R basics and data visualization

NOMA intervention: Nightingale NMR metabolomics analyses

Jacob J. Christensen

15 desember, 2019

Contents

Info

In this file, I will go through a few basic R commands and showcase general-use data visualizations.

This whole document is taken from the Data visualization chapter in the R4DS book, by Hadley Wickham. The code was fetched from Hadleys GitHub repo r4ds/visualize.Rmd.

Introduction

> "The simple graph has brought more information to the data analyst's mind than any other device." — John Tukey

This chapter will teach you how to visualise your data using ggplot2. R has several systems for making graphs, but ggplot2 is one of the most elegant and most versatile. ggplot2 implements the **grammar of graphics**, a coherent system for describing and building graphs. With ggplot2, you can do more faster by learning one system and applying it in many places.

Prerequisites

This chapter focusses on ggplot2, one of the core members of the tidyverse. To access the help pages and functions that we will use in this chapter, load the tidyverse by running this code:

```
library(tidyverse)
```

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

If you run this code and get the error message "there is no package called 'tidyverse' ", you'll need to first install it, then run `library()` once again.

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

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

If we need to be explicit about where a function (or dataset) comes from, we'll use the special form `package::function()`. For example, `ggplot2::ggplot()` tells you explicitly that we're using the `ggplot()` function from the ggplot2 package.

First steps

Let's use our first graph to answer a question: is there an inverse association between triglycerides and HDL-C at baseline in the NOMA study? You probably already have an answer, but try to make your answer precise. What does the relationship between triglycerides and HDL-C look like? Is it positive? Negative? Linear? Nonlinear?

The `data_noma` data frame

You can test your answer with the **`data_noma` data frame**. A data frame is a rectangular collection of variables (in the columns) and observations (in the rows).

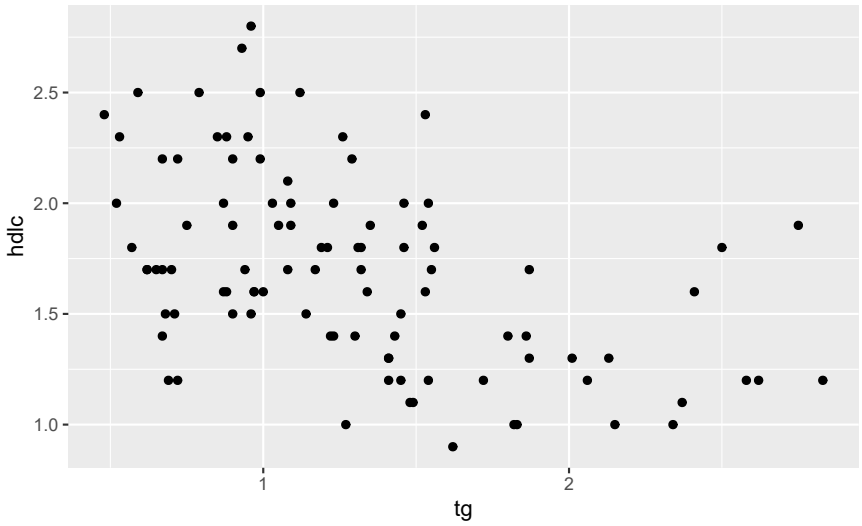
Among the variables in `data_noma` are:

1. `tg`, plasma triglyceride concentration, in mmol/L.
2. `hdlc`, plasma HDL-C concentration, in mmol/L.

Creating a ggplot

To plot `data_noma`, run this code to put `tg` on the x-axis and `hdlc` on the y-axis:

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc))
```



The plot shows a negative relationship between triglycerides (`tg`) and HDL-C (`hdlc`). In other words, persons with higher triglycerides have lower HDL-C. Does this confirm or refute your hypothesis about HDL-C and triglycerides?

With ggplot2, you begin a plot with the function `ggplot()`. `ggplot()` creates a coordinate system that you can add layers to. The first argument of `ggplot()` is the dataset to use in the graph. So `ggplot(data = data_noma)` creates an empty graph, but it's not very interesting so I'm not going to show it here.

You complete your graph by adding one or more layers to `ggplot()`. The function `geom_point()` adds a layer of points to your plot, which creates a scatterplot. ggplot2 comes with many geom functions that each add a different type of layer to a plot. You'll learn a whole bunch of them throughout this chapter.

Each geom function in ggplot2 takes a `mapping` argument. This defines how variables in your dataset are mapped to visual properties. The `mapping` argument is always paired with `aes()`, and the `x` and `y` arguments of `aes()` specify which variables to map to the x and y axes. ggplot2 looks for the mapped variables in the `data` argument, in this case, `data_noma`.

A graphing template

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

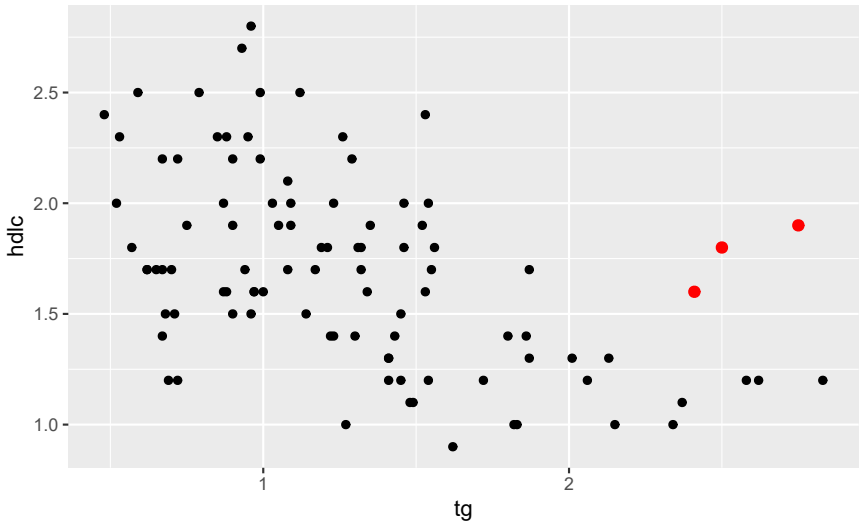
```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

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

Aesthetic mappings

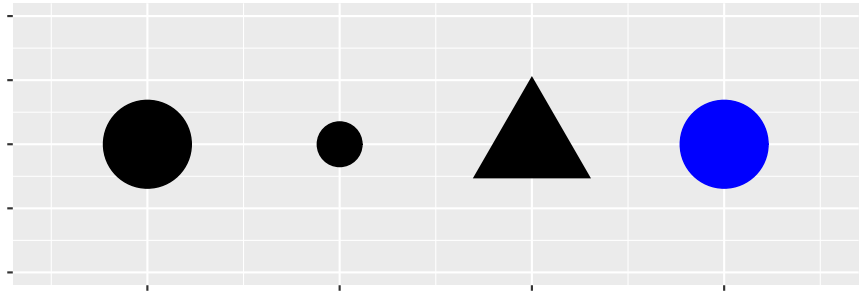
> "The greatest value of a picture is when it forces us to notice what we never expected to see." — John Tukey

In the plot below, one group of points (highlighted in red) seems to fall outside of the linear trend. These persons have a higher HDL-C than you might expect. How can you explain these persons?



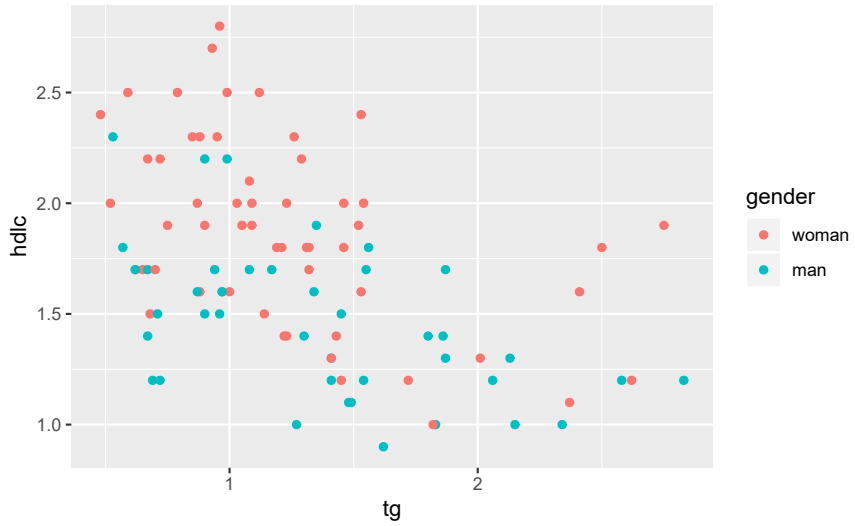
Let's hypothesize that the individuals are more healthy than "expected". One way to test this hypothesis is to look at the `gender` variable. Women, on average, have higher HDL-C than men.

You can add a third variable, like `gender`, to a two dimensional scatterplot by mapping it to an **aesthetic**. An aesthetic is a visual property of the objects in your plot. Aesthetics include things like the size, the shape, or the color of your points. You can display a point (like the one below) in different ways by changing the values of its aesthetic properties. Since we already use the word "value" to describe data, let's use the word "level" to describe aesthetic properties. Here we change the levels of a point's size, shape, and color to make the point small, triangular, or blue:



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

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc, color = gender))
```



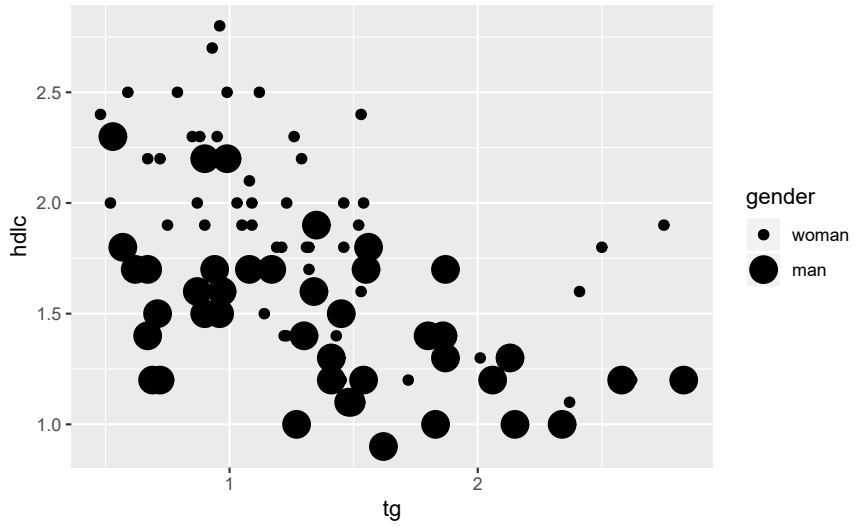
(If you prefer British English, like Hadley Wickham, you can use `colour` instead of `color`.)

To map an aesthetic to a variable, associate the name of the aesthetic to the name of the variable inside `aes()`. ggplot2 will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable, a process known as **scaling**. ggplot2 will also add a legend that explains which levels correspond to which values.

The colors reveal that the unusual points are women.

In the above example, we mapped `gender` to the color aesthetic, but we could have mapped `gender` to the size aesthetic in the same way. In this case, the exact size of each point would reveal its gender affiliation. In this case, the ordering of genders is completely arbitrary; however, sometimes (for example when performing a linear regression analysis), we might want to explicitly compare women to men, and not opposite. Then we would adjust the ordering of the variable accordingly.

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc, size = gender))
```



Or we could have mapped `gender` to the *alpha* aesthetic, which controls the transparency of the points, or to the shape aesthetic, which controls the shape of the points.

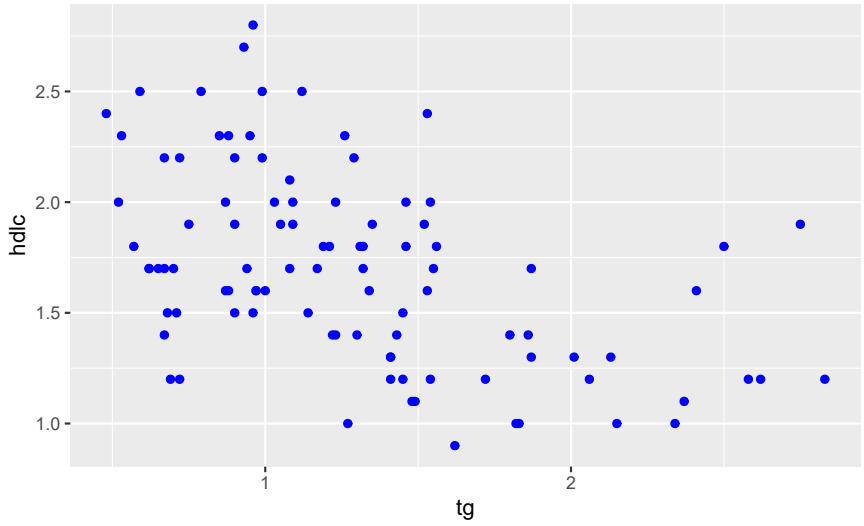

```
# Left
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc, alpha = gender))

# Right
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc, shape = gender))
```

For each aesthetic, you use `aes()` to associate the name of the aesthetic with a variable to display. The `aes()` function gathers together each of the aesthetic mappings used by a layer and passes them to the layer's mapping argument. The syntax highlights a useful insight about `x` and `y`: the x and y locations of a point are themselves aesthetics, visual properties that you can map to variables to display information about the data.

Once you map an aesthetic, ggplot2 takes care of the rest. It selects a reasonable scale to use with the aesthetic, and it constructs a legend that explains the mapping between levels and values. For x and y aesthetics, ggplot2 does not create a legend, but it creates an axis line with tick marks and a label. The axis line acts as a legend; it explains the mapping between locations and values.

You can also *set* the aesthetic properties of your geom manually. For example, we can make all of the points in our plot blue:

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc), color = "blue")
```



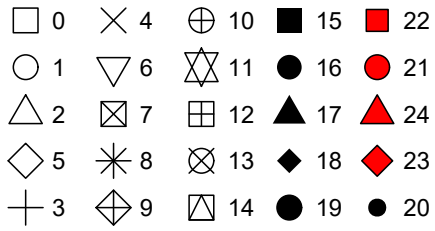


Figure 1: R has 25 built in shapes that are identified by numbers. There are some seeming duplicates: for example, 0, 15, and 22 are all squares. The difference comes from the interaction of the 'colour' and 'fill' aesthetics. The hollow shapes (0–14) have a border determined by 'colour'; the solid shapes (15–18) are filled with 'colour'; the filled shapes (21–24) have a border of 'colour' and are filled with 'fill'.

Here, the color doesn't convey information about a variable, but only changes the appearance of the plot. To set an aesthetic manually, set the aesthetic by name as an argument of your geom function; i.e. it goes *outside* of `aes()`. You'll need to pick a level that makes sense for that aesthetic:

- The name of a color as a character string.
- The size of a point in mm.
- The shape of a point as a number.

Common problems

As you start to run R code, you're likely to run into problems. Don't worry — it happens to everyone. Hadley Wickham has been writing R code for years, and every day he still write code that doesn't work!

Start by carefully comparing the code that you're running to the code in the book. R is extremely picky, and a misplaced character can make all the difference. Make sure that every `(` is matched with a `)` and every `"` is paired with another `"`. Sometimes you'll run the code and nothing happens. Check the left-hand of your console: if it's a `+`, it means that R doesn't think you've typed a complete expression and it's waiting for you to finish it. In this case, it's usually easy to start from scratch again by pressing ESCAPE to abort processing the current command.

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

```
ggplot(data = data_noma)
+ geom_point(mapping = aes(x = tg, y = hdlc))
```

If you're still stuck, try the help. You can get help about any R function by running `?function_name` in the console, or selecting the function name and pressing F1 in RStudio. Don't worry if the help doesn't seem that helpful - instead skip down to the examples and look for code that matches what you're trying to do.

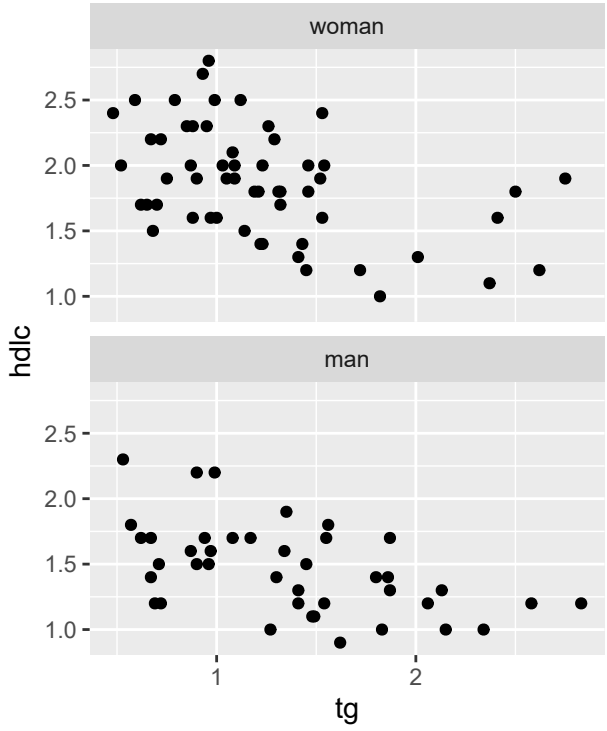
If that doesn't help, carefully read the error message. Sometimes the answer will be buried there! But when you're new to R, the answer might be in the error message but you don't yet know how to understand it. Another great tool is Google: try googling the error message, as it's likely someone else has had the same problem, and has gotten help online.

Facets

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

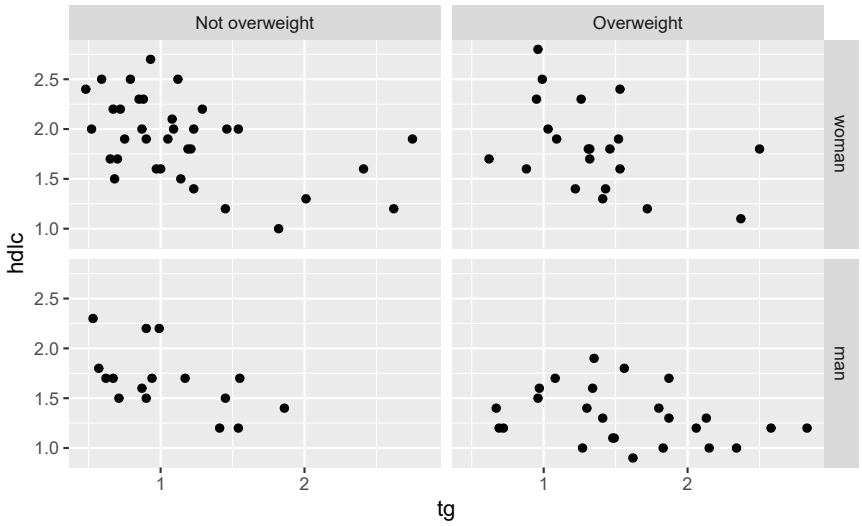
To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` should be a formula, which you create with ~ followed by a variable name (here "formula" is the name of a data structure in R, not a synonym for "equation"). The variable that you pass to `facet_wrap()`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

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc)) +
  facet_wrap(~ gender, nrow = 2)
```



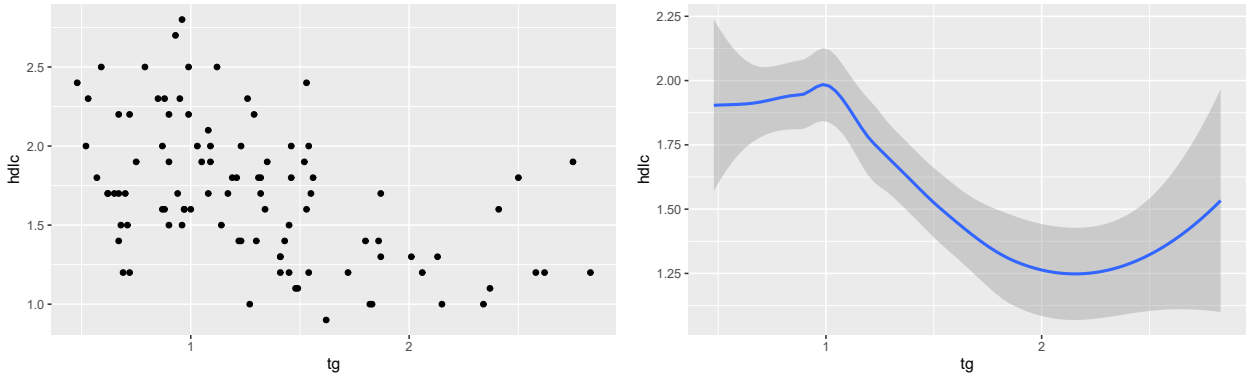
To facet your plot on the combination of two variables, add `facet_grid()` to your plot call. The arguments to `facet_grid()` are either `rows` or `cols`, or both. Supply variable names inside the `vars` selector.

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc)) +
  facet_grid(rows = vars(gender), cols = vars(overweight))
```



Geometric objects

How are these two plots similar?



Both plots contain the same x variable, the same y variable, and both describe the same data. But the plots are not identical. Each plot uses a different visual object to represent the data. In ggplot2 syntax, we say that they use different **geoms**.

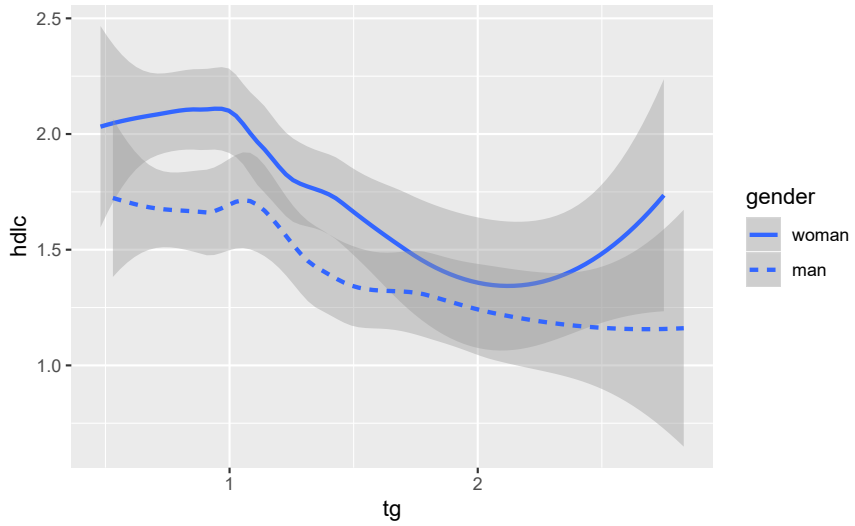
A **geom** is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on. Scatterplots break the trend; they use the point geom. As we see above, you can use different geoms to plot the same data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

To change the geom in your plot, change the geom function that you add to `ggplot()`. For instance, to make the plots above, you can use this code:

```
# left
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc))

# right
ggplot(data = data_noma) +
  geom_smooth(mapping = aes(x = tg, y = hdlc))
```

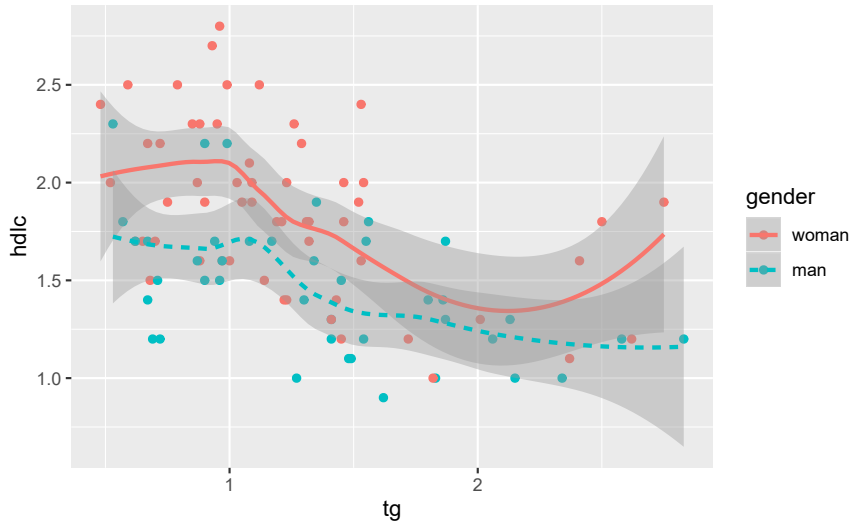
Every geom function in ggplot2 takes a `mapping` argument. However, not every aesthetic works with every geom. You could set the shape of a point, but you couldn't set the "shape" of a line. On the other hand, you *could* set the linetype of a line. `geom_smooth()` will draw a different line, with a different linetype, for each unique value of the variable that you map to linetype.

```
ggplot(data = data_noma) +
  geom_smooth(mapping = aes(x = tg, y = hdlc, linetype = gender))
```



Here `geom_smooth()` separates the persons into two lines based on their `gender` value. One line describes all of the points with a `woman` value, and one line describes all of the points with an `man` value.

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

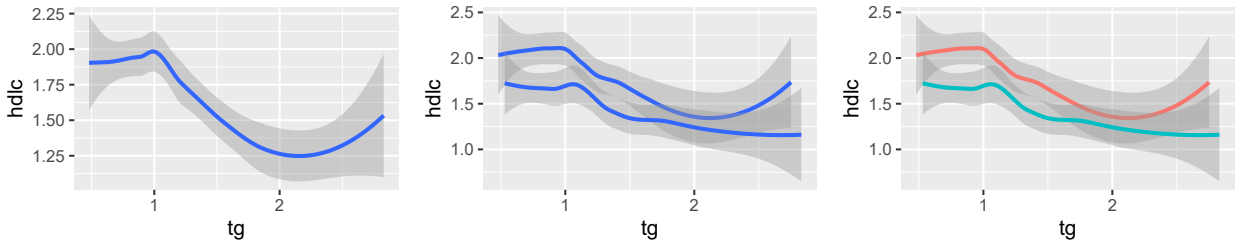


Notice that this plot contains two geoms in the same graph! If this makes you excited, buckle up. We will learn how to place multiple geoms in the same plot very soon.

ggplot2 provides over 30 geoms, and extension packages provide even more (see https://www.ggplot2-exts.org for a sampling). The best way to get a comprehensive overview is the ggplot2 cheatsheet, which you can find at http://rstudio.com/cheatsheets. To learn more about any single geom, use help: `?geom_smooth`.

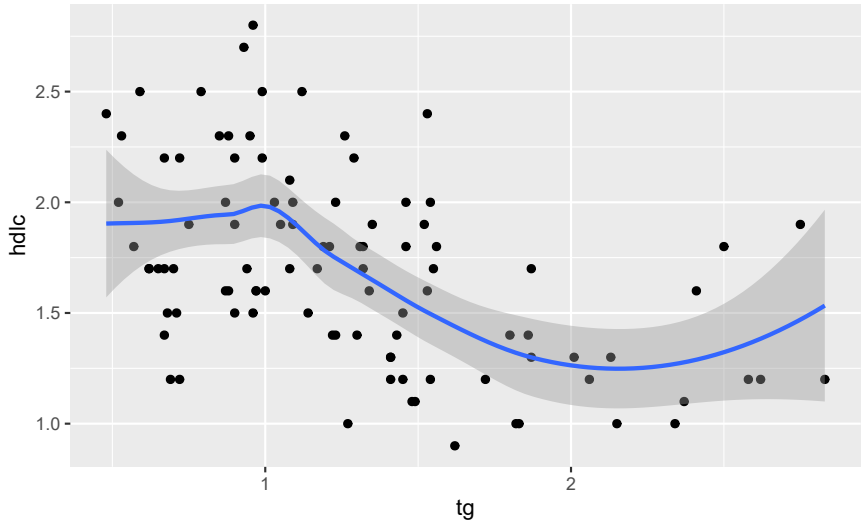
Many geoms, like `geom_smooth()`, use a single geometric object to displays multiple rows of data. For these geoms, you can set the `group` aesthetic to a categorical variable to draw multiple objects. ggplot2 will draw a separate object for each unique value of the grouping variable. In practice, ggplot2 will automatically group the data for these geoms whenever you map an aesthetic to a discrete variable (as in the `linetype` example). It is convenient to rely on this feature because the group aesthetic by itself does not add a legend or distinguishing features to the geoms.

```
ggplot(data = data_noma) +
  geom_smooth(mapping = aes(x = tg, y = hdlc))

ggplot(data = data_noma) +
  geom_smooth(mapping = aes(x = tg, y = hdlc, group = gender))

ggplot(data = data_noma) +
  geom_smooth(
    mapping = aes(x = tg, y = hdlc, color = gender),
    show.legend = FALSE
  )
```



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

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc)) +
  geom_smooth(mapping = aes(x = tg, y = hdlc))
```

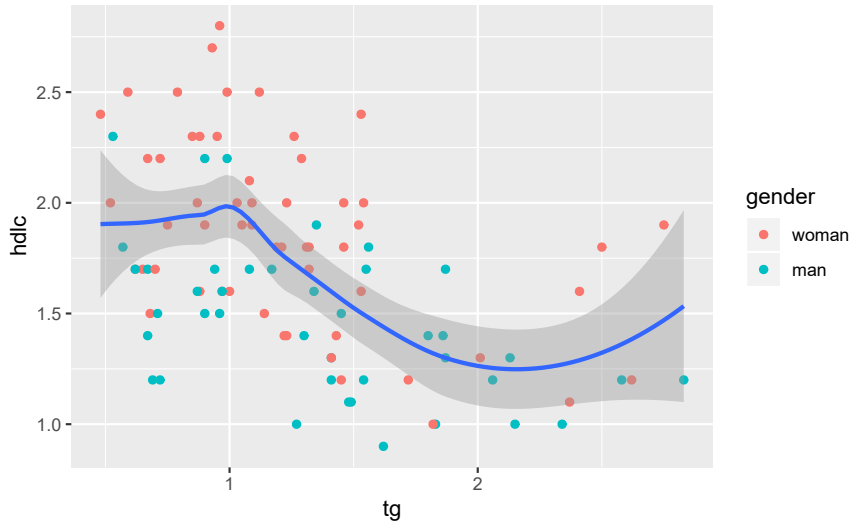


This, however, introduces some duplication in our code. Imagine if you wanted to change the y-axis to display `apoa1` instead of `hdlc`. You'd need to change the variable in two places, and you might forget to update one. You can avoid this type of repetition by passing a set of mappings to `ggplot()`. ggplot2 will treat these mappings as global mappings that apply to each geom in the graph. In other words, this code will produce the same plot as the previous code:

```
ggplot(data = data_noma, mapping = aes(x = tg, y = hdlc)) +
  geom_point() +
  geom_smooth()
```

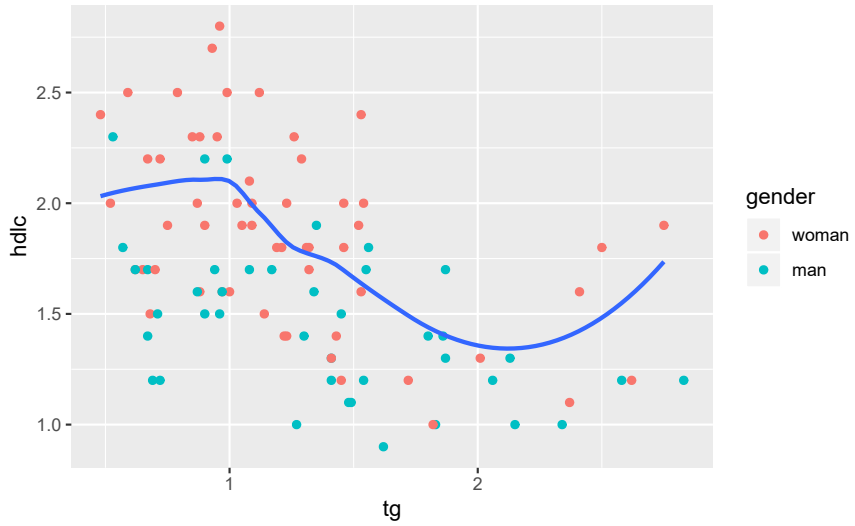
If you place mappings in a geom function, ggplot2 will treat them as local mappings for the layer. It will use these mappings to extend or overwrite the global mappings *for that layer only*. This makes it possible to display different aesthetics in different layers.

```
ggplot(data = data_noma, mapping = aes(x = tg, y = hdlc)) +
  geom_point(mapping = aes(color = gender)) +
  geom_smooth()
```



You can use the same idea to specify different `data` for each layer. Here, our smooth line displays just a subset of the `data_noma` dataset, the subcompact cars. The local data argument in `geom_smooth()` overrides the global data argument in `ggplot()` for that layer only.

```
ggplot(data = data_noma, mapping = aes(x = tg, y = hdlc)) +
  geom_point(mapping = aes(color = gender)) +
  geom_smooth(data = filter(data_noma, gender == "woman"), se = FALSE)
```

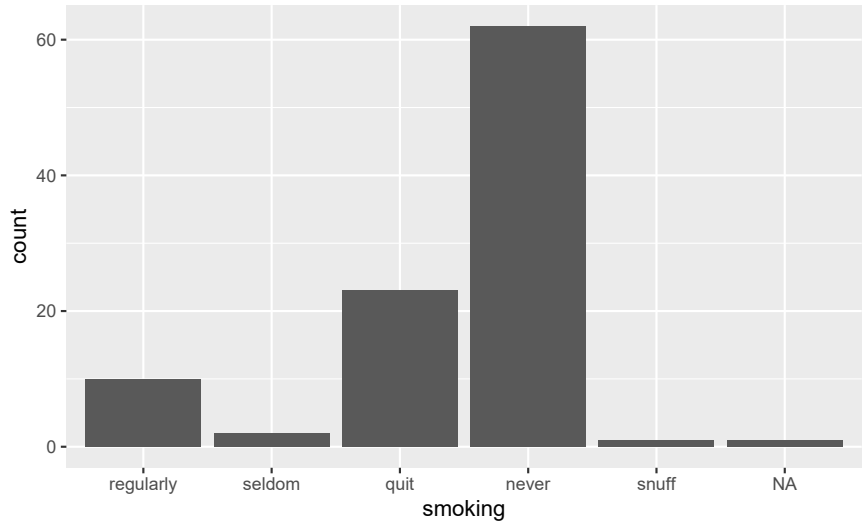


(You'll learn how `filter()` works in the chapter on data transformations: for now, just know that this command selects only women.)

Statistical transformations

Next, let's take a look at a bar chart. Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with `geom_bar()`. The following chart displays the number of persons in the `data_noma` dataset, grouped by `smoking`.

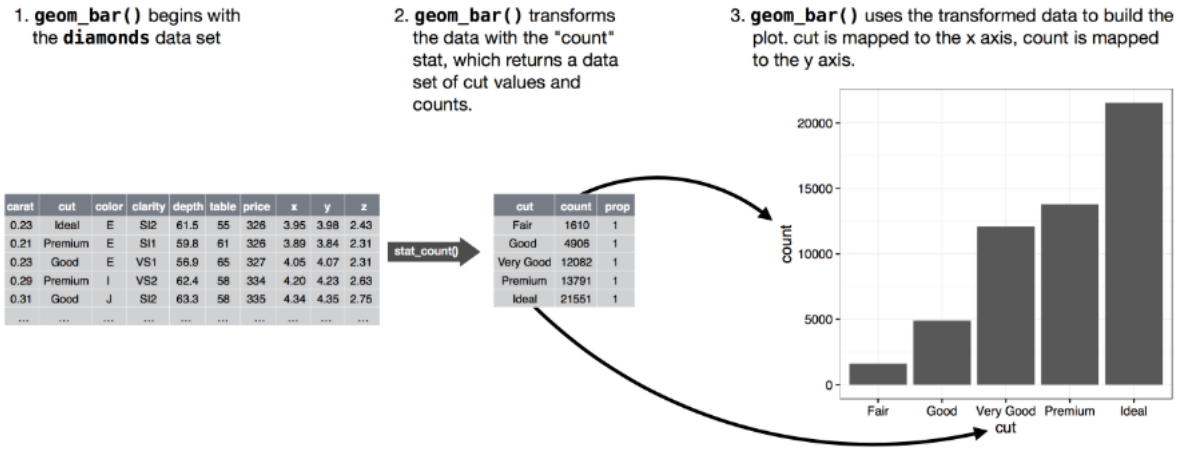
```
ggplot(data = data_noma) +
  geom_bar(mapping = aes(x = smoking))
```



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

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

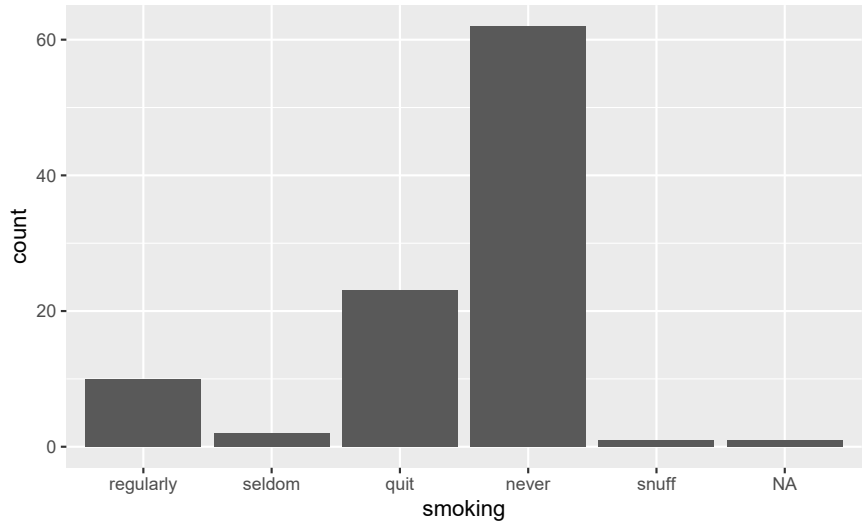
The algorithm used to calculate new values for a graph is called a **stat**, short for statistical transformation. The figure below describes how this process works with `geom_bar()`.



You can learn which stat a geom uses by inspecting the default value for the `stat` argument. For example, `?geom_bar` shows that the default value for `stat` is "count", which means that `geom_bar()` uses `stat_count()`. `stat_count()` is documented on the same page as `geom_bar()`, and if you scroll down you can find a section called "Computed variables". That describes how it computes two new variables: `count` and `prop`.

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

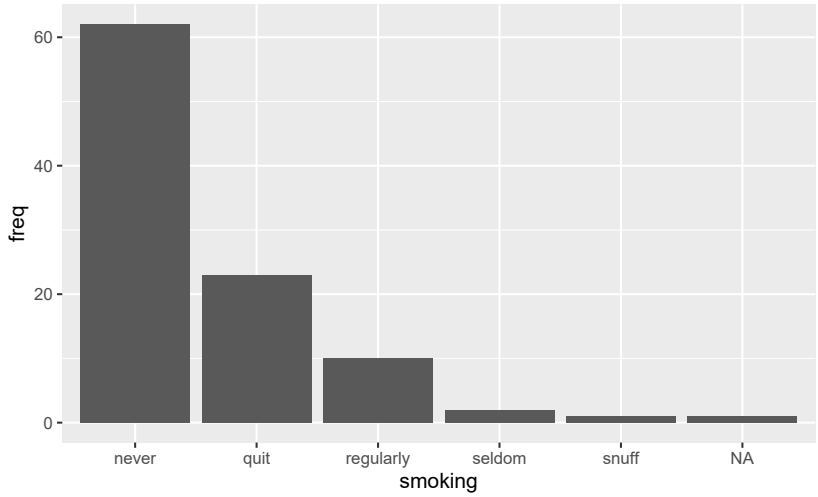
```
ggplot(data = data_noma) +
  stat_count(mapping = aes(x = smoking))
```



This works because every geom has a default stat; and every stat has a default geom. This means that you can typically use geoms without worrying about the underlying statistical transformation. There are three reasons you might need to use a stat explicitly:

1. You might want to override the default stat. In the code below, I change the stat of `geom_bar()` from count (the default) to identity. This lets me map the height of the bars to the raw values of a y variable. Unfortunately when people talk about bar charts casually, they might be referring to this type of bar chart, where the height of the bar is already present in the data, or the previous bar chart where the height of the bar is generated by counting rows.

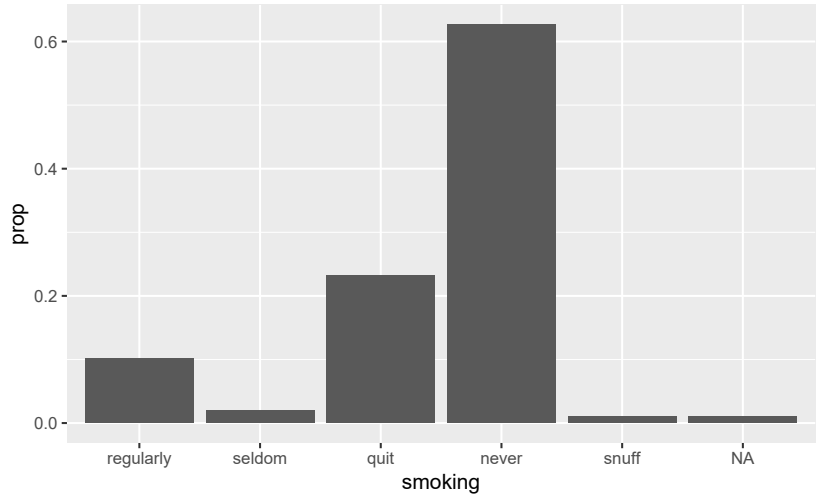
```
demo <- tribble(
  ~smoking, ~freq,
  "regularly", 10,
  "seldom", 2,
  "quit", 23,
  "never", 62,
  "snuff", 1,
  NA, 1
)

ggplot(data = demo) +
  geom_bar(mapping = aes(x = smoking, y = freq), stat = "identity")
```



(Don't worry that you haven't seen `<-` or `tribble()` before. You might be able to guess at their meaning from the context, and you'll learn exactly what they do soon!)

2. You might want to override the default mapping from transformed variables to aesthetics. For example, you might want to display a bar chart of proportion, rather than count:

```
ggplot(data = data_noma) +
  geom_bar(mapping = aes(x = smoking, y = ..prop.., group = 1))
```

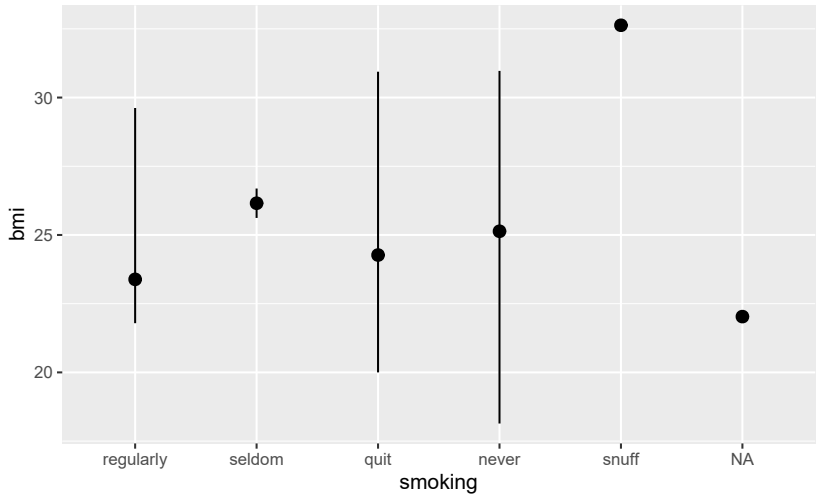


To find the variables computed by the stat, look for the help section titled "computed variables".

3. You might want to draw greater attention to the statistical transformation in your code. For example, you might use `stat_summary()`, which summarises the y values for each unique x value, to draw attention to the summary that you're computing:

```
ggplot(data = data_noma) +
  stat_summary(
    mapping = aes(x = smoking, y = bmi),
    fun.ymin = min,
    fun.ymax = max,
```

```
    fun.y = median
  )
```

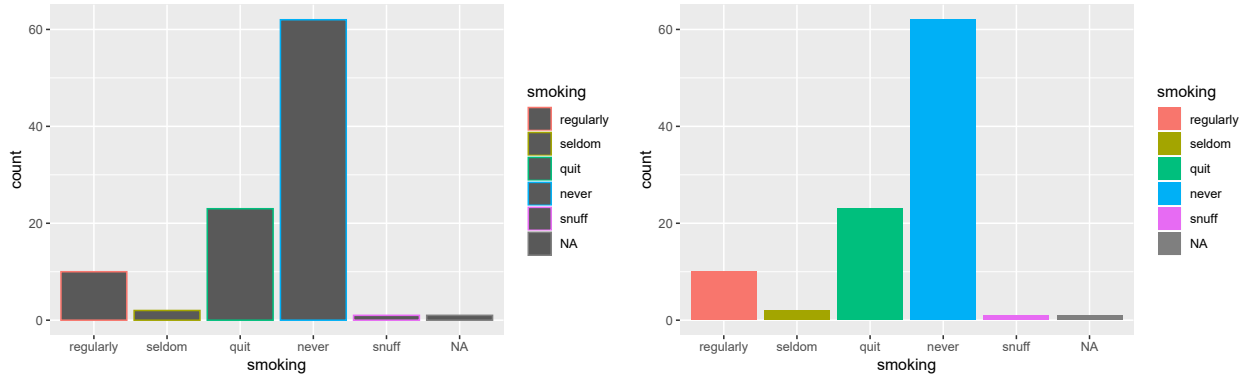


ggplot2 provides over 20 stats for you to use. Each stat is a function, so you can get help in the usual way, e.g. `?stat_bin`. To see a complete list of stats, try the ggplot2 cheatsheet.

Position adjustments

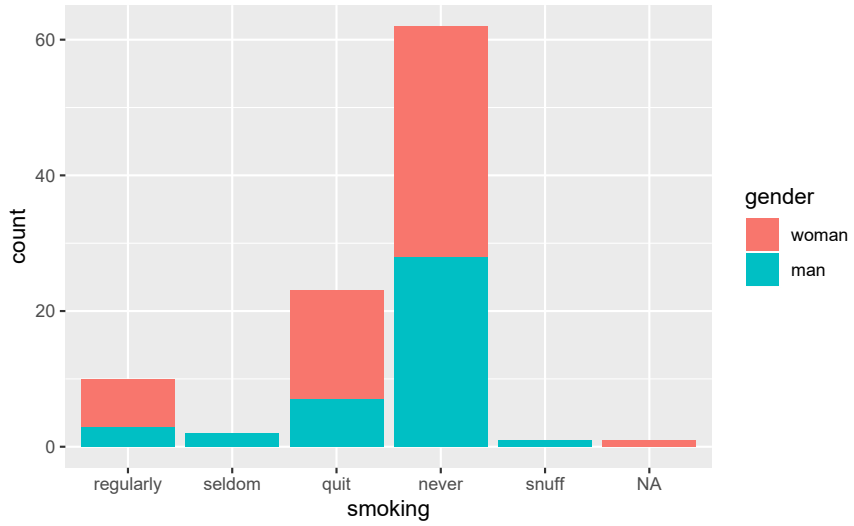
There's one more piece of magic associated with bar charts. You can colour a bar chart using either the `colour` aesthetic, or, more usefully, `fill`:

```
ggplot(data = data_noma) +
  geom_bar(mapping = aes(x = smoking, colour = smoking))
ggplot(data = data_noma) +
  geom_bar(mapping = aes(x = smoking, fill = smoking))
```



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

```
ggplot(data = data_noma) +
  geom_bar(mapping = aes(x = smoking, fill = gender))
```



The stacking is performed automatically by the **position adjustment** specified by the `position` argument. If you don't want a stacked bar chart, you can use one of three other options: `"identity"`, `"dodge"` or `"fill"`.

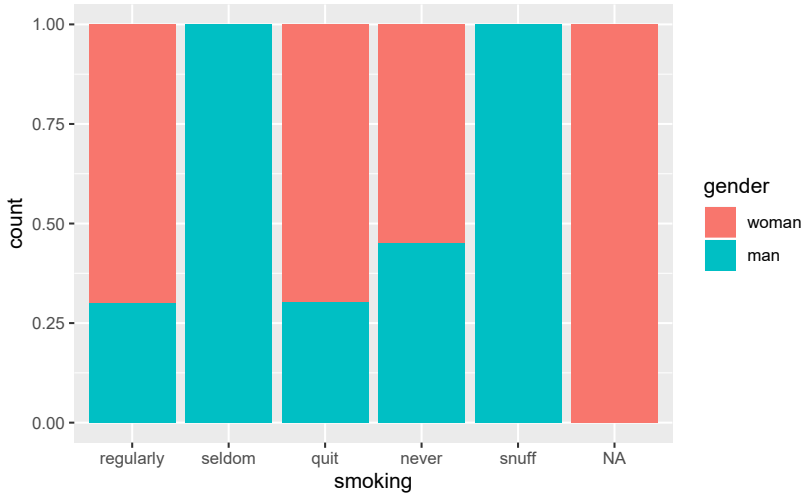
- `position = "identity"` will place each object exactly where it falls in the context of the graph. This is not very useful for bars, because it overlaps them. To see that overlapping we either need to make the bars slightly transparent by setting `alpha` to a small value, or completely transparent by setting `fill = NA`.

```
ggplot(data = data_noma, mapping = aes(x = smoking, fill = gender)) +
  geom_bar(alpha = 1/5, position = "identity")
ggplot(data = data_noma, mapping = aes(x = smoking, colour = gender)) +
  geom_bar(fill = NA, position = "identity")
```

The identity position adjustment is more useful for 2d geoms, like points, where it is the default.

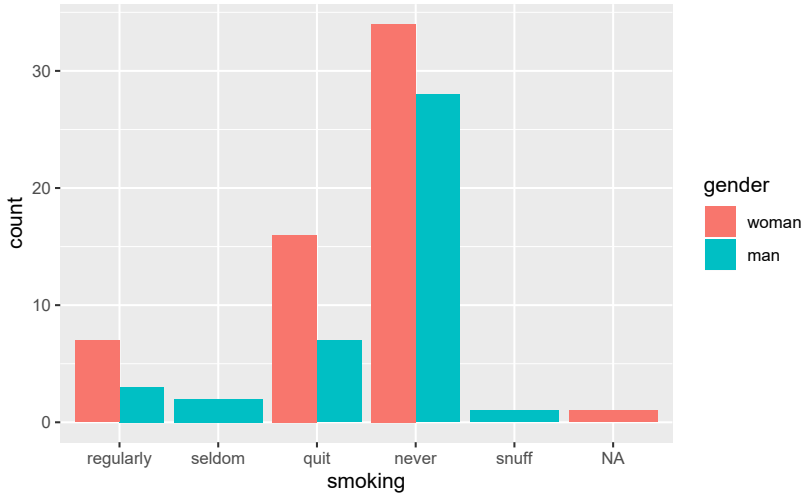
- `position = "fill"` works like stacking, but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.

```
ggplot(data = data_noma) +
  geom_bar(mapping = aes(x = smoking, fill = gender), position = "fill")
```

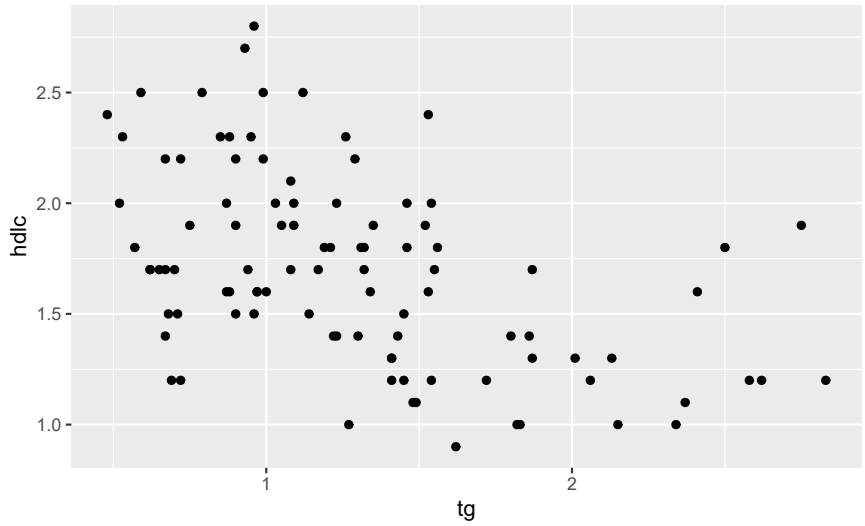


- `position = "dodge"` places overlapping objects directly *beside* one another. This makes it easier to compare individual values.

```
ggplot(data = data_noma) +
  geom_bar(mapping = aes(x = smoking, fill = gender), position = "dodge")
```



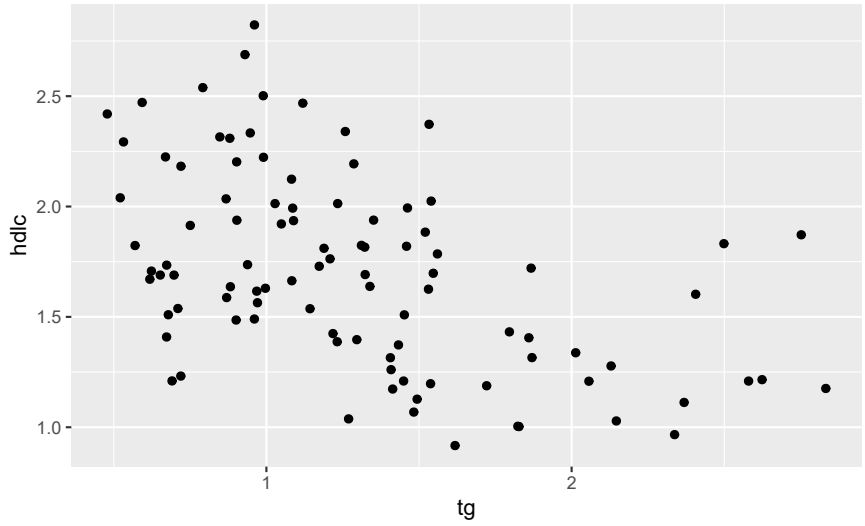
There's one other type of adjustment that's not useful for bar charts, but it can be very useful for scatterplots. Recall our first scatterplot. Did you notice that the plot displays fewer points than rows in the dataset?



The values of (especially) `hdlc` are rounded so the points appear on a grid and many points overlap at/near each other. This problem is known as **overplotting**. This arrangement makes it hard to see where the mass of the data is. Are the data points spread equally throughout the graph, or is there one special combination of `hdlc` and `tg` that contains a large number of values?

You can avoid this gridding by setting the position adjustment to "jitter". `position = "jitter"` adds a small amount of random noise to each point. This spreads the points out because no two points are likely to receive the same amount of random noise.

```
ggplot(data = data_noma) +
  geom_point(mapping = aes(x = tg, y = hdlc), position = "jitter")
```



Adding randomness seems like a strange way to improve your plot, but while it makes your graph less accurate at small scales, it makes your graph *more* revealing at large scales. Because this is such a useful operation, ggplot2 comes with a shorthand for `geom_point(position = "jitter")`: `geom_jitter()`.

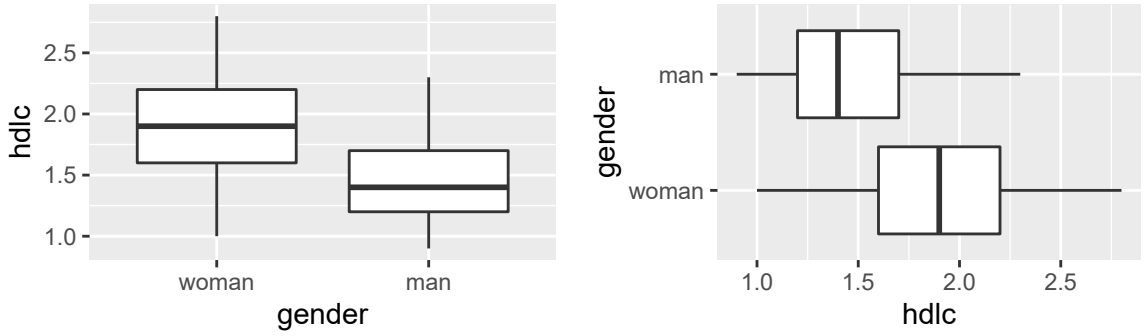
To learn more about a position adjustment, look up the help page associated with each adjustment: `?position_dodge`, `?position_fill`, `?position_identity`, `?position_jitter`, and `?position_stack`.

Coordinate systems

Coordinate systems are probably the most complicated part of ggplot2. The default coordinate system is the Cartesian coordinate system where the x and y positions act independently to determine the location of each point. There are a number of other coordinate systems that are occasionally helpful.

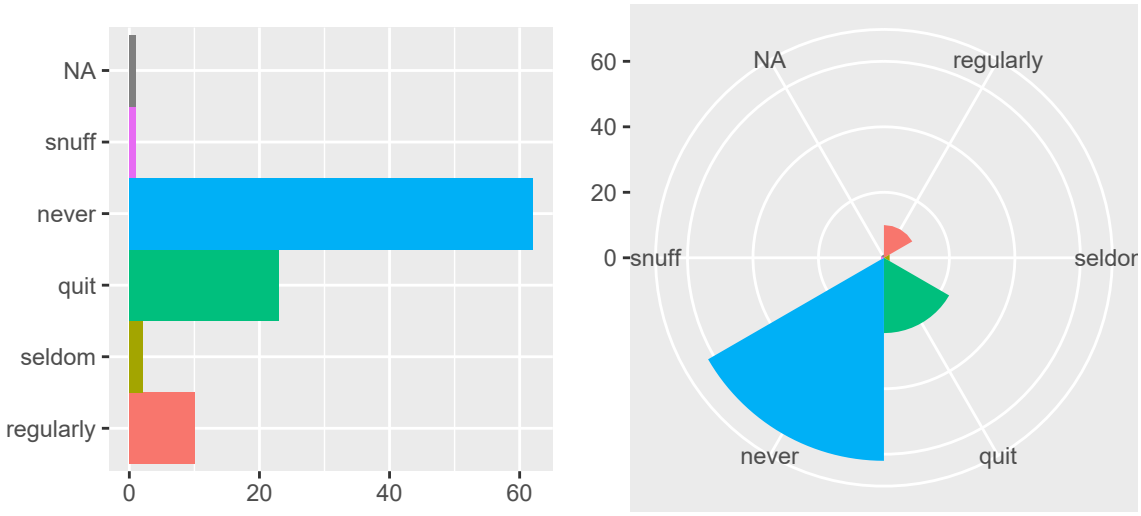
- `coord_flip()` switches the x and y axes. This is useful (for example), if you want horizontal boxplots. It's also useful for long labels: it's hard to get them to fit without overlapping on the x-axis.

```
ggplot(data = data_noma, mapping = aes(x = gender, y = hdlc)) +
  geom_boxplot()
ggplot(data = data_noma, mapping = aes(x = gender, y = hdlc)) +
  geom_boxplot() +
  coord_flip()
```



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

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

bar + coord_flip()
bar + coord_polar()
```



The layered grammar of graphics

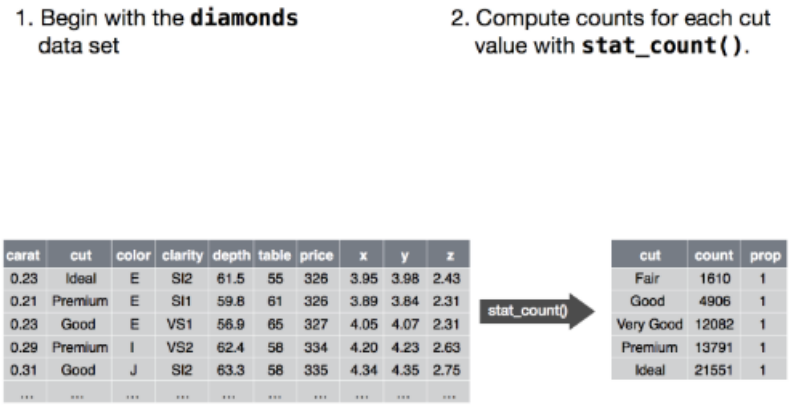
In the previous sections, you learned much more than how to make scatterplots, bar charts, and boxplots. You learned a foundation that you can use to make *any* type of plot with ggplot2. To see this, let's add position adjustments, stats, coordinate systems, and faceting to our code template:

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

Our new template takes seven parameters, the bracketed words that appear in the template. In practice, you rarely need to supply all seven parameters to make a graph because ggplot2 will provide useful defaults for everything except the data, the mappings, and the geom function.

The seven parameters in the template compose the grammar of graphics, a formal system for building plots. The grammar of graphics is based on the insight that you can uniquely describe *any* plot as a combination of a dataset, a geom, a set of mappings, a stat, a position adjustment, a coordinate system, and a faceting scheme.

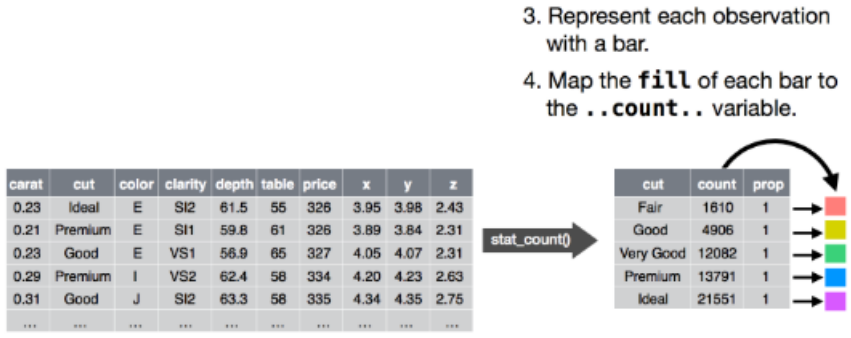
To see how this works, consider how you could build a basic plot from scratch: you could start with a dataset and then transform it into the information that you want to display (with a stat).



carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

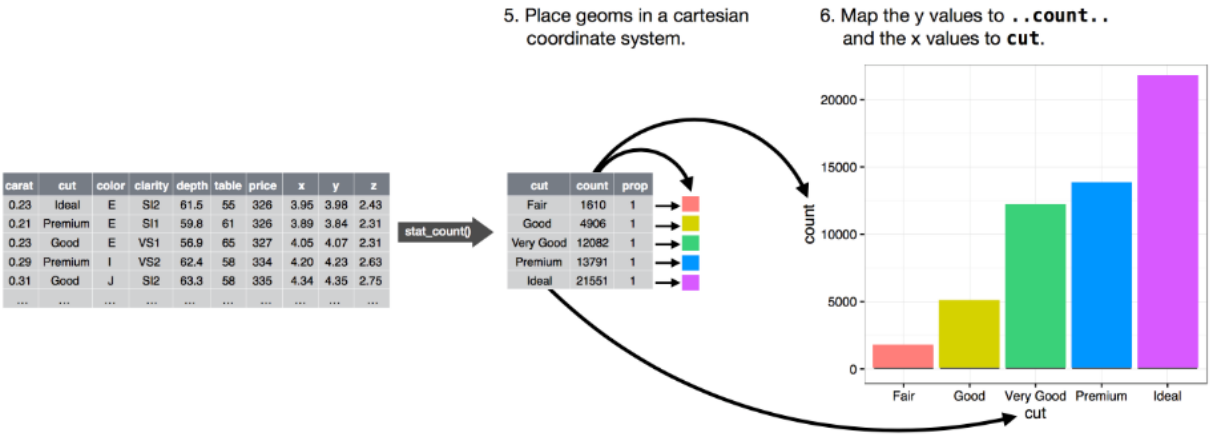
Next, you could choose a geometric object to represent each observation in the transformed data. You could then use the aesthetic properties of the geoms to represent variables in the data. You would map the values of each variable to the levels of an aesthetic.



carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

You'd then select a coordinate system to place the geoms into. You'd use the location of the objects (which is itself an aesthetic property) to display the values of the x and y variables. At that point, you would have a complete graph, but you could further adjust the positions of the geoms within the coordinate system (a position adjustment) or split the graph into subplots (faceting). You could also extend the plot by adding one or more additional layers, where each additional layer uses a dataset, a geom, a set of mappings, a stat, and a position adjustment.



carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

You could use this method to build *any* plot that you imagine. In other words, you can use the code template that you've learned in this chapter to build hundreds of thousands of unique plots.

Session info

To improve reproducibility, print out the session info for this script.

```
devtools::session_info()
#> - Session info -------------------------------------------------------------
#>
#> - Packages -----------------------------------------------------------------
#>  package      * version  date       lib source
#>  assertthat     0.2.1    2019-03-21 [1] CRAN (R 3.6.0)
#>  backports      1.1.4    2019-04-10 [1] CRAN (R 3.6.0)
#>  broom        * 0.5.2    2019-04-07 [1] CRAN (R 3.6.0)
#>  callr          3.3.0    2019-07-04 [1] CRAN (R 3.6.1)
#>  cellranger     1.1.0    2016-07-27 [1] CRAN (R 3.6.0)
#>  class          7.3-15   2019-01-01 [2] CRAN (R 3.6.0)
#>  cli            1.1.0    2019-03-19 [1] CRAN (R 3.6.0)
#>  colorspace     1.4-1    2019-03-18 [1] CRAN (R 3.6.0)
#>  crayon         1.3.4    2017-09-16 [1] CRAN (R 3.6.0)
#>  DBI            1.0.0    2018-05-02 [1] CRAN (R 3.6.0)
#>  desc           1.2.0    2018-05-01 [1] CRAN (R 3.6.0)
#>  devtools       2.1.0    2019-07-06 [1] CRAN (R 3.6.1)
#>  digest         0.6.20   2019-07-04 [1] CRAN (R 3.6.1)
#>  dplyr        * 0.8.2    2019-06-29 [1] CRAN (R 3.6.0)
#>  e1071          1.7-2    2019-06-05 [1] CRAN (R 3.6.0)
#>  ellipsis       0.2.0.1  2019-07-02 [1] CRAN (R 3.6.1)
#>  evaluate       0.14     2019-05-28 [1] CRAN (R 3.6.0)
#>  fansi          0.4.0    2018-10-05 [1] CRAN (R 3.6.0)
#>  forcats      * 0.4.0    2019-02-17 [1] CRAN (R 3.6.0)
#>  fs             1.3.1    2019-05-06 [1] CRAN (R 3.6.0)
#>  generics       0.0.2    2018-11-29 [1] CRAN (R 3.6.0)
#>  ggplot2      * 3.2.1    2019-08-10 [1] CRAN (R 3.6.1)
#>  ggrepel        0.8.1    2019-05-07 [1] CRAN (R 3.6.1)
#>  glue           1.3.1    2019-03-12 [1] CRAN (R 3.6.0)
#>  gridExtra    * 2.3      2017-09-09 [1] CRAN (R 3.6.1)
#>  gtable         0.3.0    2019-03-25 [1] CRAN (R 3.6.0)
#>  haven        * 2.1.1    2019-07-04 [1] CRAN (R 3.6.1)
#>  highr          0.8      2019-03-20 [1] CRAN (R 3.6.0)
#>  hms            0.5.0    2019-07-09 [1] CRAN (R 3.6.0)
#>  htmltools      0.3.6    2017-04-28 [1] CRAN (R 3.6.0)
#>  httr           1.4.0    2018-12-11 [1] CRAN (R 3.6.0)
#>  inline         0.3.15   2018-05-18 [1] CRAN (R 3.6.1)
#>  jsonlite       1.6      2018-12-07 [1] CRAN (R 3.6.0)
#>  knitr        * 1.23     2019-05-18 [1] CRAN (R 3.6.0)
#>  labeling       0.3      2014-08-23 [1] CRAN (R 3.6.0)
#>  lattice        0.20-38  2018-11-04 [2] CRAN (R 3.6.0)
#>  lazyeval       0.2.2    2019-03-15 [1] CRAN (R 3.6.0)
#>  lifecycle      0.1.0    2019-08-01 [1] CRAN (R 3.6.1)
#>  loo            2.1.0    2019-03-13 [1] CRAN (R 3.6.1)
#>  lubridate      1.7.4    2018-04-11 [1] CRAN (R 3.6.0)
#>  magrittr       1.5      2014-11-22 [1] CRAN (R 3.6.0)
```

```
#>  Matrix         1.2-17   2019-03-22 [2] CRAN (R 3.6.0)
#>  matrixStats    0.54.0   2018-07-23 [1] CRAN (R 3.6.1)
#>  memoise        1.1.0    2017-04-21 [1] CRAN (R 3.6.0)
#>  mitools        2.4      2019-04-26 [1] CRAN (R 3.6.1)
#>  modelr         0.1.4    2019-02-18 [1] CRAN (R 3.6.0)
#>  munsell        0.5.0    2018-06-12 [1] CRAN (R 3.6.0)
#>  nlme           3.1-139  2019-04-09 [2] CRAN (R 3.6.0)
#>  openxlsx     * 4.1.2    2019-10-29 [1] CRAN (R 3.6.1)
#>  packrat        0.5.0    2018-11-14 [1] CRAN (R 3.6.1)
#>  pheatmap       1.0.12   2019-01-04 [1] CRAN (R 3.6.0)
#>  pillar         1.4.2    2019-06-29 [1] CRAN (R 3.6.0)
#>  pkgbuild       1.0.3    2019-03-20 [1] CRAN (R 3.6.0)
#>  pkgconfig      2.0.2    2018-08-16 [1] CRAN (R 3.6.0)
#>  pkgload        1.0.2    2018-10-29 [1] CRAN (R 3.6.0)
#>  plyr           1.8.4    2016-06-08 [1] CRAN (R 3.6.0)
#>  prettyunits    1.0.2    2015-07-13 [1] CRAN (R 3.6.0)
#>  processx       3.4.0    2019-07-03 [1] CRAN (R 3.6.1)
#>  ps             1.3.0    2018-12-21 [1] CRAN (R 3.6.0)
#>  purrr        * 0.3.2    2019-03-15 [1] CRAN (R 3.6.0)
#>  R6             2.4.0    2019-02-14 [1] CRAN (R 3.6.0)
#>  RColorBrewer   1.1-2    2014-12-07 [1] CRAN (R 3.6.0)
#>  Rcpp           1.0.1    2019-03-17 [1] CRAN (R 3.6.0)
#>  readr        * 1.3.1    2018-12-21 [1] CRAN (R 3.6.0)
#>  readxl       * 1.3.1    2019-03-13 [1] CRAN (R 3.6.0)
#>  remotes        2.1.0    2019-06-24 [1] CRAN (R 3.6.0)
#>  reshape2       1.4.3    2017-12-11 [1] CRAN (R 3.6.0)
#>  rlang          0.4.0    2019-06-25 [1] CRAN (R 3.6.0)
#>  rmarkdown    * 1.13     2019-05-22 [1] CRAN (R 3.6.0)
#>  rprojroot      1.3-2    2018-01-03 [1] CRAN (R 3.6.0)
#>  rstan          2.19.2   2019-07-09 [1] CRAN (R 3.6.1)
#>  rstudioapi     0.10     2019-03-19 [1] CRAN (R 3.6.0)
#>  rvest          0.3.4    2019-05-15 [1] CRAN (R 3.6.0)
#>  scales         1.0.0    2018-08-09 [1] CRAN (R 3.6.0)
#>  sessioninfo    1.1.1    2018-11-05 [1] CRAN (R 3.6.0)
#>  StanHeaders    2.19.0   2019-09-07 [1] CRAN (R 3.6.1)
#>  stringi        1.4.3    2019-03-12 [1] CRAN (R 3.6.0)
#>  stringr      * 1.4.0    2019-02-10 [1] CRAN (R 3.6.0)
#>  survey         3.36     2019-04-27 [1] CRAN (R 3.6.1)
#>  survival       2.44-1.1 2019-04-01 [2] CRAN (R 3.6.0)
#>  tableone       0.10.0   2019-02-17 [1] CRAN (R 3.6.1)
#>  testthat       2.1.1    2019-04-23 [1] CRAN (R 3.6.1)
#>  tibble       * 2.1.3    2019-06-06 [1] CRAN (R 3.6.0)
#>  tidyr        * 1.0.0    2019-09-11 [1] CRAN (R 3.6.1)
#>  tidyselect     0.2.5    2018-10-11 [1] CRAN (R 3.6.0)
#>  tidyverse    * 1.2.1    2017-11-14 [1] CRAN (R 3.6.1)
#>  usethis        1.5.1    2019-07-04 [1] CRAN (R 3.6.1)
#>  utf8           1.1.4    2018-05-24 [1] CRAN (R 3.6.0)
#>  vctrs          0.2.0    2019-07-05 [1] CRAN (R 3.6.1)
#>  withr          2.1.2    2018-03-15 [1] CRAN (R 3.6.0)
#>  xfun           0.8      2019-06-25 [1] CRAN (R 3.6.0)
#>  xml2           1.2.0    2018-01-24 [1] CRAN (R 3.6.0)
#>  yaml           2.2.0    2018-07-25 [1] CRAN (R 3.6.0)
#>  zeallot        0.1.0    2018-01-28 [1] CRAN (R 3.6.0)
```

```
#>  zip            2.0.4    2019-09-01 [1] CRAN (R 3.6.1)
#>
#> [1] C:/Users/jacobjc/Documents/R/win-library/3.6
#> [2] C:/Program Files/R/R-3.6.0/library
```