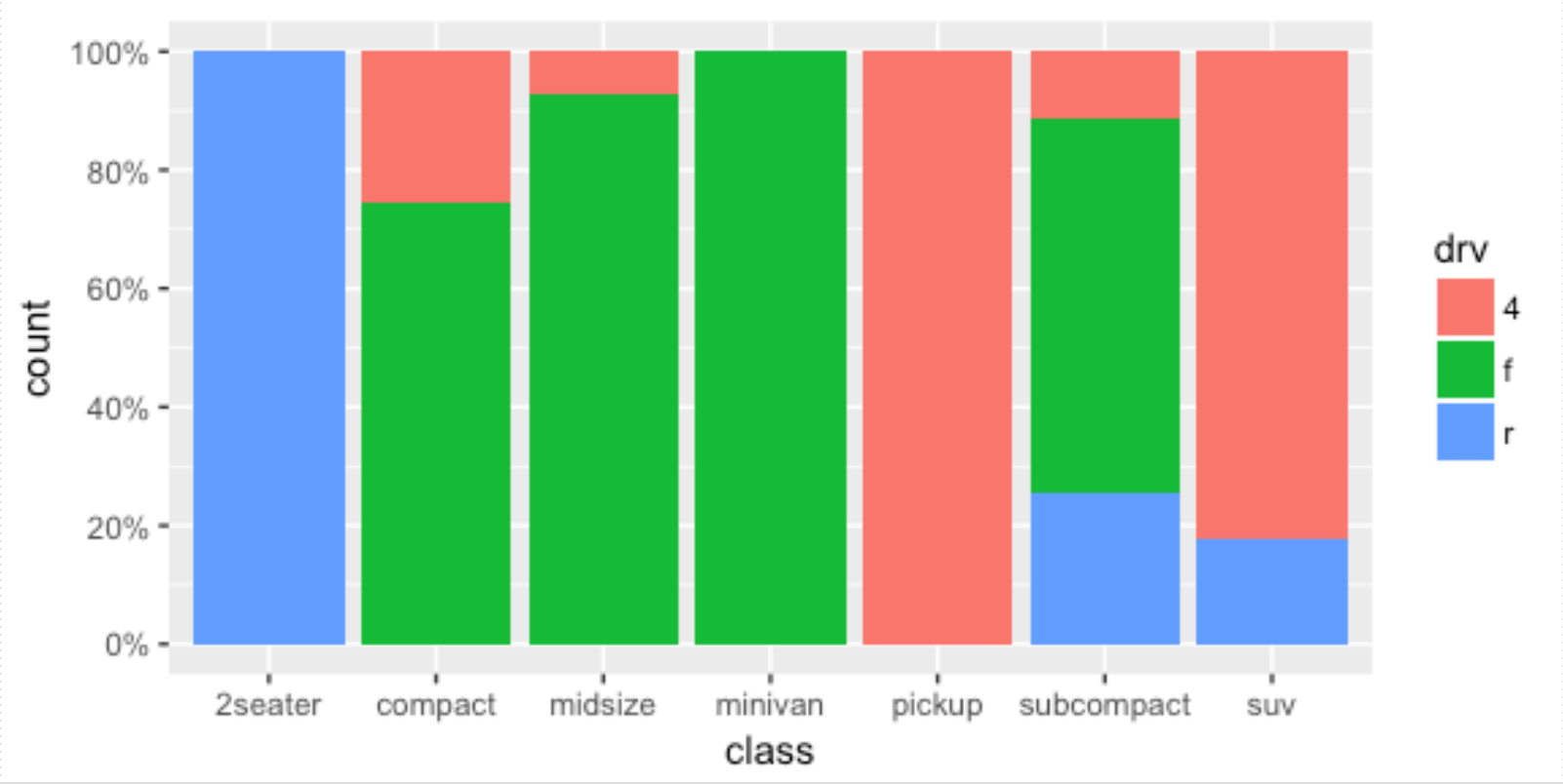


Managing Scales

- Similarly, you can use scale_x_log10() and scale_x_sqrt() to transform your scale. You can also use scales to format your axes:

```
ggplot(mpg, aes(x = class, fill = drv)) +
geom_bar(position = "fill") +
scale_y_continuous(breaks = seq(0, 1, by = .2), labels = scales::percent)
```

Managing Scales



Managing Scales

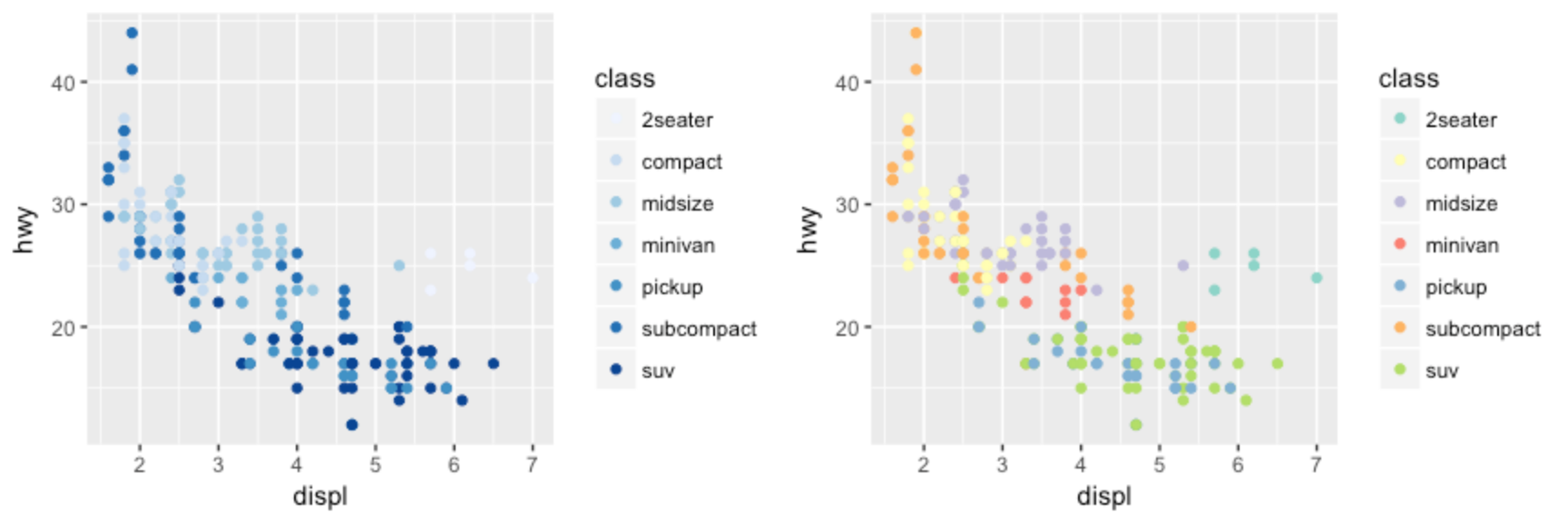
- A common parameter to change is which set of colors to use in a plot. While you can use the default coloring, a more common option is to leverage the pre-defined palettes from **colorbrewer.org**. These color sets have been carefully designed to look good and to be viewable to people with certain forms of color blindness. We can leverage color brewer palletes by specifying the scale_color_brewer() function, passing the pallete as an argument.

Managing Scales

```
# default color brewer
ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
  geom_point() +
  scale_color_brewer()

# specifying color palette
ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
  geom_point() +
  scale_color_brewer(palette = "Set3")
```

Managing Scales



Managing Scales

- Note that you can get the palette name from the colorbrewer website by looking at the scheme query parameter in the URL. Or see the diagram at https://bl.ocks.org/mbostock/5577023 and hover the mouse over each palette for the name.
- You can also specify continuous color values by using a gradient scale, or manually specify the colors you want to use as a named vector.

Coordinate Systems

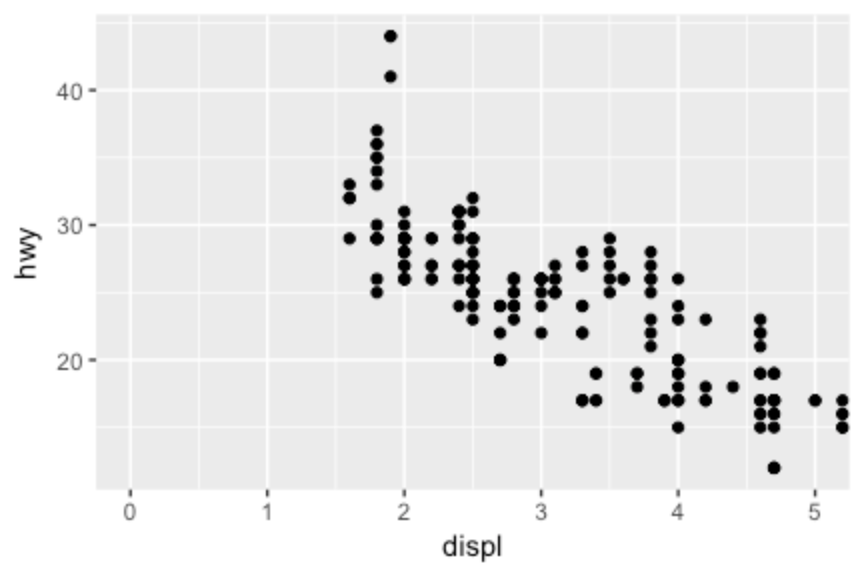
- The next term from the Grammar of Graphics that can be specified is the **coordinate system**. As with scales, coordinate systems are specified with functions that all start with coord_ and are added as a layer. There are a number of different possible coordinate systems to use, including:
  - coord_cartesian the default **cartesian coordinate system**, where you specify x and y values (e.g. allows you to zoom in or out).
  - coord_flip a cartesian system with the x and y flipped
  - coord_fixed a cartesian system with a "fixed" aspect ratio (e.g., 1.78 for a "widescreen" plot)
  - coord_polar a plot using **polar coordinates**
  - coord_quickmap a coordinate system that approximates a good aspect ratio for maps. See documentation for more details.

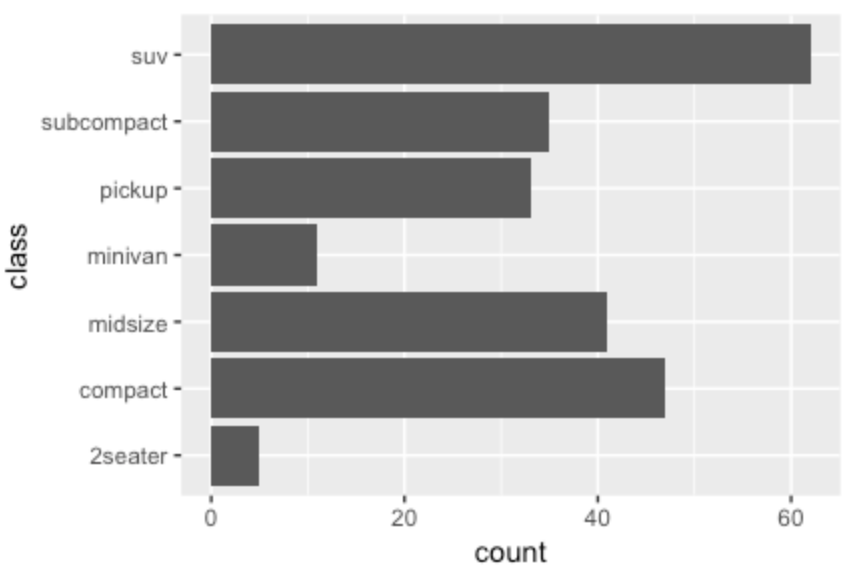
Coordinate Systems

```
# zoom in with coord_cartesian
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  coord_cartesian(xlim = c(0, 5))

# flip x and y axis with coord_flip
ggplot(mpg, aes(x = class)) +
  geom_bar() +
  coord_flip()
```

Coordinate Systems



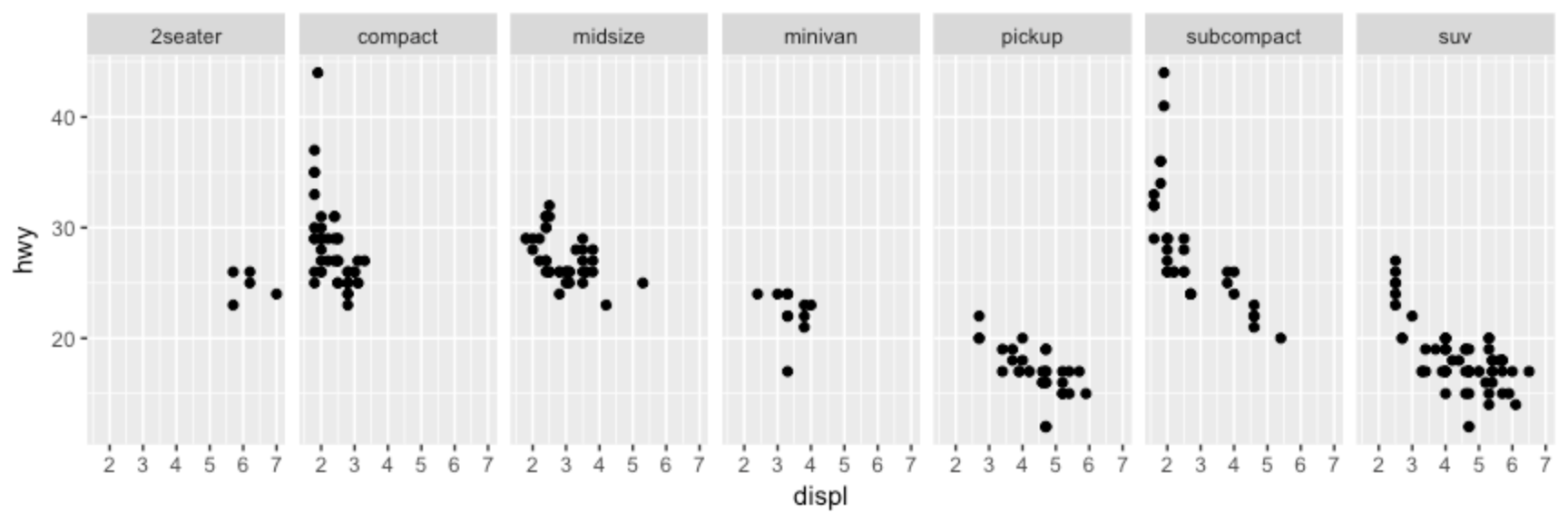


Facets

- **Facets** are ways of grouping a data plot into multiple different pieces (subplots). This allows you to view a separate plot for each value in a categorical variable. You can construct a plot with multiple facets by using the facet_wrap() function. This will produce a "row" of subplots, one for each categorical variable (the number of rows can be specified with an additional argument):

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

Facets

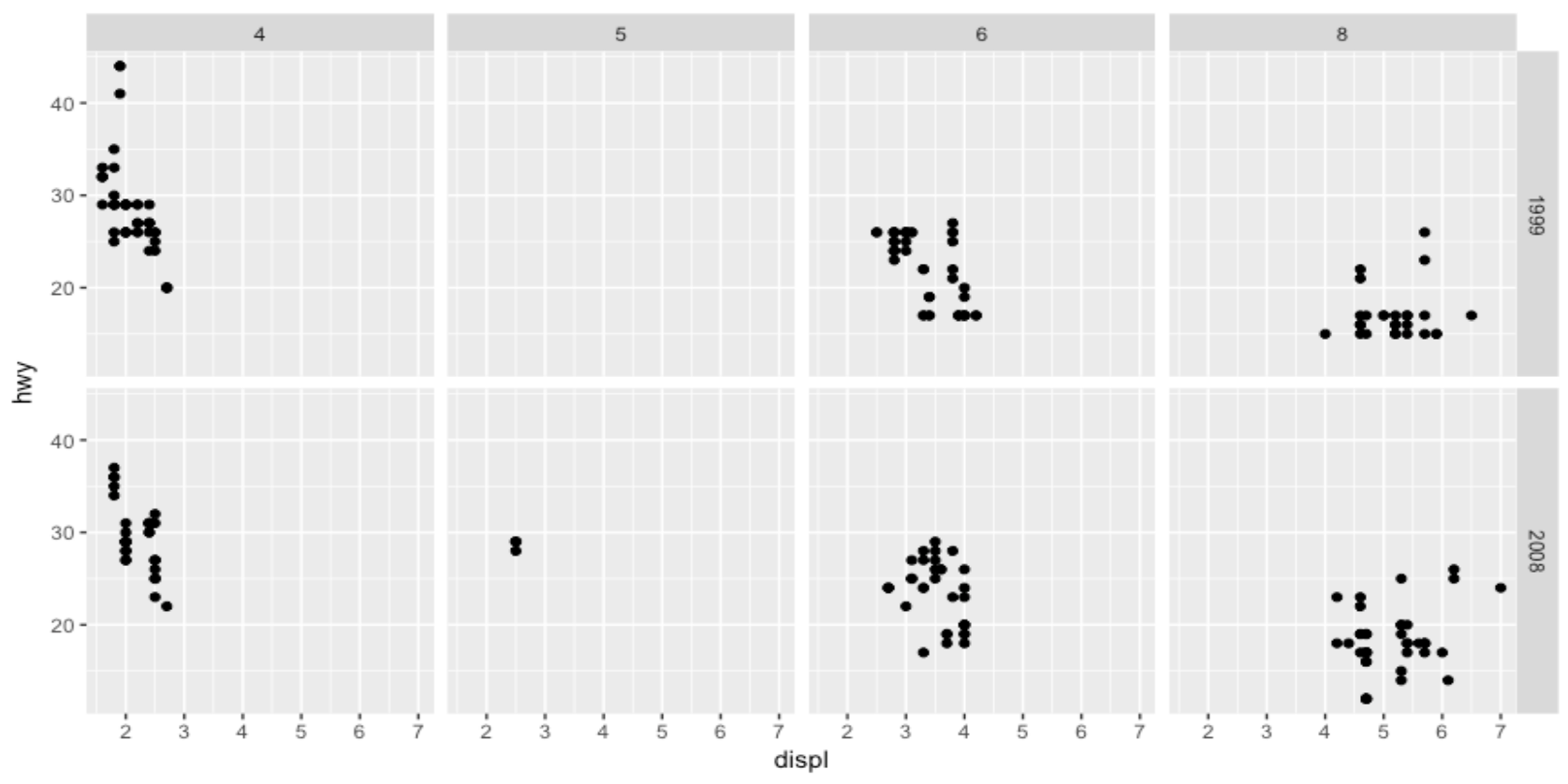

2seater
compact
midsize
minivan
pickup
subcompact
suv
hwy
40
30
20
2 3 4 5 6 7
displ

Facets

- You can also facet_grid to facet your data by more than one categorical variable. Note that we use a tilde (~) in our facet functions. With facet_grid the variable to the left of the tilde will be represented in the rows and the variable to the right will be represented across the columns.

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

Facets



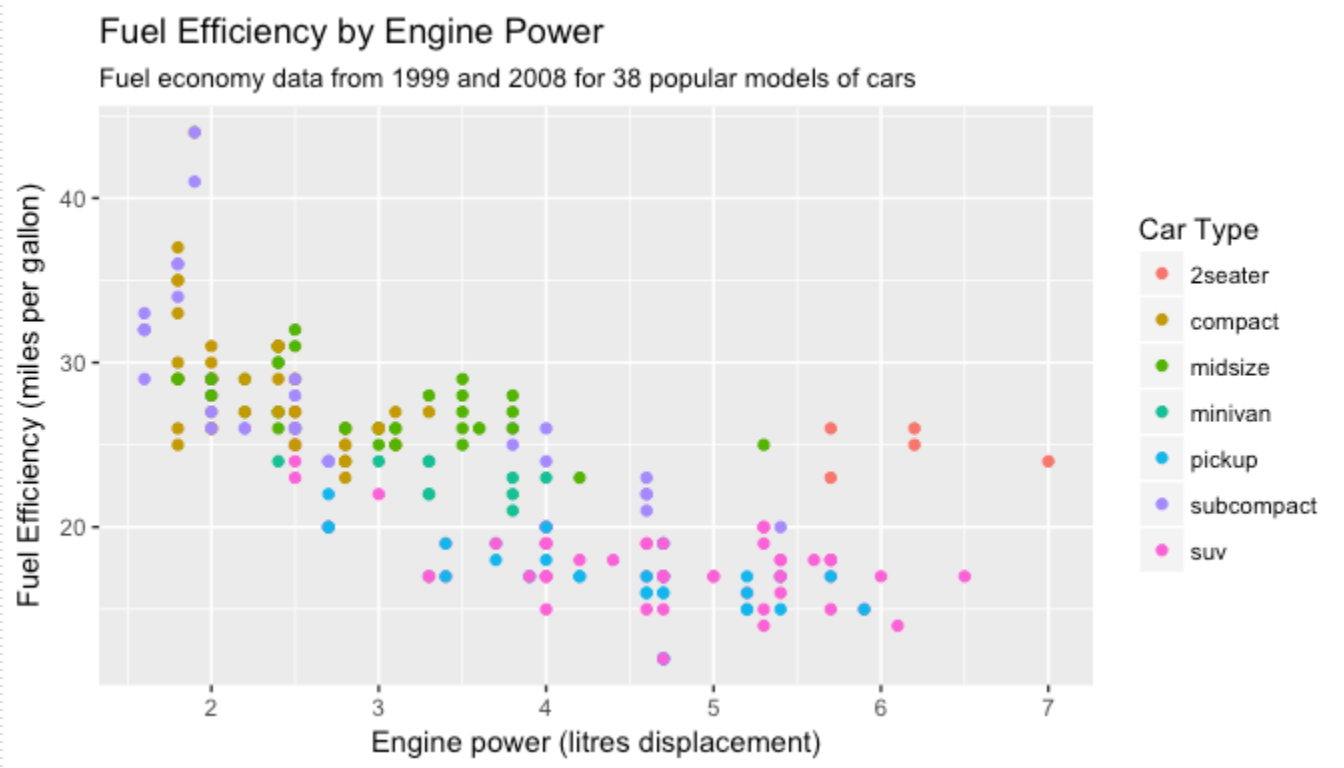
Labels & Annotations

- Textual labels and annotations (on the plot, axes, geometry, and legend) are an important part of making a plot understandable and communicating information. Although not an explicit part of the Grammar of Graphics (they would be considered a form of geometry), ggplot makes it easy to add such annotations.
- You can add titles and axis labels to a chart using the labs() function (not labels, which is a different R function!):

Labels & Annotations

```
ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
geom_point() +
labs(title = "Fuel Efficiency by Engine Power",
     subtitle = "Fuel economy data from 1999 and 2008 for
     38 popular models of cars",
     x = "Engine power (litres displacement)",
     y = "Fuel Efficiency (miles per gallon)",
     color = "Car Type")
```

Labels & Annotations



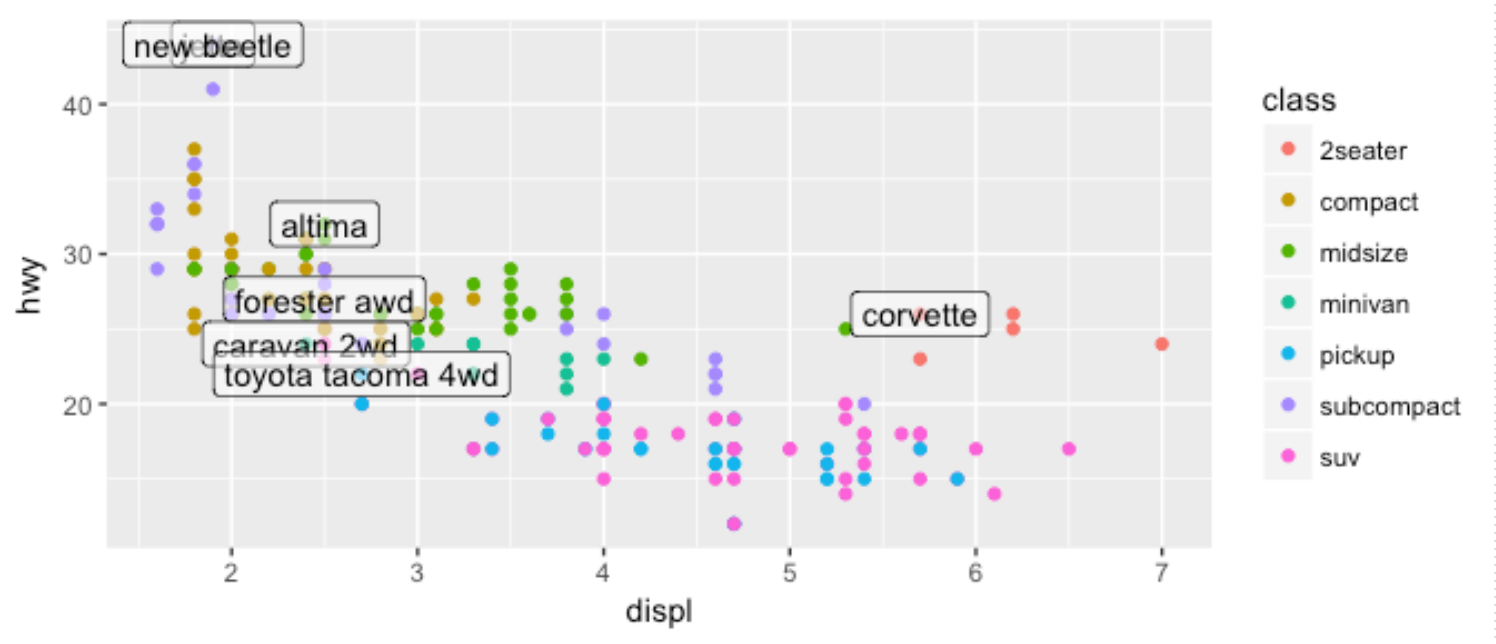
Labels & Annotations

- It is also possible to add labels into the plot itself (e.g., to label each point or line) by adding a new geom_text or geom_label to the plot; effectively, you're plotting an extra set of data which happen to be the variable names:

```
library(dplyr)
# a data table of each car that has best efficiency of its type
best_in_class <- mpg %>% group_by(class) %>% filter(row_number(desc(hwy)) == 1)
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_label(data = best_in_class, aes(label = model), alpha = 0.5)
```

labels 50% transparent

Labels & Annotations



The Operator %>%

- The infix operator %>% is not part of base R, but is in fact defined by the package magrittr (CRAN) and is heavily used by dplyr (CRAN).
- What the function does **is to pass the left hand side of the operator to the first argument of the right hand side of the operator.** In the following example, the data frame iris gets passed to head():

The Operator %>%

```
library(magrittr)
iris %>% head()
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	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	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

- Thus, iris %>% head() is equivalent to head(iris).

The Operator %>%

- Often, %>% is called multiple times to "chain" functions together, which accomplishes the same result as nesting. For example in the chain below, iris is passed to head(), then the result of that is passed to summary().

```
iris %>% head() %>% summary()
```

- Thus iris %>% head() %>% summary() is equivalent to summary(head(iris)). Some people prefer chaining to nesting because the functions applied can be read from left to right rather than from inside out.

Operator %>%

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

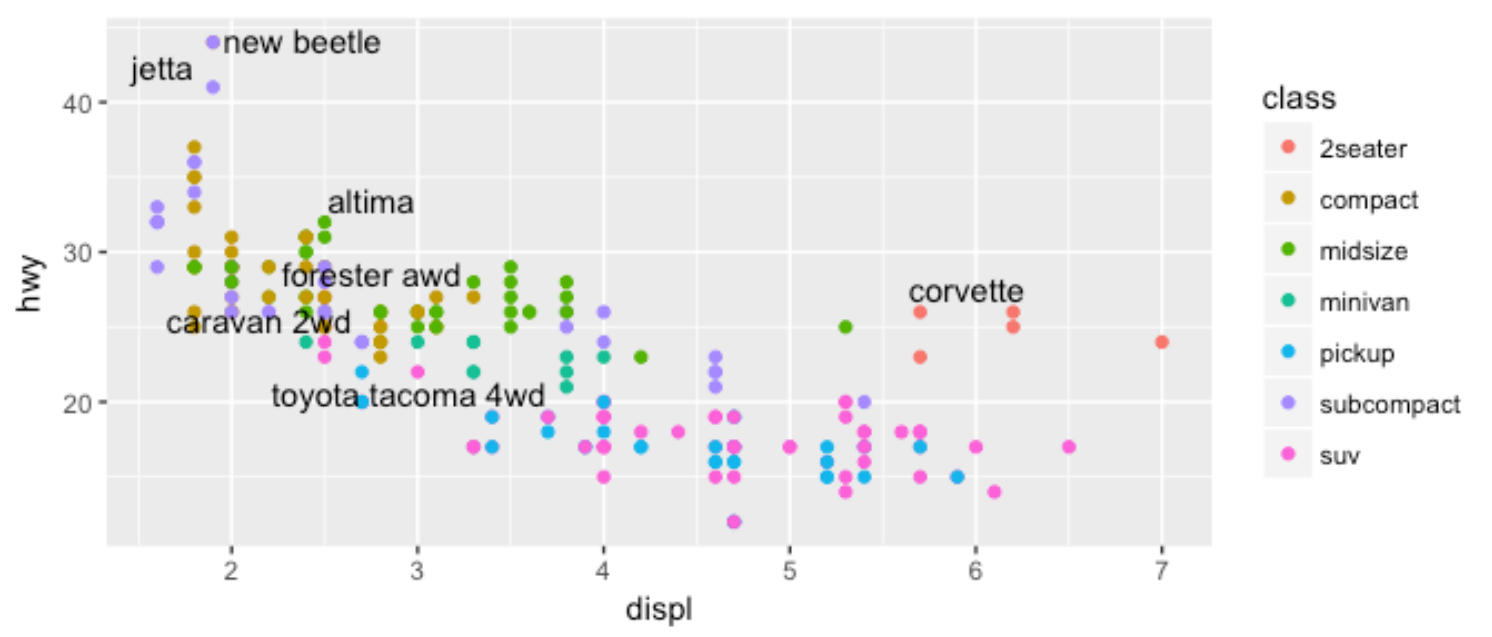
- In the above we further use the functions group_by and filter from the package dplyr.
- At the beginning the dataset mpg is grouped by class of the car.
- In the resulting object we then apply function filter that returns rows with the condition row_number(desc(hwy)) == 1; i.e. the row in each car with the highest hwy (highway miles per gallon).
- The result therefore is the car in each class with the highest highway miles per gallon.

Labels & Annotations

- Back to the plot we produced.
- Notice that two labels overlap one-another in the top left part of the plot. We can use the geom_text_repel function from the ggrepel package to help position labels.

```
library(ggrepel)
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_text_repel(data = best_in_class, aes(label = model))
```

Labels & Annotations



Other Visualization Libraries

- ggvis is a library that uses the Grammar of Graphics (similar to ggplot), but for interactive visualizations.
- plotly is a open-source library for developing interactive visualizations. It provides a number of "standard" interactions (pop-up labels, drag to pan, select to zoom, etc) automatically. Moreover, it is possible to take a ggplot2 plot and wrap it in Plotly in order to make it interactive. Plotly has many examples to learn from, though a less effective set of documentation.
- htmlwidgets provides a way to utilize a number of JavaScript interactive visualization libraries. JavaScript is the programming language used to create interactive websites (HTML files), and so is highly specialized for creating interactive experiences.