Reporting data results #3

Shiny applications

Resources for learning Shiny

Here are some ways you can learn more about making Shiny apps:

- There is an excellent tutorial to get you started at RStudio: `http://shiny.rstudio.com/tutorial/lesson1/`.
- There are also several great sites that show you both Shiny examples and their code: `http://shiny.rstudio.com/gallery/` and `http://www.showmeshiny.com`.

Many of the examples and ideas in the course notes this week come directly or are adapted from RStudio's Shiny tutorial.

Starting out with Shiny

To start, Shiny has several example apps that you can try out. These are all available through your R session once you install the Shiny package.

You can find the pathname to where these are stored using the command `system.file()`:

```
install.packages("shiny")
library(shiny)
system.file("examples", package = "shiny")
```

Basics of Shiny apps

Once you have Shiny installed, you can run the examples using the `runExample()` command. For example, to run the first example, you would run:

```r
runExample("01_hello")
```

Examples include:

- `02_text`
- `03_reactivity`
- `04_mpg`
- `05_sliders`
- `06_tabsets`
- `07_widgets`
- `08_html`
- `09_upload`
- `10_download`
- `11_timer`

Basics of Shiny apps

When you run any of the examples, a window will come up in your R session that shows the Shiny App, and your R session will pay attention to commands it gets from that application until you close the window.

If you look at your R console, you'll see something like:

```
Listening on http://127.0.0.1:6424
```

Basics of Shiny apps

If you scroll down, you'll be able to see the code that's running behind the application:

Hello Shiny!

by RStudio, Inc.

This small Shiny application demonstrates Shiny's automatic UI updates. Move the *Number of bins* slider and notice how the `renderPlot` expression is automatically re-evaluated when its dependant, `input$bins`, changes, causing a histogram with a new number of bins to be rendered.

server.R ui.R show with app

```r
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  # Expression that generates a histogram. The expression is
  # wrapped in a call to renderPlot to indicate that:
  #
  #  1) It is "reactive" and therefore should be automatically
  #     re-executed when inputs change
  #  2) Its output type is a plot

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]  # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })

})
```

Shiny app structure

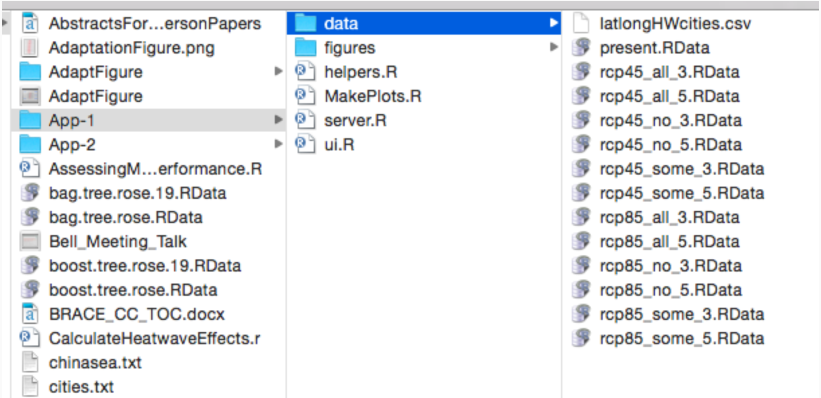
Generally, each application takes two files:

1. A user interface file ("ui.R")
2. A server file ("source.R")

At its heart, an R shiny app is just a directory on your computer or a server with these two files (as well as any data files or complementary code scripts) in it.

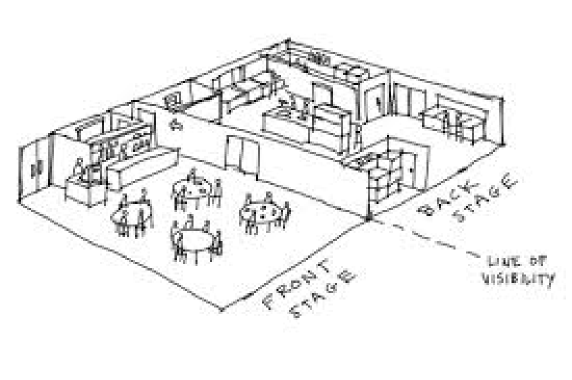
Shiny app structure

For example, here's a visual of an App I wrote to go with a paper:



Shiny app structure

You can kind of think of the two elements of an R shiny app, the user interface and the server, as two parts of a restaurant.



- User interface = the dining room
- Server = the kitchen

Sharing a Shiny App

Once you have a Shiny app running, if you have an account for the Shiny server, you can choose to "Publish" the application to the Shiny server.

Once you publish it there, anyone can access and use it online (this service is free up to a certain number of apps and a certain number of visitors per time– unless you make something that is very popular, you should be well within the free limit).

`server.R` file

The `server.R` file tells R what code to run, based on the inputs it gets from a user making certain selections.

For the example "01_hello", this file tells R to re-draw a histogram of the data with the number of bins that the user specified.

Once the computer is through with all the code for what to do, this file also will have code telling R what to send back to the application for the user to see (in this case, a picture of a histogram made with the specified number of bars).

server.R file

Here is the skeleton of the code in the server.R file for the histogram example, "01_hello":

```
library(shiny)

# Code to draw histogram
shinyServer(function(input, output) {

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]  # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with `bins` number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })

})
```

`server.R` file

Notice that "interior" code here is regular R code. In the "01_hello" example, the `server.R` file includes some code to figure out the breaks for histogram bins, based on how many total bins you want, and draw a histogram with those bin breaks:

```
x    <- faithful[, 2]  # Old Faithful Geyser data
bins <- seq(min(x), max(x), length.out = input$bins + 1)

# draw the histogram with the specified number of bins
hist(x, breaks = bins, col = 'darkgray', border = 'white')
```

server.R file

This code is then "wrapped" in two other functions in the server.R code.

First, this code is generating a plot that will be posted to the application, so it's wrapped in a renderPlot function to send that plot as output back to the application:

```
output$distPlot <- renderPlot({
    x    <- faithful[, 2]  # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
```

`server.R` file

This code puts the results of `renderPlot` into a slot of the object `output` named `distPlot`.

We could have used any name we wanted to here, not just `distPlot`, for the name of the slot where we're putting this plot, but it is important to put everything into an object called `output`.

Now that we've rendered the plot and put it in that slot of the `output` object, we'll be able to refer to it by its name in the user interface file, when we want to print it in the app.

server.R file

All of this is wrapped up in another wrapper:

```r
# Define server logic required to draw a histogram
shinyServer(function(input, output) {
  output$distPlot <- renderPlot({
    x    <- faithful[, 2]  # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
})
```

`server.R` file

The `server.R` file also has a line to load the `shiny` package.

You should think of apps as being like Rmd files— if there are any packages or datasets that you need to use in the code in that file, you need to load it within the file, because R won't check in your current R session to find it when it runs the file.

`ui.R` file

The other file that a Shiny app needs is the user interface file (`ui.R`).

This is the file that describes how the application should look. It will write all the buttons and sliders and all that you want for the application interface.

This is also where you specify where you want outputs to be rendered and put in any text that you want to show up.

`ui.R` file

For example, here is the `ui.R` file for the histogram example:

```r
library(shiny)
shinyUI(fluidPage(
  titlePanel("Hello Shiny!"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1, max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
))
```

`ui.R` file

There are a few things to notice with this code. First, there is some code that tells the application to show the results from the `server.R` code.

For example, the following code tells R to show the histogram that we put into the `output` object in the `distPlot` slot and to put that graph in the main panel of the application:

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

`ui.R` file

Other parts of the `ui.R` code will tell the application what kinds of choice boxes and sliders to have on the application, and what default value to set each to.

For example, say you want the app to:

- Have a slider bar
- Have the slider bar go from a minimum value of 1 to a maximum value of 50
- Have a default value of 30 for the slider bar
- Annotate the slider bar with "Number of bins:"
- Save the selected value to the `bins` slot of the `input` object

`ui.R` file

You can achieve that with the following code in the “ui.R” file:

```r
sliderInput("bins",
            "Number of bins:",
             min = 1,
             max = 50,
             value = 30)
```

Making a Shiny app

The first step in making a Shiny app is to make a new directory somewhere and to create R scripts for that directory called `server.R` and `ui.R`. You can set up this framework in two ways:

- You can just make these two files the normal way– within RStudio, do "New File", "R Script", and then just save them with the correct names to the directory you created for the App.
- You can go to "File" -> "New Project" -> "New Directory" -> "Shiny Web App"

Once you save a file as `ui.R`, notice that you'll have a button in the top right of the file called "Run App". When you're ready to run your application, you can either use this button or use the command `runApp`.

Starting with the `ui.R` file

Next, you'll need to put code in these files.

I suggest starting with the `ui.R` files. This file is where you get to set up how the application looks and how people will be able to interact with it.

This is because you need to have an idea of what inputs and outputs you need before you can effectively make the server file to tell R what to do.

Starting with the `ui.R` file

In the `ui.R` file, everything needs to be wrapped in a `shinyUI()` function, and then most things will be wrapped in other functions within that to set up different panels.

For example, here's a very basic `ui.R` file (adapted directly from the RStudio tutorial) that shows a very basic set up for a user interface:

```r
shinyUI(fluidPage(
        titlePanel("Tweets during Paris Attack"),
        sidebarLayout(
                sidebarPanel("Select hashtag to display"),
                mainPanel("Map of tweets")
        )
))
```

Starting with the `ui.R` file

Notice that everything that should go in certain panels of the page is wrapped in functions like `sidebarPanel` and `mainPanel` and `titlePanel`. Everything in this file will be divided up by the place you want it to go in the final version.

Note: This sidebar layout (a sidebar on one side and one main panel) is the simplest possible Shiny layout. You can do fancier layouts if you want by using different functions like `fluidRow()` and `navBarPage()`. RStudio has a layout help page with very detailed instructions and examples to help you figure out how to do other layouts.

Starting with the `ui.R` file

If I run this `ui.R`, even if my `server.R` file only includes the line `shinyServer(function(input, output) { })`, I'll get the following application:

Starting with the `ui.R` file

So far, the app doesn't have anything interactive on it, and it isn't using R at all, but it shows the basics of how the syntax of the `ui.R` file works.

As a note, I don't have all of the functions for this, like `fluidPage` and `titlePanel` memorized. When I'm working on this file, I'll either look to example code from other Shiny apps of look at RStudio's help for Shiny applications until I can figure out what syntax to use to do what I want.

Adding in widgets

Next, I'll add in a slider bar so people can chose the range of time for the tweets that are shown.

To add this on (it won't be functional, yet, but the widget will be there!), I can edit the `ui.R` script to the code in the following slide.

Adding in widgets

```
shinyUI(fluidPage(
  titlePanel("Tweets during Paris Attack"),
  sidebarLayout(position = "right",
     sidebarPanel("Choose what to display",
       sliderInput(inputId = "time_range",
             label = "Select the time range: ",
             value = c(as.POSIXct("2015-11-13 00:00:00",
                                  tz = "CET"),
                       as.POSIXct("2015-11-14 12:00:00",
                                  tz = "CET")),
             min = as.POSIXct("2015-11-13 00:00:00", tz = "CET")
             max = as.POSIXct("2015-11-14 12:00:00", tz = "CET")
             step = 60,
             timeFormat = "%dth %H:%M",
             timezone = "+0100")),
             mainPanel("Map of tweets")
      )
```

Adding in widgets

The important part of this is the new `sliderInput` call, which sets up a slider bar that users can use to specify certain time ranges to look at.

Here is what the interface of the app looks like now:

Adding in widgets

If I open this application, I can move the slider bar around, but I it isn't actually sending any information to R yet.

We'll do that by changing the "server.R" file in a minute.

Adding in widgets

Things like this slider bar are called "Control Widgets", and there's a whole list of them in the third lesson of RStudio's Shiny tutorial. There are also examples online in the Shiny Gallery.

Creating output in `server.R`

Next, I'll put some R code in the `server.R` file to create a figure and pass it through to the `ui.R` file to print out to the application interface. At first, I won't make this figure "reactive"; that is, it won't change at all when the user changes the slider bar. However, I will eventually add in that reactivity so that the plot changes everytime a user changes the slider bar.

Twitter map

I am going to create a map of all the Tweets that included certain hashtags or phrases and that were Tweeted (and geolocated) from within a five-mile radius of the center of Paris during the attacks last Friday.

Twitter map

For this, I'm going to use data on Tweets I pulled using the `twitteR` package, which syncs up with Twitter's API. Here's an example of what the data looks like:

```
library(readr); library(lubridate)
paris_twitter <- read_csv("../data/App-1/data/final_tweets.csv")
        mutate(tag = factor(tag),
               created = ymd_hms(created, tz = "Europe/Paris"),
               text = iconv(text, to='ASCII//TRANSLIT'))
paris_twitter[1:2, ]
```

```
## # A tibble: 2 x 5
##
##
## 1 RT @forza_will2006: My heart aches for the people of France
## 2 Ensemble contre la haine #jesuisparis #porteouverte #paris
## # ... with 4 more variables: created <dttm>, longitude <dbl>,
## #   latitude <dbl>, tag <fctr>
```

Twitter map

About an equal number of these have and don't have location data:

```
table(!is.na(paris_twitter$longitude))
```

```
##
## FALSE  TRUE
## 10478 10683
```

Twitter map

Here is a table of the number of tweets under the five most-tweeted tags:

Tag	# of Tweets	Example Tweet
#Paris	7502	RT taimaz Le m etro Oberkampf neutralis e paris 13novembre AFP https t co I0SNKsKIOs
#PrayForParis	7250	Tb to Paris prayforparis Paris France https t co qj0Mxrzw1J
#13novembre	1255	RT taimaz 3 personnes a terre devant le Bataclan soign ees par les pompiers AFP 13novembre Paris
#PorteOuverte	1147	RT MBrancourt PorteOuverte https t co 3VFqGFPQry
#fusillade	765	RT jacques lefort Coups de feu et haute tension a l angle rues de Charonne amp Keller Paris11 Gendarmes en position fusillade https t

Twitter map

For the Tweets that are geolocated, it's possible to map the tweet locations.

First, pull a map of Paris and geocode some of the key locations for the event:

```
library(ggmap)
paris_map <- get_map("paris", zoom = 12, color = "bw")

paris_locations <- c("Stade de France", "18 Rue Alibert",
                     "50 Boulevard Voltaire",
                     "92 Rue de Charonne",
                     "Place de la Republique")
paris_locations <- paste(paris_locations, "paris france")
paris_locations <- cbind(paris_locations,
                         geocode(paris_locations))
```

Twitter map

```
paris_locations
```

```
##                          paris_locations      lon      lat
## 1        Stade de France paris france 2.360165 48.92446
## 2         18 Rue Alibert paris france 2.367858 48.87161
## 3  50 Boulevard Voltaire paris france 2.370648 48.86300
## 4     92 Rue de Charonne paris france 2.381989 48.85376
## 5 Place de la Republique paris france 2.364267 48.86779
```

Twitter map

Next, I wrote a function that inputs a dataframe (`df`, default is the Paris tweets dataframe) and a Twitter tag (`tag`). The output is a map of Paris mapping geolocated tweets with that tag.

A bare-bones version of the function is on the next page (see the course notes for the full function).

Twitter map

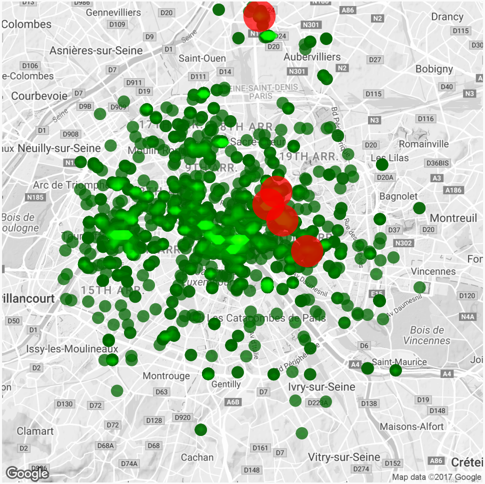
```r
plot_map <- function(tag = "all", df = paris_twitter){
  to_plot <- dplyr::select(df, tag, latitude, longitude) %>%
    filter(!is.na(longitude)) %>%
    mutate(tag = as.character(tag))
  if(tag != "all"){to_plot <- to_plot[to_plot$tag == tag, ]}

  my_map <- ggmap(paris_map, extent = "device") +
    geom_point(data = to_plot,
               aes(x = longitude, y = latitude)) +
    geom_density2d(data = to_plot,
               aes(x = longitude, latitude)) +
    stat_density2d(data = to_plot,
               aes(x = longitude, y = latitude,
                   fill = ..level.., alpha = ..level..),
                   geom = "polygon") +
    geom_point(data = paris_locations, aes(x = lon, y = lat),
                   color = "red", size = 5, alpha = 0.75)
```

Twitter map

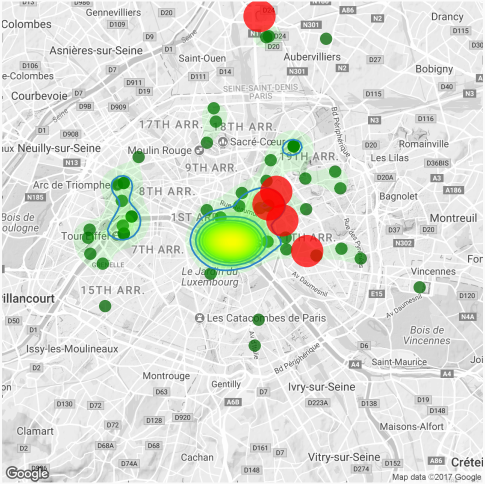
Here is an example of running the full function:

```
plot_map()
```



Twitter map

```r
plot_map(tag = "#PorteOuverte")
```

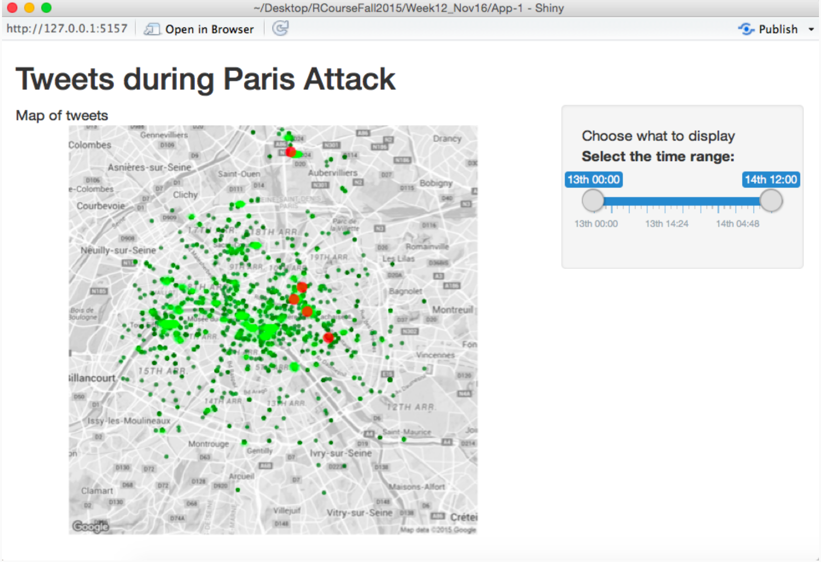


Twitter map

To print this out in the application, I'll put all the code for the mapping function in a file called `helper.R`, source this file in the `server.R` file, and then I can just call the function within the server file.

Twitter map

The application will look as follows after this step:



Twitter map

To complete this, I first changed the `server.R` file to look like this:

```
library(dplyr)
library(readr)
library(lubridate)

source("helper.R")

paris_twitter <- read_csv("data/final_tweets.csv") %>%
        mutate(tag = factor(tag),
               created = ymd_hms(created, tz = "Europe/Paris"))

shinyServer(function(input, output) {
        output$twitter_map <- renderPlot({ plot_map() })
})
```

server.R

Notice a few things here:

1. I'm loading the packages I'll need for the code.
2. I'm running all the code in the `helper.R` file (which includes the function I created to plot this map) using the `source()` command.
3. I put the code to plot the map (`plot_map()`) inside the `renderPlot({})` function.
4. I'm putting the plot in a `twitter_map` slot of the `output` object.
5. All of this is going inside the call `shinyServer(function(input, output){  })`.

server.R

One other change is necessary to get the map to print on the app. I need to add code to the `ui.R` file to tell R where to plot this map on the final interface. The addition to the `ui.R` file looks like this:

```
shinyUI(fluidPage(
        titlePanel("Tweets during Paris Attack"),
        sidebarLayout(position = "right",
                # I'm skipping the code for the sidebar
                # panel here
                mainPanel("Map of tweets",
                          plotOutput("twitter_map"))
        )
))
```

Making the output reactive

Now almost all of the pieces are in place to make this graphic reactive.

First, I added some options to the function in `helper.R` to let it input time ranges and only plot the tweets within that range. Next, I need to use the values that the user selects from the slider in the call for plotting the map.

To do this, I can use the values passed from the slider bar in the `input` object into the code in the `server.R` file.

Making the output reactive

Here is the new code for the `server.R` file:

```
library(ggmap); library(ggplot2)
library(dplyr); library(readr)
library(lubridate)

source("helper.R")

paris_twitter <- read_csv("data/final_tweets.csv") %>%
        mutate(tag = factor(tag))
paris_twitter$created <- as.POSIXct(paris_twitter$created,
                                      tz = "CET")

shinyServer(function(input, output) {
        output$twitter_map <- renderPlot({
                plot_map(start.time = input$time_range[1],
                         end.time = input$time_range[2])
                })
```

Making the output reactive

The only addition from before is to use the `start.time` and `end.time` options in the `plot_map` function and to set them to the first, `[1]`, and second, `[2]`, values in the `time_range` slot of the `input` object. Remember that we chose to label the input from the slider bar `time_range` when we set up the `ui.R` file.

This final app is deployed on shinyapps: `https://brookeanderson.shinyapps.io/ClassExampleTweets`.

You can also see all the code on GitHub: `https://github.com/geanders/RProgrammingForResearch/tree/master/data/App-1`

Fancier version

This Shiny app is pretty simple, to walk you through the ideas.

I've also created a (much) fancier version of a Shiny App looking at this Twitter data that you can check out at `https://brookeanderson.shinyapps.io/TweetsParisAttacks`).

The code for that app is also on GitHub: `https://github.com/geanders/RProgrammingForResearch/tree/master/data/App-1`