Introduction to Shiny

February, March 2022

What and Why Shiny?

- Shiny is another R extension created to deploy `R` procedures through a web page
- It starts a local web engine at 127.0.0.0 or localhost
- Why: Users cannot run or manage your R script, and they provide relevant input and collect the relevant output.

Features

- **Reactive Programming**: The program is waiting for something happening from user actions or from whatever event generated in some way. It is the paradigm of *event programming* (different from console input).
- Shiny allows to collect input and output of an R procedure:
  - think about regression:
    - input X, Y
    - output $Y|X$
- If not installed in your R ecosystem (not the case on actual R versions), type `install.packages(shiny)`.

What's a Shiny App

- A Shiny App is a web page:
- local if you run locally on your machine;
- on a server, if you publish it on
  https://www.shinyapps.io
  (or other places);
- This web page is controlled by a Shiny server that does something based on the interaction on the User Interface (UI) page;
- Users interact with the UI, and the server part does calculations and provides output to be placed on the UI page.

Basic structure

- Here is an example of UI (empty) and server.
- These are the main arguments of a Shiny App:

```
library(shiny)

ui = fluidPage()

server = function(input, output) {
}

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

. . . where

- **ui**: implements the UI;
- **server**: implements the *server side*;
- **shinyApp**: merges both.

Implementation on separate files:

- You can have only one file `app.R` or two files with UI and server separately: **ui.R** y **server.R**:
- **ui.R**:
  `fluidPage()`
- **server.R**:
  `function(input, output){}`
- **NOTE**: there is no need to make a call to `shinyApp()`
- both `UI.R` and `server.R` files are in the same folder

Shiny Hello World

UI Side

```
library(shiny)

# Defines la UI
ui = shinyUI(fluidPage(
 titlePanel("Hello World!"),

  sidebarLayout(
  sidebarPanel(
      sliderInput("obs",
                  "Sample size:",
                  min=1, max=1000, value=500)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  ) ))
```

Server-side

```
server = shinyServer(function(input, output) {

    output$distPlot = renderPlot({
        dist = rnorm(input$obs)
        hist(dist, xlab = "samples", ylab = "Frequency", col = "violet",
            main = "Histogram")
    })
})

# ..................................

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

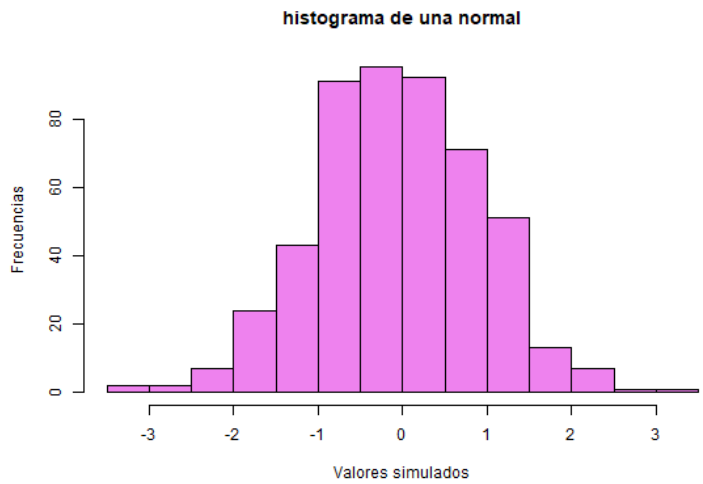
it should be like this (try)

Hola Shiny :-)

Numero de observaciones:

1 500 1,000

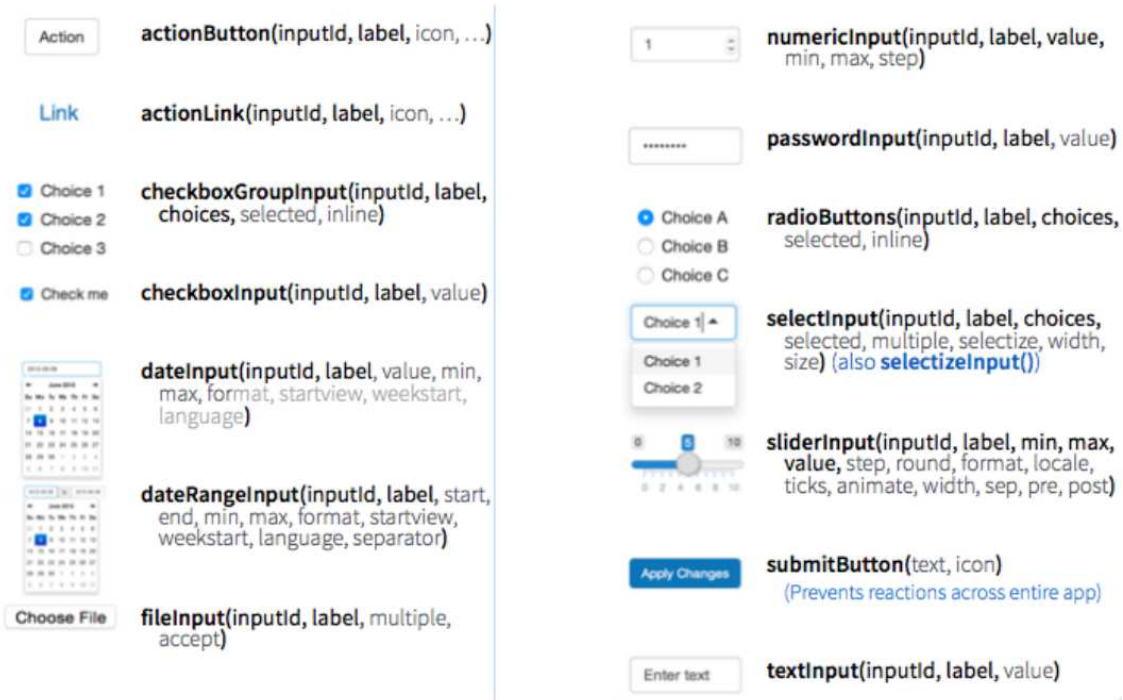
1 101 201 301 401 501 601 701 801 901 1,000



Main elements in the UI

- Shiny implements this common elements of many web pages
  - *for input*: Text field, RadioButtons, Sliders, SelectButtons, ...
  - *for output*: the same as in RMD (html_documents): tables, graphs, etc. ...
- you can use CSS, Javascript and R (you would prefer to use this latter) to customize the above elements,

Type of Input elements



Type of output elements

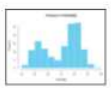
DT::**renderDataTable(expr**, options, callback, escape, env, quoted**)**

works with

dataTableOutput(outputId, icon, …**)**

renderImage(expr, env, quoted, deleteFile**)**

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**

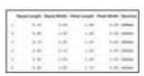


renderPlot(expr, width, height, res, …, env, quoted, func**)**

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**

renderPrint(expr, env, quoted, func, width**)**

verbatimTextOutput(outputId)



renderTable(expr,…, env, quoted, func**)**

tableOutput(outputId)

foo

renderText(expr, env, quoted, func**)**

textOutput(outputId, container, inline**)**

renderUI(expr, env, quoted, func**)**

uiOutput(outputId, inline, container, …**)**
& **htmlOutput(outputId**, inline, container, …**)**

Box-Plot example

```
library(shiny)

ui = fluidPage (
titlePanel(title='Boxplot'),

sidebarLayout(
sidebarPanel(
numericInput(inputId='n', 'Sample size', value=10),
           textInput(inputId='titulo', 'Title' ,'Boxplot of'),
               radioButtons(inputId='color',
                    'Color', list('Blue','Green'),
                                 'Green'),
                 submitButton('Apply/Run/Submit')),
    mainPanel(
      plotOutput(outputId='box'))))
```

Box-Plot code

```
server = function(input, output) {
    output$box = renderPlot({
        boxplot(rnorm(input$n), col = input$color, main = input$titulo,
            xlab = "Data")
    })
}

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

. . . it should appear as. . .

Boxplot al dente

Tamaño muestral

100

Titulo de

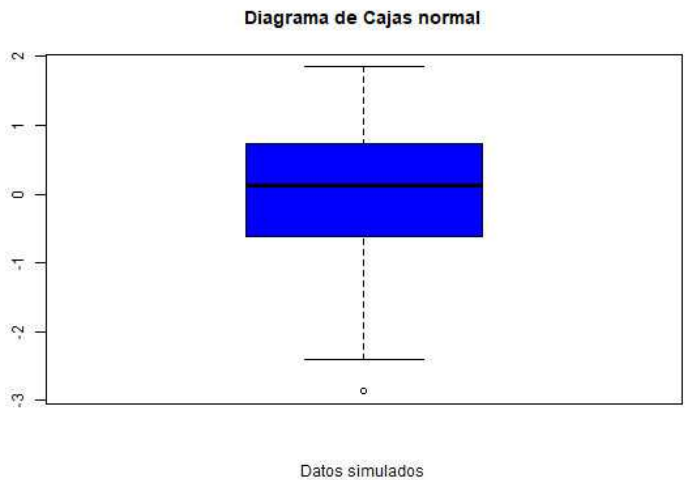
Diagrama de Cajas normal

Elige ...

Blue

Green

Aplica cambios



Beta-Binomial shiny app (you already saw this in Bayesian Analysis)

UI side

```
library(latex2exp)

ui = fluidPage(
titlePanel("Beta-Binomial model"),

sidebarLayout(
 sidebarPanel(
  sliderInput("alfa", "alfa:", min=0.01, max=20,
              value=0.5, step=0.1),
  sliderInput("beta", "beta:", min=0.01, max=20,
              value=0.5, step=0.1),
  sliderInput("y", "y:", min=0, max=20,
              value=1, step=1),
  sliderInput("n", "n:", min=1, max=40,
              value=2, step=1)
      ),
```

Server-side (calculus)

```
    mainPanel(plotOutput("distPlot") )
   ) )

# Defines el server
server = function(input, output){

 output$distPlot = renderPlot( {
  xs = seq(1e-2, 1-1e-2, length=100)

    dprior = dbeta(xs,
              shape1=input$alfa, shape2=input$beta)

      dpost = dbeta(xs, shape1=input$alfa + input$y,
              shape2=input$beta + input$n - input$y)
```

Server-side (output)

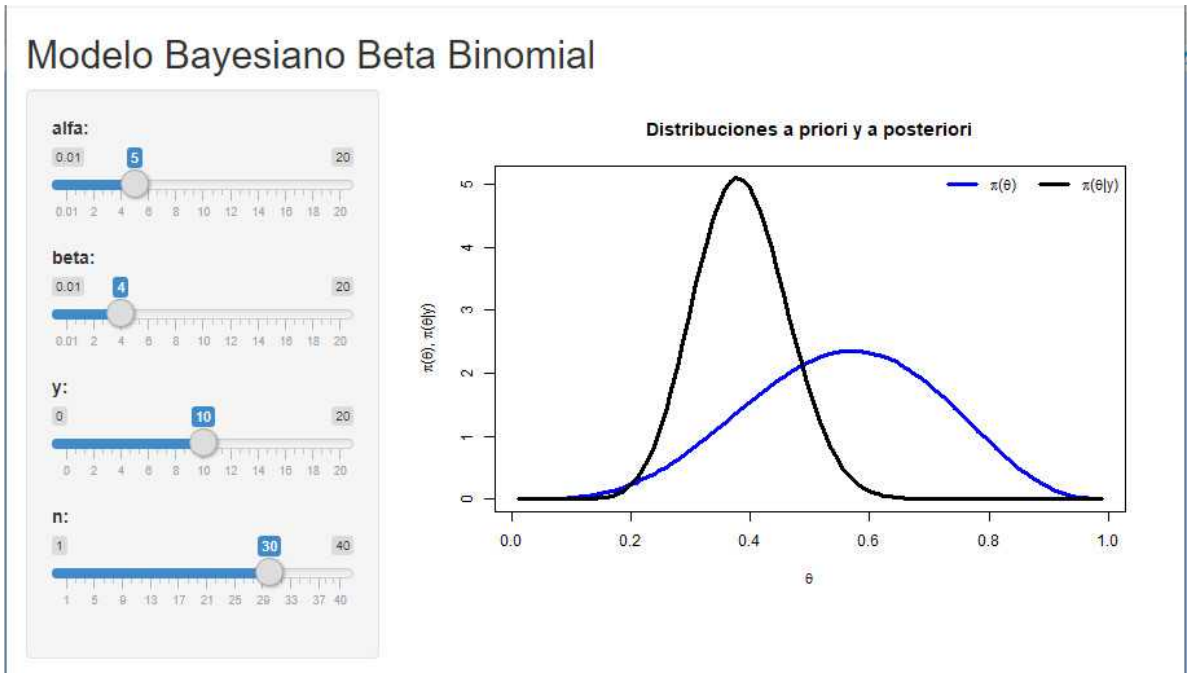
```
plot(xs, dprior, col="blue", lwd=3,
  main="Distribuciones a priori y a posteriori",
  xlab=expression(theta),
  ylab=TeX("$\\pi(\\theta)$,$\\pi(\\theta | y)$"),
  ylim=range(c(dprior, dpost)), type="l")

  lines(xs, dpost, lwd=3)

  legend("topright", c(TeX("$\\pi(\\theta)$"),
     TeX("$\\pi(\\theta | y)$")),
     col=c("blue","black"), lwd=3, bty="n", ncol=2)
      } ) }

# Run Application
shinyApp(ui=ui, server=server)
```

it should be like this. . .



Reactivity: Concept

- This concept is related to a reaction to some event.
- These types of functions `render*()`, es decir, `renderText()`, `renderPlot()`, `renderTable()`.
- are reactive to *changes in the objects that they use inside*
- shiny take track of changes in these objects;
- changes can be due to some user input or some modification of later finished calculations.

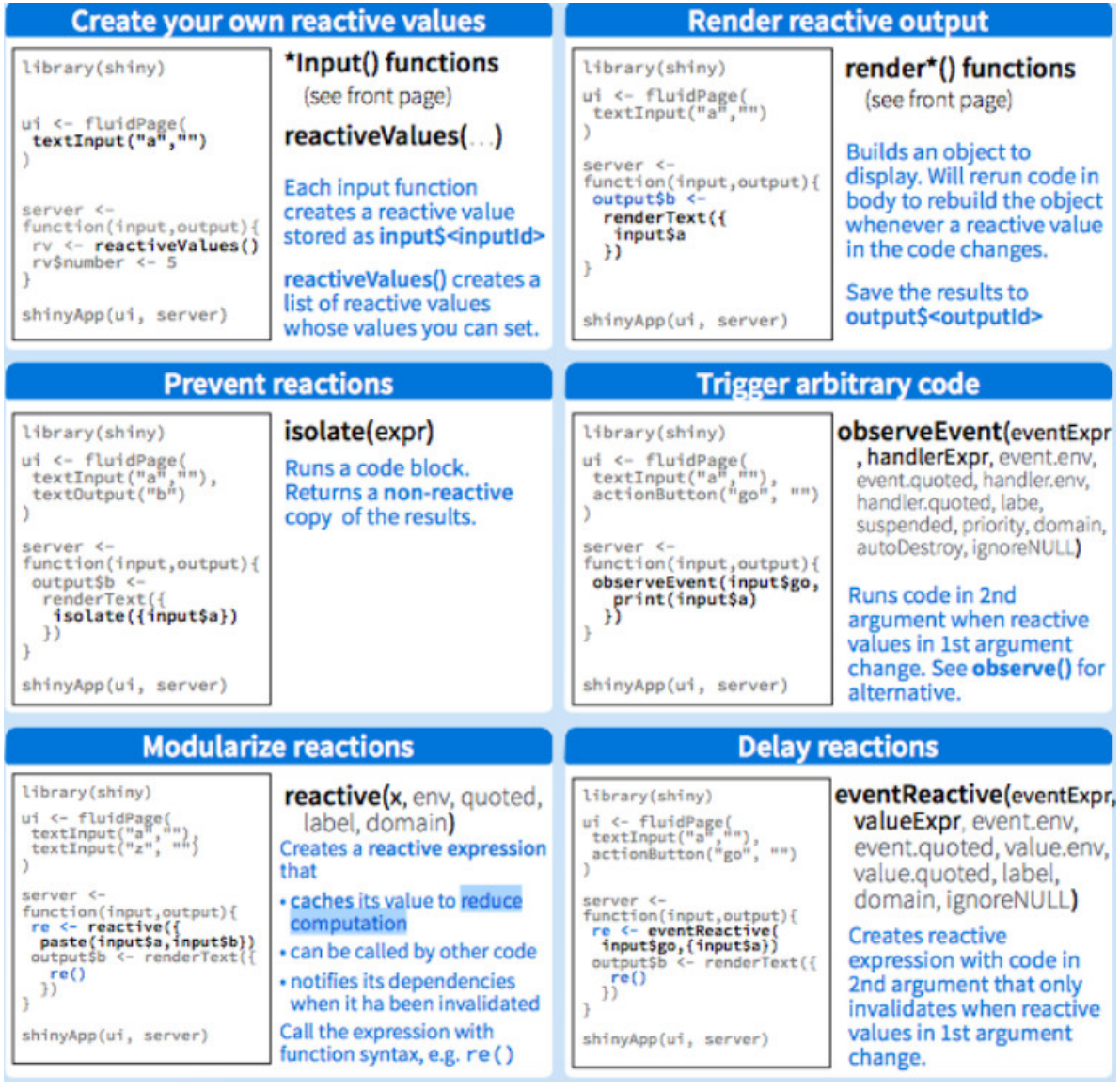
Reactivity: practical implementation

- keep in mind that if you want reactivity from user input, you have to wait to finish all calculations: sometimes these are quick,
- but it may be not always the case: use a button submit.
- Generally, we can define other reactive functions using `reactive()`.

Features of reactivity

- **Functionality**: function is run whenever something pointed as reactive changes. Such that Shiny is monitoring its change:
- be careful with loops!
- think about the event before writing the code; otherwise, the app could end up being too sl*ooo. . . oo*w..
- **Usability**: with reactivity, you are building functions of functions implicitly without explicit calls or reimplements them inside others;
- **Advantage**: no changes no function re-run.

A scheme of reactive functions



Examples of Shiny app we could expect from your project...

- One from Ernesto Rogado (Graduate student) that implemented a statistical analysis http://appstatistics.shinyapps.io/tfgshiny
- Other from me: Student Evaluation: https://evaluationuc3m.shinyapps.io/abilityevaluation/

Some links

- Shiny on CRAN: cran.r-project.org/web/packages/shiny/index.html
- Shiny Home: www.rstudio.com/shiny
- Tutorial: rstudio.github.io/shiny/tutorial/#welcome
- CheatSheet: resources.rstudio.com/spanish-pdfs/shiny-spanish
- Mastering Shiny: mastering-shiny.org
- A public server: www.shinyapps.io
  (see how to publish a Shiny App)