Exercises: htmlwidgets and RPubs

Exercises: Exercises: htmlwidgets and RPubs

Overview of Exercises

The interactive web is driven by JavaScript*, the majority of interactive elements that you use on websites are written in JavaScript - from interactive maps to auto-completing pop-up menus. Like in R, there are hundreds of different JavaScript libraries dedicated to various visualisation tasks. There is a tool called `htmlwidgets` that allows R developers to easily build bindings to JavaScript libraries, allowing incredibly rich interactive content for the web to be built just with the R language.

These bindings to JavaScript libraries are typically distributed as individual R packages; an individual R package for an individual JavaScript library. The htmlwidgets.org website provides a showcase of some of the `htmlwidget` dependent bindings that are available through CRAN.

Important: There will be functions mentioned in these tutorials that you may not have used before and some descriptions may appear deliberately misleading. However, they are an attempt to get you to think about how your code is constructed. Please do ask questions at any point!

Worked examples

The tutor will work through a number of worked examples on the projector, utilising the following datasets. Note that these are also provided in the `htmlwidgets.R` file and you are advised to copy and paste them from there and NOT from this PDF. Copying code from a PDF into a script file is a recipe for disaster, there are likely hidden characters and all sorts of nastiness.

```
africa_data_points = data.frame(
  lat = rnorm(26, mean = 6.9, sd = 10),
  lng = rnorm(26, mean = 17.7, sd = 10),
  size = runif(26, 5, 10),
  label = letters
  )
```

```
## WDI data
library(WDI)
world_data <- WDI(country=c("GB","EG","SA","EE","CA"), indicator=c("EG.GDP.PUSE.KO.PP","EN.ATM.CO2E.PP.G
                  start=1990, end=2000)

wdi_codes <- list(
  "EG.GDP.PUSE.KO.PP" = "GDP per unit of energy use (PPP $ per kg of oil equivalent)",
  "EN.ATM.CO2E.PP.GD.KD" = "CO2 emissions (kg per 2005 PPP $ of GDP)",
  "NY.GDP.MKTP.KD" = "GDP (constant 2000 US$)"
)
```

Exercise 0: R language checks

If you feel you would benefit from some revision of the basics of R syntax, then you are invited to complete the following exercise, otherwise please continue to Exercise 1.

Shiny is very easy to use but does expect knowledge of the basic R language - particularly an understanding of the different types of brackets and assignments. Many new users of R feel frustrated because of confusion about what brackets are for, to ensure that in later exercises you can build Shiny apps please consider the following guide:

- Round brackets () encapsulate the arguments for a function, in the case of `rep("Hello World", 2)` the round brackets encapsulate the two arguments passed to the function `rep` - arguments are therefore deliminated by commas.
- Square brackets [] are used for extracting parts (rows, columns, individual elements) from data structures - that's there only use
- Braces {} are used for containing expressions - when writing mathematical expressions by hand round brackets are usually used for controlling precedence (order of operations), but in R you should write 2*{x+1}^2.

Braces are necessary where *more than one thing* is being done in an individual argument

```
rep(
  "strings",
  {
    no1 <- 2
    no1 +3
  }
)
```

```
## [1] "strings" "strings" "strings" "strings" "strings"
```

With this in mind, work through the exercises in “Scripts-to-Fix.R.

Exercise 1: interactive map

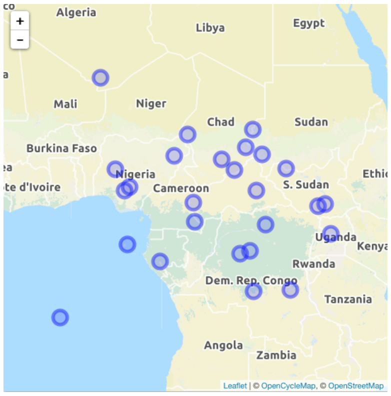
Start a new script file for this exercise with appropriate comments.

1.1 Create a basic `leaflet` map wth the following code:

```
leaflet() %>%
  addTiles()
```

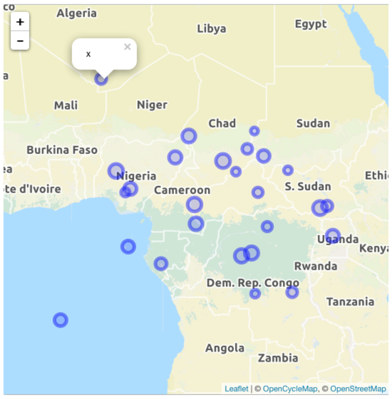
1.2 Refer to rstudio.github.io/leaflet/basemaps.html to change the map to use the attractive “Thunderforest.OpenCycleMap” tiles

1.3 Combine the code from 1.2 with the code for visualising `africa_data_points` to obtain something similar to the following visualisation

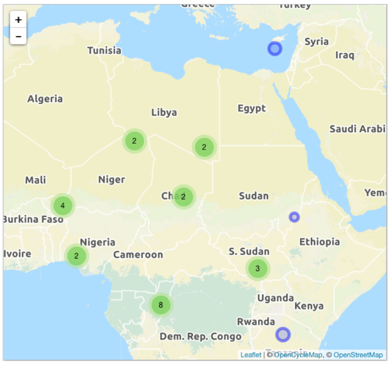


1.4 In the `plotly` worked example it was shown how to access columns from a `data.frame` within a `htmlwidget`. Use this knowledge to add two features to the map:

- scale the size of the circles by the size column of the dataset
- add a tooltip (also called popup) to the circles that shows the label of each point



1.5 Consult the documentation for `addCircleMarkers` and find out how to cluster the circles as you zoom out.

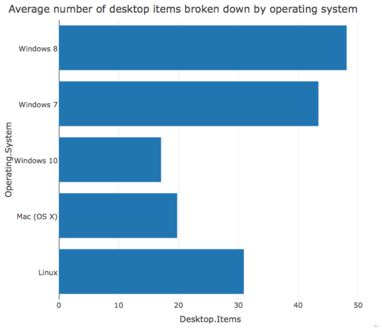


Exercise 2: interactive chart

This exercise uses an example dataset deposited on Figshare about the number of desktop items on University member computers, dx.doi.org/10.6084/m9.figshare.3425729. Please consider adding to the dataset on your own machine - http://goo.gl/forms/IehEi6dyCEBIlbXW2.

Start a new script file for this exercises with appropriate comments.

In this exercise you will simply create a [relatively boring] bar chart with `plotly` that looks like this:



2.1 Use the function `read_csv` from `readr` (a part of the `tidyverse`) to import the data and store it against an appropriate variable.

2.2 A good feature of `read_csv` is that it preserves spaces in column names, a bad feature of `formula` is that they doen't handle these. Use the function `make.names` to sanitise the `colnames` of your variable.

3.3 Using the `dplyr` library [part of `tidyverse`] perform the following actions, note that you may want to refer to the cheatsheet under Help in the menubar:

- Select only the columns containing the number of desktop items and the operating system
- Group the data by the Operating System
- Mutate the column containing the number of desktop items to contain the mean of the number of desktop items (the previous grouping step will ensure this is a factored mean)
- Use unique() to ensure that duplicate rows are removed
- Store this subsetted dataset against an appropriate symbol

3.4 Provide this dataset to `plot_ly` with `type = "bar"` to generate a simple barchart

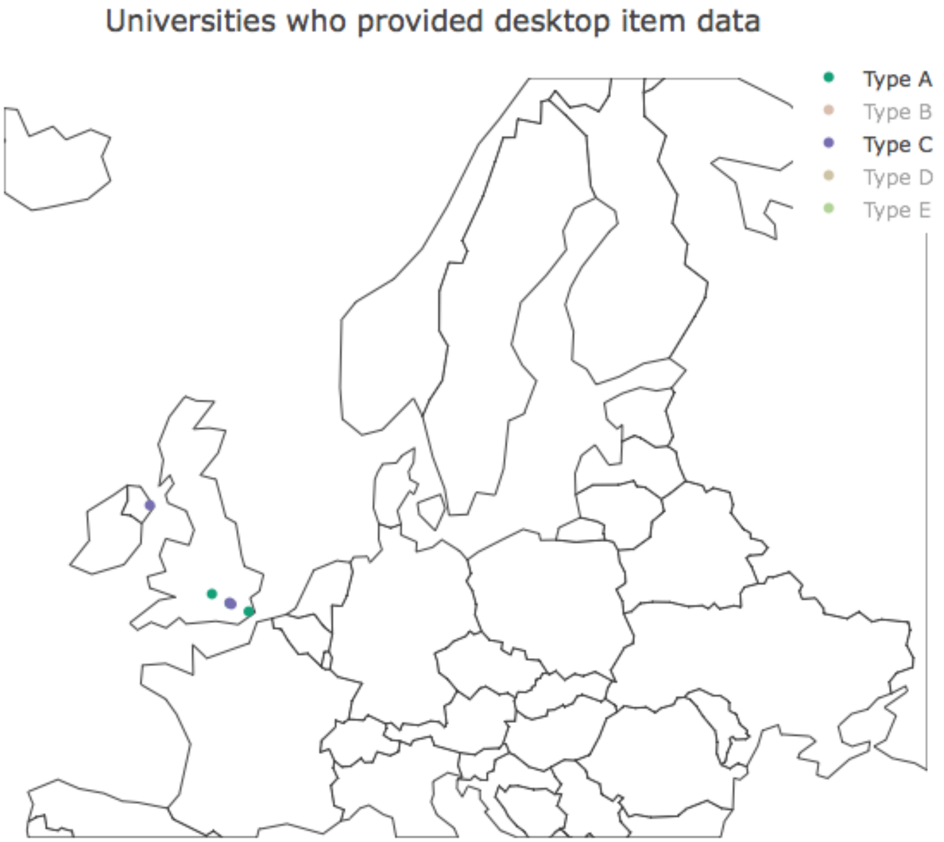
3.5 Pipe the chart into `layout` and specify an appropriate title for the chart.

3.6 If you wish to further modify the chart, refer to https://plot.ly/r/reference/

Exercise 3: maps with plotly

Leaflet is extremely powerful and you're advised to use that library for "geo-accurate" maps. If you're interested in "representative" maps then plotly (and highcharter) provides both choropleth and scattergeo functionality. This exercise is a basic introduction to scattergeos with plotly, you'll create the following map:

```
# uni_locations <- read_csv("https://ndownloader.figshare.com/files/5449670")
# colors <- rep(c("#1b9e77", "#d95f02", "#7570b3", "#e7298a", "#66a61e"), 3)
# labels <- rep(c("Type A", "Type B", "Type C", "Type D", "Type E"), 3)
# uni_locations <- uni_locations %>%
#   mutate(color = colors, label = labels)
# uni_locations %>%
#   group_by(label) %>%
#   plot_ly(
#         lat = ~Latitude,
#         lon = ~Longitude,
#         colors = ~color,
#         type = "scattergeo",
#         mode = "markers",
#         color = ~label
#         ) %>%
#   layout(title = "Universities who provided desktop item data",
#          geo = list(scope = "europe"),
#          legend = list(xanchor = "auto",
#                        yanchor = "top"))
library(grid)
img <- readPNG("images/plotly_scattergeo.png")
grid.raster(img)
```



Start a new script file for this exercises with appropriate comments.

3.1 Use the function `read_csv` from `readr` (a part of the `tidyverse`) to import the data file at this url https://ndownloader.figshare.com/files/5449670 and store it against an appropriate symbol.

3.2 To specify a particular type of chart to plotly, you must use the argument `type`. Create a `scattergeo` chart from the dataset above, noting that rather than `x` and `y` you must specify the `lat` and `lon` columns.

3.3 The layout of a plotly chart is controlled with the `layout()` function, pipe the output from above into layout and provide an appropriate `title`.

3.4 There is a `geo` argument for `layout` to which a `scope` can be provided, refer to https://plot.ly/r/reference for an appropriate value for `scope` to display this data well.

3.5 The dataset doesn't include any groupings for the universities, add the following vectors as columns to your data.frame so that you can use them within the plotly map:

```
colors <- rep(c("#1b9e77", "#d95f02", "#7570b3", "#e7298a", "#66a61e"), 3)
labels <- rep(c("Type A", "Type B", "Type C", "Type D", "Type E"), 3)
```

3.6 Use `dplyr` to group the data.frame by the `label` column you just added, and pipe this into your map. Ensure that the `plot_ly` map uses this grouping by specificying two additional arguments:

- color: this is the column by which groupings should be detected
- colors: this is the column containing the colours for each group

Exercise 3: Network

Networks/Graph are used to study connections between entities, where these entities may be telephones and the connections between then SMS messages. The `htmlwidget` library called `visNetwork` is the most widely recommended tool for visualising this data, it is based on the vis.JS library and supports `igraph` objects to.

For demonstration purposes, we require a dataset that is not yet a graph but can readily be converted into one. We use the `quanteda` library to generate collocations for a given text, i.e. which words appear adjacent in a corpus.

3.1 Install and load the `quanteda` library

```
## quanteda version 0.9.8.3
```

```
##
## Attaching package: 'quanteda'
```

```
## The following object is masked from 'package:base':
##
##     sample
```

3.2 Use the following code to generate a collocation table

```
nineteen_eighty_four <- c("Exactly. By making him suffer. Obedience is not enough. Unless he is sufferin

collocs_1984 <- collocations(nineteen_eighty_four, punctuation = "dontspan")
```

3.3 The object created by `collocations` is a data.table, which is a slightly different beast than a data.frame. We can pretty much ignore the differences for now, but it might be wise to convert it into one of the `tibbles` that the `tidyverse` uses:

```
collocs_1984 <- as_data_frame(collocs_1984)
```

3.4 The visNetwork library expects two data structures; a data.frame of nodes and edges. From the `collocs_1984` object create two objects with the following properties:

- Nodes: – id: Column containing unique ids for the nodes (can be strings) – label: Column containing labels for each node (displayed in networks) – title: Column containing titles for each node (displayed as tooltips in networks)

- Edges: – from: node from which an egde leaves – to: node into which an edge enters

3.5 Ensure you have the `visNetwork` library on your machine, consult the documentation for the `visNetwork` function and provide your data structures from above as the first two arguments of the function. NOTE: It may take some time to display the network with defaults, which is a shame.

3.6 The network takes a long time to display because the layout algorithm is inefficient. The `igraph` library is a fantastic tool for network analysis [written in C/C++] and provides a much faster layout alogirthm. Pipe your network from above into `visIraphLayout` to see the difference, this is typically a very good layout.

3.7 The `nodes` and `edges` objects can be provided with additional attributes to style the graph, add the following columns to your data structures and observe the change to the network when re-evaluated:

- edges: column called "width" populated with the "count" column from the `collocs_1984` collocations data.
- nodes: column called "color" populated with an appropriate vector containning named colours from this list: c("blue","red","green","purple")

3.8 Consult the documentation to modify your network accordingly (these are increasingly difficult):

- Prevent users from dragging nodes
- Change the nodes to be squares

- When a node is selected (clicked), highlight those nodes directly connected to your selection
- Make the edges curved lines
- Create a legend for your network