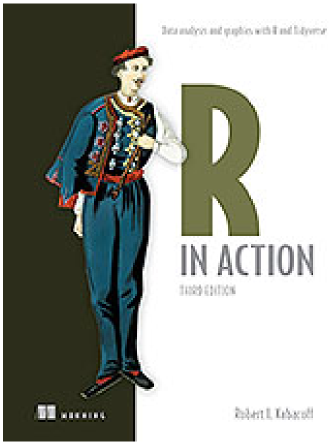

19 Advanced graphics

This chapter covers

- Customizing ggplot2 graphs
- Adding annotations
- Combining multiple graphs into a single plot
- Creating interactive graphs

There are many ways to create a graph in R. We've focused on the use of `ggplot2` because of its coherent grammar, flexibility, and comprehensiveness. The `ggplot2` package was introduced in chapter 4, with coverage of geoms, scales, and facets, and titles. In chapter 6, we created bar charts, pie charts, tree maps, histograms, kernel density plots, box and violin plots, and dot plots. Graphics for regression and ANOVA models were covered in chapters 8 and 9. Scatter plots, scatter plot matrices, bubble plots, line charts, corrgrams, and mosaic charts were covered in chapter 11. Other chapters have covered graphs as needed to visualize the topics at hand.

This chapter will continue the coverage of `ggplot2`, but with a focus is on customization – creating a graph that precisely meet your needs. Graphs help you to uncover patterns, and describe trends, relationships, differences, compositions, and distributions in data. The primary reason to customize a `ggplot2` graph is to enhance your ability to explore the data or communicate your findings to others. A secondary goal is to meet the look and feel requirements of an organization or publisher.

In this chapter, we'll explore the use `ggplot2` scale functions to customize axes and colors. The `theme()` function will be used to customize a graph's overall look and feel, including the appearance of text, legends, grid lines, and plot background. Geoms will be used to add annotations such as reference lines and labels. Additionally, the `patchwork` package will be used to combine several plots into one complete graph. Finally, the `plotly` package will be used to convert static `ggplot2` graphs into interactive web graphics that let you explore the data more fully.

The `ggplot2` package has an enormous number of options for customizing graph elements. The `theme()` function alone has over 90 arguments! Here, we'll focus on the most frequently used functions and arguments. If you're reading this chapter in greyscale, I encourage you to run the code so that you can see the graphs in color. Simple datasets are used so that you can focus on the code itself.

At this point, you should already have the `ggplot2` and `dplyr` packages installed. Before continuing, you'll need several additional packages including `ISLR` and data `gapminder`, and `ggrepel`, `showtext`, `patchwork`, and `plotly` for enhanced graphing. You can install them all using `install.packages(c("ISLR", "gapminder", "scales", "showtext", "ggrepel", "patchwork", "plotly"))`.

19.1 Modifying scales

The scale functions in `ggplot2` control the mapping of variable values to specific plot characteristics. For example, the `scale_x_continutous()` function creates a mapping of the values of a quantitative variable to positions along the *x*-axis. The `scale_color_discrete()` function creates a mapping between the values of a categorical variable and a color values. In this section, you'll use scale functions to customize a graph's axes and plot colors.

19.1.1 Customizing axes

In `ggplot2`, the *x*- and *y*-axes in a graph are controlled with `scale_x_*` and `scale_y_*` functions where `*` specifies the type of scale. The most common functions are given in table 19.1. The primary reason for customizing these axes is to make the data easier to read or make trends more obvious.

Table 19.1 Functions that specify axis scales

Function	Description
`scale_x_continuous`, `scale_y_continuous`	Scales for continuous data
`scale_x_binned`, `scale_y_binned`	Scales for binning continuous data

`scale_x_discrete, scale_y_discrete`	Scales for discrete (categorical) data
`scale_x_log10, scale_y_log10`	Scales for continuous data on a logarithmic scale (base 10)
`scale_x_date, scale_y_date`	Scales for date data. Other variants include datetime and time.

CUSTOMIZING AXES FOR CONTINUOUS VARIABLES

In the first example, we'll use the mtcars data frame, a dataset with car characteristics for 32 automobiles. The `mtcars` data frame is included in base R. Let's plot fuel efficiency (`mpg`) by automobile weight (`wt`) in 1000 lbs.

```
library(ggplot2)
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  labs(title = "Fuel efficiency by car weight")
```

The graph is given in figure 19.1. By default, major breaks are labeled. For `mpg`, these occur at positions 10 to 35 by 5. Minor breaks occur evenly between major breaks and are not labeled.

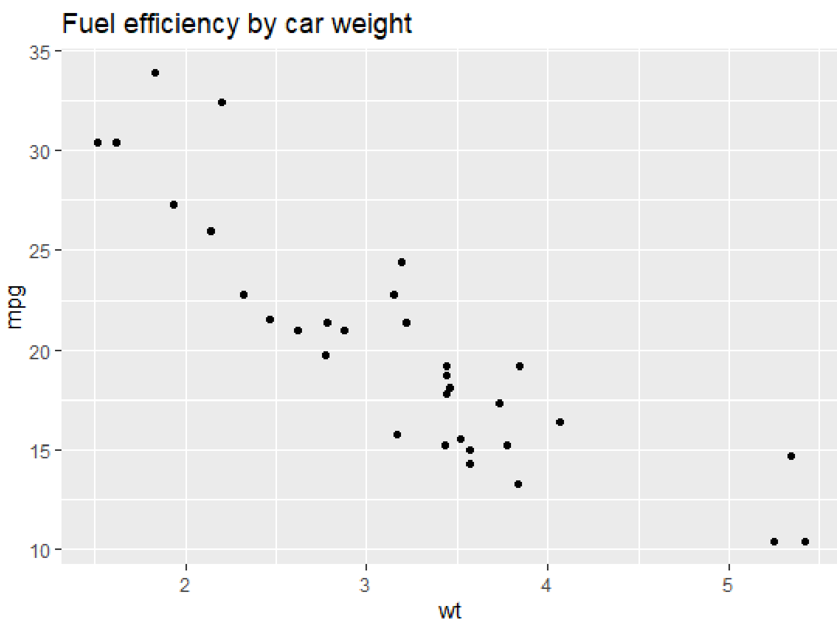

Figure 19.1 Default `ggplot2` scatter plot of miles per gallon by car weight (1000 lbs) for the 32 automobiles in the `mtcars` dataset.

Looking at the graph, what's the weight of the heaviest car in this plot? What's the mpg for the third lightest car? It takes some work to determine the values from these axes. We may want to tweak the *x*- and *y*-axes to make reading the values off the plot easier.

Since `wt` and `mpg` are continuous variables, we'll use the `scale_x_continuous()` and `scale_y_continuous()` functions to modify the axes. Common options for these functions are given in table 19.2.

Table 19.2 Some common scale_*_continuous options

Argument	**Description**
`name`	Name of the scale. Same as using the labs(x = , y =) function.
`breaks`	Numeric vector of positions for major breaks. Major breaks are labeled automatically unless overridden by the labels option. Use NULL to suppress breaks.
`minor_breaks`	Numeric vector of positions for minor tick marks. Minor breaks are not labeled. Use NULL to suppress minor breaks.
`n.breaks`	Integer guiding the number of major breaks. The number is taken as a suggestion. The function may vary this number to ensure attractive break labels.
`labels`	Character vector giving alternative break labels (must be same length as breaks)
`limits`	Numeric vector of length 2 giving minimum and maximum value
`position`	Axis placement (left/right for y-axis, top/bottom for x-axis)

Let's make the following changes. For weight,

- label the axis "Weight (1000 lbs.)"
- have the scale range from 1.5 to 5.5

- use 10 major breaks
- suppress minor breaks

For miles per gallon,

- label the axis "Miles per gallon"
- have the scale range from 10 to 35
- place major breaks at 10, 15, 20, 25, 30, and 35
- place minor breaks at one-gallon intervals

The code is given in listing 19.1.

Listing 19.1 Plot of fuel efficiency by car weight with customized axes

```
library(ggplot2)
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  scale_x_continuous(name = "Weight (1000 lbs.)",   #A
                     n.breaks = 10,                 #A
                     minor_breaks = NULL,           #A
                     limits = c(1.5, 5.5)) +        #A
  scale_y_continuous(name = "Miles per gallon",     #B
                     breaks = seq(10, 35, 5),       #B
                     minor_breaks = seq(10, 35, 1), #B
                     limits = c(10, 35)) +          #B
  labs(title = "Fuel efficiency by car weight")
```

#A Modify x-axis
#B Modify y-axis

The new graph is given in figure 19.2. We can see that the heaviest car is almost 5.5 tons, and that the 3rd lightest car gets 34 miles per gallon. Notice that you specified 10 major breaks for `wt`, but the plot only has 9. The `n.breaks` argument is taken as a suggestion. The argument may be replaced with a close number if it gives nicer labels. We'll continue to work with this graph in later sections.

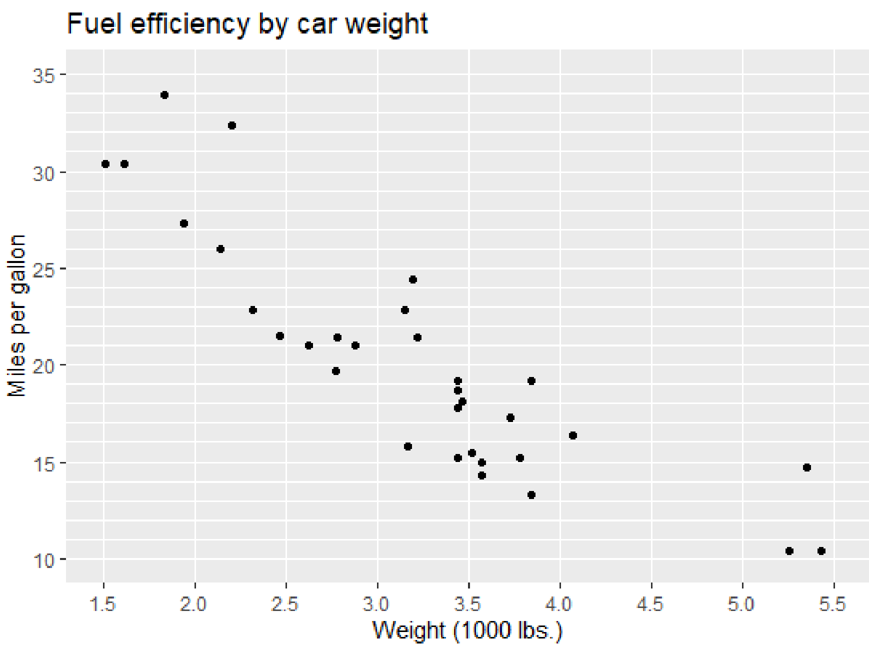

Figure 19.2 `ggplot2` scatter plot of miles per gallon by car weight (1000 lbs) with modify x- and y-axes. It is now easier to read off the values of the points.

CUSTOMIZING AXES FOR CATEGORICAL VARIABLES

The previous example involved customizing axes for continuous variables. In the next example, you'll customize the axes for categorical variables. The data come from the `Wage` data frame in the `ISLR` package. The data frame contains wage and demographic information on 3000 male workers in the Mid-Atlantic region of the United States collected in 2011. Let's plot the relationship between race and education in in this sample. The code is

```
library(ISLR)
library(ggplot2)
ggplot(Wage, aes(race, fill = education)) +
  geom_bar(position = "fill") +
  labs(title = "Participant Education by Race")
```

and the graph is given in Figure 19.3.

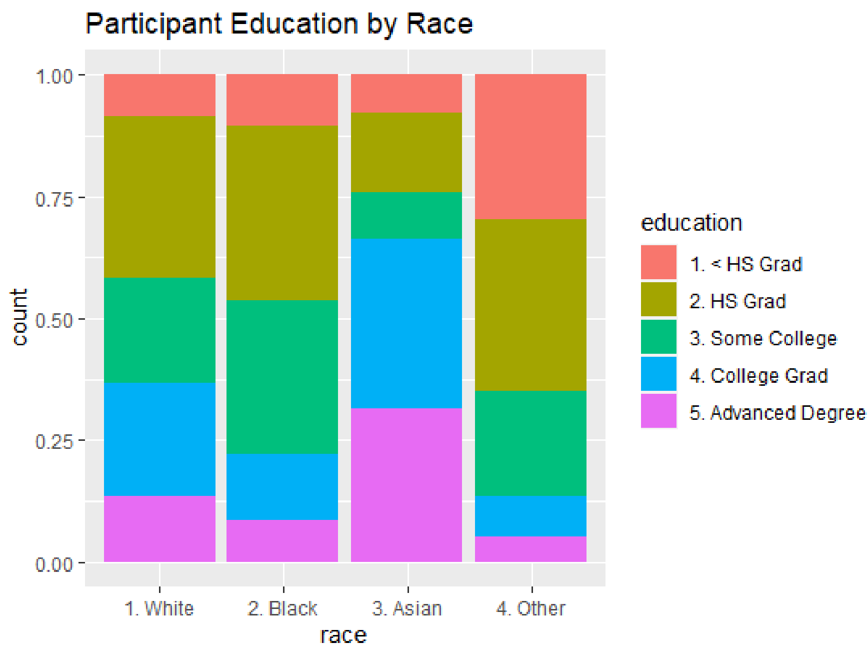

Figure 19.3 Participant education by race for a sample of 3000 Mid-Atlantic male workers in 2011.

Note that the numbering on the `race` and `education` labels is actually coded in the data.

```
> head(Wage[c("race", "education")], 4)
            race       education
231655 1. White    1. < HS Grad
86582  1. White 4. College Grad
161300 1. White 3. Some College
155159 3. Asian 4. College Grad
```

We can improve the graph by removing the numbers from the race category labels (they aren't ordinal categories), using percent formatting on the y-axis, using better scale labels, and reordering the race categories by percent with higher degrees. You may also want to drop the race-Other category since the composition of this group is unknown.

Modifying scales for categorical variables involve using scale_*_discrete() functions. Common options are given in table 19.3. You can order (and/or omit) the discrete values using the `limits` argument and change their labels using `labels` argument.

Table 19.3 Some common scale_*_discrete options

Argument	**Description**
`name`	Name of the scale. Same as using the labs(x = , y =) function.
`breaks`	A character vector of breaks.
`limits`	A character vector that defines the values of the scale and their order.
`labels`	A character vector giving the labels (must be same length as breaks). Use `labels=abbreviate` will shorten long labels
`position`	Axis placement (left/right for y-axis, top/bottom for x-axis)

The revised code is given in listing 19.2 and the graph is presented in figure 19.4.

Listing 19.2 Plot of eduction by race with customized axes

```
library(ISLR)
library(ggplot2)
library(scales)
ggplot(Wage, aes(race, fill=education)) +
  geom_bar(position="fill") +
  scale_x_discrete(name = "",                                         #A
                   limits = c("3. Asian", "1. White", "2. Black"),    #A
                   labels = c("Asian", "White", "Black")) +           #A
  scale_y_continuous(name = "Percent",                                #B
                     label = percent_format(accuracy=2),              #B
                     n.breaks=10) +                                   #B
  labs(title="Participant Education by Race")
```

#A Modify x-axis
#B Modify y-axis

The horizontal axis represents a categorical variable, so it's customized using the `scale_x_discrete()` function. The race categories are reordered using `limits` and relabeled using `labels`. The "Other" category is omitted from the graph by leaving it out of these specifications. The axis title is dropped by setting the name to "".

The vertical axis represents a numeric variable, so it's customized using the `scale_y_continuous()` function. The function is used to modify the title axis title and change the axis labels. The `percent_format()` function from the `scales` package reformats the axis labels to percents. The argument `accuracy=2` specifies the number of significant digits to print for each percent.

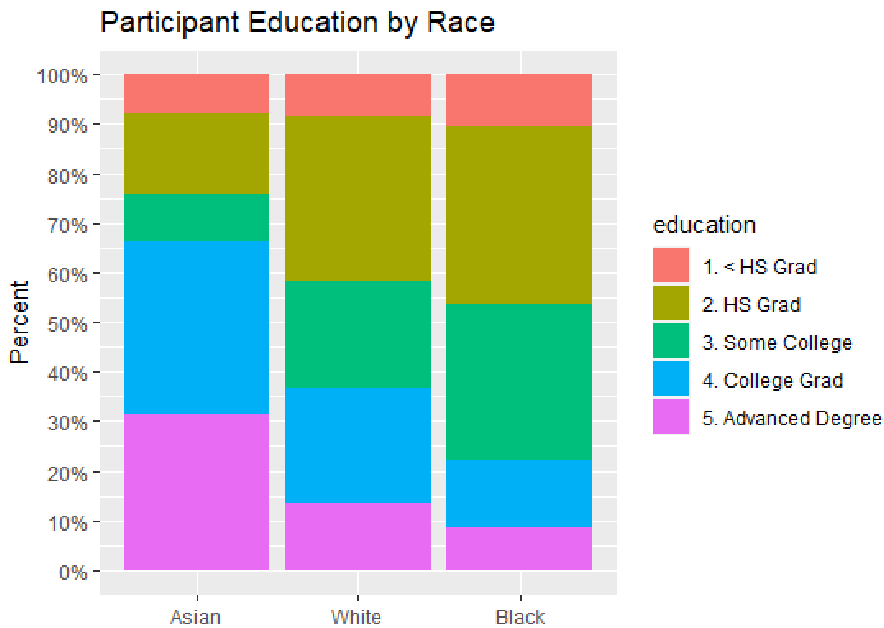

Figure 19.4 Participant education by race for a sample of 3000 Mid-Atlantic male workers in 2011. The race categories have been rearranged and relabeled. The "Other" category has been omitted. The x-axis label has been omitted and the y-axis is now formatted as percentages.

The `scales` package can be very useful for formatting axes. There are options for formatting monetary values, dates, percents, commas, scientific notation, and more. See https://scales.r-lib.org/ for details. The `ggh4x` and `ggprism` packages provide additional capabilities for customizing axes, including greater customization of major and minor tick marks.

In the previous example, education was represented on a discrete color scale. Customizing colors is considered next.

19.1.2 Customizing colors

The ggplot2 package provides functions for mapping both categorical and numeric variables to color schemes. The functions are described in table 19.3. The `scale_color_*()` functions are used for points, lines, borders, and text. The `scale_fill_*()` functions are used for objects that have area, such as rectangles, and ovals.

Color palettes can be *sequential*, *diverging*, or *qualitative*. Sequential palettes are used to map colors to a monotonic numeric variable. A diverging palette is used for numeric variables that have a meaningful middle or zero point. It's two sequential palettes that share an endpoint at the central value. For example, a diverging palette is often used to represent the values of correlation coefficients (see section 11.3). A qualitative color scale maps the values of a categorical variable to discrete colors.

Table 19.3 Functions for specifying color scales

Function	Description
`scale_color_gradient()` `scale_fill_gradient()`	Gradient color scale for a continuous variable. Specify the `low` color and the `high` color. Use the `*_gradient2()` versions to specify `low`, `mid`, and `high` colors.
`scale_color_steps()` `scale_fill_steps()`	Binned gradient color scale for a continuous variable. Specify the `low` color and the `high` color. Use the `*_steps2()` versions to specify `low`, `mid`, and `high` colors.
`scale_color_brewer()` `scale_fill_brewer()`	Sequential, diverging, and qualitative color schemes from ColorBrewer (https://colorbrewer2.org). Primary argument is `palette=`. See `?scale_color_brewer` for a list of palettes.
`scale_color_grey()` `scale_fill_gray()`	Sequential grey color scale. Optional arguments are `start` (grey value at the low end) and `end` (grey value at the high end). The defaults are 0.2 and 0.8 respectively.
`scale_color_manual()` `scale_fill_manual()`	Create your own color scale for a discrete variable by specifying a vector of colors in the `values` argument.
`scale_color_virdis_*` `scale_fill_virdis_*`	Viridis color scales from `viridisLite` package. Designed to be perceived by viewers with common forms of color blindness and prints well in black-and-white. Use *_d for discrete, *_c for

continuous, and *_b for binned scales. For example `scale_fill_viridis_d()` would provide safe color fills for a discrete variable. The argument `option` provides 4 color scheme variations ("inferno", "plasma", "viridis" (the default), and "cividis").

CONTINUOUS COLOR PALETTES

Let's look at examples of mapping a continuous quantitative variable to color palette. In figure 19.1 fuel efficiency was plotted against car weight. We'll add a third variable to the plot by mapping engine displacement to point color. Since engine displacement is a numeric variable, you create a color gradient to represent its values. Listing 19.3 demonstrates several possibilities.

Listing 19.3 Color gradients for continuous variables

```
library(ggplot2)
p <- ggplot(mtcars, aes(x=wt, y=mpg, color=disp)) +
  geom_point(shape=19, size=3) +
  scale_x_continuous(name = "Weight (1000 lbs.)",
                     n.breaks = 10,
                     minor_breaks = NULL,
                     limits=c(1.5, 5.5)) +
  scale_y_continuous(name = "Miles per gallon",
                     breaks = seq(10, 35, 5),
                     minor_breaks = seq(10, 35, 1),
                     limits = c(10, 35))

p + ggtitle("A. Default color gradient")

p + scale_color_gradient(low="grey", high="black") +
  ggtitle("B. Greyscale gradient")

p + scale_color_gradient(low="red", high="blue") +
  ggtitle("C. Red-blue color gradient")

p + scale_color_steps(low="red", high="blue") +
  ggtitle("D. Red-blue binned color Gradient")

p + scale_color_steps2(low="red", mid="white", high="blue",
                       midpoint=median(mtcars$disp)) +
  ggtitle("E. Red-white-blue binned gradient")
```

```
p + scale_color_viridis_c(direction = -1) +
  ggtitle("F. Viridis color gradient")
```

The code creates the plots in figure 19.5. The `ggtitle()` function is equivalent to the `labs(title=)` used elsewhere in this book. If you are reading a greyscale version of this book, be sure to run the code yourself so that you can appreciate the color variations.

Plot A uses the `ggplot2` default. Plot B shows the plot in greyscale. Plots C and D use a red to blue gradient. A binned color gradient takes a continuous gradient and divides it up into discrete values (usually 5). Plot E demonstrates a divergent color gradient, moving from red (low) to white (midpoint) to blue (high). Finally Plot F shows a viridis color gradient. The option `direction = -1` in Plot F reverses the color anchors leading to darker colors for greater engine displacement.

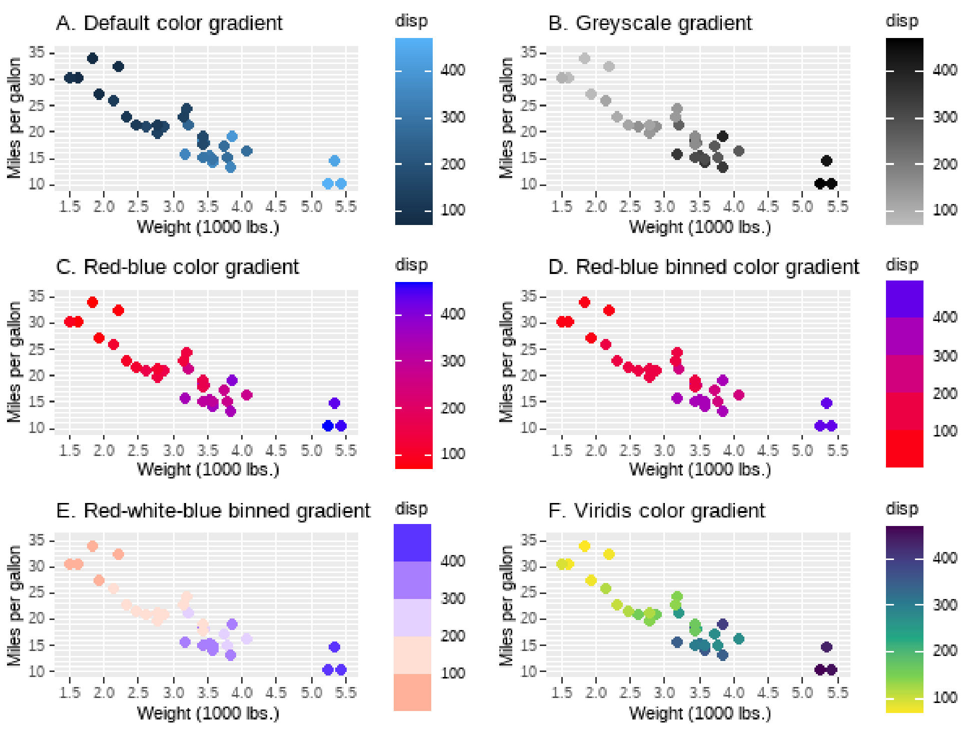

Figure 19.5 Plot of fuel efficiency by car weight. Color is used to represent engine displacement. Six color schemes are presented. A is the default. B is greyscale. Both C and D go from red to blue, but D is binned into 5 discrete colors. E goes from red to white (at the median) to blue. F uses a viridis color scheme. In

each graph, engine place increases with car weight and lower gas mileage.

Qualitative color palettes

Qualitative color schemes are demonstrated in listing 19.4. Here education is the categorical variable mapped to discrete colors. The resulting graphs are provided in figure 19.6.

Listing 19.4 Color schemes for categorical variables

```
library(ISLR)
library(ggplot2)
p <- ggplot(Wage, aes(race, fill=education)) +
  geom_bar(position="fill") +
  scale_y_continuous("Percent", label=scales::percent_format(accuracy=2),
                     n.breaks=10) +
  scale_x_discrete("",
                   limits=c("3. Asian", "1. White", "2. Black"),
                   labels=c("Asian", "White", "Black"))

p + ggtitle("A. Default colors")

p + scale_fill_brewer(palette="Set2") +
  ggtitle("B. ColorBrewer Set2 palette")

p + scale_fill_viridis_d() +
  ggtitle("C. Viridis color scheme")

p + scale_fill_manual(values=c("gold4", "orange2", "deepskyblue3",
                               "brown2", "yellowgreen")) +
  ggtitle("D. Manual color selection")
```

Plot A uses the `ggplot2` default colors. Plot B uses the ColorBrewer qualitative palette "Set2". Other qualitative ColorBrewer palettes include Accent, Dark2, Paired, Pastel1, Pastel2, Set1, and Set3. Plot C demonstrates the default Viridis discrete scheme. Finally, plot D demonstrates a manual scheme, proving that I have no business picking colors on my own.

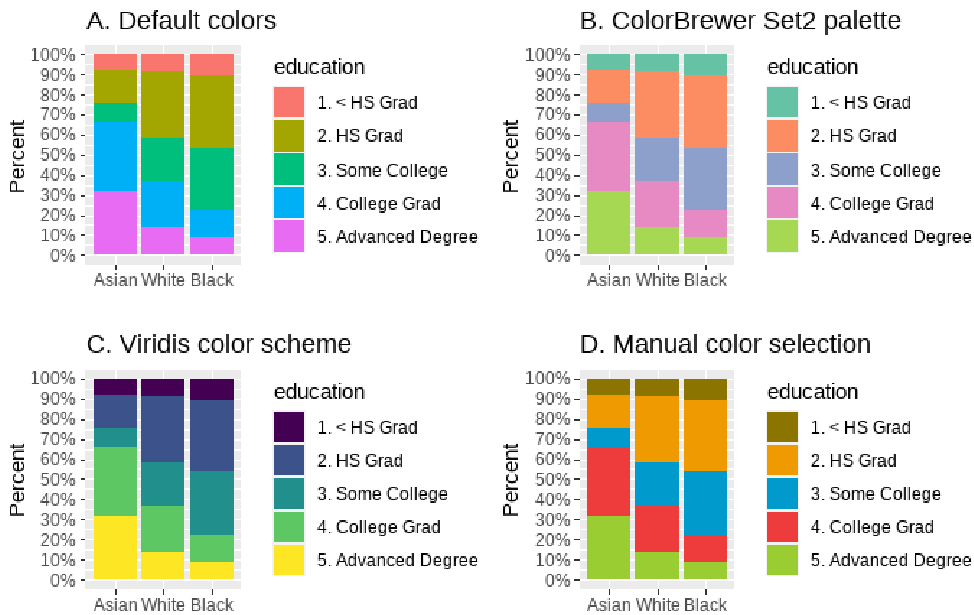

Figure 19.6 Participant education by race for a sample of 3000 Mid-Atlantic male workers in 2011. Four different color schemes are displayed. A is the default. B and C are preset color schemes. In D, the colors are specified by the user.

R packages provide a wide variety of color palettes for use in `ggplot2` graphs. Emil Hvitfeldt has created a comprehensive repository at https://github.com/EmilHvitfeldt/r-color-palettes (almost 600 at last count!). Choose the scheme that (a) you find appealing, and (b) helps communicate the information most effectively. Can the reader easily see the relationships, differences, trends, composition, or outliers that you are trying to highlight?

19.2 Modifying themes

The `ggplot2 theme()` function allows you to customize the non-data components of your plot. The help for the function (`?theme`) describes arguments for modifying a graph's titles, labels, fonts, background, gridlines, and legends.

For example, in the code

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()+
  theme(axis.title = element_text(size = 14, color = "blue"))
```

the `theme()` function renders the *x*- and *y*-axis titles in a 14pt blue font. Functions are typically used to provide the values for `theme` arguments (see table 19.3).

Table 19.3 Theme elements

Function	Description
`element_blank()`	Blank out element (useful for removing text, lines, etc.)
`element_rect()`	Specify rectangle characteristics. Arguments include `fill`, `color`, `size`, and `linetype`. The last 3 refer to the border.
`element_line()`	Specify line characteristics. Arguments include `color`, `size`, `linetype`, `lineend` ("round", "butt", "square"), and `arrow` (created with the `grid::arrow()` function).
`element_text()`	Specify text characteristics. Arguments include `family` (font family), `face` ("plain", "italic", "bold", "bold.italic"), `size` (text size in pts.), `hjust` (horizontal justification in [0,1]), `vjust` (vertical justification in [0,1]), angle (in degrees), and `color`.

First, we'll look at some preconfigured themes that change numerous elements at once to provide a cohesive look and feel. Then we'll dive into customizing individual theme elements.

19.2.1 Prepackaged themes

The ggplot2 package comes with eight preconfigured themes that can be applied to ggplot2 graphs via `theme_*()` functions. Four of the most popular are demonstrated in listing 19.5 and figure 19.7. The `theme_grey()` function is the default theme, while `theme_void()` creates a completely empty theme.

Listing 19.5 Demonstration of 4 preconfigured ggplot2 themes

```
library(ggplot2)
p <- ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  labs(x = "Weight (1000 lbs)",
       y = "Miles per gallon")

p + theme_grey() + labs(title = "theme_grey")
p + theme_bw() + labs(title = "theme_bw")
p + theme_minimal() + labs(title = "theme_minimal")
```

```
p + theme_classic() + labs(title = "theme_classic")
```

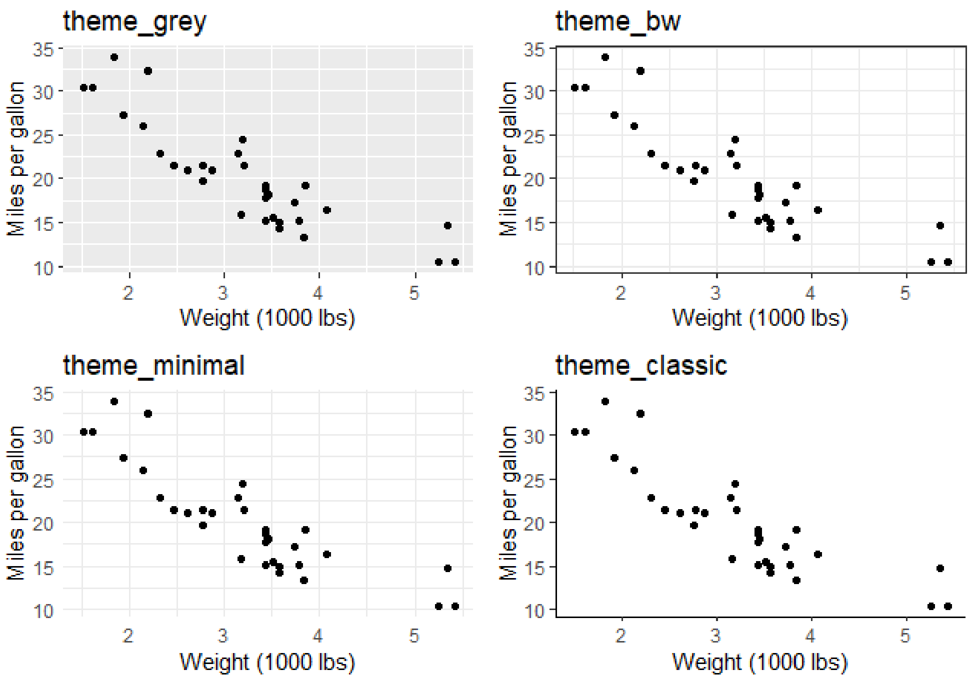

Figure 19.7 Examples of 4 preconfigured themes. By default, `ggplot2` uses `theme_grey()`.

Additional themes are provided by the `ggthemes`, `hbrthemes`, `xaringanthemer`, `tgamtheme`, `cowplot`, `tvthemes`, and `ggdark` packages. Each is available from CRAN. Additionally, some organizations provide preconfigured themes to their employees to assure a consistent appearance in reports and presentations.

In addition to preconfigured themes, you can modify individual theme elements. In the following sections you'll use theme arguments to customize fonts, legends, and other graph elements.

19.2.2 Customizing fonts

It is important to use typography to help communicate meaning without distracting or confusing the reader (see http://mng.bz/5Z1q). For example, Googles Roboto and Lora typefaces are often recommended for clarity. Base R has limited native font handling capabilities. The `showtext` package greatly expands these capabilities, allowing you to add system fonts and Google Fonts to plots.

The steps are:

1. Load local and/or Google fonts
2. Set `showtext` as the output graphics device
3. Specify fonts in the `ggplot2 theme()` function

When considering local fonts, the location, number, and type of fonts vary greatly from computer to computer. In order use local fonts (other than R's defaults), you need to know the names and locations of the font files on your system. Currently supported formats include TrueType fonts (*.ttf, *.ttc) and OpenType fonts (*.otf).

The `font_paths()` function lists the location of font files, while `font_files()` list the font files and their characteristics. Listing 19.6 provides a short function for locating font files on your local system. Here the function is used to locate the font files for the Comic Sans MS font. Since the results depend on your system (I'm using a Windows PC), your results are likely to vary.

Listing 19.6 Locating local font files

```
> findfont <- function(x){
    suppressMessages(require(showtext))
    suppressMessages(require(dplyr))
    filter(font_files(), grepl(x, family, ignore.case=TRUE)) %>%
      select(path, file, family, face)
  }

> findfont("comic")

             path        file        family        face
1 C:/Windows/Fonts   comic.ttf Comic Sans MS     Regular
2 C:/Windows/Fonts comicbd.ttf Comic Sans MS        Bold
3 C:/Windows/Fonts  comici.ttf Comic Sans MS      Italic
4 C:/Windows/Fonts  comicz.ttf Comic Sans MS Bold Italic
```

Once you've located local font files, use `font_add()` to load them. For example, on my machine

```
font_add("comic", regular = "comic.ttf",
          bold = "comicbd.ttf", italic="comici.ttf")
```

makes the Comic Sans MS the font available in R under the arbitrary name "comic".

To load Google Fonts (https://fonts.google.com/), use the statement

`font_add_google(`*`name`*`, `*`family`*`)`

where *`name`* is the name of the Google font, and *`family`* is the arbitrary name you'll use to refer to the font in later code. For example

```
font_add_google("Schoolbell", "bell")
```

loads the Schoolbell Google Font under the name "bell".

Once you've loaded the fonts, the statement `showtext_auto()` will set `showtext` as the output device for new graphics.

Finally use the `theme()` function to indicate which elements of the graph will use which fonts. Table 19.4 lists the theme arguments related to text. You can specify font family, face, size, color, and orientation using `element_text()`.

Table 19.4 `theme()` arguments related to the text.

Argument	Description
`axis.title`, `axis.title.x`, `axis.title.y`	Axis titles.
`axis.text` with the same variations as `axis.title`	Tick labels along axes.
`legend.text`, `legend.title`	Legend item labels and legend title.
`plot.title`, `plot.subtitle`, `plot.caption`	Plot title, subtitle, and caption.
`strip.text`, `strip.text.x`, `strip.text.y`	Facet labels.

Listing 19.7 demonstrates customizing a `ggplot2` graph with 2 local fonts from my machine (Comic Sans MS and Caveat) and 2 Google Fonts (Schoolbell and Gochi Hand). The graph is displayed in figure 19.8.

Listing 19.7 Customizing fonts in a ggplot2 graph

```
library(ggplot2)
library(showtext)

font_add("comic", regular = "comic.ttf",                  #1
         bold = "comicbd.ttf", italic="comici.ttf")
```

```
font_add("caveat", regular = "caveat-regular.ttf",
         bold = "caveat-bold.ttf")

font_add_google("Schoolbell", "bell")                        #2
font_add_google("Gochi Hand", "gochi")

showtext_auto()                                              #3

ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  labs(title = "Fuel Efficiency by Car Weight",
       subtitle = "Motor Trend Magazine 1973",
       caption = "source: mtcars dataset",
       x = "Weight (1000 lbs)",
       y = "Miles per gallon") +

   theme(plot.title    = element_text(family = "bell", size=14),   #4
         plot.subtitle = element_text(family = "gochi"),
         plot.caption  = element_text(family = "caveat", size=15),
         axis.title    = element_text(family = "comic"),
         axis.text     = element_text(family = "comic",
                                  face="italic", size=8))
```

#1 Load local fonts
#2 Load Google fonts
#3 Use showtext as the graphic device
#4 Specify plot fonts

The resulting graph is presented in figure 19.8. The graph is used for demonstrations purposes only. Using several fonts in a single plot is often distracting and takes away from the message the graph was designed to convey. Pick the one or two fonts that best highlight the information and stick with them. A useful starting guide is *Choosing fonts for your data visualiz*ation, by Tiffany France (http://mng.bz/nrY5).

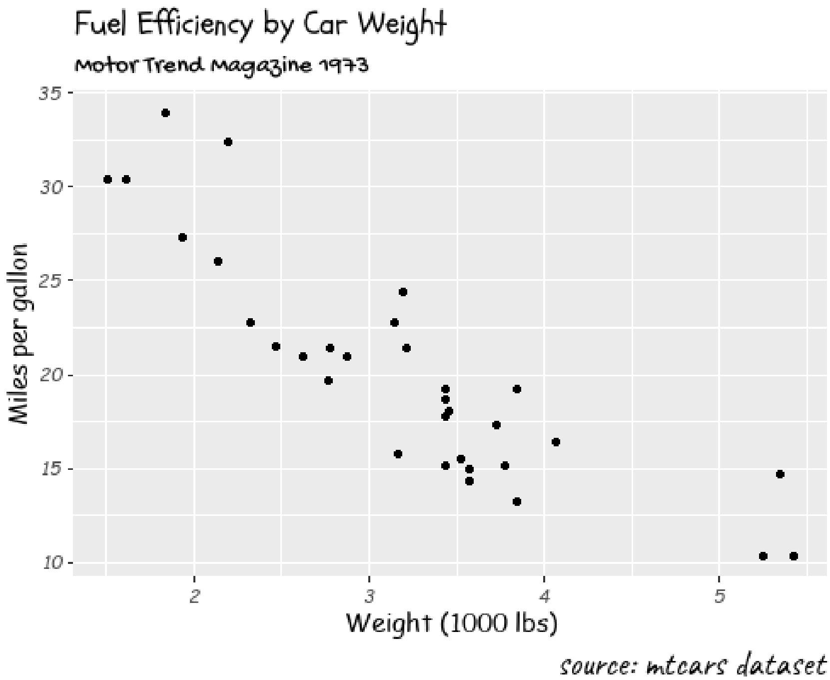

Figure 19.8 Graph using several fonts (SchoolBell for title, Gochi Hand for the subtitle, Caveat for the caption, and Comic Sans MS for the axis titles and text).

19.2.3 Customizing legends

The ggplot2 package creates legends whenever variables are mapped to color, fill, shape, line type, or size (basically any scaling the does not involve positional scales). You can modify the appearance of a legend using the theme() arguments in table 19.5.

The most frequently used argument is `legend.position`. Setting the argument to "top", "right" (default), "bottom" or "left" allows you to position the legend on any side of the graph. Alternatively, a two-element numeric vector (*x*, *y*) positions the legend on the *x*- and *y*-axis, with the *x* coordinate ranging from 0-left to 1-right, and the *y* coordinate ranging from 0-bottom to 1-top.

Table 19.5 `theme()` arguments related to the plot legend.

Argument	**Description**

`legend.background`, `legend.key`	Background of the legend and the legend key (symbols). Specify with `element_rect()`.
`legend.title`, `legend.text`	Text characteristics for the legend title and text. Specify values with `element_text()`..
`legend.position`	Position of the legend. Values are "none", "left", "right", "bottom", "top", or two-element numeric vector (each between 0-left/bottom and 1-right/top).
`legend.justification`	If `legend.postion` is set with a two-element numeric vector, `legend.justification` gives the *anchor point within the legend*, as a two-element vector. For example, if `legend.position = c(1, 1)` and `legend.justification = c(1, 1)`, the anchor point is the right corner of the legend. This anchor point is placed in the top right corner of the plot.
`legend.direction`	Legend direction as "horizontal" or "vertical".
`legend.title.align`, `legend.text.align`	Alignment of legend title and text (number from 0-left to 1-right).

Let's create a scatterplot for the `mtcars` data frame. Place `wt` on the *x*-axis, mpg on the *y*-axis, and color the points by the number engine cylinders. Using table 19.5, customize the graph by

- placing the legend in the upper right-hand corner of the plot
- titling the legend "Cylinders"
- listing the legend categories horizontally rather than vertically
- setting the legend background to light gray and removing the background around the key elements (the colored symbols)
- placing a white border around the legend

The code is provided in listing 19.8 and the result plot is given in figure 19.9.

Listing 19.8 Customizing a plot legend

```
library(ggplot2)
```

```
ggplot(mtcars, aes(wt, mpg, color = factor(cyl))) +
         geom_point(size=3) +
  scale_color_discrete(name="Cylinders") +
  labs(title = "Fuel Efficiency for 32 Automobiles",
       x = "Weight (1000 lbs)",
       y = "Miles per gallon") +
  theme(legend.position = c(.95, .95),
        legend.justification = c(1, 1),
        legend.background = element_rect(fill = "lightgrey",
                                         color = "white",
                                         size = 1),
        legend.key = element_blank(),
        legend.direction = "horizontal")
```

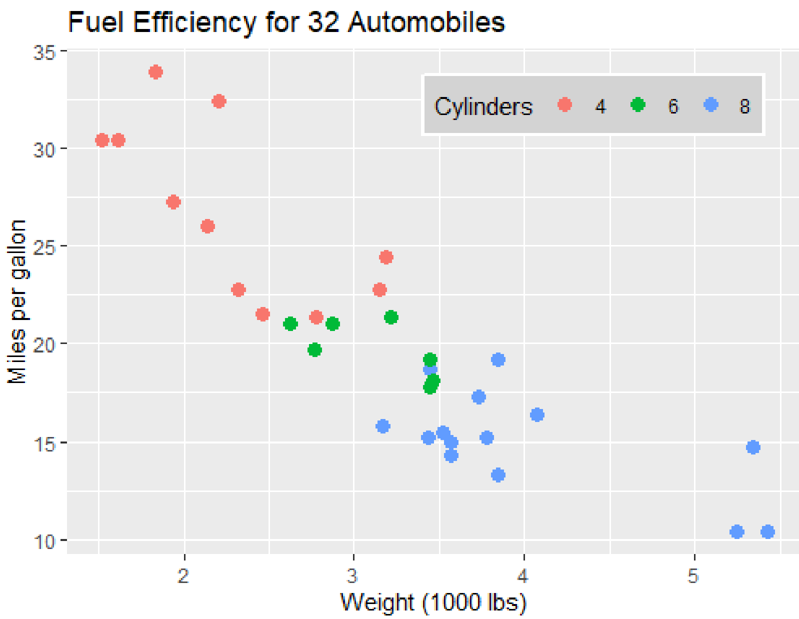

Figure 19.9. Plot with a customized legend. The upper right corner of the legend is placed in the upper left corner of the plot. The legend is printed horizontally, with a grey background, a solid white border, and a title.

Again, this graph is provided for demonstration purposes. It's actually easier to relate the legend to the graph if was it's placed on the right side and vertical (the default in this case). See the *Data Visualization Standard* website (http://mng.bz/6m15) for recommendations regarding legend formatting.

19.2.4 Customizing the plot area

The theme arguments in table 19.6 allow you to customize the plot area. The most common changes are the background color and major and minor grid lines. Listing 19.9 demonstrates customizing many features of the plot area for a faceted scatterplot.

Table 19.6 `theme()` arguments related to plot area.

Argument	**Description**
`plot.background`	Background of the entire plot. Specify with `element_rect()`.
`plot.margin`	Margin around entire plot. Use `units()` function with sizes for top, right, bottom, and left margins.
`panel.background`	Background for plotting area. Specify with `element_rect()`.
`strip.background`	Background of the facet strip label.

`panel.grid,`
`panel.grid.major,`
`panel.grid.minor,`
`panel.grid.major.x`
`panel.grid.major.y`
`panel.grid.minor.x`
`panel.grid.minor.y`
	Grid lines, major grid lines, minor grid lines, or specific major or minor grid lines. Specify with `element_line()`.

`axis.line, axis.line.x,`
`axis.line.y,`
`axis.line.x.top,`
`axis.line.x.bottom,`
`axis.line.y.left,`
`axis.line.y.right`
	Lines along the axes (`axis.line`), the lines for each plane (`axis.line.x`, `axis.line.y`), or individual lines for each axis (`axis.line.x.bottom`, etc.). Specify with `element_line()`.

The resulting graph is presented in figure 19.10.

Listing 19.9 Customizing the plot area

```
library(ggplot2)
mtcars$am <- factor(mtcars$am, labels = c("Automatic", "Manual"))
ggplot(data=mtcars, aes(x = disp, y = mpg)) +
  geom_point(aes(color=factor(cyl)), size=2) +                         #1
  geom_smooth(method="lm", formula = y ~ x + I(x^2),                   #2
              linetype="dotted", se=FALSE) +
  scale_color_discrete("Number of cylinders") +
  facet_wrap(~am, ncol=2) +                                            #3
  labs(title = "Mileage, transmission type, and number of cylinders",
       x = "Engine displacement (cu. in.)",
       y = "Miles per gallon") +
  theme_bw() +                                                        #4
  theme(strip.background = element_rect(fill = "white"),              #5
        panel.grid.major = element_line(color="lightgrey"),
        panel.grid.minor = element_line(color="lightgrey",
                                        linetype="dashed"),
        axis.ticks = element_blank(),
        legend.position = "top",
        legend.key = element_blank())
```

#1 Grouped scatterplot
#2 Fit line
#3 Faceting
#4 Set black-white theme
#5 Modify theme

The code creates a graph with engine displace (`disp`) on the *x*-axis and miles per gallon (mpg) on the *y*-axis. The number of cylinders (`cyl`) and transmission type (`am`) are originally coded as numeric but are converted to factors for plotting. For `cyl`, converting to a factor assures a single color for each number of cylinders. For `am`, this provides better labels than 0 and 1.

A scatterplot is created with enlarged points, colored by number of cylinders #1. A quadratic line of best fit is then added #2. A quadratic fit line allows for a line with one bend (see section 8.2.3). A faceted plot for each transmission type is then added #3.

To modify the theme, we started with `theme_bw()` #4, and then modified it with the `theme()` function #5. The strip background color is set to white. Major grid lines are set to solid light grey and minor grid lines are set to dashed light grey. Axis tick marks are removed. Finally, the legend is placed at the top of the graph and the legend keys (symbols) are given a blank background.

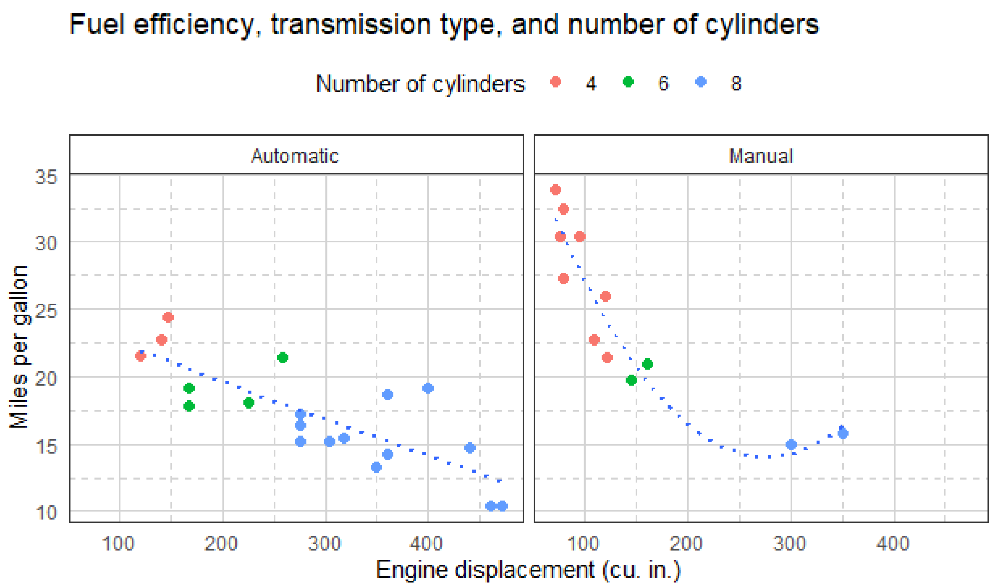

Figure 19.10 Faceted scatter plot with fit lines. The final theme is based on a modified version of the black and white theme.

19.3 Adding annotations

Annotations allow you to add additional information to a graph, making it easier for the reader to discern relationships, distributions, or unusual observations. The most common annotations are reference lines and text labels.

Table 19.7 Functions for adding annotations.

Function	**Description**
`geom_text, geom_label`	`geom_text()` adds text to a plot. `geom_label()` is similar, but draws a rectangle around the text.
`geom_text_repel, geom_label_repel`	These are functions from the `ggrepel` package. They are similar to `geom_text()` and `geom_label()`, but avoid overlapping text.

geom_hline, geom_vline, geom_abline	Adds horizontal, vertical, and diagonal reference lines.
geom_rect	Adds rectangles to the graph. Useful for highlighting areas of the plot.

LABELING POINTS

In figure 19.1 we plotted the relationship between car weight (wt) and fuel efficiency (mpg). However, the reader can't determine which cars are represented by which points without reference to the original data set. Listing 19.10 adds this information to the plot. The graph is given in figure 19.11.

Listing 19.10 Scatter plot with labeled points

```
library(ggplot2)
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point(color = "steelblue") +
  geom_text(label = row.names(mtcars)) +
  labs(title = "Fuel efficiency by car weight",
       x = "Weight (1000 lbs)",
       y = "Miles per gallon")
```

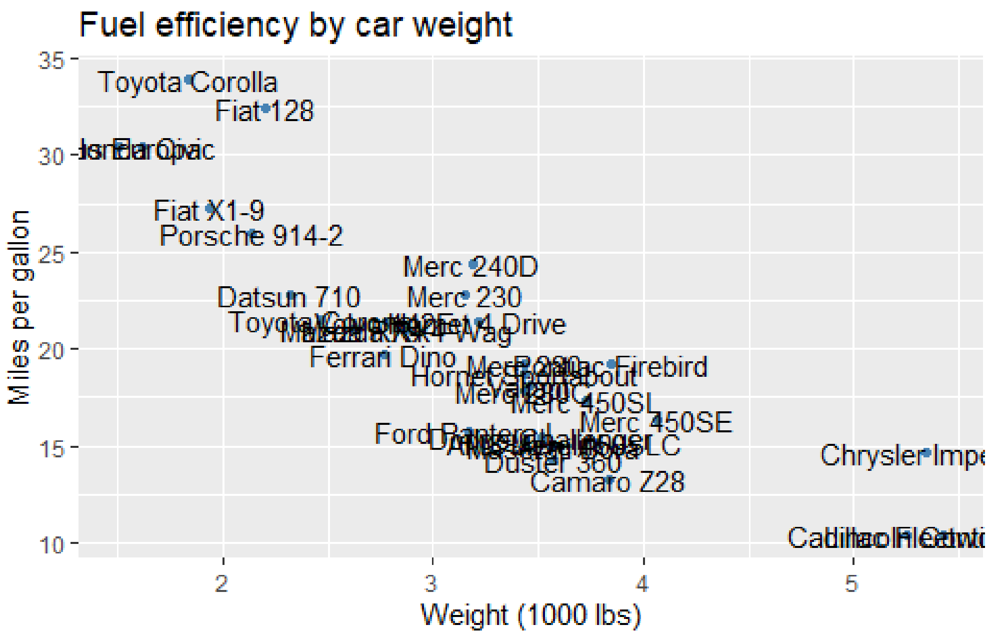

Figure 19.11 Scatterplot of car weight by miles per gallon. Points are labeled with car names.

The resulting graph is difficult to read due to overlapping text. The `ggrepel` package addresses this limitation by repositioning text labels to avoid overlaps. We recreate the graph using this package to label the points. Additionally, we'll add a reference line and label indicating the median MPG. The code is given in listing 19.11 and the graph is provided in figure 19.12.

Listing 19.11 Scatter plot with labeled points using ggrepel

```
library(ggplot2)
library(ggrepel)
ggplot(data = mtcars, aes(x= wt, y = mpg)) +
  geom_point(color = "steelblue") +
  geom_hline(yintercept = median(mtcars$mpg),                    #1
             linetype = "dashed",
             color = "steelblue") +
  geom_label(x = 5.2, y = 20.5,                                   #2
             label = "median MPG",
             color = "white",
```

```
           fill = "steelblue",
           size = 3) +
  geom_text_repel(label = row.names(mtcars), size = 3) +         #3
  labs(title = "Fuel efficiency by car weight",
       x = "Weight (1000 lbs)",
       y = "Miles per gallon")
```

#1 Reference line
#2 Reference line label
#3 Point labels

The reference line indicates which cars are above or below the median miles per gallon #1. The line is labeled using `geom_label` #2. The proper placement of the line label (*x*, *y*) takes some experimentation. Finally, the geom_text_repel() function is used to label the points. The size of the labels is also decreased from a default of 4 to 3. This graph is much easier to read and interpret.

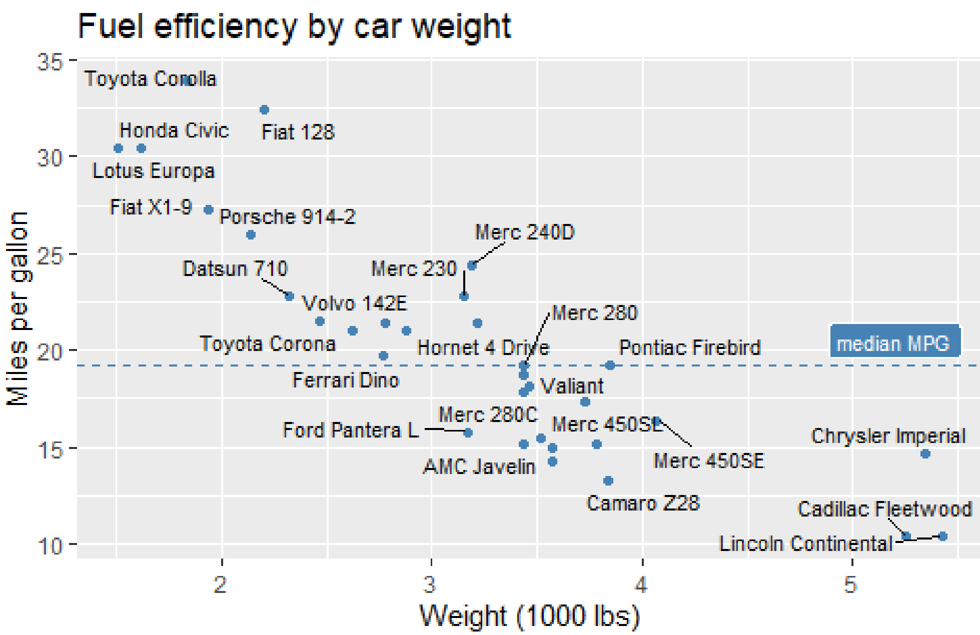

Figure 19.12 Scatterplot of car weight by miles per gallon. Points are labeled with car names. The `ggrepel` package has been used to reposition labels to avoid overlapping text. Additionally, a reference line and label have been added.

LABELING BARS

Labels can be added to bar charts to clarify the distribution of a categorical variable or the composition of a stacked bar chart. Adding percentage labels to each bar is a two-step process. First calculate the percentages for each bar. Then create the bar chart using these percentages directly, adding the labels via the `geom_text()` function. The process is demonstrated in listing 19.12. The graph is given in figure 19.13.

Listing 19.12 Adding percent labels to a bar chart

```
library(ggplot2)
library(dplyr)
library(ISLR)

plotdata <- Wage %>%                                                    #1
  group_by(race) %>%
  summarize(n = n()) %>%
  mutate(pct = n / sum(n),
         lbls = scales::percent(pct),
         race = factor(race, labels = c("White", "Black",
                                        "Asian", "Other")))

plotdata

## # A tibble: 4 x 4
##   race          n    pct lbl
##   <fct>     <int>  <dbl> <chr>
## 1 1. White  2480 0.827  82.7%
## 2 2. Black   293 0.0977 9.8%
## 3 3. Asian   190 0.0633 6.3%
## 4 4. Other    37 0.0123 1.2%

ggplot(data=plotdata, aes(x=race, y=pct)) +
  geom_bar(stat = "identity", fill="steelblue") +                       #2
  geom_text(aes(label = lbls),                                          #3
            vjust = -0.5,
            size = 3) +
  labs(title = "Participants by Race",
       x = "",
       y="Percent") +
  theme_minimal()
```

#1 Calculate percentages
#2 Add bars
#3 Add bar labels

The percentages for each race category are calculated #1, and formatted labels (`lbls`) are created using the `percent()` function in the `scales` package. A bar chart is then created using this summary data #2. The option `stat = "identity"` in the `geom_bar()` function tells `ggplot2` to use the *y*-values (bar heights) provided, rather than calculating them. The geom_text() function is then used to print the bar labels #3. The `vjust = -0.5` parameter raises the text slightly above the bar.

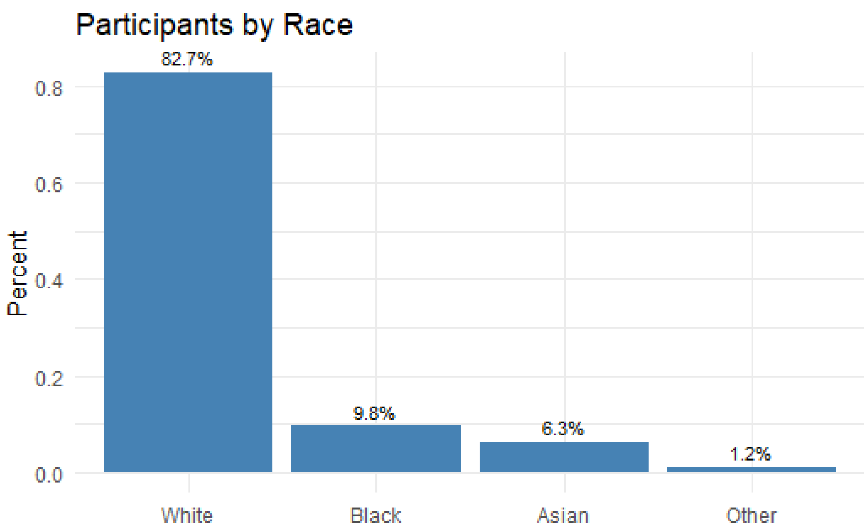

Figure 19.13. Simple bar chart with percent labels.

You can also add percentage labels to stacked bar charts. In listing 19.13 the filled bar chart from figure 19.4 is reproduced with added percent labels. The final plot is given in figure 19.14.

Listing 19.13 Adding percent labels to a stacked (filled) bar chart

```
library(ggplot2)
library(dplyr)
library(ISLR)

plotdata <- Wage %>%                                                    #1
  group_by(race, education) %>%
  summarize(n = n()) %>%
  mutate(pct = n/sum(n),
         lbl = scales::percent(pct))

ggplot(plotdata, aes(x=race, y=pct, fill=education)) +
  geom_bar(stat = "identity",
           position="fill",
           color="lightgrey") +
  scale_y_continuous("Percent",                                         #2
                     label=scales::percent_format(accuracy=2),
                     n.breaks=10) +
```

```
scale_x_discrete("",
                 limits=c("3. Asian", "1. White", "2. Black"),
                 labels=c("Asian", "White", "Black")) +
geom_text(aes(label = lbl),                                               #3
          size=3,
          position = position_stack(vjust = 0.5)) +
labs(title="Participant Education by Race",
     fill = "Education") +
theme_minimal() +                                                         #4
theme(panel.grid.major.x=element_blank())
```

#1 Calculate percentages
#2 Customize y- and x-axes
#3 Add percent labels
#4 Customize theme

This code is similar to the previous code. The percentages for each race by education combination are calculated #1, and the bar charts are produced using these percentages. The x- and y-axes are customized to match listing 19.2 #2. Next, the `geom_text()` function is used to add the percent labels. #3. The `position_stack()` function assures that the percentages for each stack segment are placed properly. Finally, the plot and fill titles are specified, and a minimal theme is chosen without *x*-axis grid lines (they aren't needed).

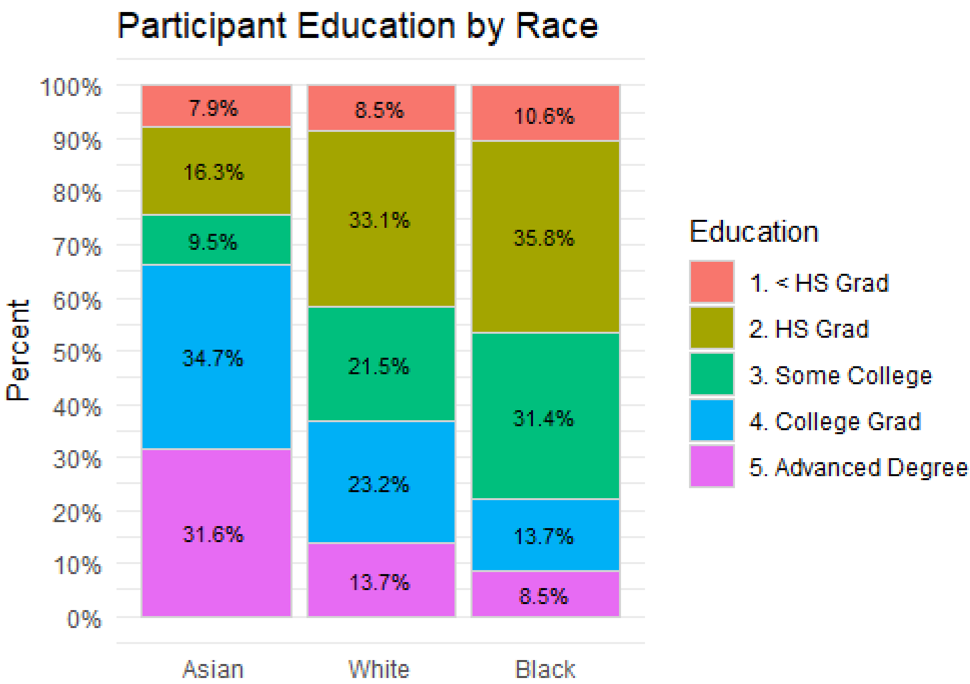

Figure 19.14 Stacked (filled) bar chart with percent labels.

HIGHLIGHTING DETAILS

A final example demonstrates the use of annotation to highlight information in a complex graph. The `gapminder` data frame in the `gapminder` package contains the average annual life expectancy for 142 countries recorded every 5 years from 1952 to 2002. Life expectancy in Cambodia differs greatly from other Asian countries in this data set. Let's create a graph that highlights these differences. The code is given in listing 19.14. The results are provided in figure 19.15.

Listing 19.14 Highlighting one trend among many

```
library(ggplot2)
library(dplyr)
library(gapminder)
plotdata <- gapminder %>%
  filter(continent == "Asia")                                      #1

plotdata$highlight <- ifelse(plotdata$country %in%                 #2
                             c("Cambodia"), "y", "n")
```

```
ggplot(plotdata, aes(x = year, y = lifeExp,                           #3
                     group = country,
                     size = highlight,
                     color = highlight)) +
  scale_color_manual(values=c("lightgrey", "red")) +
  scale_size_manual(values=c(.5, 1)) +
  geom_line() +
  geom_label(x=2000, y= 52, label="Cambodia",                          #4
             color="red", size=3) +
  labs(title="Life expectancy for Asian countries",
       x="Year",
       y="Life expectancy") +
  theme_minimal() +
  theme(legend.position="none",
        text=element_text(size=10))
```

#1 Subset Asian countries
#2 Create indicator variable for Cambodia
#3 Visually highlight Cambodia
#4 Add an annotation label

First, the data is subset to include only Asian countries #1. Next, a binary variable is created to indicate Cambodia vs. other countries #2. Life expectancy is plotted against year and separate line is plotted for each country #3. The Cambodian line is thicker than other countries and colored red. All other country lines are colored light grey. A label for the Cambodian line is added #4. Finally, plot and axis labels are specified, and a minimal theme is added. The legends (size, color) are suppressed (they aren't needed), and the base text size is decreased.

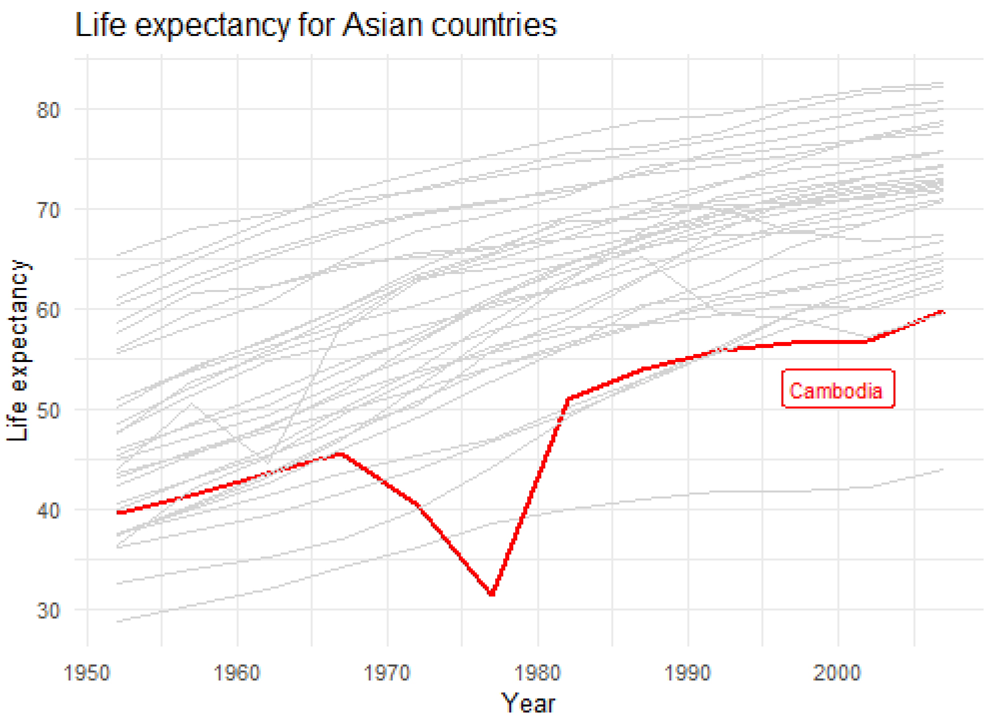

Figure 19.15. Average life expectancy trends for 33 Asian countries. The trend for Cambodia is highlighted. Although the trend is positive for each country, Cambodia had a sharp decline from 1967 to 1977.

Looking at the graph, it is clear average life expectancy has increased for each country. But the trend for Cambodia is quite different, with a major decline between 1967 and 1977. This is likely due to the genocide carried out by Pol Pot and the Khmer Rouge during that period.

19.4 Combining Graphs

Combining related `ggplot2` graphs into a single overall graph can often help emphasize relationships and differences. I used this when creating several plots in the text (see figure 19.7 for an example). The `patchwork` package provides a simple yet powerful language for combining plots. To use it, save each ggplot2 graph as an object. Then use the vertical bar (|) operator to combine graphs horizontally and the forward slash (/) operator to combine graphs vertically. You can use parentheses () to create subgroups of graphs. Figure 19.16 demonstrates various plot arrangements.

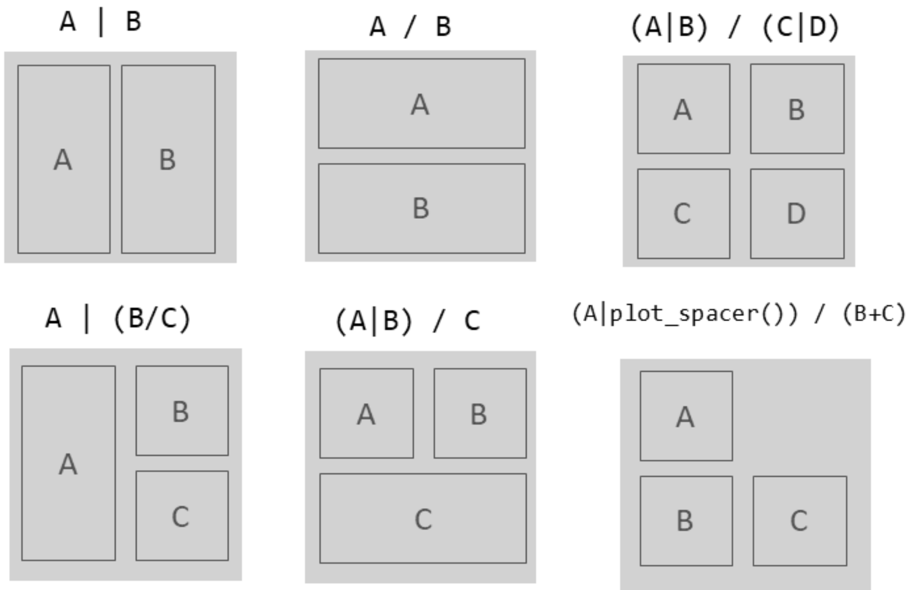

Figure 19.16 The patchwork package provides a simple set of arithmetic symbols for arranging multiple graphs in a single plot.

Let's create several `mpg`-related plots from the `mtcars` data frame and combine them into a single graph. The code is given in listing 19.15 and the graph is displayed in figure 19.17.

Listing 19.15 Combining graphs using the patchwork package

```
library(ggplot2)
library(patchwork)

p1 <- ggplot(mtcars, aes(disp, mpg)) +                                  #1
  geom_point() +
  labs(x="Engine displacement",
       y="Miles per gallon")

p2 <- ggplot(mtcars, aes(factor(cyl), mpg)) +
  geom_boxplot() +
  labs(x="Number of cylinders",
       y="Miles per gallon")

p3 <- ggplot(mtcars, aes(mpg)) +
  geom_histogram(bins=8, fill="darkgrey", color="white") +
  labs(x = "Miles per gallon",
       y = "Frequency")
```

```
(p1 | p2) / p3 +                                                    #2
  plot_annotation(title = 'Fuel Efficiency Data') &
  theme_minimal() +
  theme(axis.title = element_text(size=8),
        axis.text = element_text(size=8))
```
#1 Three graphs are created
#2 The graphs are combined into one plot

Three separate graphs are created and saved as p1, `p2`, and `p3` #1. The code `(p1| p2)/p3` indicates that the first two plots should be placed in the first row and the third plot should take up the entire second row #2.

The resulting graph is also a `ggplot2` graph and can be further edited. The `plot_annotation()` function adds a title to the combined graph (rather than to one of the subplots). Finally, the theme is modified. Note the use of the ampersand `(&)` to add theme elements. If you had used a plus (+) sign, the changes would only apply to the *last* subplot (`p3`). The `&` sign indicates that the theme functions should apply to each subplot (`p1`, `p2`, and `p3`).

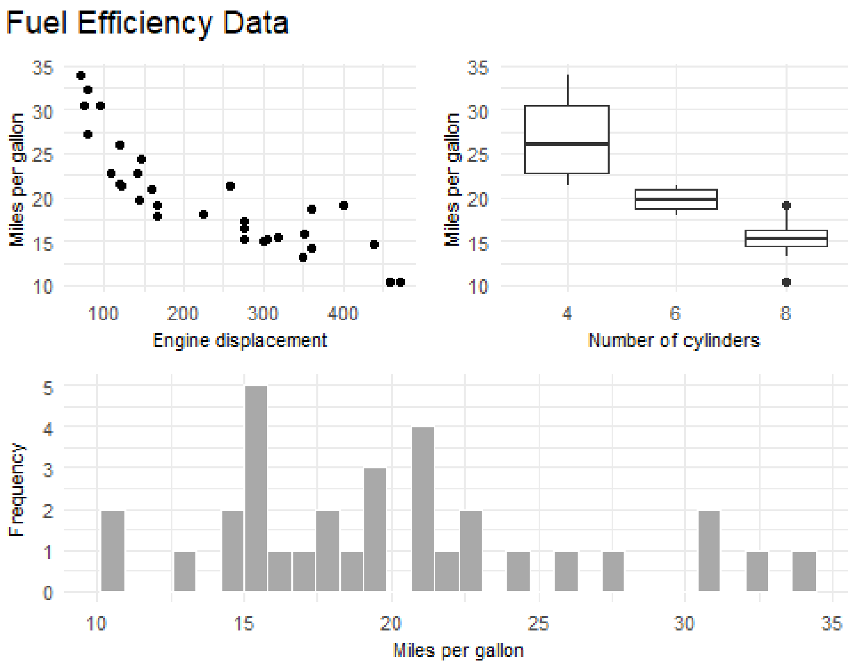

Figure 19.17. Three `ggplot2` plots are combined into one graph using the `patchwork` package.

The `patchwork` package has many additional options. To learn more, see the package reference site (https://patchwork.data-imaginist.com/).

19.5 Making graphs interactive

With few exceptions, the graphs in this book have been static images. There are several reasons to create *interactive graphs*. Interactive graphs allow you to focus on interesting results and call up additional information to understand patterns, trends, and unusual observations. Additionally, they're often more engaging than static graphs.

There are several packages in R that can be used to create interactive visualizations, including `leaflet`, `rbokeh`, `rCharts`, and `highlighter`, and `plotly`. In this section, we'll focus on the `plotly` package.

The *Plotly R Open Source Graphics Lib*rary (https://plotly.com/r/) can be used creating high end interactive visualizations. A key advantage is its ability to convert a static `ggplot2` graph into an interactive web graphic.

Creating an interactive graph using the `plotly` package is an easy two-step process. First, save a `ggplot2` graph as an object. Then pass the object to the `ggplotly()` function.

In listing 19.16, a scatterplot between miles per gallon and engine displacement is created using ggplot2. The points are colored to represent the number of engine cylinders. The plot is then passed to the `ggplotly()` function in the `plotly` package, producing an interactive web based visualization. A screenshot is provided in figure 19.18.

Listing 19.16 Converting a ggplot2 graph to an interactive plotly graph

```
library(ggplot2)
library(plotly)
mtcars$cyl <- factor(mtcars$cyl)
mtcars$name <- row.names(mtcars)

p <- ggplot(mtcars, aes(x = disp, y= mpg, color = cyl)) +
  geom_point()
ggplotly(p)
```

When you mouse over the graph, a toolbar will appear on the top right-hand side of the plot. The toolbar allows you to zoom, pan, select areas, download an image and more (see figure 19.19). Additionally, tooltips will pop up when the mouse cursor move over the plot area. By default, the tooltip displays the variable values use to create the graph (`disp`. `mpg`, and `cyl` in this example). Additionally, clicking on a key (symbol) in the legend will toggle that data on and off. This allows you to easily focus on subsets of the data.

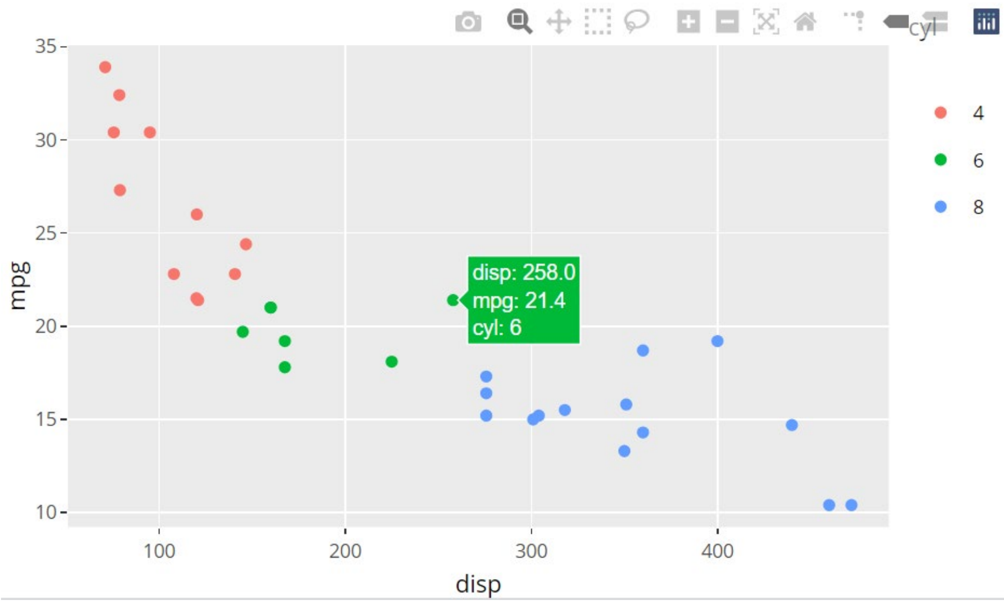

Figure 19.18 Screenshot of a plotly interactive web graphic created from a static ggplot2 graph.

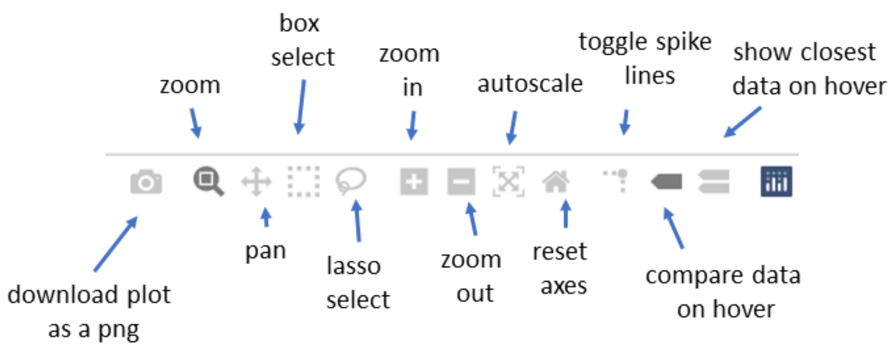

Figure 19.19. The plotly graph toolbar. The easiest way to understand these tools is to try them out one at a time.

There are two simple ways to customize the tooltip. You can add additional variables to the tooltip by including `label1 = var1, label2 = var2`, etc. in the `ggplot aes()` function. For example

```
p <- ggplot(mtcars, aes(x = disp, y= mpg, color = cyl,
                        label1 = gear, label2 = am))) +
  geom_point()
ggplotly(p)
```

will create a tooltip with `disp`, `mpg`, `cyl`, `gear`, and `am`.

Alternatively, you can use an undocumented `text` argument in the aes() function to build the tooltip from an arbitrary text string. An example is given in listing 19.17. A screenshot of the result is provided in figure 19.20.

Listing 19.17 Customizing the plotly tooltip

```
library(ggplot2)
library(plotly)
mtcars$cyl <- factor(mtcars$cyl)
mtcars$name <- row.names(mtcars)

p <- ggplot(mtcars,
            aes(x = disp, y=mpg, color=cyl,
                text = paste(name, "\n",
                             "mpg:", mpg, "\n",
                             "disp:", disp, "\n",
                             "cyl:", cyl, "\n",
                             "gear:", gear))) +
  geom_point()
ggplotly(p, tooltip=c("text"))
```

The `text` approach gives you great control over the tooltip. You can even include HTML markup in the text string, allowing you customize the text output further.

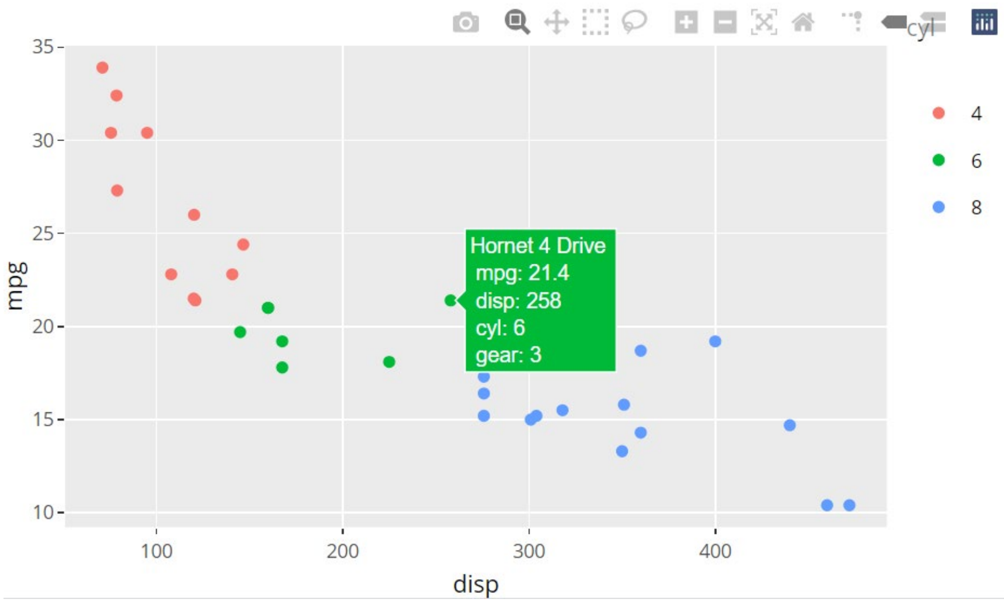

Figure 19.20. Screenshot of an interactive plotly graph with a custom tooltip created from within the ggplot2 code.

This chapter has covered a variety of methods for customizing ggplot2 graphs. Remember that the goal of customization is to enhance your insight into the data and improve communication of those insights to others. Anything added to a graph that detracts from these goals is mere decoration (also known as chart junk). Always try to avoid chart junk! See the *Data Visualization Standards* website (https://xdgov.github.io/data-design-standards/) for additional recommendations.

19.6 Summary

- The `ggplot2` scale functions map variable values to visual aspects of a graph. They are particularly useful for customizing the axes and color palette.
- The `ggplot2` theme() function controls non-data elements of the graph. It is useful for customizing the appearance of fonts, legends, grid lines, and the plot background.
- The `ggplot2 geom` functions are useful for annotating a graph – adding useful information such as reference lines and labels.
- Two or more graphs can be combined into a single graph using the `patchwork` package.
- Almost any `ggplot2` graph can be converted from a static image to an interactive web graphic using the `plotly` package.