Basic reactivity

Capítulo 3: Reactividad básica

- **Idea clave**: Especificar un gráfico de dependencias para que, al cambiar una entrada, todas las salidas relacionadas se actualicen automáticamente.

1. **Función `server`**: Explicación de los argumentos `input` y `output`.

2. **Reactividad básica**: Conexión directa entre entradas y salidas.

3. **Expresiones reactivas**: Eliminar trabajo duplicado y optimizar el código.

La función server

- El `ui` es **simple** porque todos los usuarios ven el mismo HTML.
- No cambia entre sesiones.

1. **`server`**:
    - Se ejecuta **cada vez que un nuevo usuario inicia una sesión**.
    - Cada usuario obtiene una **versión independiente** de la aplicación.
    - Shiny invoca la función `server` automáticamente al iniciar una sesión.

Las función del `server` toman tres parámetros: `input`, `output`, y `session`

La función server

```
library(shiny)
ui <- fluidPage(
  # front end interface
)
server <- function(input, output, session) {
  # back end logic
}
shinyApp(ui, server)
```

Input

El argumento `input` es un objeto tipo lista, que contiene todos los datos de entrada enviados desde el navegador, nombrados según el ID de entrada.

```
ui <- fluidPage(
  numericInput("count", label = "Number of values", value = 100)
)
```

Input

los objetos `input` son de solo lectura.

Si intenta modificar una entrada dentro de la función del `server`, recibirá un error:

```
ui <- fluidPage(
  numericInput("count", label = "Number of values", value = 100)
)

server <- function(input, output, session) {
  input$count <- 10
}

shinyApp(ui, server)
```

Input

Para leer desde un `input`, debes estar en un **contexto reactivo** creado por una función como -`renderText()` -reactive()`.

Un “contexto reactivo” es un entorno donde la aplicación está atenta a los cambios en los datos de entrada y reacciona automáticamente cuando algo cambia.

```
server <- function(input, output, session) {
  message("The value of input$count is ", input$count)
}

shinyApp(ui, server)
#> Error: Can't access reactive value 'count' outside of reactive consumer.
#> ℹ Do you need to wrap inside reactive() or observer()?
```

Output

- Es similar a `input`: es un objeto en forma de lista cuyos nombres dependen del **ID de salida**.
- **Diferencia clave**:
  - `input` recibe datos del usuario.
  - `output` envía datos de salida a la interfaz.
- Siempre se usa junto con una **función de renderizado**, como en este ejemplo:

Output

```
library(shiny)
ui <- fluidPage(
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText("Hola RLadies!!")
}
shinyApp(ui, server)
```

Output

Hola RLadies!!

Función render

1. **Establecen un contexto reactivo**:
   - Rastrean automáticamente las entradas que utiliza la salida.
2. **Convierten el código R en HTML**:
   - Transforman la salida de R en un formato adecuado para mostrarse en una página web.
3. **Uso estricto**:
   - Al igual que `input`, el objeto `output` debe usarse correctamente dentro de funciones de renderizado.

Aparecera un error si: Olvidaste la `render`función.

Sin función render

```
server <- function(input, output, session) {
  output$greeting <- "Hola RLadies"
}
shinyApp(ui, server)
```

Con función render

```
library(shiny)
ui <- fluidPage(
  textOutput("greeting")
)
server <- function(input, output, session) {
  output$greeting <- renderText("Hola RLadies")
}
shinyApp(ui, server)
```

Con función render

Hola RLadies

Programación reactiva

- Una aplicación será bastante aburrida si solo tiene entradas o solo salidas.
- La verdadera magia de Shiny surge cuando tienes una aplicación con ambas.

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)
server <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Hello ", input$name, "!")
  })
}
shinyApp(ui, server)
```

Programación reactiva

What's your name?

Hello !

Programación imperativa vs. declarativa

- Programación Imperativa:
  - Das órdenes específicas que se ejecutan de inmediato.
  - Común en *scripts* de análisis: cargar datos, transformarlos, visualizarlos y guardar resultados.
- Programación Declarativa:
  - Describes objetivos o restricciones de alto nivel.
  - El sistema decide cómo y cuándo ejecutarlos.
  - Shiny usa este estilo: defines “recetas” y Shiny las ejecuta cuando es necesario.

Laziness (o “evaluación perezosa”)

Significa que las expresiones reactivas **solo se ejecutan cuando realmente se necesitan**.

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)
server <- function(input, output, session) {
  output$greting <- renderText({
    paste0("Hello ", input$name, "!")
  })
}
shinyApp(ui, server)
```

- Escribí `greting`en lugar de `greeting`.
- Esto no generará un error en Shiny, pero no hará lo que quieres.
- output$greting no existe, por lo que el código que contiene`renderText()`nunca se ejecutará.

Si está trabajando en una aplicación Shiny y no puede entender por qué su código nunca se ejecuta, verifique que las funciones de su interfaz de usuario y del server estén usando los mismos identificadores.

El gráfico reactivo

- La **laziness** de Shiny tiene otra característica clave.
- En R tradicional, el orden de ejecución se entiende leyendo el código de arriba a abajo.
- En Shiny, esto no aplica, ya que el código solo se ejecuta cuando es necesario.
- Para entender el flujo, debes analizar el **gráfico reactivo**, que muestra cómo se relacionan las entradas y salidas.

El gráfico reactivo

1. El **gráfico reactivo** usa símbolos para representar entradas y salidas, conectándolos cuando una salida accede a una entrada.

2. Este gráfico muestra que `greeting` se recalculará cada vez que `name` cambie.

3. Esta relación se describe como una **dependencia reactiva**: `greeting` depende de `name`.

Expresiones reactivas

La expresión reactiva reduce la duplicación en su código reactivo al añadir nodos adicionales al gráfico reactivo.

Agregamos una para que pueda ver cómo afecta al gráfico reactivo.

```
server <- function(input, output, session)
            {string <- reactive(paste0("Hello",
            input$name, "!"))
            output$greeting <- renderText(string()) }
```

Orden de ejecución

- En Shiny, el **orden de ejecución** no depende del orden de las líneas de código, sino del **gráfico reactivo**.
- Esto es diferente al código R tradicional, donde el orden de ejecución sigue el orden en que están escritas las líneas.
- Por ejemplo, en una función `server`, puedes invertir el orden de dos líneas sin afectar el resultado, ya que Shiny sigue las dependencias reactivas.

```
server <- function(input, output, session) {
          output$greeting <- renderText(string())
          string <- reactive(paste0("Hello ", input$name, "!")) }
```

Orden de ejecución

1. Parecería un error, ya que `output$greeting` depende de un *string* reactivo que aún no existe.

2. Shiny es “lazy”: el código se ejecuta al iniciar la sesión, cuando el string ya existe.

3. Este código produce el mismo gráfico reactivo, con el mismo orden de ejecución.

4. El orden de ejecución del código reactivo lo define el gráfico reactivo, no su ubicación en la función.

Ej 1 server 1

Dada esta interfaz de usuario: Corrija los errores simples encontrados en cada una de las tres funciones del `server` a continuación.

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server1 <- function(input, output, server) {
  input$greeting <- renderText(paste0("Hello ", name)) }

shinyApp(ui, server1)
```

`output$greeting` debe usarse dentro de `output$`, no en `input$`

Ej 1 server 1 corregido

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server1 <- function(input, output, server) {
  output$greeting <- renderText(paste0("Hello ", input$name)) }

shinyApp(ui, server1)
```

Ej 1 server 1 corregido

What's your name?

Hello

Ej 1 server 2

```
ui <- fluidPage(   textInput("name", "What's your name?"),
                   textOutput("greeting") )

server2 <- function(input, output, server)
{greeting <- paste0("Hello ", input$name)
  output$greeting <- renderText(greeting) }

shinyApp(ui, server2)
```

- El tercer argumento en server debe llamarse session, no server.
- greeting <- paste0(“Hello”, input$name) falla porque input name se usa fuera de un contexto reactivo.
- En Shiny, input$ debe usarse dentro de una función reactiva (reactive() o renderText()).

Ej 1 server 2 corregido

```
library(shiny)

ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server2 <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Hello ", input$name)
  })
}

shinyApp(ui, server2)
```

Ej 1 server 2 corregido

What's your name?

Hello

Ej 1 server 3

```
library(shiny)

ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server3 <- function(input, output, server) {
  output$greting <- paste0("Hello", input$name)
}

shinyApp(ui, server3)
```

- la variable definida en `ui` es `greeting` (falta una “e” en `greeting`).
- `output$greeting` debe ser una función `renderText({ ... })`

Ej 1 server 3 corregido

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server3 <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Hello, ", input$name)
  })
}

shinyApp(ui, server3)
```

Ej 1 server 3 corregido

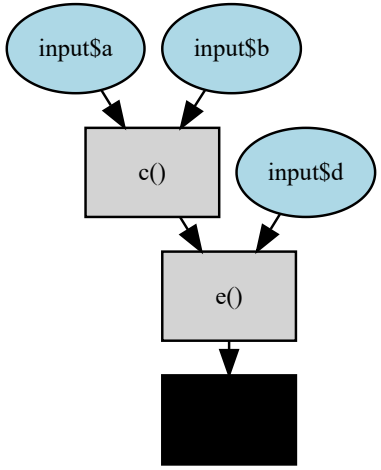
What's your name?

Hello,

Ej 2 Server 1

Dibujar el gráfico reactivo para los siguientes `server`:

```
server1 <- function(input, output, session) {
  c <- reactive(input$a + input$b)
  e <- reactive(c() + input$d)
  output$f <- renderText(e())
}
shinyApp(ui, server1)
```



Ej 2 server 1 errores

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)
server1 <- function(input, output, session) {
  c <- reactive(input$a + input$b)
  e <- reactive(c() + input$d)
  output$f <- renderText(e())
}
shinyApp(ui, server1)
```

- Faltan los inputs `input$a`, `input$b`, y `input$d` en la UI
- No se ha definido la salida `output$greeting`
- Uso incorrecto del `output$f`

Ej 2 server 1 corregido

```
library(shiny)
ui <- fluidPage(
  textInput("name", "What's your name?"),
  numericInput("a", "Valor A", value = 1),
  numericInput("b", "Valor B", value = 1),
  numericInput("d", "Valor D", value = 1),
  textOutput("greeting")  # Aquí debe coincidir con el nombre en el server
)
server1 <- function(input, output, session) {
  # Definir las funciones reactivas correctamente
  c <- reactive(input$a + input$b)
  e <- reactive(c() + input$d)

  # Mostrar el saludo con el nombre ingresado
  output$greeting <- renderText({
    paste0("Hello ", input$name, " - Resultado: ", e())
  })
}
shinyApp(ui, server1)
```

Ej 2 server 1 corregido

What's your name?

Valor A

1

Valor B

1

Valor D

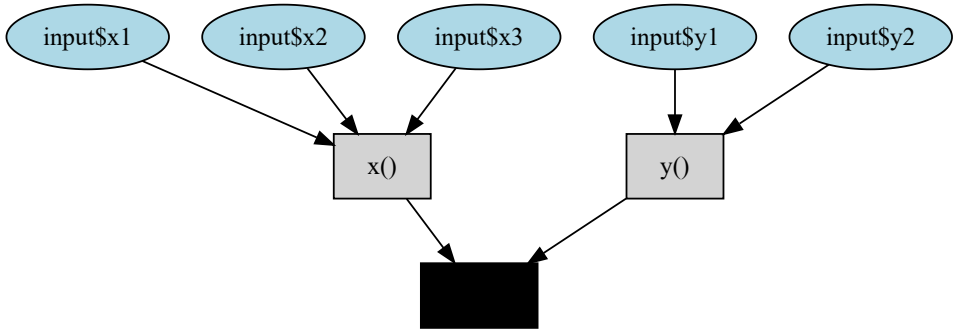
1

Hello - Resultado: 3

Ej 2 server 2

```
server2 <- function(input, output, session) {
  x <- reactive(input$x1 + input$x2 + input$x3)
  y <- reactive(input$y1 + input$y2)
  output$z <- renderText(x() / y())
}

shinyApp(ui, server2)
```



Ej 2 server 2 errores

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)
server2 <- function(input, output, session) {
  x <- reactive(input$x1 + input$x2 + input$x3)
  y <- reactive(input$y1 + input$y2)
  output$z <- renderText(x() / y())
}
shinyApp(ui, server2)
```

- Falta de los inputs `x1`, `x2`, `x3`, `y1`, `y2` en la UI:
- Salida `output$z` no está definida correctamente en la UI:
- Error en la interacción entre inputs y outputs:

Ej 2 server 2 corregido

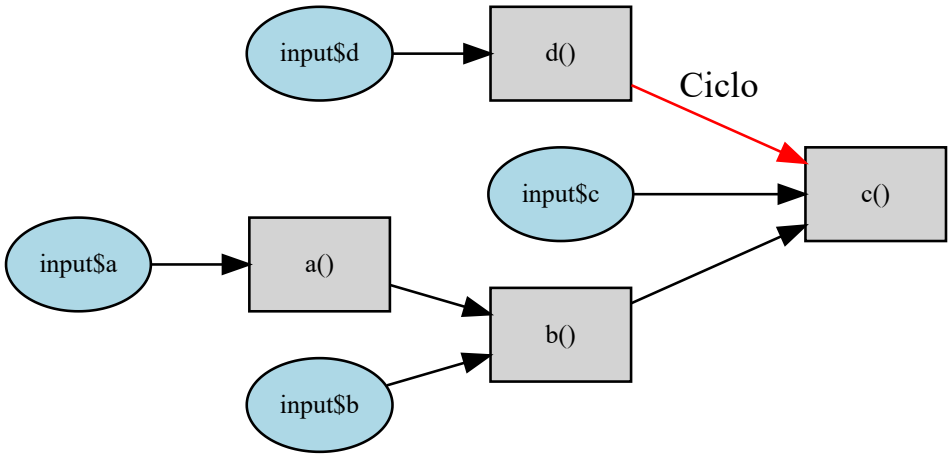
```
library(shiny)

ui <- fluidPage(
  textInput("name", "What's your name?"),
  numericInput("x1", "Valor X1", value = 1),
  numericInput("x2", "Valor X2", value = 1),
  numericInput("x3", "Valor X3", value = 1),
  numericInput("y1", "Valor Y1", value = 1),
  numericInput("y2", "Valor Y2", value = 1),
  textOutput("greeting"),
  textOutput("z")
)
server2 <- function(input, output, session) {

  x <- reactive(input$x1 + input$x2 + input$x3)
  y <- reactive(input$y1 + input$y2)
  output$greeting <- renderText({
  paste0("Hello ", input$name)
  })
   output$z <- renderText({
    paste0("Resultado de la división: ", x() / y())
  })
}
shinyApp(ui, server2)
```

Ej 2 server 3

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)
server3 <- function(input, output, session) {
  d <- reactive(c() ^ input$d)
  a <- reactive(input$a * 10)
  c <- reactive(b() / input$c)
  b <- reactive(a() + input$b)
}
shinyApp(ui, server3)
```



Ej 2 server 3 errores

El código, estás creando funciones reactivas que se refieren entre sí de forma circular:

- `d` depende de `c()`, pero `c` depende de `b()`, y `b` depende de `a()`.
- `a()` depende de `input$a`, y luego `b()` depende de `a()` y así sucesivamente.
- Uso de `c()` en vez de `input$c`:

Ej 2 server 3 corregido

```
  ui <- fluidPage(
  textInput("name", "What's your name?"),
  numericInput("a", "Input A", value = 1),
  numericInput("b", "Input B", value = 1),
  numericInput("c", "Input C", value = 1),
  numericInput("d", "Input D", value = 1),
  textOutput("greeting")
)
server3 <- function(input, output, session) {
  a <- reactive(input$a * 10)
  b <- reactive(a() + input$b)
  c <- reactive(b() / input$c)
  d <- reactive(c() ^ input$d)
  output$greeting <- renderText({
    paste0("Hello, ", input$name, "!")
  })
}
shinyApp(ui, server3)
```

Ej 2 server 3 corregido

What's your name?

Input A

1

Input B

1

Input C

1

Input D

1

Hello, !

¿Por qué fallará este código?

```
var <- reactive(df[[input$var]])
range <- reactive(range(var(), na.rm = TRUE))
```

- ¿Por qué hay nombres `range()` `var()` para `reactive`?
- nombrar las funciones reactivas con el mismo nombre que las funciones estándar de R

Ejercicio de Motivación

Se quiere comparar dos conjuntos de datos simulados con un gráfico y una prueba de hipótesis. Se ha realizado algunos experimentos creado las siguientes funciones:

- `freqpoly()` visualiza las dos distribuciones con polígonos de frecuencia.
- `t_test()` utiliza una prueba t para comparar medias y resume los resultados con una cadena:

Ejercicio de Motivación

```
library(ggplot2)
freqpoly <- function(x1, x2, binwidth = 0.1, xlim = c(-3, 3)) {
  df <- data.frame(
    x = c(x1, x2),
    g = c(rep("x1", length(x1)), rep("x2", length(x2)))
  )
  ggplot(df, aes(x, colour = g)) +
    geom_freqpoly(binwidth = binwidth, size = 1) +
    coord_cartesian(xlim = xlim)
}


t_test <- function(x1, x2) {
  test <- t.test(x1, x2)
  sprintf(
    "p value: %0.3f\n[%0.2f, %0.2f]",
    test$p.value, test$conf.int[1], test$conf.int[2]
  )
}
```

Ejercicio de Motivación

Si tengo algunos datos simulados, puedo usar estas funciones para comparar dos variables:

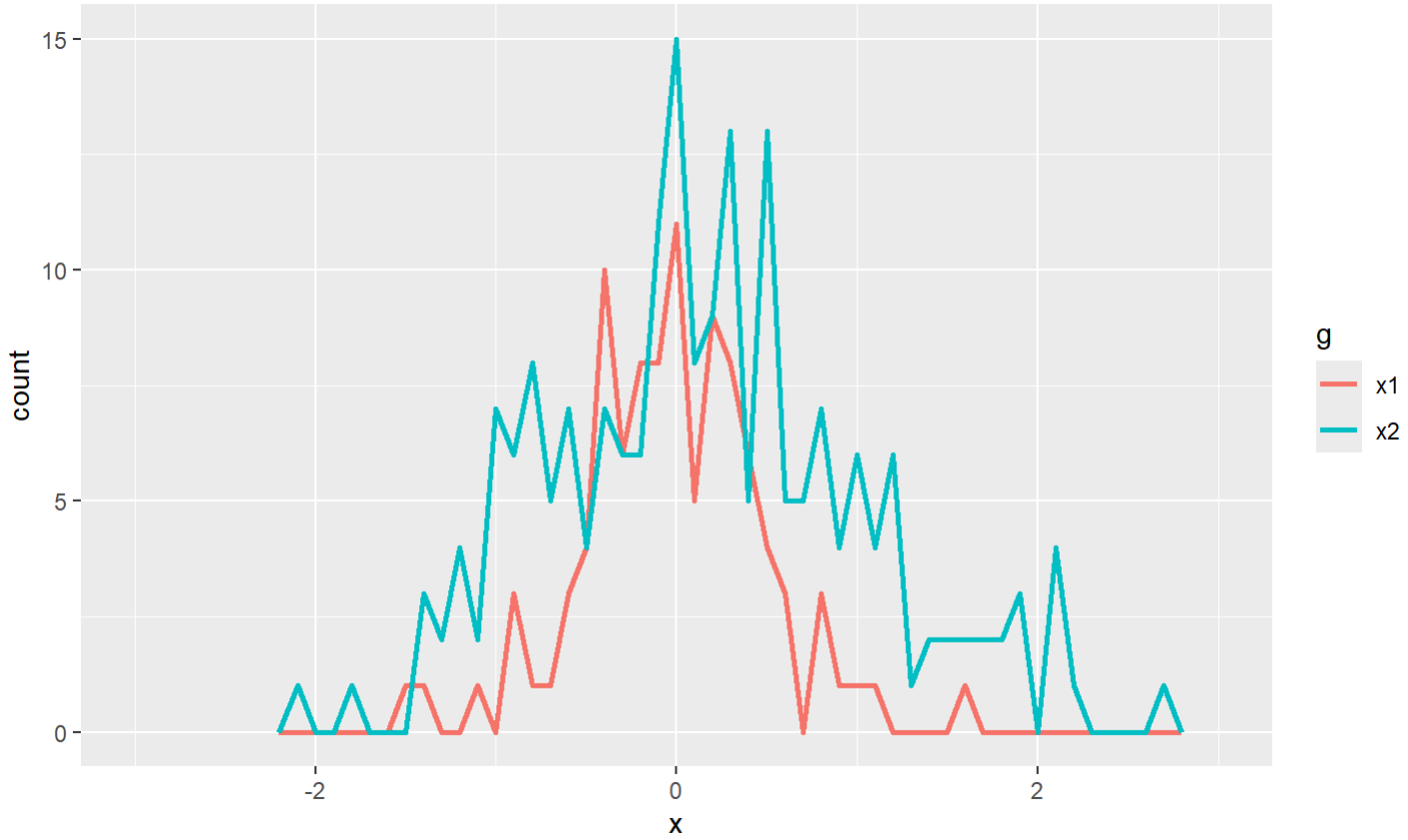
```
x1 <- rnorm(100, mean = 0, sd = 0.5)
x2 <- rnorm(200, mean = 0.15, sd = 0.9)

freqpoly(x1, x2)

print(t_test(x1, x2))
```

Ejercicio de Motivación

```
x1 <- rnorm(100, mean = 0, sd = 0.5)
x2 <- rnorm(200, mean = 0.15, sd = 0.9)
freqpoly(x1, x2)
```




```
cat(t_test(x1, x2))
```

Ejercicio de Motivación

La primera fila tiene tres columnas para los controles de entrada (distribución 1, distribución 2 y controles de gráfico).

```
library(ggplot2)
ui <- fluidPage(
  fluidRow(
    column(4,
      "Distribution 1",
      numericInput("n1", label = "n", value = 1000, min = 1),
      numericInput("mean1", label = "μ", value = 0, step = 0.1),
      numericInput("sd1", label = "σ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Distribution 2",
      numericInput("n2", label = "n", value = 1000, min = 1),
      numericInput("mean2", label = "μ", value = 0, step = 0.1),
      numericInput("sd2", label = "σ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Frequency polygon",
      numericInput("binwidth", label = "Bin width", value = 0.1, step = 0.1)# continua....
```

Ejercicio de Motivación

La segunda fila tiene una columna ancha para el gráfico y una columna estrecha para la prueba de hipótesis.

```
fluidRow(
    column(9, plotOutput("hist")),
    column(3, verbatimTextOutput("ttest"))
  )
)
```

Ejercicio de Motivación

```
library(ggplot2)
server <- function(input, output, session) {
  output$hist <- renderPlot({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    freqpoly(x1, x2, binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    t_test(x1, x2)
  })
}

shinyApp(ui, server)
```

Ejercicio de Motivación

Distribution 1

n

1000

μ

0

σ

0,5

Distribution 2

n

1000

μ

0

σ

0,5

Frequency polygon

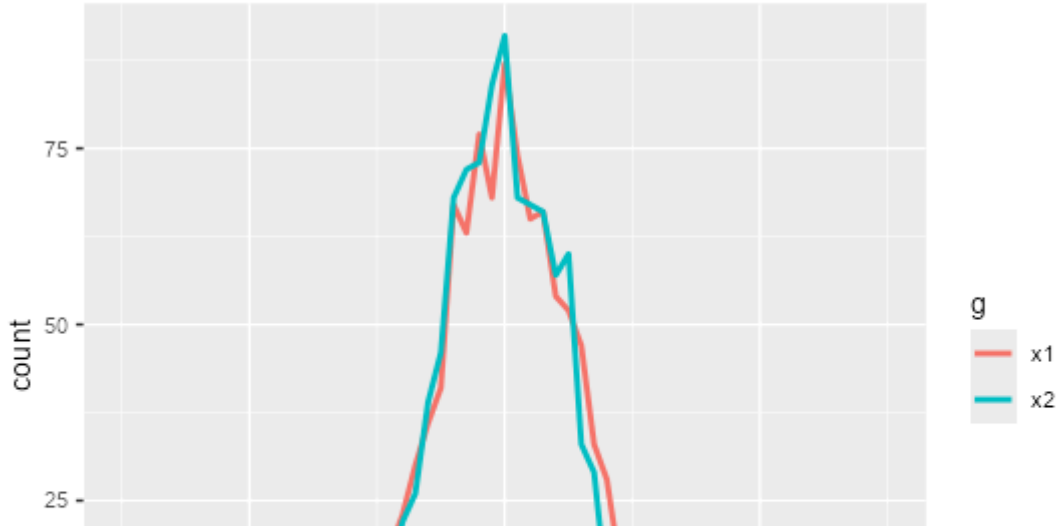
Bin width

0,1

range

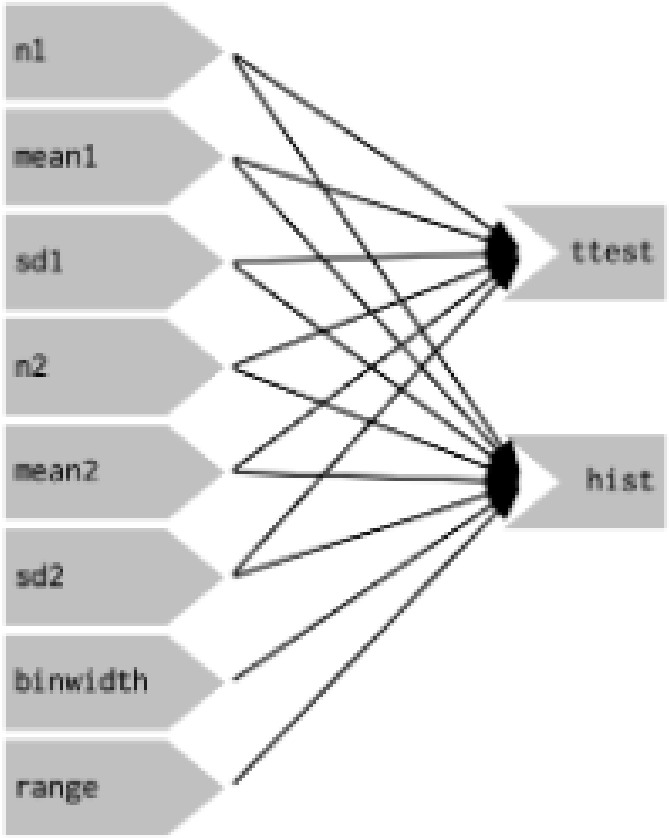
-5 -3 3 5

-5 -4 -3 -2 -1 0 1 2 3 4 5




```
p value: 0.753
[-0.05, 0.04]
```

El gráfico reactivo



El gráfico reactivo

- El gráfico reactivo muestra que cada salida depende de cada entrada, creando un diseño denso y poco claro.
- Esto genera dos problemas:
  - La aplicación es difícil de entender y analizar debido a sus múltiples conexiones.
  - Es ineficiente: realiza cálculos innecesarios (por ejemplo, si cambian los cortes del gráfico, se recalculan los datos).

El gráfico reactivo

- Otro problema: el polígono de frecuencias y la prueba t usan extracciones aleatorias independientes, lo que es engañoso, ya que deberían trabajar con los mismos datos.
- La solución: usar expresiones reactivas para extraer y reutilizar cálculos repetidos.

Simplificando la gráfica

```
server <- function(input, output, session) {
  x1 <- reactive(rnorm(input$n1, input$mean1, input$sd1))
  x2 <- reactive(rnorm(input$n2, input$mean2, input$sd2))

  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}

shinyApp(ui, server)
```

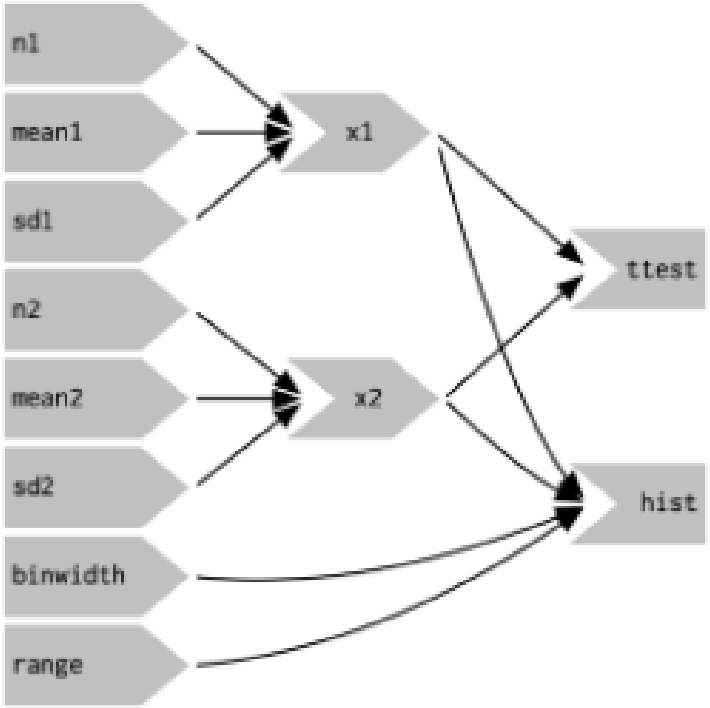
Simplificando la gráfica

En este gráfico los componentes se pueden analizar de forma aislada.

Los parámetros de distribución solo afectan la salida a través de `x1` y `x2`.*

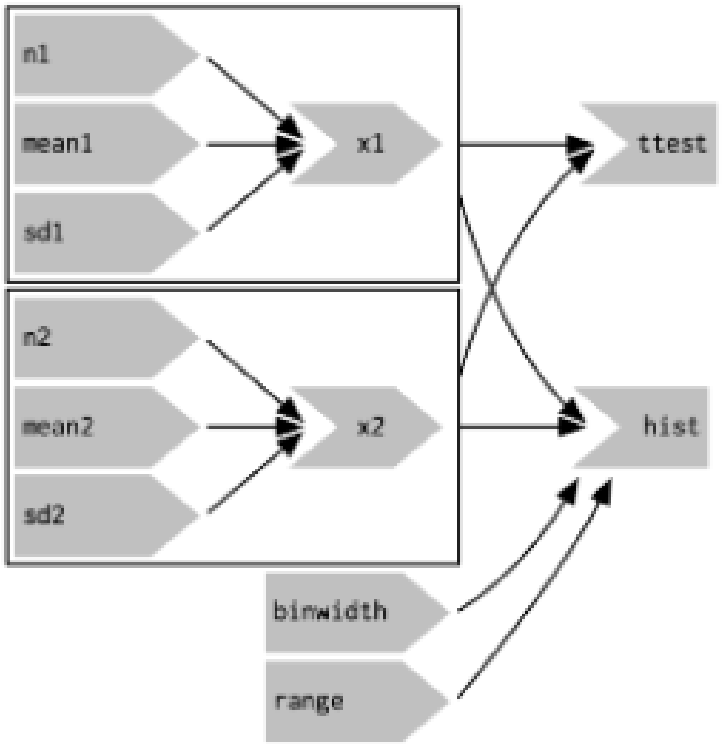
La reescritura también mejora la eficiencia: al cambiar `binwidth` o `range`, solo se actualiza el gráfico, no los datos subyacentes.*

Simplificando la gráfica



Simplificando la gráfica

Para enfatizar esta modularidad, la siguiente figura dibuja recuadros alrededor de los componentes independientes.



La “regla de tres”

La “regla de tres” en programación: si copias y pegas algo tres veces, debes eliminar la duplicación (generalmente con una función).

En Shiny, aplica la “regla de uno”: si copias y pegas algo una vez, considera extraer el código repetido en una expresión reactiva.

Simplificando la gráfica

Distribution 1

n

1000

μ

0

σ

0,5

Distribution 2

n

1000

μ

0

σ

0,5

Frequency polygon

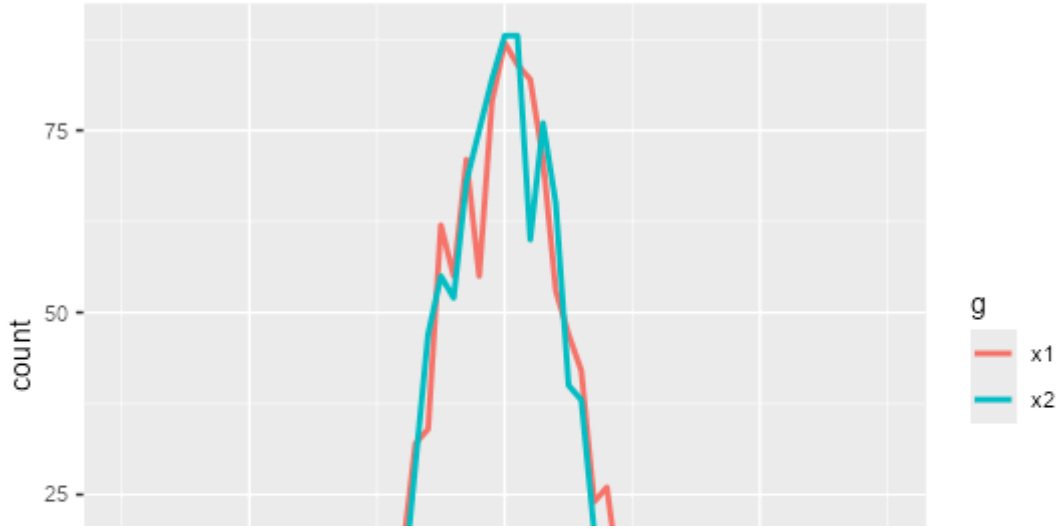
Bin width

0,1

range

-5 -3 3 5

-5 -4 -3 -2 -1 0 1 2 3 4 5




```
p value: 0.787
[-0.04, 0.05]
```