HOUSE OF INNOVATION

MODULE 6: CREATING INTERACTIVE OUTPUTS & CONCLUSION

7316 - INTRODUCTION TO DATA ANALYSIS WITH R

Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

1. Creating interactive outputs

In the two previous session, we learnt how to create figures (with `ggplot2`), tables (with *e.g.* `stargazer`) and documents (with `knitr` and RMarkdown). Those, however, are static outputs: this means that your audience cannot interact with those (*e.g.*, moving the graph, ordering the table, changing the variables, etc.). In some cases, you may want to give more freedom to your audience, so that they can better understand your results and findings. R offers you multiple tools to allows users interacting with your figures, your tables, or your data.

1.1 Interactive figures with `plotly`

`plotly` is yet another tool to generate graphs, like `ggplot2`, where the generated graphs will not be an image, but a webpage, *i.e.* your audience will be able to perform actions on the graph with the mouse in the web browser. Actions includes zooming in the plot, rotating the plot (useful for 3D plots) and selecting observations.

1.1.1 A simple example

- **Warning:** The figures bellow will not show properly in the pdf, because pdf documents are not interactive. If you want to interact with the figure, you have to run the code in R, or regenerate the document in an html output.

```r
df <- rio::import("data/7316_module_6_data/graph_reg.Rds")

# Loading plotly
library(plotly)
```

- Plotting a simple graph

```r
plot_ly(df,
        x = ~iv,
        y = ~dv,
        type="scatter")
```

- Adding a third dimensions

```r
plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv)
```

- Adding groups

```r
plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv,
        color = ~group)
```

- Changing colors of the groups

```r
plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv,
        color = ~group,
        colors = c('#BD382B', '#0C5B8F'))
```

- Changing size

```
# Create a new variable to add as size
df$random_size <- rnorm(nrow(df), 1, 100)

plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv,
        color = ~group,
        colors = c('#BD382B', '#0C5B8F'),
        size = ~random_size)
```

1.1.2 A more complex example: drawing maps

```
# Map example, from https://plotly.com/r/mapbox-county-choropleth/

# Get data
url <- 'https://raw.githubusercontent.com/plotly/datasets/master/geojson-counties-fips.json'
url2 <- "https://raw.githubusercontent.com/plotly/datasets/master/fips-unemp-16.csv"

counties <- rjson::fromJSON(file = url)
df <- read.csv(url2, colClasses = c(fips = "character"))

# Create the map
fig_map <- plot_ly() %>%
  add_trace(
    type = "choroplethmapbox",
    geojson = counties,
    locations = df$fips,
    z = df$unemp
  ) %>%
  layout(
    mapbox = list(
      style = "carto-positron",
      zoom = 2,
      center = list(lon = -95.71,
                    lat =  37.09))
  )

fig_map
```

Maps are a form of graphs. It is possible to simply draw them with ggplot2, with the following properties:

- The x and y axis are often hidden. In ggplot2, you can use `theme_void()` to hide the axes.
- The x and y axis are used as projection of geographical coordinates: latitude and longitude. Because the earth is a sphere, the choice of projection you do will change the appearance of your map.
- city, region, or country border are usually depicted with polygons. Internet is full of geographical polygon databases that you can use as a background layer of your maps. One nice source of polygon data is *Natural Earth*, integrated in R with the package `rnaturalearth`.

Once you chose your polygon database for the part of the globe you wish to project, you may add other layers to your graph as you usually do with ggplot, using `geom_point()` for dots at specific latitudes and longitudes, `geom_sf()` to fill surfaces (*polygons*) with colors, or `geom_sf_label()` and `geom_sf_text()` to add labels and texts to your map.

1.1.2.1 Example: plotting Europe

1. First, we want to load and plot the polygons of Europe as a background layer.

```r
# Get country polygon from the rnaturalearth package
world <- rnaturalearth::ne_countries(scale = "medium",
                                     returnclass = "sf")

# we want to draw a map of Europe. Let's subset this part of the world
Europe <- world[world$continent == "Europe",]
```

- In the chunk of code above, `rnaturalearth::ne_countries()` create a spatial polygons dataframe, containing the country polygons of the world.
    - The `scale` indicate the zoom level of the map.
    - the `returnclass` provide the spatial representation you would like to create. It provides two options: `sp`, or `sf`. I suggest you always use `sf`, as `sp` will be deprecated in future versions.

2. Now, you have enough to draw your map with ggplot.

```r
# Load ggplot2
library(ggplot2)

# We use the Europe dataset that we loaded
map_1 <- ggplot(data = Europe) +
  # This is a spatial "sf" representation
  geom_sf() +
  # We want to zoom the map on Europe latitude and longitude
  coord_sf(xlim = c(-20, 40),
           ylim = c(35, 70))

# We plot the map
map_1
```

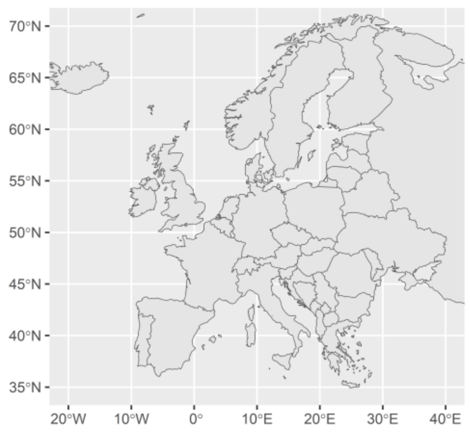


Figure 1: A simple map with polygons

- In the example above, we simply plot the country polygons that we loaded above. We can remove the coordinates background using `theme_void()`.

```
# We remove the axes from the map
map_1 + theme_void()
```

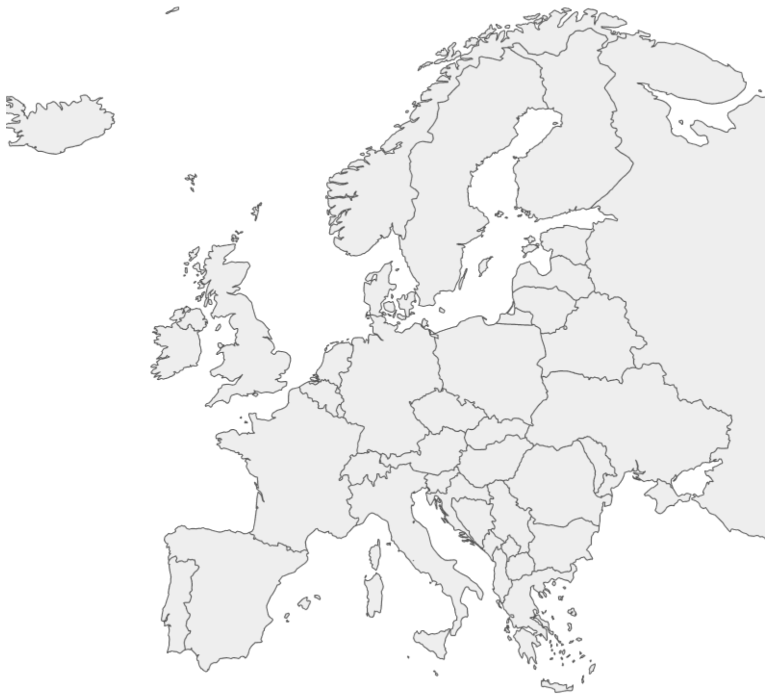


Figure 2: The map has no axes

Now, we have a template of Europe with gray colored countries. We may want to add colors and dots.

3. Let's add dots to represent the cities of Stockholm and Göteborg.

```
# Let's create a dataframe containing the coordinates of Stockholm and Göteborg
cities <- data.frame(city = c("Stockholm", "Göteborg"),
                     latitude   = c(59.334591, 57.708870),
                     longitude  = c(18.063240, 11.974560),
                     stringsAsFactors = FALSE)

# Let's add the points to the map
map_1 <- map_1 +
  geom_point(data = cities,
             mapping = aes(x = longitude, y = latitude),
             colour = "#EC3D2F",
             size = 3)

# Display the map
map_1
```

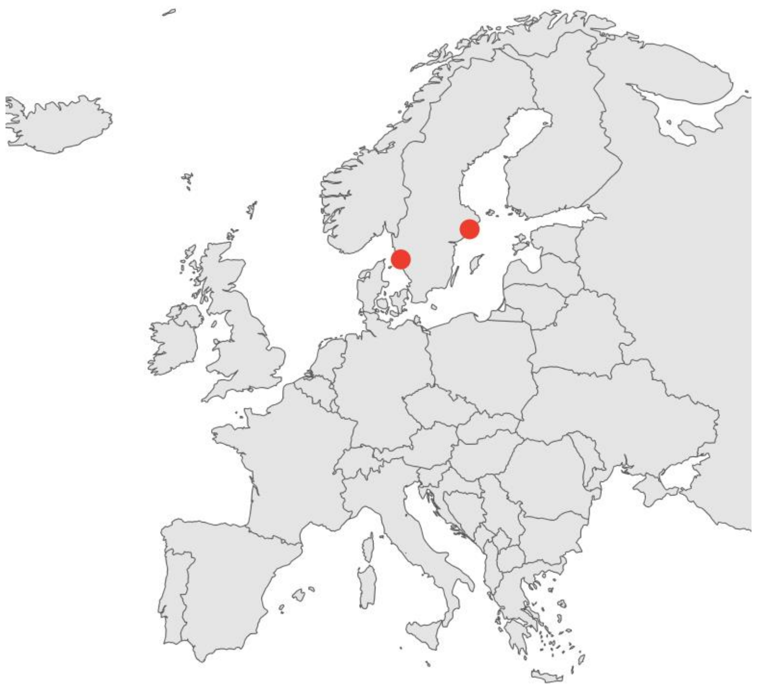


Figure 3: We add points to the map

- In the code above, we add `geom_point()` to the `map_1` graph. The data for the position of the points are in the dataframe cities. `x` corresponds to the `longitude` variable, and `y` to the `latitude`.
- We also choose a color and a size for the points. This is optional.

4. Now, let's imagine that you would like to color the country based on some country characteristics: level of hapiness, GDP, etc. you may simply add a column in the polygon dataset containing the value of the variable you want to depict.

WARNING: make sure your country (or city, or region) identifiers are spellt exactly the same as in the polygon dataset. Otherwise the `merge()` will create missing values.

```
# We have a country dataset measure the level of levelness
mydf <- data.frame(country   = c("Albania", "Austria", "Belgium", "Estonia",
                                 "Finland", "France", "Faeroe Is.",
                                 "United Kingdom", "Greece", "Sweden"),
                   levelness = c(1, 3, 4, 2, 6, 1, 2, 1, 3, 7))

# We merge mydf with the Europe polygon dataset
Europe <- merge(x = Europe, y = mydf,
                by.x = "name", by.y = "country",
                all.x = TRUE, all.y = FALSE)

# Warning: if a country is missing in our levelness dataset,
#  we still want to keep the polygon (hence, all.x = TRUE, all.y = FALSE)
```

5. Now, we can color the polygon with gradient

```
# We updated the Europe dataset. Let's reload the data in the graph
map_2 <- ggplot(data = Europe) +
  geom_sf() +
  coord_sf(xlim = c(-20, 40),
           ylim = c(35, 70)) +
  theme_void()

# Adding aestetics with filling on the map_2
map_2 <- map_2 + aes(fill = levelness)

# Display the map
map_2
```

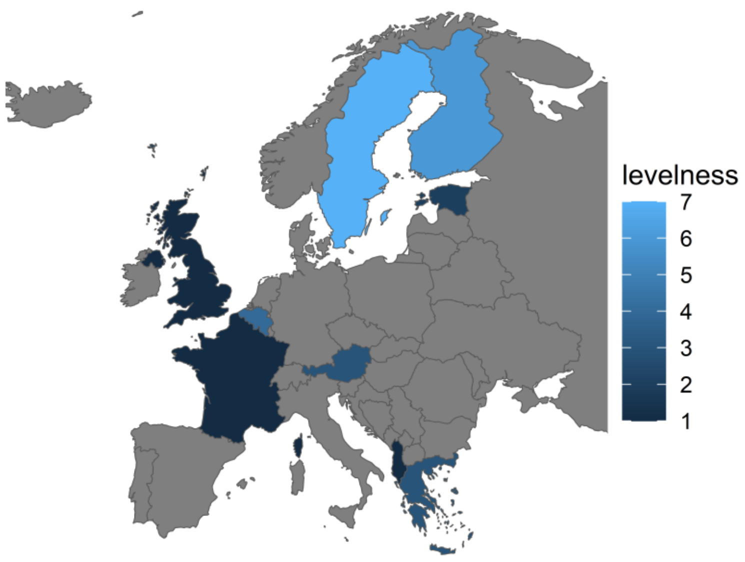


Figure 4: We add colors to the map

Remember that the variables defining the colors of the polygons need to be added to the polygon dataframe before the map is created. Otherwise, R will tell you that the variable does not exist.

You can then adjust the colors and choice of variable as you like, following what we have seen for ggplot2 customization during the module.

1.2 Interactive outputs with `shiny`

Shiny is a very powerful subset of tools, that allows creating webpages that include buttons, sliders, or text forms. To use shiny, you will have to create a specific shiny project (*i.e.*, this is not a normal R project).

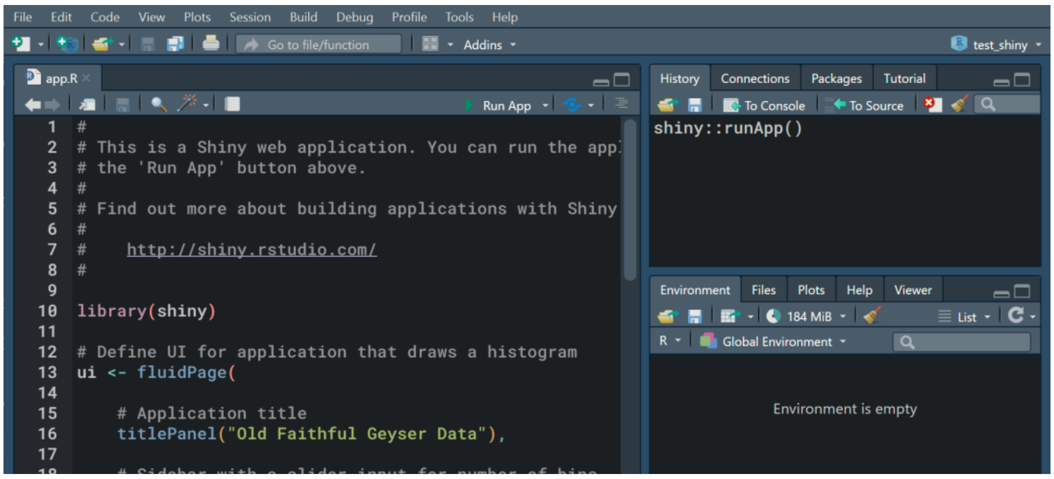
Shiny then contains two elements:

1. A frontend (*i.e.* the webpage), that contains the buttons and the outputs that you want in your interactive page.

2. A backend (*i.e.* a server part), that will analyze the request from the webpage and prepare the requested outputs. The R code that you use to type is in the backend. The frontend is mostly composed of html markup.

1.2.1 Create a shiny project

- To create a shiny app, go to **File** > **New project...** > **New directory** > **Shiny Application** and give it a name.
- Once you created the new project, a default file called "app.R" appears: this is a shiny app that you can run.



```r
#
# This is a Shiny web application. You can run the app
# the 'Run App' button above.
#
# Find out more about building applications with Shiny
#
#    http://shiny.rstudio.com/
#

library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

```

```
shiny::runApp()
```

Environment is empty

The shiny app

- To run the shiny app, click `Run app`. It will starts a server, and open the app in the web browser.

1.2.2 The Shiny UI

- The shiny app contains two parts. The first part is a `ui` function (_i.e. User Interface):

```r
# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30)
        ),

        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
    )
)
```

- The function contains the elements that you want to display on the page. The app above contains:

1. A title, created with `titlePanel()`
2. A slider input, created with `sliderInput()`
3. A plot output, created with `plotOutput()`

- The you see that the slider and the plot are respectively placed in a `sidebarPanel()` and a `mainPanel()`. If you run the shiny app, you will see the elements placed on your screen.
- Shiny apps can contains many other elements, but they are always placed on the screen using the same logic: invoking the name of the object with a function.
- The list of objects you can place are available here: https://shiny.rstudio.com/reference/shiny/1.6.0/

1.2.3 The Shiny Server

- The second important part of a shiny app is the **server**. This is what gets the inputs (such as the `sliderInput()` above) and generate the outputs (such as the `plotOutput()` above).
- The server side is also a function:

```r
# Define server logic required to draw a histogram
server <- function(input, output) {

    output$distPlot <- renderPlot({
        # Here, you get the data that you want to display:
        #   this could come from a data file.
        x    <- faithful[, 2]

        # Here, getting the input from the UI slider, you create
        #   a bin variable
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # draw the histogram with the specified number of bins
        #  --> This is the thing your renderPlot function returns
        hist(x, breaks = bins, col = 'darkgray', border = 'white')
    })
}
```

- The function contains two parameters: `input` and `output`. They are the two lists of inputs and outputs define in the UI. For example, in the UI, we created a `sliderInput()` called `"bins"`: it is accessible from the server side, as `input$bins`. In this case, as `bins` is a slider that is set to a value between 1 and 50, `input$bins` is a value between 1 and 50. Every time the slider is moved by the user, the value of `input$bins` is updated. This value can be used in any R code.
- The other parameter is `output`. This one expects element to render on the webpage. For example, the UI contains a plot called `"distPlot"` (see the UI code, above). This means that the server needs to render a plot and save it in the list of outputs, in an object called

`distPlot`: then, the UI will be able to access it. This is what we have in the code, with the function `renderPlot({})`, assigned to `output$distPlot`.

- Every time the user interface is updated (for example, the slider is moved), the functions of the servers are run again, and the new plot is rendered.

1.2.4 Deploy the app

- The app that you have designed is run locally, on your computer. This can be handy during presentations sometimes, but this is not enough to let other people interact with it.
- If you want to share it online, it is possible to deploy it on a web server. This requires some skills in web deployment. However, you can also find ready-made cloud solution to deploy your app. For example, RStudio allows you to deploy your app (*i.e.* make it accessible through the web browser of other people) on their website, here: https://www.shinyapps.io/
- Shinyapps.io is not free for large scale application, but you can test it and develop personal project with the free version.

1.2.5 Read more about `shiny`...

- The shiny developers created a complete course accessible for free, if you would like to learn more about it: https://shiny.rstudio.com/tutorial/

2. Final words, don't forget style and elegance

So far, we used short and easy examples. If you plan to continue using R, either for your thesis, or in your job, you will soon end up with more complex dataset, and more complex projects. In this final section, I give you some advice to improve the speed and usability of your code:

2.1 Advice on speed

2.1.1 Use vectors!

R is a vector language. Manipulate vectors, use vectors. To the contrary

2.1.2 Don't use loops!

In R, loops are very, **VERY** inefficient! If you have one million observations, it might take hours to run your loop. This does not work. R is designed to manipulate vectors. As much as you can, manipulate vectors. Examples: see module 5!

2.1.3 data.frame are slow!

- Accessing a `data.frame` takes time for R, especially if the `data.frame` contains hundreds of thousands of variables.
- If you have to access a data.frame multiple times for a specific variables (for example, in a loop that you cannot avoid), it is faster to extract the variable from the data.frame, compute it, and then save it back.
- **Example:** The code below is faster (and gets even faster compared to the other code provided if `df` contains many variables):

```r
var_1 <- df$var_1
for (i in 1:length(var_1)) {
  var_1[i] <- var_1[i] + 2
}
df$var_1 <- var_1
```

- than this:

```r
for (i in 1:length(var_1)) {
  df$var_1[i] <- df$var_1[i] + 2
}
```

- **Note** that both codes above make no sense: you should use vectors to compute var_1 in both cases.

2.2 Advice on memory

2.2.1 Mind you data types!

- If your computer has memory issues, you may want to reduce the size of objects. Not all objects in R are made equal! A `logical` takes much less space than a `character`. For example

```r
# A dummy variable with 200000 observations, NOT properly coded
student_level <- c(rep("undergrad.", 100000), rep("postgrad.", 100000))
object.size(student_level)
```

```
1600176 bytes
```

```r
# Now, the same information, as a logical variable
is_undergrad <- c(student_level == "undergrad.")
object.size(is_undergrad)
```

```
800048 bytes
```

- You will note that the `is_undergrad` object (logical) above is twice smaller than the `student_level` object (character) although they contain the same information.
- In general:
  - Using `factor` is cheaper in memory than using `characters`
  - Using `integer` (if possible) is cheaper than using `numeric`
  - `logical` is the cheapest (if possible)

2.2.2 Clean after you

- If you stored and created many objects in your environment, you may run out of memory. To get free space again, think of deleting the objects and freeing the memory.
- To delete an object

```r
rm(tmp)
```

- To collect garbage (= free memory that is not used anymore)

```r
gc(reset = TRUE)
```

```
          used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 1893760 101.2    3826000 204.4  1893760 101.2
Vcells 3416066  26.1    8388608  64.0  3416066  26.1
```

2.3 Advice on code

2.3.1 Dummy variable: be affirmative!

- It is often the case, when you see a dummy variable in dataset, that it is coded as 0, 1, with an obscure name (for example, `student_educ_level`) and a label indicating the levels (*e.g.* `bachelor`, `master`...). This is not ideal, because the variable name does not indicate what coding corresponds to 0 or to 1.
- You should use instead **affirmative** statements in the dummy variable names. For example, `is_bachelor_student` is better than a dummy `student_educ_level` variable, because the statement `is_bachelor_student` can be `TRUE` or `FALSE`. Student education level, on the contrary, cannot be `TRUE` or `FALSE`, hence not conveying the right information.

2.3.2 Use functions!

It is very often that you want to proceed to multiple steps in your analyses. The usual practice is simply to write hundreds of lines of codes in a row. This is a bad practice, because it makes it very difficult to track changes on your variables in the long term.

- If you want to perform any operation, such as, computing `var_3`, create a function for this:

```
compute_var3 <- function(var_1, var_2) {
  # Your code
}
```

- By doing so, you know exactly where is all the code related to the creation of your var_3 variable. You can easily run it independently, and even store it in a different file.

2.3.3 Separate data manipulation from data analyses!

- In general, when you analyze data, you should store the preparation of the dataset and the analysis in different files. This avoids having multiple version of the same variable, used in different analysis (for example, if you run a linear model with `var_3`, then make changes on `var_3`, then run another linear model, you will not be able to tell that `var_3` changed in the meantime by looking at the output).
- In any data analyses process, you should follow those steps:

1. Load the raw data
2. Process the data with any necessary steps (remove missing, compute new variables, etc.)
3. **Save the complete data manipulation in a new file, in `.Rds` format**
    1. Do not forget that other format may be lossy: information from data.frame may not be recorded properly in Stata files, Excel files, or in other file formats.
    2. If data are not anonymized, you should always prefer to anonymize or pseudonymize them before saving your cleaned data file. In case of pseudonymization, you should store the identification keys separately from the data.
    3. **DO NOT FORGET: NEVER UPLOAD A DATA FILE ON GITHUB!**

4. When creating the output and analysis, only use the cleaned file, in `.Rds` format, that you created: you should not change anything in your data anymore at this stage, to make sure everything is transparent and reproducible.
    1. Save ALL you analyses in R scripts or in Quarto files, so that you can reproduce them later, if required

3. Conclusion

This was only a short introduction on R. As every language, the best way now to move further is to start speaking… or writing in that case. And don't forget, if there is something you don't know how to do yet, the answer is probably on StackOverflow…