Data Visualisation Week 5: Customising plots

What's happening today?

Today, we're going to have a bit of a change of pace. In the last few weeks, we've introduced how to graph different kinds of variables and different kinds of relationships, including thinking about different ways to get our data into the kind of format that works for the graphs we want to draw.

While the graphs we've been drawing have been looking increasingly professional, they haven't been very *flexible*: we've been using default colour schemes, default layouts, and so on. While we looked at how to add labels, and change the order of levels in a factor, we haven't done much else. What if you like different colour schemes? What if you don't like your scatterplots to have grey backgrounds?

We also haven't discussed what we do once we've made our graphs. How do we export them? And how do we integrate them into other things we're doing? Let's address those issues now.

Getting started

You know how this works at this point. Let's load some packages, including a new one, and let's load some data. (You're free to load other datasets that we've already used.)

As a reminder, when we load new packages, we first have to install them once only. Let's do so now.

```
install.packages("ggthemes")
```

As a reminder, **you only have to do that once.**

```
library(tidyverse)
library(lubridate)
library(ggthemes)
```

```
taylor <- read_csv("https://bit.ly/swifty-data")
```

Back to scatterplots, with some colour

Let's remind ourselves of the relationship between tempo and danceability among Taylor Swift's tracks, coloured according to which album they're on.

```
ggplot(data = taylor) +
  aes(x = tempo,
      y = danceability,
      colour = reorder(album,
                       ymd(release_date))) +
  geom_point()
```

This looks OK (except for the legend taking up more than half the space, but you'll remember how we can deal with this issue.) But what if we want to use a different colour scheme?

Here we're going to introduce two approaches to colour palettes that are quite easy to use in R – **ColorBrewer** and **viridis**. Both come preloaded with ggplot2, so we don't need to do anything special to load them. Let's start with a colour scheme from ColorBrewer:

```
ggplot(data = taylor) +
  aes(x = tempo,
      y = danceability,
      colour = reorder(album,
                       ymd(release_date))) +
  geom_point() +
  scale_colour_brewer(palette = "Paired")
```

… and follow it up with a colour scheme from viridis.

```
ggplot(data = taylor) +
  aes(x = tempo,
      y = danceability,
      colour = reorder(album,
                       ymd(release_date))) +
  geom_point() +
  scale_colour_viridis_d(option = "D")
```

With viridis, guessing how you might find the other colour schemes is easy: you just swap out "D" for different letters of the alphabet and see what happens. For ColorBrewer, it's a bit less obvious: Google ColorBrewer and R and the full list of palettes will come up.

Alternatively, you can run the following command and the available palettes will be plotted for you:

```
RColorBrewer::display.brewer.all()
```

Note that here the two colons (::) are telling R to pull the function from the RColorBrewer library, even when that library is not loaded directly.

- One other thing to mention on these colour schemes is that, with ColorBrewer, you specify the colour scheme to use in the same way regardless of whether you're using a continuous or discrete colour scheme. With viridis, it's a bit different; if you're mapping colour to a categorical, factor, or **discrete** variable, you want to add scale_colour_viridis_d(); if you're mapping colour to a continuous variable, you want to add scale_colour_viridis_c(). Like so:

```
ggplot(data = taylor) +
  aes(x = tempo,
      y = danceability,
      colour = acousticness) +
  geom_point() +
  scale_colour_viridis_c(option = "D")
```

Back to bar charts, with some colour

Let's revisit another graph, from a few weeks ago: the balance between major and minor on each of Taylor Swift's albums.

```
ggplot(data = taylor) +
  aes(x = fct_reorder(album,
                      ymd(release_date)),
      fill = mode) +
  geom_bar(position = "dodge")
```

What if you wanted your own colour scheme? What if you thought of major tracks being a particular shade of purple, and minor tracks being a particular shade of green?

```
ggplot(data = taylor) +
  aes(x = fct_reorder(album,
                      ymd(release_date)),
      fill = mode) +
  geom_bar(position = "dodge") +
  scale_fill_manual(values = c("darkorchid2",
                               "forestgreen"))
```

You could do something like this, with scale_fill_manual. (You can imagine a similar thing for scale_colour_manual if you were colouring a scatterplot.) You might wonder how I knew to use the colours *darkorchid2* and *forestgreen*. All the colours available in R are available at this guide (http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf).

An alternative to using named colours is to use **HEX codes**. These start with a # and are used to represent different colours in digital applications. Let's instead change our colours to "#5AD9ED" and "#5AED9F" and see what happens:

```
ggplot(data = taylor) +
  aes(x = fct_reorder(album,
                      ymd(release_date)),
      fill = mode) +
  geom_bar(position = "dodge") +
  scale_fill_manual(values = c("#5AD9ED",
                               "#5AED9F"))
```

Tidying up facets

Let's revisit facets, by looking at how the distribution of keys varies across Taylor Swift albums.

```
ggplot(data = taylor) +
  aes(x = key) +
  geom_bar() +
  facet_wrap(~ album)
```

Any problems with this?

One issue is that there's several albums that don't include every key. This isnt a problem per se, but you can imagine other settings where it is, particularly if you're looking at a categorical variable that varies a lot according to your faceting variable. You can also imagine settings where the number of observations within facets varies a lot: under those circumstances, having a common y-axis might make it difficult to understand what's going on in some facets.

What we can do is free up the axes: x, y, or both. Here's how we do it. Compare the following four options: which one works best? Which might work best in other contexts?

```
ggplot(data = taylor) +
  aes(x = key) +
  geom_bar() +
  facet_wrap(~ album)

ggplot(data = taylor) +
  aes(x = key) +
  geom_bar() +
  facet_wrap(~ album,
             scales = "free_x")

ggplot(data = taylor) +
  aes(x = key) +
  geom_bar() +
  facet_wrap(~ album,
             scales = "free_y")

ggplot(data = taylor) +
  aes(x = key) +
  geom_bar() +
  facet_wrap(~ album,
             scales = "free")
```

Who likes Tufte?

Who likes Tufte? Some people like Tufte. If you're one of them, you might find yourself thinking "these boxplots use up too much ink, I wish ggplot2 would let me use the Tufte approach". It's easier to make your graphics look like Tufte's if you've loaded the **ggthemes** package (which hopefully you were able to do from the start of this workshop).

If you want to look at how the tempo across Taylor Swift's different albums have varied, why not compare:

```
ggplot(data = taylor) +
  aes(x = reorder(album,
                  ymd(release_date)),
      y = tempo) +
  geom_boxplot()
```

with:

```
ggplot(data = taylor) +
  aes(x = reorder(album,
                  ymd(release_date)),
      y = tempo) +
  geom_tufteboxplot()
```

(As I mentioned last week, my instinct is that there are better ways to communicate this information than in box plots. But if you like box plots, and you like Tufte, here's a way of indulging that combination.)

Prepackaged themes

You can change each element of a ggplot object separately, if you want. You can change the background colour, the colours of individual objects, the fonts used in the legend, and so on. Sometimes this is useful and practical: for example, if you're a journalist who's using a house style, it's easier to manually set this up using ggplot2 than it is to tweak the graph afterwards using Illustator or Photoshop. (This isn't a made-up example: you can read things John Burn-Murdoch at the FT has written about arranging his ggplot2 workflow so that ggplot graphics can go straight into the paper.)

You can also use prepackaged themes, if there are particular styles that you want to mimic. Not everyone likes the grey grid with white gridlines that ggplot2 uses by default, for example. (I'm not the biggest fan.) Several themes come preloaded with ggplot2, and **ggthemes** adds several more; if you've found another theme that you like the look of, there's often a package available for that as well, you'll just have to install it.

Let's say we want to tweak our last graph so that we lose the grey grid, swapping it for something a bit more minimal.

```
ggplot(data = taylor) +
  aes(x = tempo,
      y = danceability,
      colour = acousticness) +
  geom_point() +
  scale_colour_viridis_c(option = "D") +
  theme_bw()
```

How do you feel about that?

Having done that, type it out again, but pause once you've typed as far as theme_. This will give you a sense of which other themes are available. Some of them I'd advise against – the Excel theme is only really in there as a joke, for example – but see what you like and what you get on with.

Annotations

Quickly: annotations. Our graphics have been fairly sparse so far. This isn't a problem per se – sparse graphs are often clear graphs – but you might find yourself wondering how it is that you sometimes see graphics with text on them. This is broadly pretty easy to do in ggplot, and this is how.

```
ggplot(data = taylor) +
  aes(x = tempo,
      y = danceability,
      colour = acousticness) +
  geom_point() +
  theme_light() +
  annotate("text",
           x=175,
           y=0.8,
           label="No Taylor songs \n are both fast \n and danceable")
```

Here, what I've done is add an annotate() in my ggplot2 code. You can see it needs four arguments: what kind of annotation it is, where it's going on each of the x and y axes, and a label. (You'll also note I've broken the label up into three lines.)

You'll often want to annotate graphs if you think particular areas of them are interesting. (You might also want to annotate particular points, and you won't always want to do this manually. We won't go over this now, but if this is likely to be a focus for you, I'd recommend the **ggrepel** package.)

You'll also probably be thinking — what's with the "backslash n"s in the middle of the text? These are a way to indicate to ggplot that you want to put a line break (new line) in your text.

Exporting graphics

Once we've made graphs, then what? We don't want to take a photo on our phones, at the very least.

You'll note in RStudio that there's a little "export" button in the graphics pane. You can use this if you want – it's no-frills, but it normally works.

However, I normally use the **ggsave** command. One thing about ggsave is it starts to raise questions of what the *working directory* is, which I've been largely avoiding up to this point. But when you save graphics, you're saving graphics somewhere, and you need to know where that is. Let's find out with the getwd() command.

```
getwd()
```

You can also set the working directory. Here's how it works on the computer I'm writing this handout on; I'll also demonstrate this in the workshop.

```
setwd("~/Library/Mobile Documents/com~apple~CloudDocs/r/smi105-data_viz_materials/
smi105-teaching-materials/week-05/handout")
```

Now I've done this, I can both load data from this folder, like so (imagine I've saved **taylor.csv** in this folder):

```
taylor_again <- read_csv("taylor.csv")
```

but I can also *save* the graphic I plotted most recently:

```
ggsave("taylor_plot_01.png")
```

You'll note this is fairly sparse. The **ggsave** command is flexible, but simple to use if you want it to be. (I normally want it to be simple.) One thing you'll sometimes want to do is change the dimensions of the graph – maybe you want a long and thin graph, maybe you want a short and fit graph. By default ggsave uses inches as its units, but you can tweak this. Here's how you can change the dimensions of a graph.

```
ggsave("taylor_plot_01.png", width = 10, height = 5)
```

And so on. (You can also export to different file formats, like .jpg, and .eps; if you want to do this, just change the file extension in the command.)

Finally: RMarkdown

How do I write these handouts? How might you effectively integrate graphics, and other processing, into your work in a more straightforward way, so you don't have to keep exporting? You can use RMarkdown. In the last part of this session, I'll demonstrate how this works.

Task

No task this week – you've got an assessment due in for me soon, so please dedicate the rest of the week to working on that assessment.