Desarrollo de aplicaciones web con Shiny

Máster en Data Science y Big Data en Finanzas (MDSF)

Máster en Data Science y Big Data (MDS)

Rocío Parrilla

rocio.parrilla@atresmedia.com

Enero 2022

Contenidos

1. La librería *shiny*.
    - Estructura de una aplicación shiny.
    - Interactividad.
    - Estilos.
2. Cuadros de mando con shinydashboard.
3. Mapas con *leaflet*.
4. Ejercicios.
5. Referencias.

1 | La librería shiny

Introducción

- shiny es una librería de R que permite el desarrollo de aplicaciones web (o, más concretamente, aplicaciones shiny).
- Una aplicación shiny es una página web mantenida por una sesión de R.
- Ejemplos:
  - Histograma de una distribución uniforme aquí.
  - k-means clustering aquí.
  - Word Cloud aquí.
  - Explorador de películas aquí.

Inputs y outputs

Lo más habitual es construir una aplicación shiny en torno a entradas (*inputs*) y salidas (*outputs*).

- Los *inputs* son elementos de la aplicación manipulables por el usuario y que proporcionan información a la aplicación.
  - *Sliders*.
  - Botones.
  - Desplegables.
  - Campos de texto.

- Los *outputs* son elementos visibles que pueden **reaccionar** a las interacciones del usuario con la aplicación.
  - Tablas.
  - Gráficos.

Programación reactiva

Dado el siguiente fragmento de código:

```
a <- 4
b <- a + 3
a <- 5
```

¿Cuál es el valor de `b` al final de su ejecución?

Programación reactiva

Dado el siguiente fragmento de código:

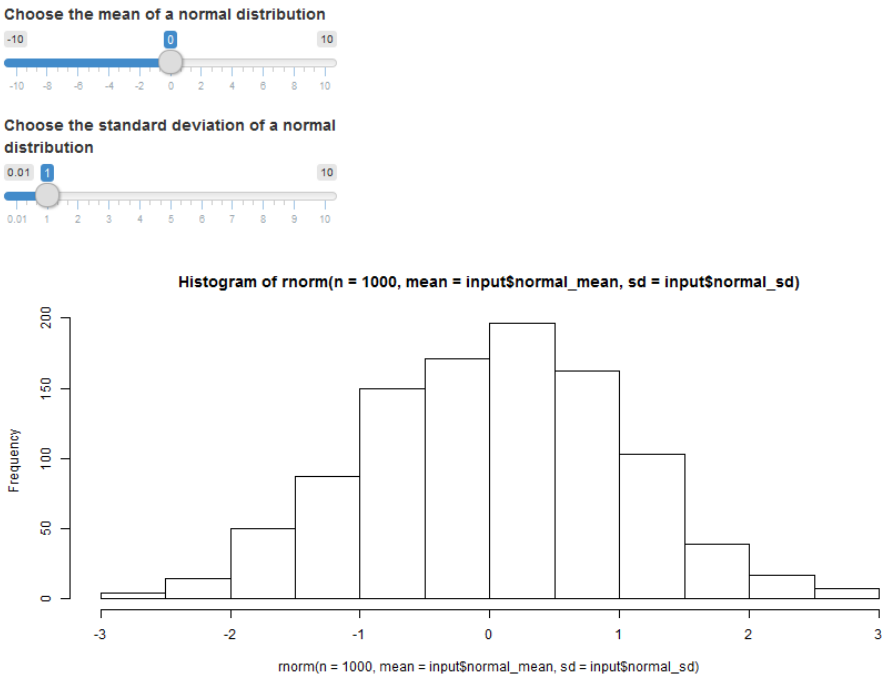
```
a <- 4
b <- a + 3
a <- 5
```

¿Cuál es el valor de `b` al final de su ejecución?

- En general, `b = 7`.
- Si `b` es una variable reactiva, la actualización de `a` desencadena una actualización en `b`, por lo que el valor final de `b` es `8`.
- El uso de *outputs* reactivos frente a cambios en los *inputs* es la base de funcionamiento de una aplicación `shiny`.

Una primera aplicación (I)

A modo de ejemplo, vamos a construir la siguiente aplicación:



Una primera aplicación (II)

La inmensa mayoría de aplicaciones shiny (y, al principio, es recomendable que todas) pueden construirse a partir de la siguiente plantilla (`shiny/00_template.R`):

```
library(shiny)
ui <- fluidPage(…)
server <- function(input, output) {…}
shinyApp(ui = ui, server = server)
```

Una primera aplicación (III)

- La instrucción `ui <- fluidPage(…)` define la interfaz de la aplicación: una página web *responsive* estructurada en base al *grid system* de Bootstrap3.

- `fluidPage` es una función. Sus parámetros son (esencialmente) los inputs y outputs que deben mostrarse. En nuestro ejemplo:

  - Un *slider* para elegir la media.
  - Un *slider* para elegir la desviación típica.
  - El histograma.

Una primera aplicación (IV)

- Los *inputs* se definen mediante *input functions*. La mayoría pueden encontrarse aquí, en el apartado “UI inputs”. Por ejemplo, para el primer *slider*, haríamos

```
sliderInput(
 inputId = 'normal_mean',
 label = 'Choose the mean of a normal distribution',
 min = -10, max = 10, value = 0
)
```

- Cada *input* tiene un `inputId` que debe ser único, en el que se almacenará el valor que el usuario introduzca en cada momento.

Una primera aplicación (V)

- Los *outputs* se definen mediante *output functions.* La mayoría pueden encontrarse aquí, en el apartado “UI outputs”. Por ejemplo, para el histograma, haríamos

```
plotOutput(
  outputId = 'normal_hist'
)
```

- Cada *output* tiene un `outputId` que debe ser único, al que deberemos asignar aquello que queramos mostrar en el espacio creado por el *output function* correspondiente.

- El código necesario para generar la interfaz de nuestra primera aplicación se encuentra en `shiny/01_easy_app_ui.R`.

Una primera aplicación (VI)

- Como hemos visto, la aplicación no muestra nada en el histograma (y esto es natural porque en ningún sitio le hemos dicho que lo haga).
- El servidor se encarga de enlazar *inputs* y *outputs* en nuestra aplicación.
- Tres reglas fundamentales:
  - Asignar aquellos objetos que quieran mostrarse en `output$...` .
  - Construir aquellos objetos que quieran mostrarse con *render functions* (la mayoría de ellas pueden consultarse aquí, en el apartado “rendering functions”.
  - Usar valores de entrada mediante `input$...` .

Una primera aplicación (VII)

- El código a ejecutar por el servidor en nuestra aplicación es

```
server <- function(input, output){

  output$normal_hist <- renderPlot(
    hist(rnorm(n = 1000, mean = input$normal_mean, sd = input$normal_sd))
  )
}
```

- ¡Ojo! output$normal_hist es una variable reactiva (y, necesariamente, tiene que serlo).

- El código completo de nuestra aplicación se encuentra en `shiny/02_easy_app.R`.

Compartir aplicaciones shiny

- Existen, en principio, dos posibilidades para compartir una aplicación shiny:
  - Usar shinyapps.io.
  - Construir un servidor shiny.
- Para publicar en shinyapps.io, la aplicación debe encontrarse en un fichero llamado `app.R`, y éste debe estar en el mismo directorio que todos los ficheros que se utilicen en la aplicación.
- Publicar en shinyapps.io es muy sencillo, y puede hacerse desde RStudio.
- shinyapps.io es gratis… hasta cierto punto.

¿Qué más podemos hacer?

- Personalizar la reactividad de los *outputs*.
  - Mostrar distintos tipos de *output.*
  - Decidir qué objetos son reactivos o no, y cuándo deben reaccionar.

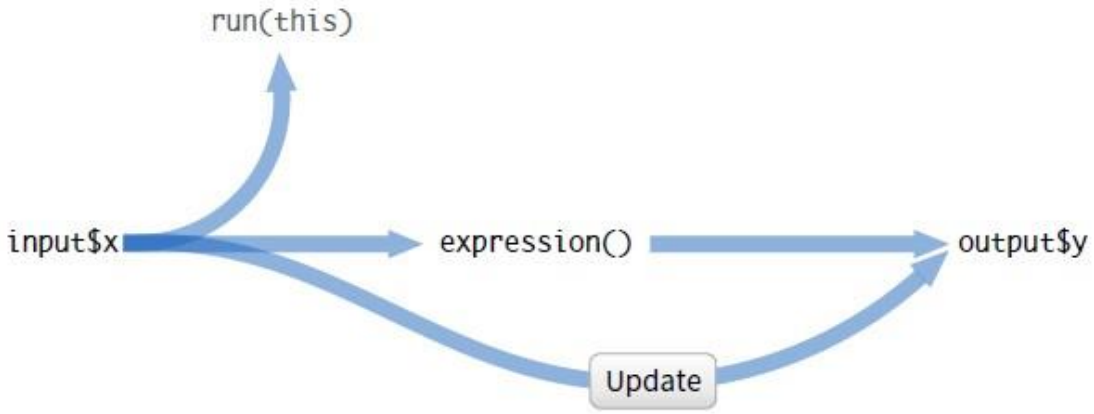
- Personalizar la apariencia.
  - Añadir contenido HTML.
  - Decidir qué espacio ocupará cada elemento.
  - Modificar el estilo visual de los elementos que se muestren.

Reactive functions

- Dependiendo del tipo de salida que pretenda mostrarse, es necesario utilizar una u otra *render function*:
  - renderTable()
  - renderDataTable()
  - renderImage()
  - renderPlot()
  - renderPrint()
  - renderText()
  - renderUI()

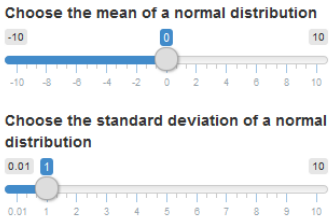
Personalización de reacciones

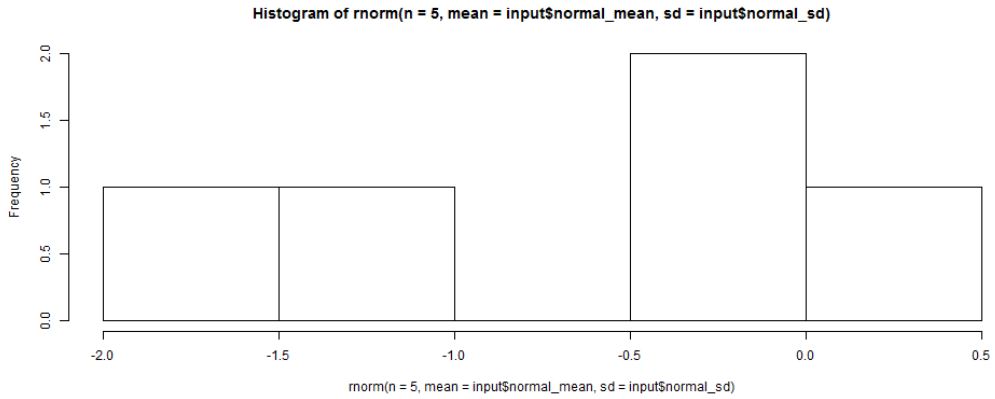
- La forma más sencilla de reactividad en una aplicación shiny consiste en que cambios en las entradas se propagan a las salidas y producen cambios sobre éstas.
- Hay cosas más complicadas que pueden hacerse:



Modularizar código con reactive() (I)

Empecemos por un ejemplo sencillo (`shiny/03_easy_app_two_outputs_wrong.R`).






```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.6430 -1.0010 -0.9961 -0.5362 -0.4533  1.4120
```

Modularizar código con reactive() (II)

- En la interfaz hemos definido dos salidas:

```
plotOutput(outputId = 'normal_hist'),
verbatimTextOutput(outputId = 'normal_summary')
```

- En el servidor las asociamos a las entradas:

```
output$normal_hist <- renderPlot(
        hist(rnorm(n = 5, mean = input$normal_mean, sd = input$normal_sd))
        )
output$normal_summary <- renderPrint(
        summary(rnorm(n = 5, mean = input$normal_mean, sd = input$normal_sd))
        )
```

- Esto está mal.

Modularizar código con reactive() (III)

Lo natural (`shiny/04_easy_app_two_outputs_wrong.R`) sería hacer lo siguiente:

```
x <- rnorm(n = 5, mean = input$normal_mean, sd = input$normal_sd)

output$normal_hist <- renderPlot(
  hist(x)
)

output$normal_summary <- renderPrint(
  summary(x)
)
```

Modularizar código con reactive() (IV)

Eso también está mal:

```
> shinyApp(ui = ui, server = server)

Listening on http://127.0.0.1:5785
Warning: Error in .getReactiveEnvironment()$currentContext: Operation not allowed without an active reactive context. (You tried to do some
thing that can only be done from inside a reactive expression or observer.)
Stack trace (innermost first):
    50: .getReactiveEnvironment()$currentContext
    49: .subset2(x, "impl")$get
    48: $.reactivevalues
    47: $
    46: rnorm
    45: server [#3]
     4: <Anonymous>
     3: do.call
     2: print.shiny.appobj
     1: <Promise>
Error in .getReactiveEnvironment()$currentContext() :
  Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive express
ion or observer.)
```

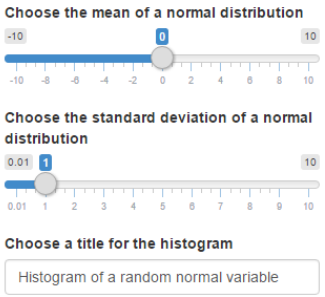
Modularizar código con reactive() (V)

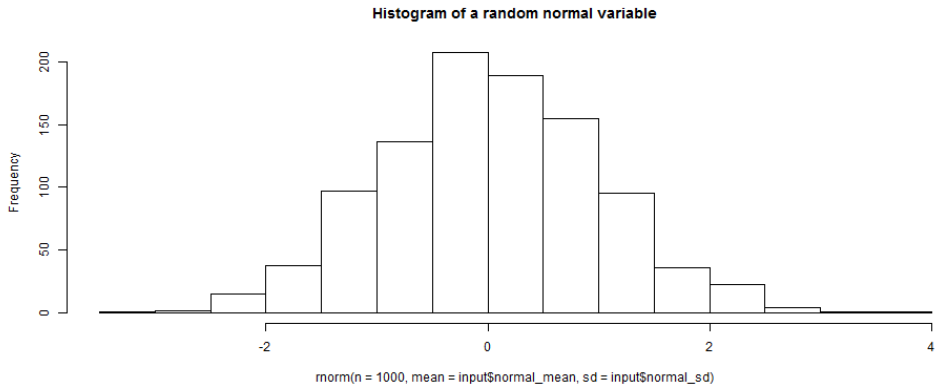
Lo correcto (shiny/05_easy_app_two_outputs.R) es definir x como una variable reactiva (o, más precisamente, como una función reactiva).

```
x <- reactive(rnorm(n = 5, mean = input$normal_mean, sd = input$normal_sd))

output$normal_hist <- renderPlot(
  hist(x())
)

output$normal_summary <- renderPrint(
  summary(x())
)
```

Retrasar reacciones con isolate() (I)

- En principio, un cambio en cualquiera de los *inputs* que intervienen en la construcción de un *output* desencadenan que la función que renderiza el output en el servidor vuelva a ejecutarse.
- Hay situaciones (`shiny/06_easy_app_title_wrong.R`) donde puede que no queramos que todos los *inputs* desencadenen reacciones.





Retrasar reacciones con isolate() (II)

- Con isolate(), se elimina una variable de entrada del flujo de actualización.
- Las salidas se refrescarán cuando cambios en el resto de *inputs* desencadenen el proceso reactivo.

```
output$normal_hist <- renderPlot(
  hist(rnorm(n = 1000, mean = input$normal_mean, sd = input$normal_sd),
    main = isolate(input$histogram_title))
)
```

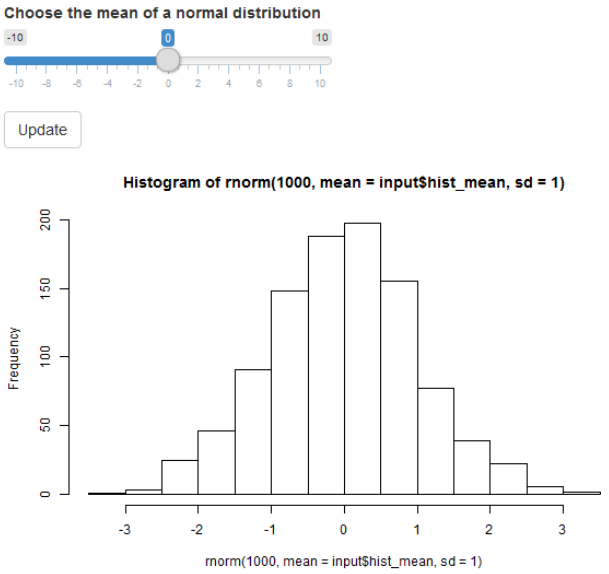
(Código completo en `shiny/07_easy_app_title.R`).

Retrasar reacciones con eventReactive() (I)

- En programación web, las reacciones se dan cuando se produce algún "evento".
  - Click.
  - Pasar el ratón por un área de la pantalla.
  - Introducir un texto.
  - …
- La forma general de programar esta reactividad es la siguiente:
  - Hay un "observador de eventos" que está continuamente en ejecución, a la espera de que se produzca tal evento.
  - Cuando el evento se produce, el observador hace que se ejecute el código que desencadena los cambios que tengan que producirse.
- Hasta ahora, los observadores de eventos han sido transparentes para nosotros.

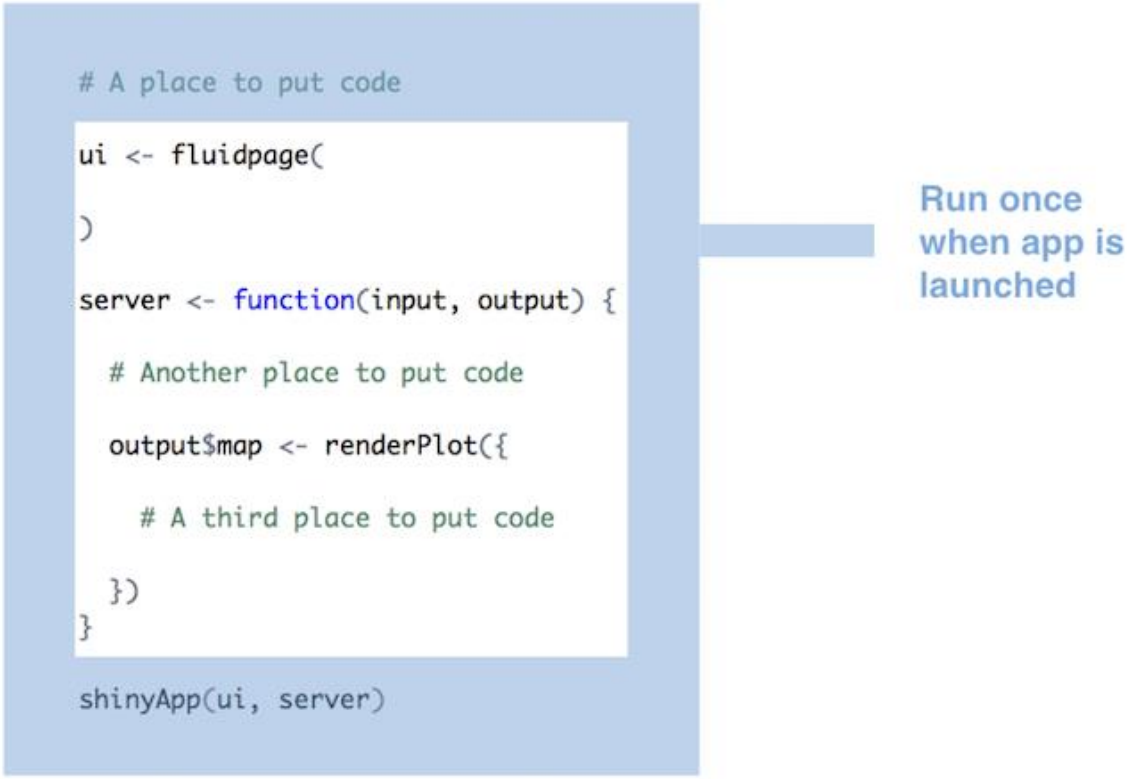
Retrasar reacciones con eventReactive() (II)

- Con eventReactive() podemos declarar expresiones reactivas ante valores concretos.
  - El ejemplo de referencia debe ser “actualizar datos cuando se clica un botón”.



(Código completo en `shiny/09_events_no.R` y `shiny/10_events.R`).

¿Dónde incluyo mi código en una app de shiny?




```
# A place to put code

ui <- fluidpage(

)

server <- function(input, output) {

  # Another place to put code

  output$map <- renderPlot({

    # A third place to put code

  })
}

shinyApp(ui, server)
```

¿Dónde incluyo mi código en una app de shiny?

```
# A place to put code

ui <- fluidpage(

)

server <- function(input, output) {

  # Another place to put code

  output$map <- renderPlot({

    # A third place to put code

  })
}

shinyApp(ui, server)
```

Run once each time a user visits the app

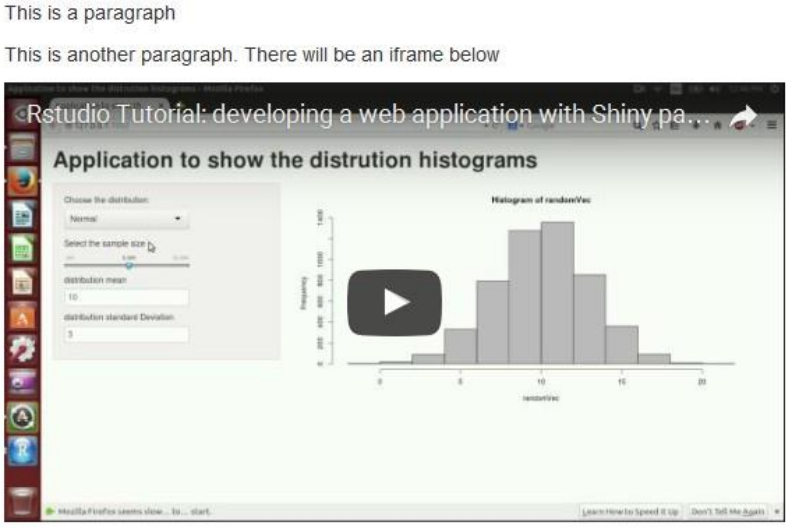
¿Dónde incluyo mi código en una app de shiny?

```
# A place to put code

ui <- fluidpage(

)

server <- function(input, output) {

  # Another place to put code

  output$map <- renderPlot({

    # A third place to put code

  })
}

shinyApp(ui, server)
```

Run once each time a user changes a widget that output$map depends on

Incluir código HTML

- Para ello, se puede usar la función `HTML()`, que toma como parámetro un *string* que contiene el código HTML que queremos incluir en la aplicación.



(Código en `shiny/12_html.R`).

Colocación de los objetos (I)

- Documentación general aquí.
- Se pueden emplear *layout functions* para posicionar los objetos que definamos en la página.
- El *layout* de una aplicación shiny está basado en el *grid system* de Bootstrap3.

.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1
.col-md-8								.col-md-4			
.col-md-4			.col-md-4					.col-md-4			
.col-md-6						.col-md-6					

Colocación de los objetos (II)

- Una página se compone de filas (*row*, fluidRow).
- Cada fila se compone de columnas (*col*, column).

```
ui <- fluidPage(
  fluidRow(
    column(width = 4, sliderInput(…), offset = 0),
    column(width = 4, sliderInput(…), offset = 4)
  ),
  fluidRow(
    column(width = 8, plotOutput(…), offset = 2)
  )
)
```

(Código completo en `shiny/13_layout.R`).

Paneles

Los paneles sirven para agrupar distintos elementos visuales.

- wellPanel() agrupa elementos dentro de un rectángulo sombreado.
- tabPanel() sirve para crear pestañas.
- tabsetPanel() y navlistPanel() sirven para decidir de qué forma se navega por las pestañas.

(Código en `shiny/14_tabsetpanel.R` y `shiny/15_navlistpanel.R`).

CSS (I)

- CSS (Cascading Style Sheets) es un lenguaje que sirve para modificar los aspectos visuales de elementos de una página web.

CSS (II)

- Ya hemos usado CSS:
  - fluidPage().
  - fluidRow(), column().
- Se puede incluir una hoja CSS para modificar los estilos del HTML “propio” que incluyamos.
- Más información aquí.

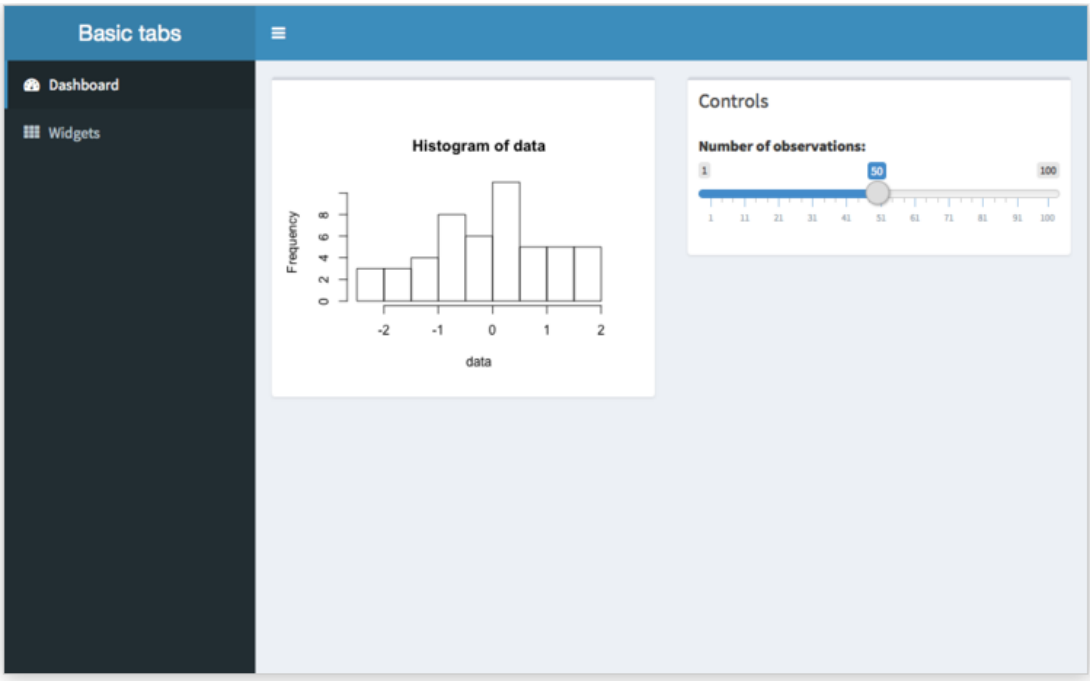
2 | Cuadros de mando con shinydashboard

Shinydashboard

- shinydashboard es un paquete de R que permite crear cuadros de mando interactivos, definiendo el *layout* apropiado para ello.
- Ejemplos:
  - Twin cities buses aquí.
  - Popularidad de paquetes de R aquí.

Estructura

- Un *dashboard* se compone de tres elementos:
  - Una cabecera (*header*).
  - Una barra lateral (*sidebar*).
  - Un cuerpo (*body*).



Plantilla

La mayoría de *dashboards* pueden crearse a partir de la siguiente plantilla (`dashboard/01_template.R`):

```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
      dashboardHeader(),
      dashboardSidebar(),
      dashboardBody())
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

Cabecera y *skins*

- Hay diversos temas (*skins)* que pueden especificarse para un *dashboard*, que le dan propiedades visuales globales.
- Generalmente, la cabecera se emplea para mostrar el título de la aplicación y poco más. Para que sea visualmente atractivo, se ha de ajustar el ancho del título con el de la barra lateral.

```
ui <- dashboardPage(
  skin = 'purple',
  dashboardHeader(title = 'Probability distributions',
                  titleWidth = 250),
  dashboardSidebar(width = 250),
  dashboardBody()
)
```

(Código completo en `dashboard/02_title.R`).

Sidebar

- La barra lateral se emplea, frecuentemente, como un menú donde se pueden seleccionar distintas pestañas que aparecerán en el cuerpo.
- Para esto, se ha de definir un sidebarMenu. Cada uno de sus menuItems deberá referenciar (por nombre) a la pestaña a la que corresponda.

```
dashboardSidebar(width = 250,
                 sidebarMenu(
                   menuItem('Normal', tabName = 'Normal'),
                   menuItem('Uniform', tabName = 'Uniform')
                 ))
```

(Código completo en `dashboard/03_menu.R`).

Pestañas

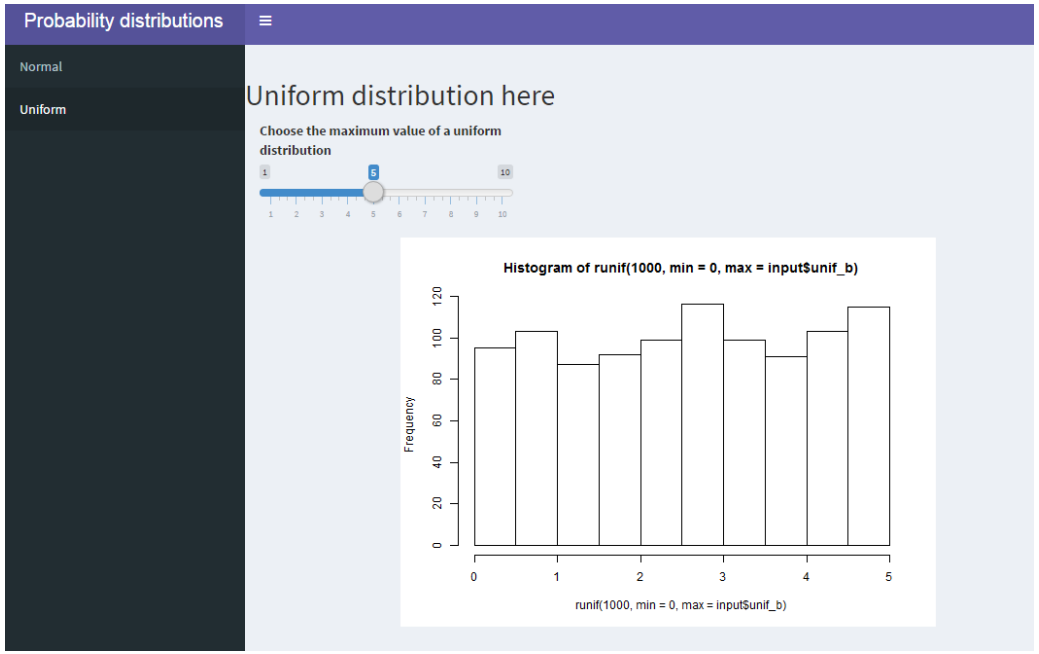
- Para incluir una lista de pestañas en el cuerpo, debe incluirse un `tabItems`.
- El elemento `tabItems` estará compuesto por distintos `tabItem`, cada uno de los cuales representa una pestaña, que deberá enlazarse por nombre al `menuItem` correspondiente.

```
dashboardBody(
        tabItems(
          tabItem(tabName = 'Normal',
                h2('Normal distribution here')),
          tabItem(tabName = 'Uniform',
                h2('Uniform distribution here'))
        )
      )
```

(Código completo en `dashboard/04_tabs.R`).

Una aplicación completa

- Todo lo que hemos hecho hasta ahora se refiere únicamente a la interfaz.
- ¡El servidor no se entera del *layout*!



(Código completo en `dashboards/05_full_app.R`).

3 | Mapas con leaflet

leaflet

- Leaflet es una librería *open-source* de Javascript para diseñar mapas interactivos.
- leaflet es una librería de R que *encapsula* la funcionalidad de Leaflet de modo que se puedan diseñar mapas utilizando *únicamente* código R.
- Estos mapas pueden
  - Visualizarse en el propio RStudio.
  - Incluirse en documentos RMarkdown (y derivados).
  - Incluirse en aplicaciones shiny.

Uso básico

- El diseño de un mapa con leaflet se basa en los siguientes cuatro pasos:
  1. Crear el mapa con leaflet().
  2. Añadir una capa con addTiles(), addMarkers() o addPolygons().
  3. Repetir el punto 2 cuantas veces sea necesario.
  4. Pintar el mapa.

```
library(leaflet)

map <- leaflet() %>% addTiles() %>%
        addMarkers(lng = -3.6878, lat = 40.4309, popup = 'Afi Escuela de Finanzas')

map
```

(Código en `leaflet/01_afi.R`).

Visualización inicial

- Mediante `setView()` podemos fijar el centro y el nivel de zoom al que se visualiza el mapa de inicio.
- `fitBounds()` permite definir longitud y latitud mínima y máxima que se muestra en la primera visualización del mapa.
- `addTiles(urlTemplate = …)` permite definir el mapa de *background* (aquí hay un listado con numerosos mapas de background disponibles).

(Código en `leaflet/02_afi_marcos.R` y `leaflet/03_basemaps.R`).

Markers (I)

- Los *markers* son elementos que se colocan en determinados puntos del mapa.
- Ya hemos colocado algunos fijando las coordenadas, pero lo interesante es pasar un fichero de datos con coordenadas, y colocar un punto por cada fila, en el lugar que corresponda.
- La función leaflet() toma un parámetro data que hace este papel, y puede ser
  - Un data.frame con puntos (longitud, latitud).
  - Elementos del paquete sp (SpatialPoints, SpatialLines, SpatialPolygons).
  - El data.frame devuelto por la función map() del paquete maps.
  - …

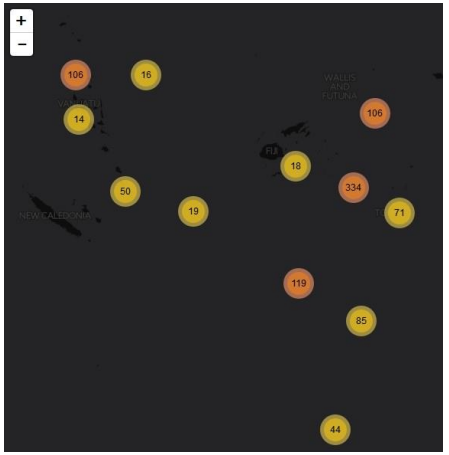
Markers (II)

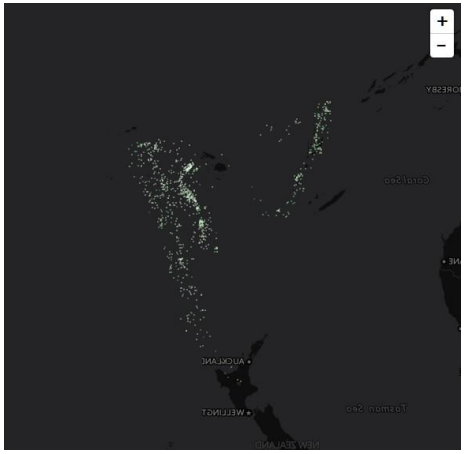
- El set de datos quakes.

	lat	long	depth	mag	stations
1	-20.42	181.62	562	4.8	41
2	-20.62	181.03	650	4.2	15
3	-26.00	184.10	42	5.4	43
4	-17.97	181.66	626	4.1	19
5	-20.42	181.96	649	4.0	11
6	-19.68	184.31	195	4.0	12
7	-11.70	166.10	82	4.8	43
8	-28.11	181.93	194	4.4	15
9	-28.74	181.74	211	4.7	35
10	-17.47	179.59	622	4.3	19
11	-21.44	180.69	583	4.4	13
12	-12.26	167.00	249	4.6	16

Markers (III)

`leaflet/04_markers.R`





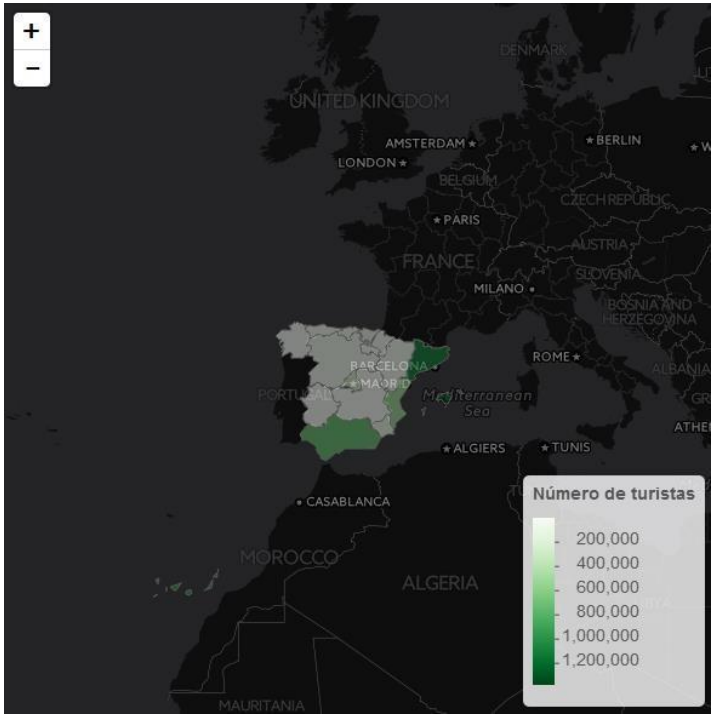
GeoJSON (I)

- GeoJSON es un formato para codificar estructuras de datos geográficos.
  - Point.
  - Polygon.
  - MultiPolygon.
- A los objetos geométricos con propiedades adicionales se les llama Feature, y un GeoJSON suele estar formado por un FeatureCollection.

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [125.6, 10.1]
  },
  "properties": {
    "name": "Dinagat Islands"
  }
}
```

GeoJSON (II)

- leaflet también soporta datos en formato GeoJSON.
  `leaflet/06_geojson.R`



Ejemplo: shiny + leaflet

`leaflet/07_leaflet_in_shiny_app_interactive.R`

Ejemplo: shiny + leaflet

`leaflet/07_leaflet_in_shiny_app_interactive.R`

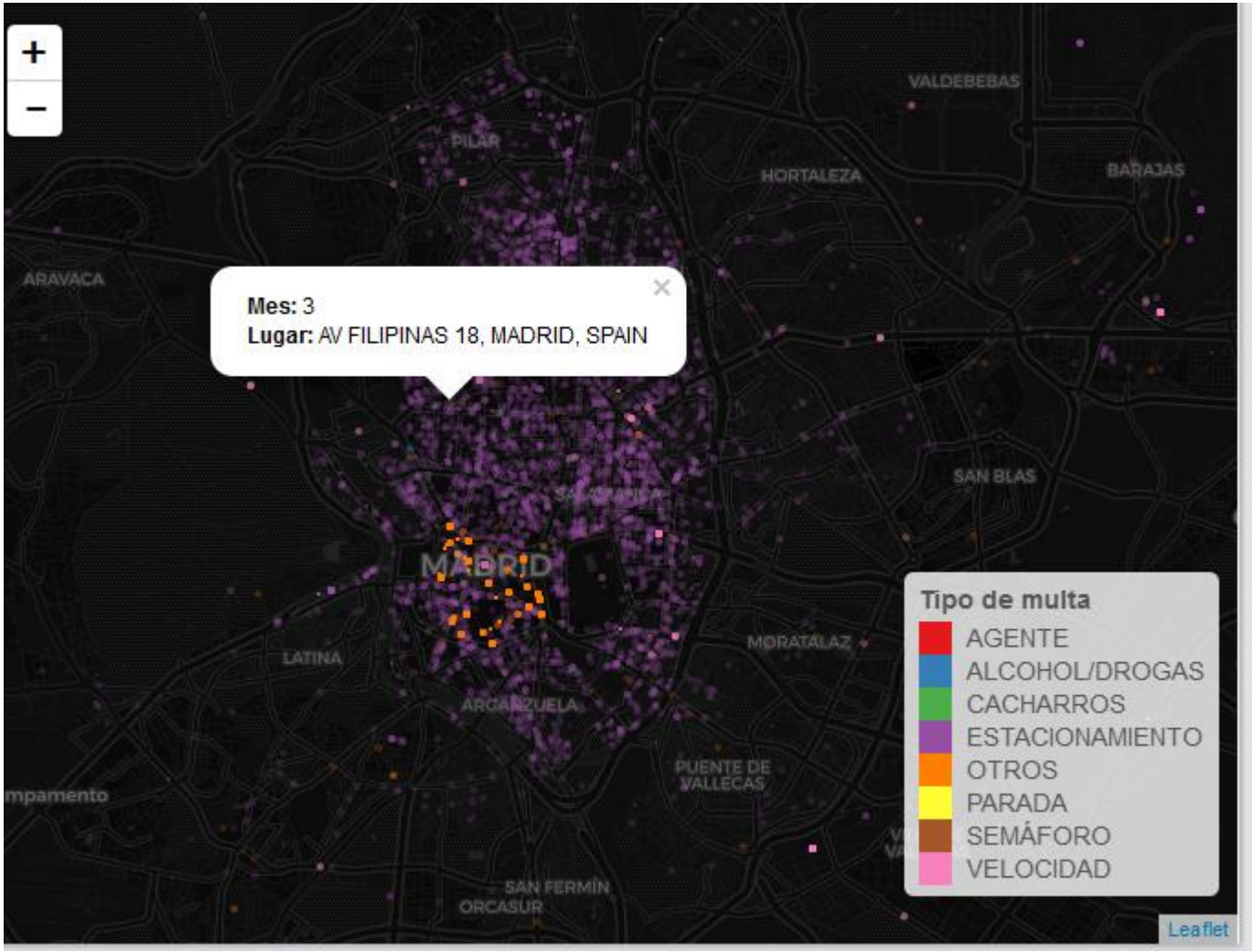
Mapas con plotly

La librería plotly también nos proporciona herramientas para visualizar mapas interactivos. Para ello usamos la función *plot_geo.*

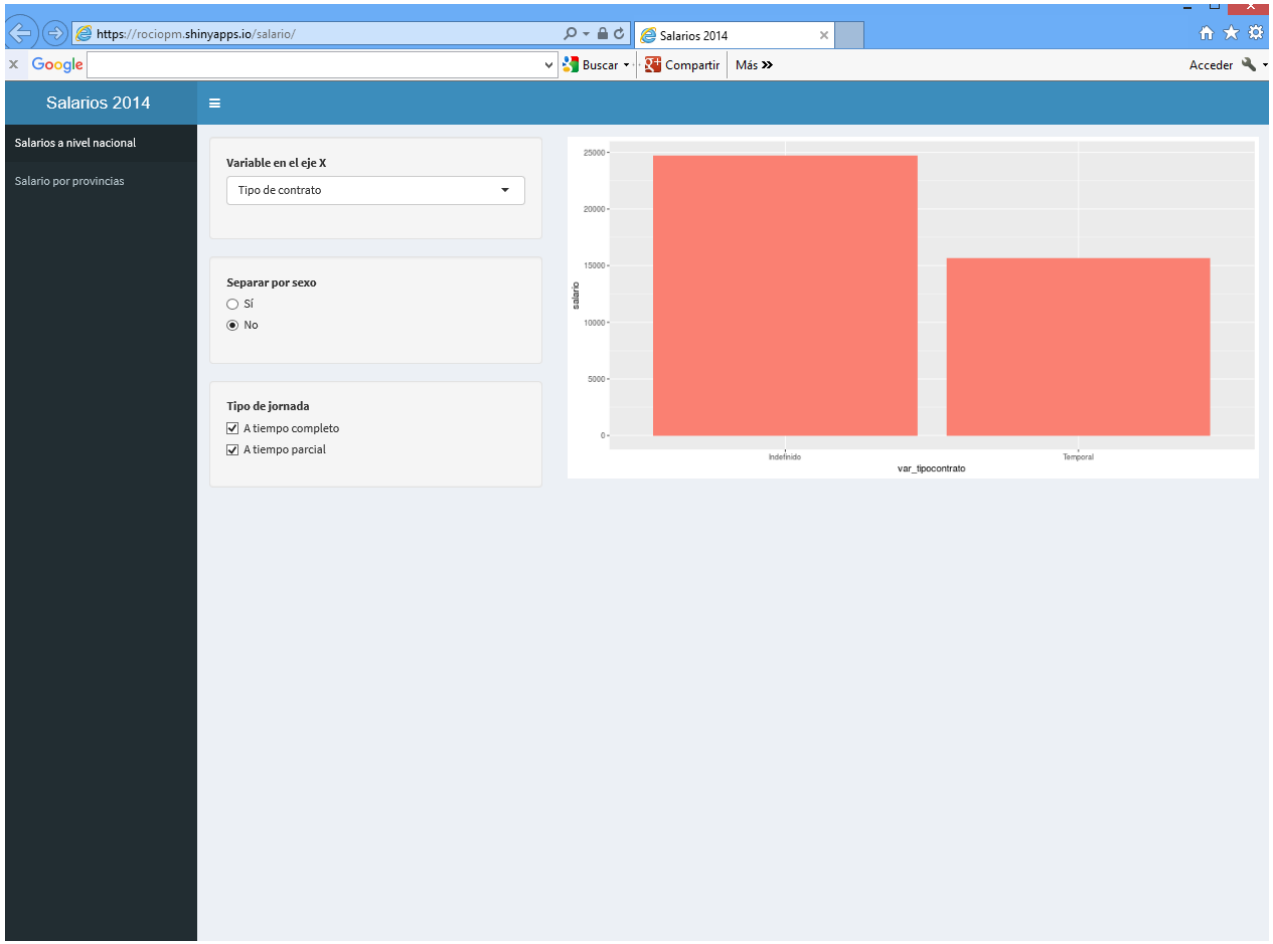
Ver esto.

4 | Ejercicios

Reproduce el siguiente mapa con *Leaflet*:



Ejercicio Shiny



5 | Referencias

Referencias

- Webinars de RStudio.
- Tutoriales de Shiny
- Documentación de leaflet para R.

Afi