A Tutorial on Shiny and Leaflet

Zachary Hoylman

11/5/2019

Shiny: Introduction, installation and example

Shiny is an R package that makes it easy to build interactive web applications (apps) straight from R. This lesson will get you started building Shiny apps right away.

If you still haven't installed the Shiny package, open an R session, connect to the internet, and run

```
install.packages("shiny")
```

First lets run a shiny app example that comes with shiny to understand what apps, in a very very very simple form, are.

```
library(shiny)
runExample("01_hello")
```

Shiny layout

There are two main components to a shiny app, a User Interface (UI) and a server. The UI is used to pass arguments to the server script which conducts the operations. These two parts of the app can either be contained in two separate scripts (ui.R and server.R) or can be combined in a single script that has both components.

```
shinyApp(ui = defines the user interface,
         server = define the operations being conducted and does things like plotting)
```

Simple script example

This script recreates the app shown above to familiarize ourselves with how shiny apps operate. We are going to be using a base R data set for this example, "faithful", which is a data set of Old Faithful geyser eruptions.

Lets take a quick look at the data

```
head(faithful)
```

```
##   eruptions waiting
## 1     3.600      79
## 2     1.800      54
## 3     3.333      74
## 4     2.283      62
## 5     4.533      85
## 6     2.883      55
```

"eruptions" represent the length of time of a single eruption, and "waiting" is the time in between eruptions. We are going to be using the second column in this data set.

Now lets look at the app.

```
shinyApp(
  # First lets build the user interface (UI).
  ui = fluidPage(
  # App title ----
  titlePanel("Old Faithful Eruptions"),
  # There are different lay out options we can use in shiny,
  # here we will be using the "sidebarLayout" option.
  sidebarLayout(
    # Now we add idebar panel for inputs ----
    sidebarPanel(
      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    # Main panel for displaying outputs ----
    mainPanel(
      # Output: Histogram named distPlot.
      # distPlot is defined below.
      plotOutput(outputId = "distPlot")
    )
  )
),
server <- function(input, output) {
  # Histogram of the Old Faithful Geyser Data
  # with requested number of bins
  # This expression that generates a histogram is wrapped in a call
  # to renderPlot to indicate that:
  #
  # 1. It is "reactive" and therefore should be automatically
  #    re-executed when inputs (input$bins) change
  # 2. Its output type is a plot
  output$distPlot <- renderPlot({
    # Fist we define what data is being used in the plot.
    # For a histogram we want our variable of interest on the x
    # The user doesnt have an option to change this... yet
    x    <- faithful$waiting
    # This is the part that is modifiied by the user (input$bins).
    # Here the user is overwriting the "bins" variable
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    #here is the plot
    hist(x, breaks = bins, col = "#75AADB", border = "white",
         xlab = "Time in between eruptions (Minutes)",
         main = "Histogram of waiting times")
    })
})
```

Multivariate example

Great! Now we have a better idea of how these apps work. Lets now do something a bit more realistic. Say you have a lot of data that you want to visualize in different ways. Lets make an app that allows the user to choose x and y variables from a list of variables, plots them and does a bit of stats. We are going to use the data set “iris” for this example. This data set has information about properties of flowers.

```r
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
shinyApp(ui = fluidPage(
  # First we will build the UI in the same fashion as before
  titlePanel("Flowers!"),
  # we are going to allow the user to choose a polynomial degree for the model
  numericInput(inputId = "obs",
                label = "Choose Polynomial Degree",
                value = 1, min = 1, max = 5),
  # this time instead of a slider, we are going to define the
  # dropdown options for selecting data from the iris dataset.
  # We are going to define the input ID, Label and Choices.
  # Choices are a name you decide, and then the respective Collumn
  # name in iris. (See prind out above for more details)
  sidebarLayout(
    sidebarPanel(
      selectInput(inputId = "x",
                label = "Choose an independent variable",
                choices = c("Sepal Length" = "Sepal.Length",
                      "Sepal Width" = "Sepal.Width",
                      "Petal Length" = "Petal.Length",
                      "Petal Width" = "Petal.Width"))
    ),
    # repeate for y
    sidebarPanel(
      selectInput(inputId = "y",
                label = "Choose an dependent variable",
                choices = c("Sepal Length" = "Sepal.Length",
                      "Sepal Width" = "Sepal.Width",
                      "Petal Length" = "Petal.Length",
                      "Petal Width" = "Petal.Width"))
    ),
  ),
  # the output will be a plot, same as the previous example
  mainPanel(
    plotOutput(outputId = "Plot")

  )
), server = function(input, output) {
  # now we define what part of the app is "reactive" in the server section.
  # in this case it will be the user defined variables to use for the modeling
  # and plotting.
  output$Plot = renderPlot({
    # using the user inputs, we will build the dataset used
    data = data.frame(x = iris[,input$x],
                      y = iris[,input$y])
    # compute a linear model with  polynomial term
    model = with(data,lm(y ~ poly(x,input$obs)))
    # predict data out for plotting
    predict_data = data.frame(x = seq(min(data$x), max(data$x), length.out = 1000))
    predicted.intervals = data.frame(predict(model, newdata = predict_data, , interval = "confidence"))
    # extract some inforamtion from the lm for the plot
    summary = summary(model)
    # build the plot itself
    plot(data$x, data$y, xlab = input$x, ylab = input$y, col = iris$Species)
    # add the lm as an and confidence intervals as lines
    lines(predict_data$x, predicted.intervals$fit,col='green',lwd=3)
    lines(predict_data$x, predicted.intervals$lwr,col='black',lwd=1)
    lines(predict_data$x, predicted.intervals$upr,col='black',lwd=1)
    # add the r2 of the regression to the plot
    mtext(paste0("r = ", round(summary$r.squared,2), side=3))
    # add a color scalling legend
    legend("topleft",legend=levels(iris$Species),col=1:3, pch=1)
  })
})
```

Bingo! We have a working multivariate app that allows for some data analysis. At this point you should start to see some of the utility of building apps. Apps allow the user to get a much more "data rich" experience by allowing them to explore the data. In other words, they can evaluate your data and analysis more on their own terms. In summary allowing for flexibility in data visualization and analysis can promote a much greater understanding of your research.

Leaflet: Introduction and installation

Leaflet is a very powerful library that allows for interactive mapping. A lot of research in the natural sciences has a spatial component and often we rely on graphical user interfaces to produce visualizations of our data. Think ESRI ArcGIS. In some cases, making quick maps for visualizing can be a pain and sometimes you want a product that is interactive for the end user without them having to have access to a GIS platform.

With this I present Leaflet, an open-source JavaScript library for mobile-friendly interactive maps. But… Now we have a R library that brings this powerful mapping service to R users without having to code in JavaScript (except for customization).

Lets install the package.

```r
install.packages("leaflet")
```

Quick Note!

In this next section I am going to be using an operator you might not be familiar with, the "pipe" operator (%>%). This is a very nifty tool that allows a user to avoid having tons of para theses, making code much more difficult to read. When you have a bunch of pipes put together complex data manipulation can be easy to read and understand by someone not familiar with your code. The main jist of the pipe operator is that the data set from the previous line is passed to the next part of the "pipe chain".

Here is an example

```r
library(leaflet)

# traditional way of doing some wierd data manipulation (HARD TO READ!!!)
round(exp(diff(log(iris$Sepal.Length))),2)

# the other non preferable option.
data = iris$Sepal.Length
log_data = log(data)
diff_log_data = diff(log_data)
exp_diff_log_data = exp(diff_log_data)
round_exp_diff_log_data = round(exp_diff_log_data,2)
print(round_exp_diff_log_data)

# with a pipe operator (much easier to read, also allows you to avoid redefining data)
iris$Sepal.Length %>%
  log()%>%
  diff()%>%
  exp()%>%
  round(., 2)
```

They all yield the same result, they just do so in different ways. When you start to have large data sets the pipe chain method saves tons of RAM space and mixed with packages that leverage C++ (like dplyr) make your code run WAAAAAYYYYYYY faster.

Simple Leaflet example

Let's say you want a simple ESRI like base map.

```r
library(leaflet)
leaflet() %>%
  addProviderTiles(providers$Esri.NatGeoWorldMap)
```

DONE! How cool is that? Now we can make the map go straight to Missoula.

```r
leaflet() %>%
  addProviderTiles(providers$Esri.NatGeoWorldMap) %>%
  setView(lat = 46.875676, lng = -113.991386, zoom = 12)
```

Now lets add some data you collected from the field

```r
# define some data with a location and a name
# this is where you would substatute your own data from the field
data_from_the_field = data.frame(names = c("Site 1", "Site 2", "Site 3"),
                                 lat = c(46.875, 46.877, 46.887),
                                 long = c(-113.991, -113.994, -114))
```

```r
leaflet() %>%
  addProviderTiles(providers$Esri.NatGeoWorldMap) %>%
  setView(lat = 46.875676, lng = -113.991386, zoom = 13) %>%
  addMarkers(data_from_the_field$long, data_from_the_field$lat, popup = data_from_the_field$names)
```

Mixing Leaflet and Shiny

This is where things get really cool. Coupling custom functions with geospatial information from leaflet and using the reactive capabilities of Shiny yield some seriously powerful tools. For example, a question I get a lot working in the climate office is "home much precipitation does [insert place here] get". I am going to share with you a tool that can answer that question for any location in the continental U.S. using gridMET data from University of Idaho.