R: Visualization 2

Sakol Suethanapornkul

07 December 2020

Review

Let's begin by loading `tidyverse` which will also call `ggplot2`:

```
library(tidyverse)
```

Review

So far, we have practiced creating a few plots with `ggplot2`.

For every plot we make, we need to begin by setting up a canvas. This translates to (1) declaring a data frame and (2) mapping variables to aesthetic attributes.

Review

So far, we have practiced creating a few plots with `ggplot2`.

For every plot we make, we need to begin by setting up a canvas. This translates to (1) declaring a data frame and (2) mapping variables to aesthetic attributes.

```
ggplot(data    = mpg,
       mapping = aes(x = displ, y = cty)
       )
```

Or

```
mpg %>%
  ggplot(mapping = aes(x = displ, y = cty)
         )
```

Review

After that, we apply geometric objects on the canvas. So far, we have seen three such objects, namely `geom_point()`, `geom_smooth()`, and `geom_text()`.

```
mpg %>%
  ggplot(mapping = aes(x = displ, y = cty)
         ) +
  geom_point(aes(color = drv)) +
  geom_smooth(method = "lm", se = FALSE
              )
```

Notice the arguments of `geom_smooth()`, which are not part of `aes()`.

Review

Recall that any aesthetic attribute declared at the top level (i.e., inside `ggplot()`) is passed down to all geoms.

```
mpg %>%
  ggplot(mapping = aes(x = displ, y = cty) ) +
  geom_point(aes(color = drv)) +
  geom_smooth(method = "lm", se = FALSE)
```

```
mpg %>%
  ggplot(mapping = aes(x = displ, y = cty,
                       color = drv) ) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

Data frame & `ggplot()`

This also means that any data frame declared at the top level is passed down to all geoms...

...and each geom can take a different data frame!

Data frame & ggplot()

EXERCISE: What will you get from running the following sets of code?

```
a <- mpg %>%
  filter(manufacturer %in% c("audi", "nissan",
                             "toyota", "honda")
         )
```

```
b <- a %>%
  group_by(manufacturer) %>%
  summarize(m = mean(cty)
            )
```

Data frame & ggplot()

Now, we can have two different data frames in one plot!

```
ggplot(data  = a,
       aes(x = manufacturer) ) +
  geom_point(aes(y = cty),
             shape = 5 ) +
  geom_point(data  = b, aes(y = m),
             color = "red",
             shape = 2 )
```

ggplot2

In this lecture, we will look at other aspects of `ggplot2` focusing on statistical transformation and faceting.

geom_bar()

Previously, we worked with `geom_point()` where each individual observation is plotted. But there's a different kind of plot, where data are summarized before being plotted. A very good example of this is a bar chart:

```
diamonds %>%
  ggplot(aes(x = cut) ) +
  geom_bar()
```

But pay attention to the y axis. Where does count come from? It isn't in our `aes()`!

geom_bar()

Geometric objects rely on *stat* (which is short for statistical transformation) to compute values. Each geom has its default stat (e.g., run `?geom_bar` and `?geom_point` to check default stats). So,

```
diamonds %>%
  ggplot(aes(x = cut) ) +
  geom_bar(stat = "count") #default; often omitted
```

is equivalent to:

```
diamonds %>%
  ggplot(aes(x = cut) ) +
  stat_count()
```

geom_bar()

If we dig deeper into `stat_count()`, we will see that it has a default geom, which is `geom = "bar"`!

This shows that `geom_bar()` and `stat_count()` are pretty much the same thing!

So why is this remotely interesting, you might ask?

geom_bar()

Having this understanding means that we are not tied only to `geom_bar()` when we want to present frequency counts. With `stat_count()`, we can use a different geom to present the counts!

```
diamonds %>%
  ggplot(aes(x = cut) ) +
  stat_count(geom = "point")
```

```
diamonds %>%
  ggplot(aes(x = cut) ) +
  stat_count(geom = "point", shape = 4)
```

geom_bar()

Knowledge about *stat* can also come in handy. In some cases, we may want to mess with how frequency counts are to be presented.

For example,

```
cnt <- diamonds %>%
  count(cut)
```

EXERCISE: What does this code do? What object does `cnt` refer to?

geom_bar()

EXERCISE: Will these two chunks of code work? Why or why not?

```
cnt %>%
  ggplot(mapping = aes(x = cut)) +
  geom_bar()
```

```
cnt %>%
  ggplot(mapping = aes(x = cut, y = n)) +
  geom_bar()
```

geom_bar()

If you want to map the values of `n` to *y*, you will need to provide a different option for `stat`. A default option is `stat = "count"`, but now we need:

```
cnt %>%
  ggplot(mapping = aes(x = cut, y = n)) +
  geom_bar(stat = "identity")
```

which is equivalent to:

```
cnt %>%
  ggplot(mapping = aes(x = cut, y = n)) +
  stat_identity(geom = "bar")
```

geom_bar()

If you'd like to sort the x axis by frequencies (or counts), use `reorder()`. This works with `geom_bar()` and `geom_boxplot()`:

```
cnt %>%
  ggplot() +
  geom_bar(aes(x = reorder(cut, -n),
               y = n),
               stat = "identity")
```

geom_bar()

But in many cases, we'd like to present frequency counts by group. For example, in the `diamonds` data set, we may want to count the number of cuts by diamond colors. The following code doesn't give us what we want:

```
diamonds %>%
  ggplot(aes(x = cut)) +
  geom_bar()
```

geom_bar()

But in many cases, we'd like to present frequency counts by group. For example, in the `diamonds` data set, we may want to count the number of cuts by diamond colors. The following code doesn't give us what we want:

```
diamonds %>%
  ggplot(aes(x = cut)) +
  geom_bar()
```

We can add another aesthetic attribute to the ggplot call:

```
diamonds %>%
  ggplot(aes(x = cut, fill = color)) +
  geom_bar()
```

geom_bar()

But we get different diamond colors stacked on top of each other. We can change this with `position`:

```
diamonds %>%
  ggplot(aes(x = cut, fill = color)) +
  geom_bar(position = "dodge")
  #or geom_bar(position = position_dodge())
  #or stat_count(position = "dodge")
```

Run `?geom_bar` and look at `position` option.

geom_bar()

Recall that in ggplot2, graphics can be created by stacking one layer on top of another. This means we can, for instance, plot error bars on top of a bar chart!

Let's start by summarizing our data:

```
diamonds %>%
  group_by(cut, color) %>%
  summarize(m  = mean(x), sd = sd(x),
            n  = n(),     se = sd/sqrt(n)
            )
```

geom_bar()

And pipe this tibble into ggplot:

```
diamonds %>%
  group_by(cut, color) %>%
  summarize(m  = mean(x), sd = sd(x),
            n  = n(),     se = sd/sqrt(n)
            ) %>%
  ggplot(aes(x = cut, y = m, fill = color) ) +
  geom_bar(stat = "identity", position = "dodge")
```

geom_bar()

Then, we add geom_errorbar() to the call:

```
diamonds %>%
  group_by(cut, color) %>%
  summarize(m  = mean(x), sd = sd(x),
            n  = n(),     se = sd/sqrt(n)
            ) %>%
  ggplot(aes(x = cut, y = m, fill = color) ) +
  geom_bar(stat = "identity", position = "dodge") +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se
                    ),
                position = "dodge"
                )
```

See this site for more examples.

Faceting

Thus far, we have learned how to create a single plot using different geoms. But oftentimes, we may need to subset our data before plotting them. It's easy to do that with `ggplot2`.

Faceting

We use `facet_wrap()` to split a plot. We can choose how the plot is split (i.e., by rows or columns).

```
mpg %>%
  ggplot(aes(x = displ, y = cty)) +
  geom_point() +
  facet_wrap(~ drv)
```

```
mpg %>%
  ggplot(aes(x = displ, y = cty)) +
  geom_point() +
  facet_wrap(~ drv, nrow = 3)
```

Faceting

Another option is to use `facet_grid()` to split a plot by rows and columns. This can be done with a formula `a ~ b`:

```
mpg %>%
  ggplot(aes(x = displ, y = cty)) +
  geom_point() +
  facet_grid(year ~ drv)
```

Faceting

You can adjust positions of x and y axes with `scales`, which works with both `facet_grid()` and `facet_wrap()`:

```
mpg %>%
  ggplot(aes(x = displ, y = cty)) +
  geom_point() +
  facet_grid(year ~ drv,
             scales = "free_x")
```

The four options are: `fixed`, `free`, `free_x`, `free_y`

To facet or not to facet

Faceting has its pros and cons. If there is not much overlap, it may be better to use aesthetic attributes instead of faceting:

```
mpg %>%
  ggplot(aes(x = displ, y = cty, color = drv)) +
  geom_point(position = "jitter")
  # or geom_jitter(width = ..., height = ...)
```

```
mpg %>%
  ggplot(aes(x = displ, y = cty)) +
  geom_point(position = "jitter") +
  facet_wrap(~ drv)
```

Themes and apperances

ggplot2 offers almost unlimited options when it comes to appearances. Let's start with labels (see this site for more information):

```
mpg %>%
  ggplot(aes(x = displ, y = cty, color = drv)) +
  geom_point(position = "jitter") +
  labs(x = "Engine size", y = "Miles per gallon",
       color = "Car type")
```

Or

```
mpg %>%
  ggplot(aes(x = displ, y = cty, color = drv)) +
  geom_point(position = "jitter") +
  labs(x = NULL, y = NULL, color = NULL)
```

Themes and apperances

You can also adjust color palette. For instance, ColorBrewer offers great options for colors (see this site):

```
mpg %>%
  ggplot(aes(x = displ, y = cty, color = drv)) +
  geom_point(position = "jitter") +
  labs(x = NULL, y = NULL, color = NULL) +
  scale_color_brewer(palette = "Set1")
```

Themes and apperances

We can also set overall theme of our graphics with `theme_` such as:

```
mpg %>%
  ggplot(aes(x = displ, y = cty, color = drv)) +
  geom_point(position = "jitter") +
  labs(x = NULL, y = NULL, color = NULL) +
  scale_color_brewer(palette = "Set1") +
  theme_bw()
```

Try: `theme_dark()`, `theme_light()`, or `theme_classic()`

Themes and apperances

And finally, we can customize all the tiny little details (non-data) in our plot with `theme()`:

```
mpg %>%
  ggplot(aes(x = displ, y = cty, color = drv)) +
  geom_point(position = "jitter") +
  labs(x = NULL, y = NULL, color = NULL) +
  scale_color_brewer(palette = "Set1") +
  theme_bw() +
  theme(legend.position = "top",
        legend.title = element_text(size = 9),
        legend.text = element_text(size = 9)
        )
```

Check out this site.

Saving graphics

We can save our plots with `ggsave()`. By default, the most recent plot is saved:

```
ggsave("file_name.png",
       dpi = 300,
       width = ...,
       height = ...,
       units = "in") # or "cm" / "mm"
```

Show & Tell