Data Visualisation Week 10: Web scraping and interactive plots

What's happening today?

Let's round off our new set of skills with two more: **web scraping** and **interactivity**.

There are still a tonne of other things you can do in `R` using additional libraries, but I wanted to end things with these for two reasons: 1) web scraping is an increasingly useful tool but is always becoming more and more difficult as data holders try to keep control of their data (and the value it's worth). A relatively new package called `polite` allows you to set up your web scraping activities so that they comply with websites terms of services. This will help prevent you from getting banned or blocked from websites when you try to web scrape 2) interactivity is a valuable tool to encode more information into your data visualisation, especially information/qualitative data that helps readers engage with the people behind the numbers.

Scraping data from the web, politely

Let's start by loading some packages. Many of these we'll be using for the first time, so remember we'll need to install them first.

```
install.packages("polite")
install.packages("rvest")

library(tidyverse)
library(polite)
library(rvest)
```

Okay, lots of packages today, and we're going to use even more when we get to interactivity. A quick rundown of what these new ones do:

- `polite` is a web scraping package that emphasises good web etiquette (not spamming websites, respecting their terms of services, etc.) It's worth saying, even if a website does not allow scraping, that doesn't mean it's illegal. You can still scrape a website despite it being against their terms of service. But, it is considered poor etiquette and, more importantly, if you don't know how to avoid drawing attention to yourself and having your account banned, your IP banned, and, if you don't know what you're doing, might accidentally do something that *is* illegal like spamming the website with requests that constitute a DDOS attack.
- `rvest` is a package for working with webpage data, in particular, stripping out the code and tags to get back to the kind of data format we need.

Okay, let's prepare to scrape a webpage. The webpage we'll be scraping is this page (https://en.wikipedia.org/wiki/List_of_countries_by_tea_consumption_per_capita) about tea consumption per capita in countries across the world. I'd recommend googling "tea consumption wikipedia" and then going to the page yourself to see what it looks like, rather than writing the url out in full.

The first step is for us to take our url and introduce ourself to the web host — we will request information about whether web scraping is allowed, and collect any rules we need to stick by (e.g. how frequently we can send requests to scrape the data), by using the `bow` function.

```
tea <- "https://en.wikipedia.org/wiki/List_of_countries_by_tea_consumption_per_cap
ita"

tea_bow <- bow(tea)
tea_bow
```

So, we've found our that the page we've provided is scrapable for us. Interestingly, you might want to try another website like X or reddit to see what a site that does not allow scraping returns.

Now we need to scrape a copy of the table that's on the page.

```
tea_html <- scrape(tea_bow)
tea_html
```

So, what is this? Well, it's the whole webpage, including all of the information about the image links, the fonts, the formatting, and so on and so on. What we need to do now is specifically extract the table from all of that information.

To do this, we need to identify the "node" or "element" that holds the table. To do this, you need to use the "Inspect element" function on a web browser (this might be hidden behind the Developer only options in your browser). What we want to find, ideally, is something called the "XPath" of the table. I'll show you how I do this on my web browser in the workshop but the process should briefly look like: 1) right click somewhere on the table in your web browser and then click Inspect Element; 2) find the line of code that contains the whole table; 3) right click, and then choose Copy -> XPath

Now we have our XPath, we can extract our table from our scraped page:

```
tea_table <- html_element(tea_html, xpath = '//*[@id="mw-content-text"]/div[1]/tab
le')

tea_table
```

Nearly there, now we just have the html code for our table and the last thing we need to do is convert that into a nice dataset object that `R` likes to use. Luckily, the rvest package has a convenient function to do this:

```
tea_table <- html_table(tea_table)

tea_table
```

A little more tidying: regex and separate

Hooray! Now we have a table that we can definitely tidy up using the data tidying skills we learned over the last few weeks. Let's start by renaming our columns, and then I'll show you another couple of neat data tidying tips.

```
tea_table <- tea_table %>%
  rename(
    rank = Rank,
    country = `Country/Region`,
    tea_consumption = `Tea consumption`
  )
tea_table
```

Okay, there's a couple of things we want to do now using our `mutate` function to change our variables:

1. We have some footnotes (e.g. [2]) that we ideally want to get rid of. We will use something called "regular expressions" for this, also known as "regex"
2. Ideally we want to split our tea consumption in kilograms (kg) from our tea consumption in pounds (lb), and then we want to convert these into just numbers. For this, we'll use the function `separate`, followed by the `parse_number` function to extract only the numbers from the values.

```
tea_table <- tea_table %>%
  separate(
    tea_consumption, sep = "kg",
    into = c("tea_consumption_kg", "tea_consumption_lb")
           ) %>%
  mutate(
    country = str_remove_all(country, "\\[[0-9]\\]"),
    tea_consumption_kg = parse_number(tea_consumption_kg),
    tea_consumption_lb = parse_number(tea_consumption_lb)
  )

tea_table
```

That looks much better. Let me attempt to explain the the regex we used:

- The regex [0-9] means "Any number in this space between zero and nine"; as you can tell, the square brackets here are performing a specific function — they indicate the range of numbers that we want to remove.
- Because square brackets have a special meaning in regex we need to do something special when we want to remove a square bracket. We need to do what is called a "string escape" using backslashes. So, if you want to remove a special character like [from a string you need to put two backslashes first, i.e. \\[

Let's make an interactive plot with plotly!

Let's install and load plotly. It's also fairly likely that R will attempt to install several other packages in order to get plotly loaded.

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

`plotly` is a tool for generating interactive graphics that isn't limited to ggplot2, and more generally isn't limited to R. What we're doing here is working within a ggplot2 framework to generate plotly objects: using R to generate interactives that can be dropped into different settings.

(In practice, the easiest thing is to integrate these interactives into .html files that you've generated using RMarkdown.)

Let's start by making a plot of our tea consumption in `ggplot2`, and then try doing the same in `plotly`.

```
tea_plot <- tea_table %>%
  ggplot() +
  aes(y = fct_reorder(country, tea_consumption_kg),
      x = tea_consumption_kg,
      fill = tea_consumption_kg) +
  geom_col() +
  ylab("") +
  xlab("Annual Tea Consumption per Capita (kg)") +
  scale_fill_gradient(low = "grey20", high = "brown") +
  theme_minimal() +
  theme(legend.position = "none")

tea_plot
```

Looks good. The nice thing about plotly is that it can take `ggplot2` plots and then translate them into a plotly plot using the `ggplotly` function: let's see how it looks.

```
ggplotly(tea_plot)
```

Try hovering over the bars on the plotly graph and see what happens. Also try resizing the plot, or opening it in your browser using the "Show in new window" button.

Custom tooltip hover text

This is a great start, but the text that's showing in our tooltip (the box that appears when we hover over) isn't particularly nice looking. How can we fix that?

Well, what we need to do is create a custom "text" aesthetic within our `aes()` function using `paste`. Copy and paste our original tea_plot ggplot2 code and add the text argument below:

```
tea_plot <- tea_table %>%
  mutate(
    emphasise = case_when(country == "China" ~ "brown",
                          country == "United Kingdom" ~ "brown",
                          country == "India" ~ "brown",
                          TRUE ~ "grey80"
                          )
  ) %>%
  ggplot() +
  aes(y = fct_reorder(country, tea_consumption_kg),
      x = tea_consumption_kg,
      fill = emphasise,
      text = paste(
        "Country:", country,
        "<br>Tea consumption (kg):", tea_consumption_kg,
        "<br>That's around", round(tea_consumption_kg*450,0), "cups of tea a yea
r!"
      )
      ) +
  geom_col() +
  ggtitle("Despite all cultures being known for their tea, the UK<br>comes third,
China comes 21st, and India comes 29th<br>in tea consumption") +
  ylab("") +
  xlab("Annual Tea Consumption per Capita (kg)") +
  scale_fill_identity() +
  theme_minimal() +
  theme(legend.position = "none",
        axis.text = element_text(size = 7))

ggplotly(tea_plot, tooltip = "text") %>%
  layout(
    margin = list(l = 75, t = 130, r = 75, b = 50)
  )
```

So, what is this doing?

- `paste` is creating a long string to display when we hover over the data. The text that is in quotation marks (“) is text that is static, but the variable names stand for things that will change when we hover over the text (e.g. the country name, the value for amount of tea drank in kilograms, etc.) By default, paste will put a space between each element. If you want it to not put any spaces between elements (for example, if you wanted to add kg after the number), you can use `paste0` instead.
- The
 tag is a HTML tag that tells the web page (which the plot is being rendered in) to include a line break to put the next bit of text on a new line. This helps makes things nice and organised.
- `tooltip = "text"` is telling ggplotly that everything we need to show in the hover text is held in the `text` aesthetic we created in the ggplot code.

I’ve also added a couple of extra flourishes here: firstly, I’ve used mutate to create a new variable of countries I want to emphasise, and then that variable contains the colours I want those countries to be on the graph. Because that variable contains valid colours, I can use `scale_fill_identity` to tell R that the variable I’m filling the bars with contains colours. In all:

- An extra thing I've added here is an *active title* — pointing something interesting out about the graph and directing my readers towards a specific couple of data points — remember that this is good for data storytelling.
- I've chosen to use a highly saturated brown colour to highlight the three countries I'm emphasising in my story, and then a desaturated grey colour for the "unimportant" countries.
- Because Plotly (and ggplot2) quite bad for long titles, I have changed the default "margins" for my plot using the `layout` plotly function, specifically to increase the amount of space at the top (`t` for top) in pixels.

What next?

That's it! Plotly can do other interesting things; if you have data over multiple years or any other time variable you can also use plotly to create some animation (`gganimate` also does this). Plotly itself is a huge package that works across javascript, python, and R. If you're interested in this, spend some time Googling plotly combined with ggplot2.

It's worth bearing in mind that plotly does have some limitations: it doesn't work so well for displaying subtitles and captions, probably because it's designed to be integrated with web pages that can add the captions separately.

There's no task this week, but there's loads more you can do with plotly.