Developing Data Products In R

DDP

Brian Caffo

Developing Data Products in R

Brian Caffo

This book is for sale at http://leanpub.com/ddp

This version was published on 2015-11-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Brian Caffo

Statistical inference for data science

Regression Models for Data Science in R

This book is dedicated to Kerri, Penelope, Scarlett and Bowie.

Contents

Preface

About this book

This book is written as a companion book to the Developing Data Products[1] Coursera class as part of the Data Science Specialization[2]. However, if you do not take the class, the book mostly stands on its own. A useful component of the book is a series of YouTube videos[3] that comprise the Coursera class.

The book is intended to be a low cost introduction to the important field of data products. The intended audience are students who are numerically and computationally literate, who would like to put those skills to use in Data Science. The book is offered for free as a series of markdown documents on github and in more convenient forms (epub, mobi) on LeanPub.

[1] https://www.coursera.org/course/devdataprod

[2] https://www.coursera.org/specialization/jhudatascience/1?utm_medium=courseDescripTop

[3] https://www.youtube.com/playlist?list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

[4] http://creativecommons.org/licenses/by-nc-sa/4.0/

What is a data product?

We'll begin this book by defining the topic of this class, data products. A data product is the production output of a data analysis. For example, a data analysis might build a clever machine learning algorithm. A data product embeds that algorithm in a web site so that users can input values and get predictions. Interactive analysis web sites, graphics, apps, R packages, presentations and reports are all data products. In this book we focus only on a few of these components. Mostly for space reasons, but also because our Coursera specialization covers others (like report writing).

Before beginning this book, you should been functional in R. This language will serve as the launching point for all of our data products. Fortunately, if you don't know R, Roger Peng has a great coursera class and LeanPub book on the subject; take and read those first. The class runs every month and both can be obtained for free.

Why R? Well for starters, it's what I know. But, also it's a very prevalent data analysis language. Thus, it's convenient to build the data product in the same language as the analysis is done in. In addition, the list of tools that one needs to learn beyond R to develop data products is massive and include: html5, javascript, D3, REST, python, AWS, and so on. In some sense, the tools we present are best thought of as prototyping tools before building a larger production endeavor. However, for many applications, they can stand alone. Shiny, in particular, is undergoing rapid adoption, development and growth.

The goal of this book

This book (and the corresponding class) has one simple goal: get you started on making data products by introducing you to some very neat tools in R. We only scratch the surface on most of these fantastic platforms, and sadly omit some important ones. It's best to pursue this book with a simple data project in mind. So, before begining, think of a data oriented web app that you'd like to create. Try using the tools in progress to create simplified versions of your app. Hopefully by the end you'll have a large enough toolkit to be able to learn what you need to build your app or product.

Manipulate

Watch this video before beginning[5]

Suppose that you want to create a quick interactive graphic and you have to do it *now*. You're not concerned about accessibility to your interactive graph, you just need it for you or others who also use RStudio. The wonderful little R package `manipulate` is for you.

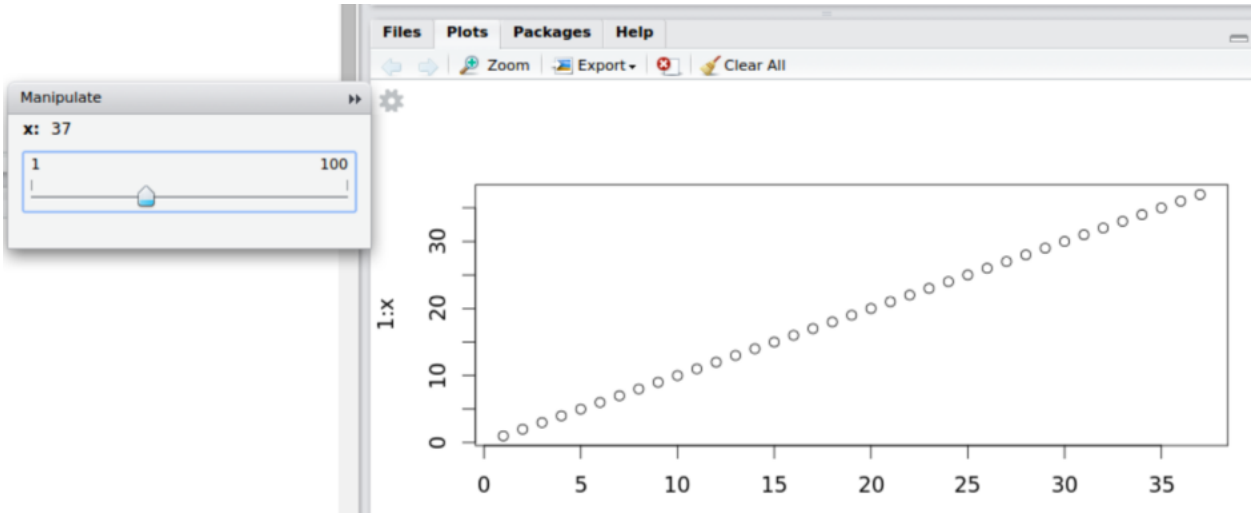
`manipulate` is an R package created by RStudio and must be used within that development environment to work. It is described in very good detail here[6]. It offers simple controls for graphics. So, you're not going to win any visualization awards for your `manipulate` output, but it will solve your problem quickly.

Installing `manipulate` couldn't be easier, `install.packages("manipulate")` will do it. Alternatively, go through RStudio's package management system.

Let's do a simple example.

```
library(manipulate)
manipulate(plot(1:x), x = slider(1, 100))
```

The end result of the procedure is:



Simple example of manipulate.

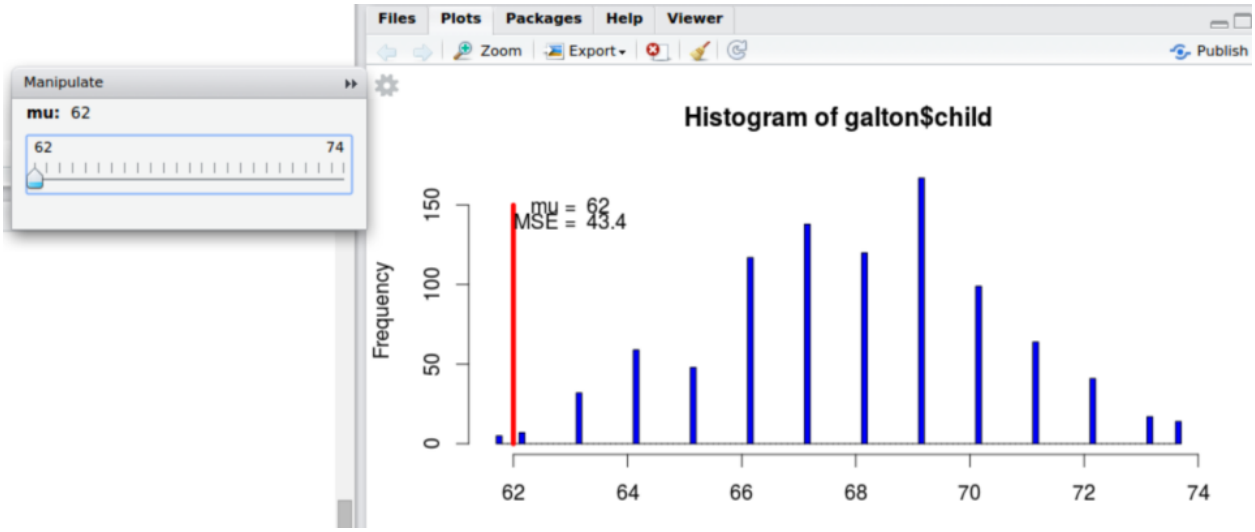
[5] https://www.youtube.com/watch?v=9vbvQa4xY3E&index=3&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

[6] http://www.RStudio.com/ide/docs/advanced/manipulate

By moving the slider, the plot executes `plot(1 : x)` for the value of `x` at the slider value. It's important to note, this interactivity only exists in RStudio, a console version of R won't show the slider.

Let's try a slightly more complicated example.

```
library(manipulate)
myHist <- function(mu){
  hist(galton$child,col="blue",breaks=100)
  lines(c(mu, mu), c(0, 150),col="red",lwd=5)
  mse <- mean((galton$child - mu)^2)
  text(63, 150, paste("mu = ", mu))
  text(63, 140, paste("MSE = ", round(mse, 2)))
}
manipulate(myHist(mu), mu = slider(62, 74, step = 0.5))
```



Second example of manipulate.

Manipulate has several types of controls available including sliders, pickers and checkboxes. Moreover, you can have more than one set of controls by simply adding more arguments to the manipulate function.

Though `manipulate` is great, you quickly realize its limitations. Foremost among these is distribution. It's difficult to share a `manipulate` interactive graphic. Our next platform, `shiny`, also from RStudio, overcomes most of these limitations while remaining in R.

Shiny, Part 1

Watch this video before beginning[7]

Shiny is an important enough topic to devote two chapters to it. Shiny is another product by RStudio and it is described by RStudio as "A web application framework for R". They further add "Turn your analyses into interactive web applications No HTML, CSS, or JavaScript knowledge required". This is mostly true, though a little HTML at least would be useful for understanding some of the concepts. Here's[8] a useful site for learning html basics. We'll proceed as if your html knowledge is very basic and no more advanced than understanding heading levels for fonts.

It is important to dinstiguish between a Shiny applications (app) and a Shiny server. A Shiny server is required to host a shiny app for the world. Otherwise, only those who have have shiny installed and have access to your code could run your web page (really defeating the purpose of making a web page in the first place). In this book, we won't cover creating a shiny server, as that requires understanding a little linux server administration. Instead, we'll run our apps locally and use RStudio's service for hosting shiny apps (their servers) on a platform called shinyapps.io[9]. In other words, RStudio does the server work for your so that all you need to worry about is building your app. Shinyapps.io is free up to a point in that you can only run 5 apps for a certain amount of time per month. This will be fine for our purposes, but if you're really going to get into making Shiny apps, you'll have to spring for a paid plan or run your own server.

Let's build or first app. Like many projects in this book, we'll start with a really simple one.

Your first app

Let's create a setting. Imagine that you created a novel prediction algorithm to predict risk for developing diabetes. You're hoping patients and caregivers will be able to enter their data and, if needed, take preventative measures. You want to create a web site so that users can input the relevant predictors and obtain their prediction. Your prediction algorithm is:

```
diabetesRisk <- function(glucose) glucose/200
```

Ok, so you're Nobel Prize for Medicine likely won't be coming for this one. Here's a link for a real diabetes prediction score.[10]. But, for our purposes this example will serve us well.

First, make sure you have the latest release of R installed. If on Windows, make sure that you have Rtools[11] installed. Then, you can install `shiny` with

[7] https://www.youtube.com/watch?v=xMIra2fmmlE&index=7&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

[8] http://www.w3schools.com/html/

[9] http://www.shinyapps.io/

[10] http://www.ncbi.nlm.nih.gov/pubmed/12610029

[11] https://cran.r-project.org/bin/windows/Rtools/

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

The makeup of a Shiny app

A shiny app consists of two files. First, a file called `ui.R` that controls the User Interface (hence the ui in the filename) and secondly, a file `server.R` that controls the shiny server (hence the server in the filename). All of the calculations and statistical computing will be done by `server.R`. If you're familiar with web development, you might find the `ui.R` file to be weird extra R code instead of good old fashioned, well understoood, html. We'll show later on how to forgo a `ui.R` file and go straight to an html file for the user interface.

In a single directory, make a file called `ui.R` and put in the following code:

```
library(shiny)
shinyUI(pageWithSidebar(
  headerPanel("Hello Shiny!"),
  sidebarPanel(
    h3('Sidebar text')
  ),
  mainPanel(
      h3('Main Panel text')
  )
))
```

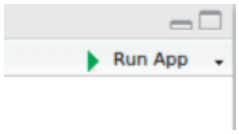
Next, create a `server.R` file (in the same directory as the `ui.R` file) containing the following code:

```
library(shiny)
shinyServer(
  function(input, output) {
  }
)
```

Now, change to the directory where these files are, make sure that shiny is loaded (with `library(shiny)`) and type

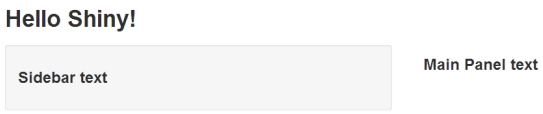
```
runApp()
```

Alternatively, `runApp` can take the path to the files. The current version of Rstudio has a "Run app" button in the upper right hand corner of the editor if a `ui.R` or `server.R` file is open.



The Run app icon.

To the right of the button is a little triangle that will give options in how the shiny app is displayed. Personally, I like the option "Run in viewer pane" so that it runs in the RStudio app itself. Here's the result of running our app.



Example of a simple shiny app.

If you're like most R users when you first encounter shiny, you're probably wondering "What is going on? Why is the syntaxt so strange?" To get used to programming shiny apps, you need to throw away a little of your thinking about R; it's a different style of programming.

Let's parse through what's going on with our two files to hopefully make things more clear. The `ui.R` function is controlling the interface. The function `shinyUI` is alerting R to that. The interior function, `pageWithSidebar` is telling `shinyUI` what kind of page to create. In this case, it's a page with a sidebar (as the function name would suggest). All, elements of the page are now input as functions. In this case we want a header `headerPanel("Hellow Shiny!")`. Then we want the `sidebarPanel` to contain certain elements. So, the `sidebarPanel` function then takes arguments (again functions) of its elements. The statement `h3('Sidebar text')` is saying that we want the text `Siderbar text` in the sidebar (since this function occurs within the function `SidebarPanel`) and we want it to be at the font size `h3`. If you know a little html, you'll recognize `h3` as the third heading level font size. Similar to the `SidebarPanel` function, the `MainPanel` function takes functional arguments for the main panel.

Probably the most frequent syntaxt error for shiny is not putting commas in the right places of `ui.R`. Remember, the page elements are input as arguments, so they need commas like all arguments to R functions.arguments

The `server.R` file is a little easier. The file `shinyServer` tells R that it's dealing with a shiny server. The server function always take an argument of a function with arguments `inputs` and `outputs`. In this case, our function doesn't do anything.

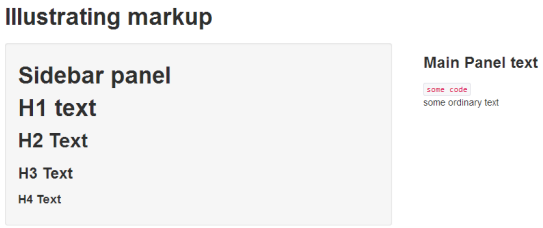
Style and markup

Perhaps the easiest way to illustrate markup is through an example. While keeping the `server.R` file the same, change the `ui.R` file to the following:

```
shinyUI(pageWithSidebar(
  headerPanel("Illustrating markup"),
  sidebarPanel(
      h1('Sidebar panel'),
      h1('H1 text'),
      h2('H2 Text'),
      h3('H3 Text'),
      h4('H4 Text')

  ),
  mainPanel(
      h3('Main Panel text'),
      code('some code'),
      p('some ordinary text')
  )
))
```

This produces the output:



Markup from shiny.

Different input types

Shiny allows for many different input types. In the following example, we change `ui.R` to allow for a few different input types. After you get the hang of these, any new ones will be easy. We leave `server.R` as is, so our inputs don't do anything. See below the numeric input, checkbox, and date input.

```
shinyUI(pageWithSidebar(
  headerPanel("Illustrating inputs"),
  sidebarPanel(
    numericInput('id1', 'Numeric input, labeled id1', 0, min = 0, max = 10, step\
 = 1),
    checkboxGroupInput("id2", "Checkbox",
                       c("Value 1" = "1",
                         "Value 2" = "2",
                         "Value 3" = "3")),
    dateInput("date", "Date:")
  ),
  mainPanel(

  )
))
```

Run this to see the inputs. They're also shown in the next image a few paragraphs below.

Refer to this lesson[12] on the shiny tutorial site for a complete list of inputs available. Once you get the hang of one or two of them, then the remainder will be easy. So, do the above example first and then move on to trying some of the others.

Right now, nothing is done with our inputs. Let's see if we can figure output how to at least get them into `server.R`.

Making your site interactive

Now that we have a bit of a handle on creating a shiny user interface, let's make `server.R` reactive to the inputs from `ui.R`. First, let's adapt our previous simple `ui.R` file to illustrate. Take the last example, where we collected a numeric input, checkbox and a date, and replace the `mainPanel` function with below.

```
mainPanel(
        h3('Illustrating outputs'),
        h4('You entered'),
        verbatimTextOutput("oid1"),
        h4('You entered'),
        verbatimTextOutput("oid2"),
        h4('You entered'),
        verbatimTextOutput("odate")
  )
```

Our goal is to have our sidebarPanel collect the inputs and our main panel display them. The variables `oid1`, `oid2` and `odate` are all outcome variables that we define in our `server.R` function. Here it is:

```
shinyServer(
  function(input, output) {
    output$oid1 = renderPrint({input$id1})
    output$oid2 = renderPrint({input$id2})
    output$odate = renderPrint({input$date})
  }
)
```

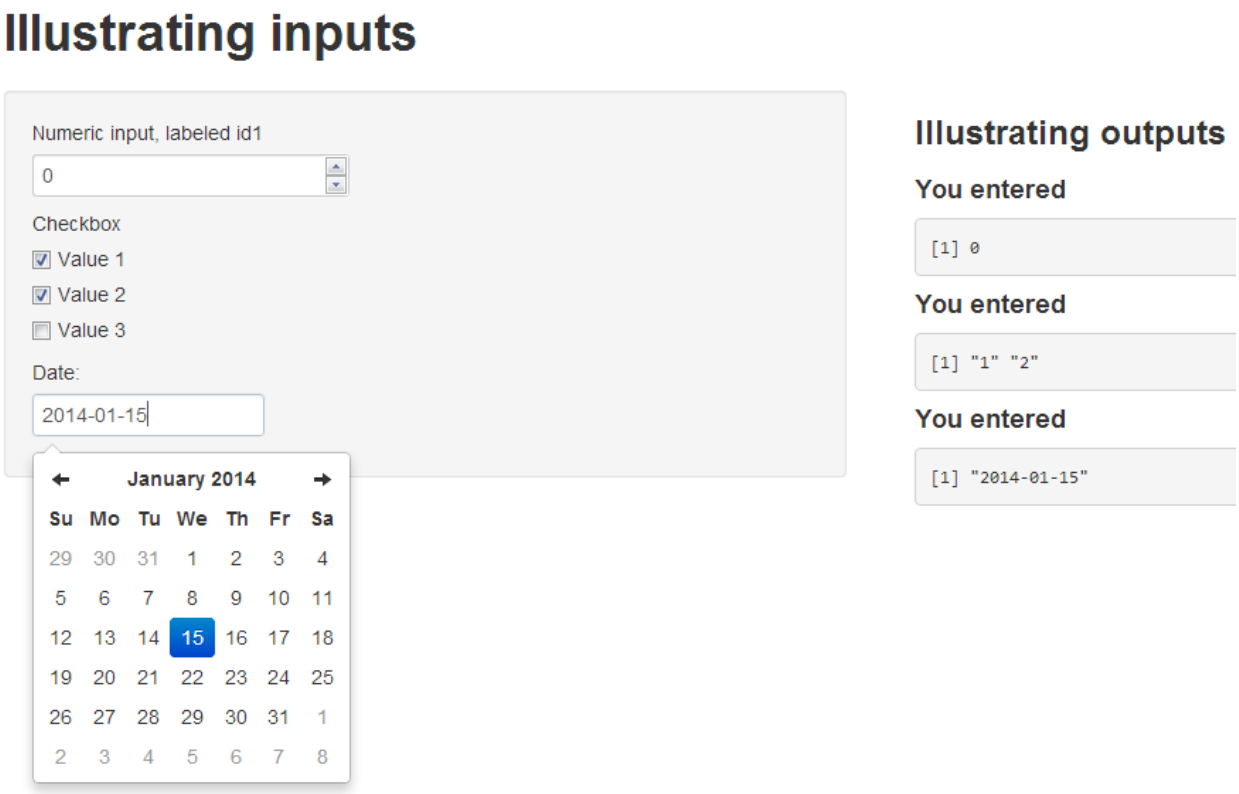
So, our function takes in `input$id1` and prints out `oid1`. Note that `id1` was the name given to our numeric input in our `ui.R` function. This is storred in `input$id1`. The label `id2` was given to our checkbox data in `ui.R` and it's value is storred in `input$id2`. Similarly, `date` was the label given to the input date and it is stored in `input$date`.

[12] http://shiny.rstudio.com/tutorial/lesson3/

It is important in `server.R` to break from how we think about R programs being executed linearly. Instead, think of the functions being executed reactively to changing input from the `ui.R` function. The `renderPrint` function take the reactive input and assigns it to the output variable. Note the peculiar syntax for `renderPrint` in the `({R Statements })`.

In this case, our `server.R` function is merely taking our inputed data and returning it right back. We named our output variables `oid1`, `oid2` and `odate`, the same names used in `ui.R`. The `renderPrint` statement says that it is being sent back to `ui.R` for formatted display.

Here's an example of the running of the code:



Example of input/output in Shiny.

Putting it all together

Now let's build our prediction algorithm. For the `ui.R` files, let's try

```
shinyUI(
  pageWithSidebar(
    # Application title
    headerPanel("Diabetes prediction"),

    sidebarPanel(
      numericInput('glucose', 'Glucose mg/dl', 90, min = 50, max = 200, step = 5\
),
      submitButton('Submit')
    ),
    mainPanel(
        h3('Results of prediction'),
        h4('You entered'),
        verbatimTextOutput("inputValue"),
        h4('Which resulted in a prediction of '),
        verbatimTextOutput("prediction")
    )
  )
)
```

So, we have a sidebar panel that takes in the glucose value. The `submitButton` puts a button that waits until the button is pressed to send values to `sever.R`. We'll discuss more on submit buttons in the next chapter. The main panel shows the output. Notice that `verbatimTextOutput` is the function that is used to display the output of our `server.R` functions.

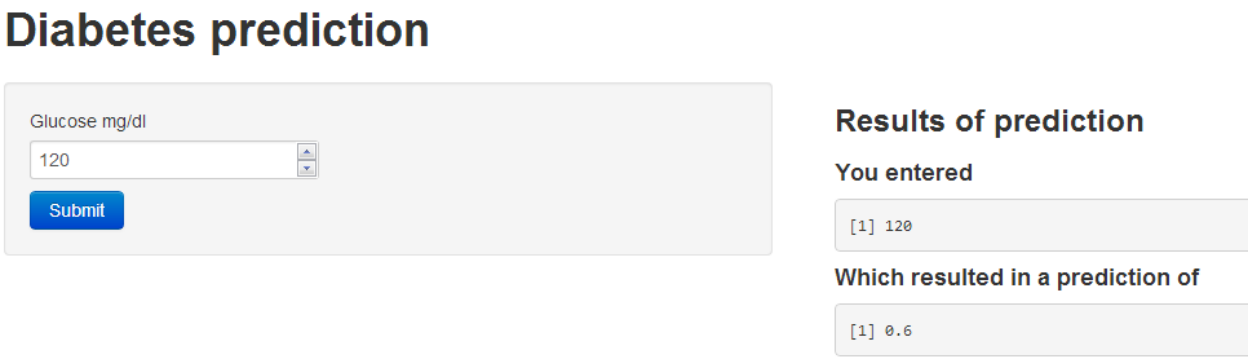
Our `server.R` function is

```
diabetesRisk <- function(glucose) glucose / 200

shinyServer(
  function(input, output) {
    output$inputValue <- renderPrint({input$glucose})
    output$prediction <- renderPrint({diabetesRisk(input$glucose)})
  }
)
```

Notice that our prediction function is defined outside of the `shinyServer` function. The `shinyServer` function takes in the input and repeats both the input glucose level and the outputted diabetes risk.

The output of the function is show below



Output of our diabetes risk assessment score.

Another example

A great way to use shiny is to create interactive graphics. Let's go through a simple example exactly like the one we did in the `manipulate` chapter. The benifit of using shiny over `manipulate` is the ability to share the app broadly as a web page.

Here's our `ui.R` function

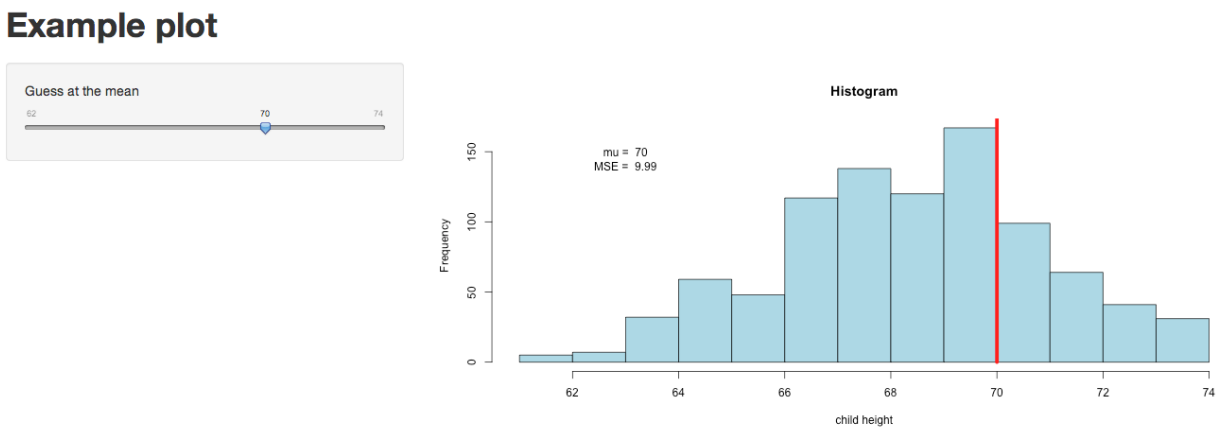
```
shinyUI(pageWithSidebar(
  headerPanel("Example plot"),
  sidebarPanel(
    sliderInput('mu', 'Guess at the mean',value = 70, min = 62, max = 74, step =\
 0.05,)
  ),
  mainPanel(
    plotOutput('newHist')
  )
))
```

Notice that `plotOutput` is the function used for plotting our generated histogram. Let's consider the `server.R` function.

```
library(UsingR)
data(galton)

shinyServer(
  function(input, output) {
    output$newHist <- renderPlot({
      hist(galton$child, xlab='child height', col='lightblue',main='Histogram')
      mu <- input$mu
      lines(c(mu, mu), c(0, 200),col="red",lwd=5)
      mse <- mean((galton$child - mu)^2)
      text(63, 150, paste("mu = ", mu))
      text(63, 140, paste("MSE = ", round(mse, 2)))
      })
  }
)
```

This shows how we created a somewhat elaborate set of code in the `renderPlot` statement that generates the plot. Specifically note the line `mu <- input$mu`. This is our slider value that we use to generate our horizontal line. The final output is shown below:



Output of our interactive plotting example.

Sharing your app

Now that we have a working app we'd like to share it with the world. You could simply post the code, and whatever data files are needed, then users could use `runApp` to see the application. However, it's much nicer to have it display as a stand alone web application. This requires running a shiny server to host the app. Instead of creating and deploying our own shiny server, we'll rely on RStudio's service `shinyapps.io`. You can go to https://www.shinyapps.io/[13] where you will be shown how to

[13] https://www.shinyapps.io/

create an account. After creating the account, you'll be able to host your shiny apps there. Note, this is a freemium service, so that if you want to host a lot of apps, and have fancy bells and whistles, you'll have to spring for the paid service (or host your own shiny server).

After getting a login, you'll have to do some basic installs. First, `devtools`. (This is an essential package for a variety of reasons.) So, do `install.packages("devtools")` at the R command prompt. This will allow you to install the `shinyapps` package directly from github with the command `devtools::install_github('rstudio/shinyapps')`.

Next you have to run some code that you can copy from the `shinyapps.io` site. It looks like

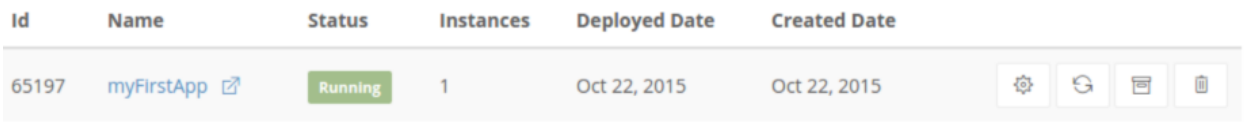
```
shinyapps::setAccountInfo(name='<ACCOUNT NAME>',
                          token='<TOKEN>',
                          secret='<SECRET>')
```

This tells RStudio how to submit your code to `shinyapps.io` and gives it the permissions to do so. Now, change to the directory where your `server.R` and `ui.R` files are at and you can submit your code with:

```
deployApp(appName = "myFirstApp")
```

Make sure that `shinyapps` is loaded (with `library(shinyapps)`). If you need a path to your files, put that in the argument to `deployApp`. The `appName` argument appears to be necessary, and you want it anyway so you know which app it is on `shinyapps.io`. If all has gone well, your app will launch and it will open up a browser window linking to it. In my case, the link was. `https://bcaffo.shinyapps.io/myFirstApp`.

You can manage your app in the browswer at `shinyapps.io`. Now when you go to `shinyapps.io` and click on Applications/All, or Applicatons/Running then you should see your app.



Example app listing at shinyapps.io

From here you can start, stop and delete your app. You can also do that in your R console, with the `shinapps` tools. I would recommend using the web browswer as it's a little easier, but if you get to the point where you're writing a lot of apps, you probably want to learn the console commands.

Shiny Part 2

Watch this video before beginning.[14]

The last chapter should give you enough to get started in using shiny. In this chapter, I'll cover some of the less basic ideas, and go over some of the intricasies of shiny that have been stumbling blocks for me.

Reactivity

When coding a shiny app, it's important to realize that the reactive statements `server.R` functions seemingly follow different rules than ordinary R functions.

Code that you put before `shinyServer` in the server.R function gets called once when you do `runApp()`. Code inside the unnamed function of `shinyServer(function(input, output){`, but not in a reactive statement, will run once for every new user (or page refresh). Code in reactive functions of `shinyServer` get run repeatedly as needed when new values are entered. Reactive functions are those like `renderPlot` and `renderPrint`.

Let's experiment with this. Create this `ui.R` file:

```
shinyUI(pageWithSidebar(
  headerPanel("Hello Shiny!"),
  sidebarPanel(
      textInput(inputId="text1", label = "Input Text1"),
      textInput(inputId="text2", label = "Input Text2")
  ),
  mainPanel(
      p('Output text1'),
      textOutput('text1'),
      p('Output text2'),
      textOutput('text2'),
      p('Output text3'),
      textOutput('text3'),
      p('Outside text'),
      textOutput('text4'),
      p('Inside text, but non-reactive'),
      textOutput('text5')
```

[14] https://www.youtube.com/watch?v=3lLqNoV_MvU&index=5&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

```
  )
))
```

Then the below `server.R` file. Set `x=0` in the console before calling `runApp(display.mode='showcase')`.

```
library(shiny)
x <<- x + 1
y <<- 0

shinyServer(
  function(input, output) {
    y <<- y + 1
    output$text1 <- renderText({input$text1})
    output$text2 <- renderText({input$text2})
    output$text3 <- renderText({as.numeric(input$text1)+1})
    output$text4 <- renderText(y)
    output$text5 <- renderText(x)
  }
)
```

Notice hitting refresh incriments `y`, but enterting values in the textbox does not. Notice `x` is always `1`. Next watch how it updated `text1` and `text2` as needed. Particularly note that it doesn't add 1 to `text1` every time a new `text2` is input.

The `displaymode='showcase'` argument for `runApp` brings up the display of the code as it's being executed in the shiny window.

More on reactive expressions

Sometimes to speed up your app, you want reactive operations (those operations that depend on widget input values) to be performed outside of a `render*` statement. For example, you want to do some code that gets reused in several `render*` statements and don't want to recalculate it for each one. The `reactive` function is made for this purpose. Here's a simple example, (use the `ui.R` function from before)

```
shinyServer(
  function(input, output) {
    x <- reactive({as.numeric(input$text1)+100})
    output$text1 <- renderText({x()                          })
    output$text2 <- renderText({x() + as.numeric(input$text2)})
  }
)
```

Notice that `x` is then used in the correct way in both `text1` and `text2`. Also notice that `x` is executed as a function in the `renderText` command. By comparison, try the following

```
shinyServer(
  function(input, output) {
    output$text1 <- renderText({as.numeric(input$text1)+100  })
    output$text2 <- renderText({as.numeric(input$text1)+100 +
        as.numeric(input$text2)})
  }
)
```

Now the input text has to have 100 added to it twice. This is, of course, inconsequential for this code. But, you can imagine settings where having reactive code outside of the standard reactive statements would be very useful and time saving.

Adding an action button

Sometimes you want shiny to wait to execute code until a button is pressed. This is especially useful when the calculations are very time consuming. The following example shows how to do this.

Here's `ui.R`

```
shinyUI(pageWithSidebar(
  headerPanel("Hello Shiny!"),
  sidebarPanel(
      textInput(inputId="text1", label = "Input Text1"),
      textInput(inputId="text2", label = "Input Text2"),
      actionButton("goButton", "Go!")
  ),
  mainPanel(
      p('Output text1'),
      textOutput('text1'),
      p('Output text2'),
```

```
      textOutput('text2'),
      p('Output text3'),
      textOutput('text3')
  )
))
```

And here's `server.R`.

```
shinyServer(
  function(input, output) {
    output$text1 <- renderText({input$text1})
    output$text2 <- renderText({input$text2})
    output$text3 <- renderText({
        input$goButton
        isolate(paste(input$text1, input$text2))
    })
  }
)
```

Notice the `isolate` statement. Therefore, `text3` isn't displayed until the action button is pressed. You can make statements conditional on the first button press as follows (replace the `output$text3` lines from above with):

```
output$text3 <- renderText({
    if (input$goButton == 0) "You have not pressed the button"
    else if (input$goButton == 1) "you pressed it once"
    else "OK quit pressing it"
})
```

More on layouts

With regard to layouts, the sidebar layout with a main panel is the easiest and best in my opinion. However, using `shinyUI(fluidpage(` is much more flexible and allows tighter access to the bootstrap styles Examples here[15]. `fluidRow` statements create rows and then the column function from within it can create columns tabsets, navlists and navbars can be created for more complex apps.

For much more complex layouts, direct use of html is preferred, http://shiny.rstudio.com/articles/html-ui.html[16]. Also, if you know web development well, you might find using R to create web layouts

[15] http://shiny.rstudio.com/articles/layout-guide.html
[16] http://shiny.rstudio.com/articles/html-ui.html

kind of annoying so this option might be compelling from the start. We don't want to go too much into html, so we're just going to give you enough to try.

First, create a directory called `www` in the same directory with `server.R` Have an `index.html` page in that directory. Your named input variables will be passed to `server.R`. For example, consider the html input type: `<input type="number" name="n" value="500" min="1" max="1000" />`. These can be accessed from `server.R` just like all of the named input variables from `ui.R`. Your server.R output will have class definitions of the form `shiny-`. As an example, your html code might have `<pre id="summary" class="shiny-text-output"></pre>`.

Uploading a file

Uploading a file to shiny is easy. Let's walk through this great example[17] by the shiny folks. There are two main intricasies to get over. First, is writing the code to accept the file

```
fileInput(<FILE ID>, <LABEL FOR THE BUTTON>,
    accept=<MEDIA FORMAT>),
```

The media format in their example was `'text/csv', 'text/comma-separated-values,text/plain', '.csv'` meaning they wanted to accept csv files. This is needed to ensure that a user doesn't upload a non-supported file format. See more about media types at https://en.wikipedia.org/wiki/Media_-type[18].

Then consider accessing the file in a reactive statement in their `server.R` function

```
    inFile = input$<FILE ID>
    read.csv(inFile$datapath)
```

where we've omitted everything except the most essential part of the code. Hopefully, now you'll be able to extend this example to fit your needs.

Summary

Shiny is a pretty amazing framework for creating really rapid client / server applications without a lot of knowledge about the underlying computing. It's perhaps most useful for rapid prototying of data science products.

[17] http://shiny.rstudio.com/gallery/file-upload.html

[18] https://en.wikipedia.org/wiki/Media_type]

Reproducible presentations, slidify

Watch this video before beginning.[19]

We don't cover making reproducible reports, as Roger Peng has an entire book devoted to the subject[20]. For the same reason we want reproducible reports, we also want reproducible presentations. In fact, in many ways, reproducible presentations should be more common as presentations are the default communication method for many projects in progress. Reports, often come later. Fortunately, the same framework accomodates both in R. Specifically, R markdown (knitr) documents.

In R markdown, one writes a text in a simplified markup language (called markdown) that is *trivial* to learn. One embeds the R code and data analysis in this document then the presentation is compiled using the R command prompt. The end result is a visually pleasing html5 presentation. Since the presentation is html5, it just runs in a browser, so no more worrying about different PowerPoint versions!

The act of embedding the code in the document helps one organize their thoughts around the presentation. Moreover, it makes all of the numbers and figures perfectly reproducible. Finally, it's incredibly useful for yourself when you come back to the presentation after a while and want to figure out what you did.

There are two main platforms for reproducible presentations in R, `slidify` and RStudio's presenter. We'll cover both.

Markdown

Watch this intro to `knitr` first.[21]

Perhaps the easiest way to learn markdown is to create a markdown file and push it to github. The benifit of doing this is that github will actually render the markdown document for you when you visit the repo there. In fact, you can just edit the document in the cloud on github if you'd like. Here's[22] a nice reference of markdown commands that I often use.

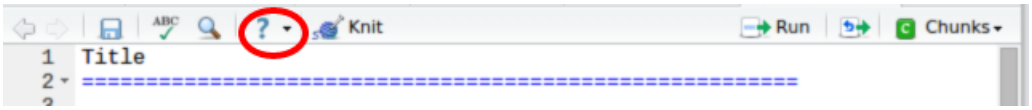
Alternatively, create an R markdown document in RStudio by doing File -> New File -> R Markdown. Then play around with documents to see the way in which markdown syntax is rendered. There's a very useful cheat sheat in RStudio that you can see by clicking on the question mark above the editor.

[19] https://www.youtube.com/watch?v=5WGFv7dkkI4&index=2&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

[20] https://leanpub.com/reportwriting

[21] https://www.youtube.com/watch?v=p3YqnE7EElo&index=8&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

[22] https://daringfireball.net/projects/markdown/syntax



Markdown quick reference.

Finally, all of the markdown slide generators create templates that guide you through the use of markdown.

Slidify

Slidify was created by Ramnath Vaidyanathan in order to streamline the process of creating and publishing R driven presentations. Slidify is an amalgamation of other technologies including knitr, Markdown, and several javascript libaries for HTML5 presentations. Slidify is infinitely extendable and customizable, yet it is easy to use! Slidify allows embedded code chunks and mathematical formulas. Slidify presentations are just HTML files, so you can view them with any web browser and share them easily on Github, Dropbox, or your own website.

To get `slidify`, first fire up RStudio. Make sure you have `devtools` installed then:

```
install.packages("devtools")
library(devtools)
install_github('slidify', 'ramnathv')
install_github('slidifyLibraries', 'ramnathv')
library(slidify)
```

To get started with slidify, set the working directory to where you want to create your Slidify project (say `setwd("~/sample/project/")`) and then create your project and give your project a name. My project is named `first_deck`

```
author("first_deck")
```

The command `author("first_deck")` causes the following to happen: 1. A directory with the name of your project is created inside of your current directory. 2. Inside of this directory an assets directory and a file called `index.Rmd` is created. 3. The assets directory is populated with the following empty folders: css, img, js, and layouts. 4. The newly created index.Rmd R Markdown file will open up in RStudio.

Any custom css, images, or javascript you want to use should respecively be put into the newly created css, img, and js folders.

The file `index.Rmd` is the R Markdown document which you will use to compose the conent of your presentation. The first part of an `index.Rmd` file is a bit of `YAML` code which will look like this:

```
title       :
subtitle    :
author      :
job         :
framework   : io2012        # {io2012, html5slides, shower, dzslides, ...}
highlighter : highlight.js  # {highlight.js, prettify, highlight}
hitheme     : tomorrow      #
widgets     : []            # {mathjax, quiz, bootstrap}
mode        : selfcontained # {standalone, draft}
```

You should edit your `YAML` to include the title, subtitle, author, and job of the author. Other fields include what slide framework you wish to use, which code highlighter you wish to use, and any widgets you want to include. `mathjax` allows for the input of LaTeX code in your document that will get rendered as nice equations in the final presentation. You can also include a logo to appear in your title slide under `logo`, the path to your assets folder and the paths to any other folders you may be using under `url`, and the specific theme for your code highlighter of choice under `hitheme`.

```
logo    : my_logo.png
url     :
    assets: ../assets
highlighter : highlight.js  # {highlight.js, prettify, highlight}
hitheme     : zenburn       #
```

Most of these I wouldn't mess with until you're more familiar with Slidify. Here's a recent YAML that I got from Jeff Leek:

```
title       : Slidify
subtitle    : Data meets presentation
author      : Jeffrey Leek, Assistant Professor of Biostatistics
job         : Johns Hopkins Bloomberg School of Public Health
logo        : bloomberg_shield.png
framework   : io2012        # {io2012, html5slides, shower, dzslides, ...}
highlighter : highlight.js  # {highlight.js, prettify, highlight}
hitheme     : tomorrow      #
url:
  lib: ../../libraries
  assets: ../../assets
widgets     : [mathjax]            # {mathjax, quiz, bootstrap}
mode        : selfcontained # {standalone, draft}
```

OK, let's make a presentation (of two slides). `author` creates two example slides under the `YAML`. You can avoid this by just creating the files yourself without using `author`. Here's the example two slides that it uses

```Rmd
## Read-And-Delete

1. Edit YAML front matter
2. Write using R Markdown
3. Use an empty line followed by three dashes to separate slides!

--- .class #id

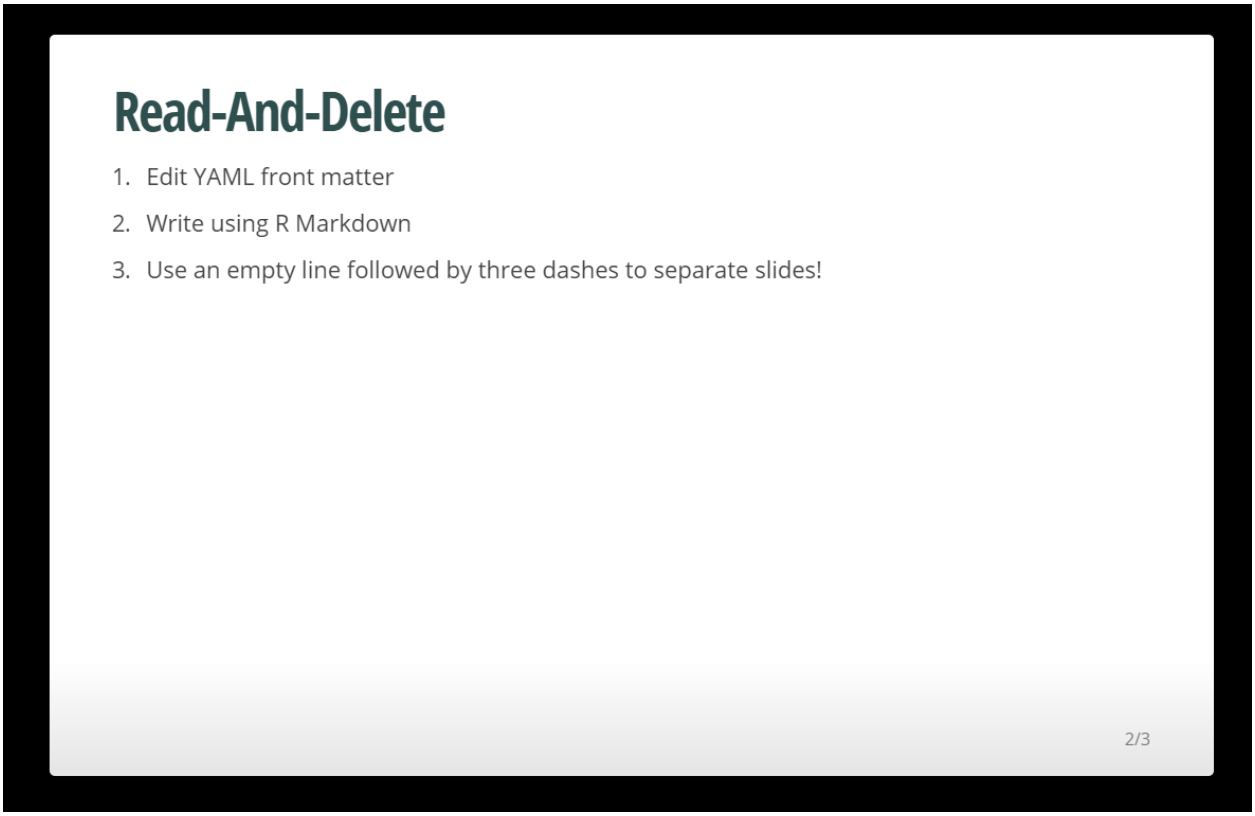
## Slide 2
```

- Whatever you put after `##` will be the title of the slide.
- `---` marks the end of the slide.
- `.class #id` are `CSS` attributes you can use to customize the slide.
- Whatever you put between `##` and `---` is up to you! As long as it is valid R\
 Markdown or `HTML`.

To compile your presentation make sure the working directory contains your `index.Rmd` file and enter the following command:

```
slidify("index.Rmd")
```

An `HTML` flie should appear in your current directory, open it with your favorite web browser and enjoy your Slidify deck! Altnertively, type `browseUrl("index.html")` from the R console to bring up the presentation. Here's the second page



Second slide of the example slidify presentation.

Note, whether your slidify presentation is viewable without an internet connection depends on whether you've downloaded the javascript libraries locally or whether you're asking slidify to grab them off of the internet. This is controlled by the YAML line `mode : selfcontained # {standalone, draft}`. If you really need that presentation to work without an internet connection, make sure to test it out!

Code chunks

As this book mirrors the class on Coursera, we're going to assume that you've seen knitr previously. Since slidify works exactly like knitr for code chunks, there's nothing new. If you haven't seen knitr before, here's the one line version of how to embed your code in slidify.

```
```{r}
rnorm(6)
```
```

Here the R code in between the set of quote tick marks gets executed. The `{r}` tells slidify that it's an R code chunk. There's lots of options that can occur in those braces, such as `{r, echo = TRUE}`

to indicate that you want the code printed as well as the results. Simply typing Tab in RStudio lists out the options.

When you slidify your document, it will run the code and stick in the results. If the option `{r, cache = TRUE}` is on, then the results will be cached so that it doesn't need to run it again if it's already been run once.

For more on embedding code chunks into knitr documents, see Roger Peng's Reproducible Report Writing book mentioned earlier.

Publishing

While it's great that you can view your slide deck on your computer, you might also like for anyone to view it anywhere. To do this, you need to publish it as a web page on a web server. There are tons of ways to create a webserver to host your presentation. An easy one is to use a github account, which as a data scientist, you probably already have.

This page[23] gives a set of commands to publish your presentation to github, dropbox and Rpubs. I'll go through manually publishing to github here, since that's how I like to do it.

Publishing to Github

Publishing to github using the `publish` command should be as simple as `publish(user = "USERNAME", repo = "REPONAME")`. However, there is the issue the git must be installed on the system and in the path. If this is an issue, you can publish to github manually.

Try these steps to buid an example. First create the slide deck with `author('testDeck')` in Slidify in R. Then created the empty repo `testDeck` on github. Add the slide deck and all of the files to the repo. The git repo should include an empty text file named `.nojekyll`.

For example, at the command line, in the directory with the slidify files

```
git init
git add *
git commit -a -m "Added all of the files"
git remote add origin *put in your github origin here*
git push origin master
~~
Then make a branch with
{lang=bash,line-numbers=off}
```

git branch gh-pages ∼∼ Then push the branch to github with {lang=bash,line-numbers=off} ∼∼∼ git push origin gh-pages ∼∼ Then the presentation can be viewed at: `http://USERNAME.github.io/REPONAME/index`

[23] http://slidify.org/publish.html

Here in this case `REPONAME` was `testDeck` and `USERNAME` is your github username. Remember, though, you have to have a branch called gh-pages and the .nojekyll file for it to show up. In general, if you have an html file on github, a `.nojekyll` file, then your page is at `http://USERNAME.github.io/REPONAME/HTMLFILENAME`.

Publishing to dropbox

Surprisingly, Dropbox is a pretty handy little webserver. You can simply copy your slidify deck to your `public` directory and get the public link to your `index.html` file. Alternatively, `publish` will accept Dropbox as an option, where it simply executes these steps.

Publishing to Rpubs

Rpubs is RStudio's presentation and reproducible document hosting site. To publish to Rpubs, simply do

```
publish(title = <TITLE>, html_file = "index.html", host = rpubs)
```

The `html_file` argument isn't needed unless you have a different name than `index.html` for the root slide document.

From my experience in the Developing Data Products class, publishing to RPubs seems to be the easiest. When sharing the link, use `http` rather than the `https`.

`HTML5` Deck Frameworks

The following frameworks are compatible with Slidify for making your presentations. Try each of them out to get a style that you like.

- io2012[24]
- html5slides[25]
- deck.js[26]
- dzslides[27]
- landslide[28]
- Slidy[29]

[24] https://code.google.com/p/io-2012-slides/
[25] https://code.google.com/p/html5slides/
[26] http://imakewebthings.com/deck.js/
[27] http://paulrouget.com/dzslides/
[28] https://github.com/adamzap/landslide
[29] http://www.w3.org/Talks/Tools/Slidy2/Overview.html#

Mathjax

You can include LaTeX math formatting as follows. Edit your YAML so that the widgets line looks like this: `widgets : [mathjax]` Enter inline math code with `$x^2$`, for example. Enter centered code with, `$$\frac{-b \pm \sqrt{b^2 - 4 a c}}{2a}$$` for example. A picture of the output is below.

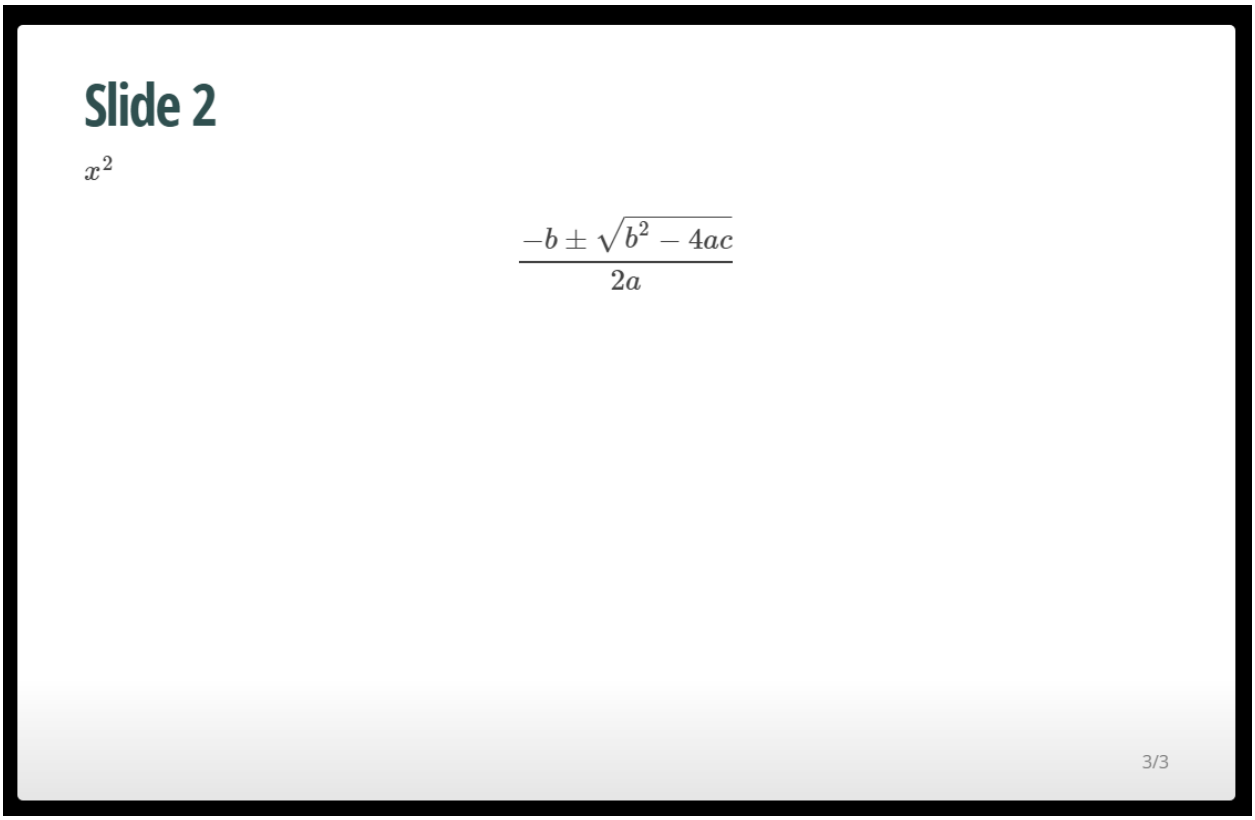


Illustration of using mathjax.

HTML

You can directly add html into your R markdown document. Just include html in the Rmd file and it will get kept as html when it's slidified Especially useful for stuff like images or tables where you need finer control of the html options Also, remember you can edit the final html slide if necessary. This isn't a good best solution, since why do a reproducible format if you're going to break that reproducibility at the last step? But, it's sometimes useful in a pinch, lke if you're frantically preparing course slides at the last minute.

Adding interactive elements to slidify

You can add interactive elements to slidify - Quiz questions - interactive Rcharts plots - Shiny apps

Of course, you could do this directly with html/js More easily, the dev version of slidify has this built in See http://slidify.github.io/dcmeetup/demos/interactive/[30]. To see a demonstration, look here.To[31]

[30] http://slidify.github.io/dcmeetup/demos/interactive/

[31] https://youtu.be/5WGFv7dkkI4?list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90&t=1002

RStudio's Presenter

Watch this video before beginning.[32]

Not to be outdone, RStudio has its own presentation software. It works much like slidify, but with small sytax differences and more integration into RStudio.

If you are familiar with slidify, you will also be familiar with this tool. First, code is authored in a generalized markdown format that allows for code chunks. Secondly, the output is an html5 presentation.

The file extension for the presenter file is `.Rpres`, which gets converted to an `.md` (markdown) file and then to an `html` file if desired. What's nice is that there's a preview tool in RStudio and GUIs for publishing to Rpubs or viewing/creating an html file.

Authoring content

There is a fairly complete guide here[33]. In general, I would say that RStudio presenter is even easier to learn than Slidify.

A quick start version is to do `file` then `New File` then `R Presentation` (use `alt-f` then `f` then `p` if you want key strokes). Use basically the same R markdown format for authoring as slidify/knitr. These include: - Single quotes for inline code - Tripple qutoes for block code - Same options for code evaluation, caching, hiding etcetera

Compiling and tools

It's quite convenient that RStudio auto formats and runs the code when you save the document. Further, the mathjax JS library is loaded by default so that `$x^2$` yields x^2.

Notice that as you type an Rpres, it gets compiled and viewed in the RStudio windo pane. There is a slide navigation button on the preview, as well as the final presentation. Clicking on the notepad icon takes you to that slide in the deck.

Clicking on `more` yields options for:"Clearning the knitr cache", "Viewing in a browser" and "Create a html file to save where you want"

[32] https://www.youtube.com/watch?v=xkidCHwI2sE&index=4&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

[33] http://www.rstudio.com/ide/docs/presentations/overview

The more button.

Finally, there's a refresh button that updates the presentation and a zoom button that brings up a full window.

Visuals

The default transitionfor RPres is `transition: linear`. However, RStudio has made it easy to get some cool html5 effects, like cube transitions with simple options in YAML-like code after the first slide. For example, try `transition: rotate` to see the cube rotation effect.

You can specify the slide transition in a slide-by-slide basis For example, just put `transition: linear` right after the slide creation. In RPres new slides are given by `===` (three equal signs or more in a row) You can see the various transition options here[34]

Hierarchical organization

There is a hierarchical organization structure to RPres documents. That is, you can have the equivalent of chapters, sections, subsections etc. In RPres, the organizational structures are: `section`, `sub-section`, `prompt` and `alert`. The section type changes the appears of the slides. If you want a hierarchical organization structure, just add a `type: typename` option after the slide.

Section appearance.

[34] http://www.rstudio.com/ide/docs/presentations/slide_transitions_and_navigation

Subsection appearance.

Two columns

You can get two columns in your presentation by doing the following. First, create your list for column one. Then put *** on a line by itself with blank lines before and after. Then create your section column.

```
Slide
===

- Bullet 1
- Bullet 2
- Bullet 3

***
- Bullet 4
- Bullet 5
- Bullet 6
```

Here's a picture of the output:

Two column format.

Changing the slide font

Changing the font family is a little challenging, but not so bad once you get the hang of it. After the slide designation (three or more equal signs), one simply has to add lines such as:

```
font-import: http://fonts.googleapis.com/css?family=Risque
font-family: 'Risque'
```

The `font-import` statement brings in a font if it's not a system font. The font names are cpecified in the same way as css font families[35] Some important caveats should be mentioned. First, fonts must be present on the system that you're presenting on, or it will go to a fallback font. Secondly, you have to be connected to the internet to use an imported font (so don't rely on this for offline presentations).

Really changing things

If you know html5 and CSS well, then you can basically change whatever you want in RPres. A css file with the same names as your presentation in that directory will be autoimported. Alternatively, you can use `css: file.css` to import a css file. In your css file, you have to create named classes and then use `class: classname` to get slide-specific style control from your css.

Finally, just as with slidify, ultimately, you have an html file, that you can edit as you wish. This should be viewed as a last resort, as the whole point is to have reproducible presentations, but may be the easiest way to get the exact style control you want for a final product

[35] http://www.w3schools.com/cssref/css_websafe_fonts.asp

Slidify versus RPres

Given the similarity between the two products, it's natural to wonder if one is better. In my opinion, this is a hard question to answer, since the two are really geared toward different types of users. My summary of the benefits of each is as follows:

Slidify

- Gives extremely flexible, control if you're willing to learn.
- Under rapid ongoing development.
- Has a large user base.
- Has lots and lots of styles and options.
- Has a steep learning curve.
- More command-line oriented.

R Studio Presenter

- Embedded in R Studio.
- Very easy to get started.
- Smaller set of easy styles and options.
- Default styles look very nice.
- Ultimately as flexible as slidify with a little CSS and HTML knowledge.

My overall summary would be: if you're sort of a hacker type and you like to tinker with things, use slidify. If you just want to get it done and not worry about it, use RPres. Either way, you really can't go wrong.

Interactive graphs

R has an increasingly diverse set of tools for creating interactive graphs. You've already seen some with `manipulate` and `shiny`. These tools, however, rely on running R in the background. Other tools convert the graphs into javascript displays that can be embedded into web pages and html presentations.

rCharts

Watch this presentation.[36]

`rCharts` is a way to create interactive javascript visualizations using R. It was created by, Ramnath Vaidyanathan, the same author of `slidify`. With Rcharts you don't have to learn complex tools, like D3. Instead, you simply work in R learning a minimal amount of new syntaxt. These notes are basically going through the notes at http://rcharts.io/[37]. Install `Rcharts` with

```
require(devtools)
install_github('rCharts', 'ramnathv')
```

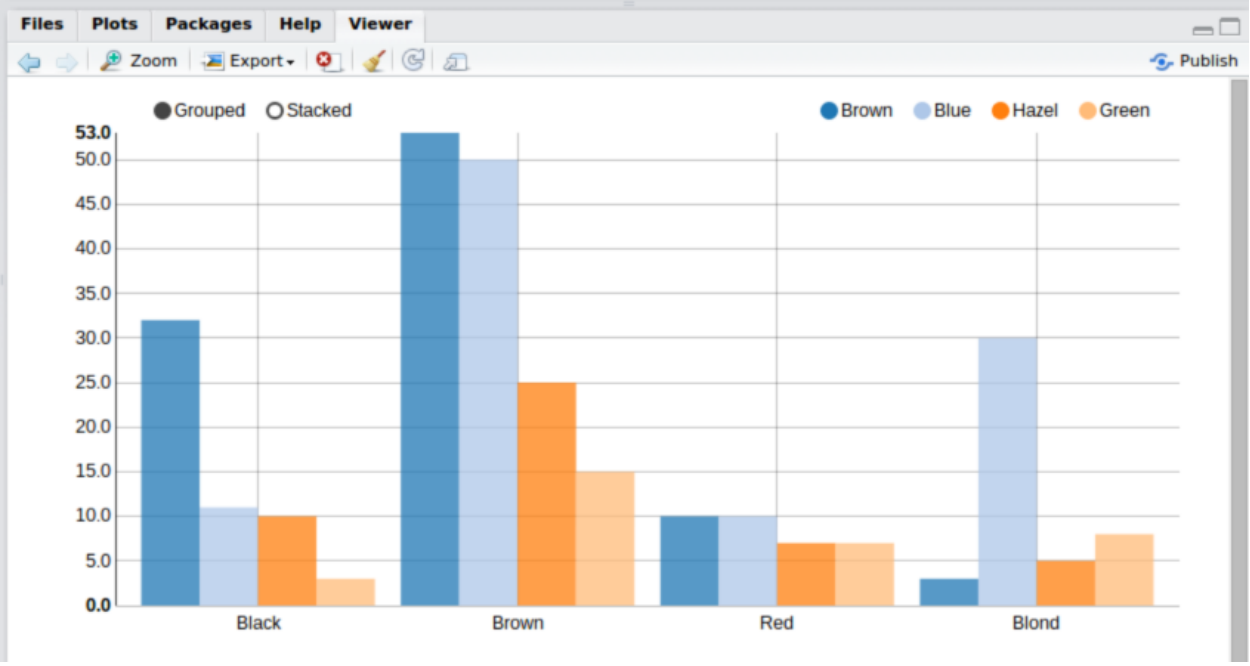
Note that you have to have `devtools` to install `RCharts` directly from github. Now let's try out a first example:

```
## Load the package
require(rCharts)
## Format the data
haireye = as.data.frame(HairEyeColor)
## Generate the plot
n1 = nPlot(Freq ~ Hair, group = 'Eye', type = 'multiBarChart',
  data = subset(haireye, Sex == 'Male')
)
## Display the plot
n1
```

The output can be seen in our `RStudio` display window as:

[36] https://www.youtube.com/watch?v=RF3DaF-ldAw&index=1&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90

[37] http://rcharts.io/



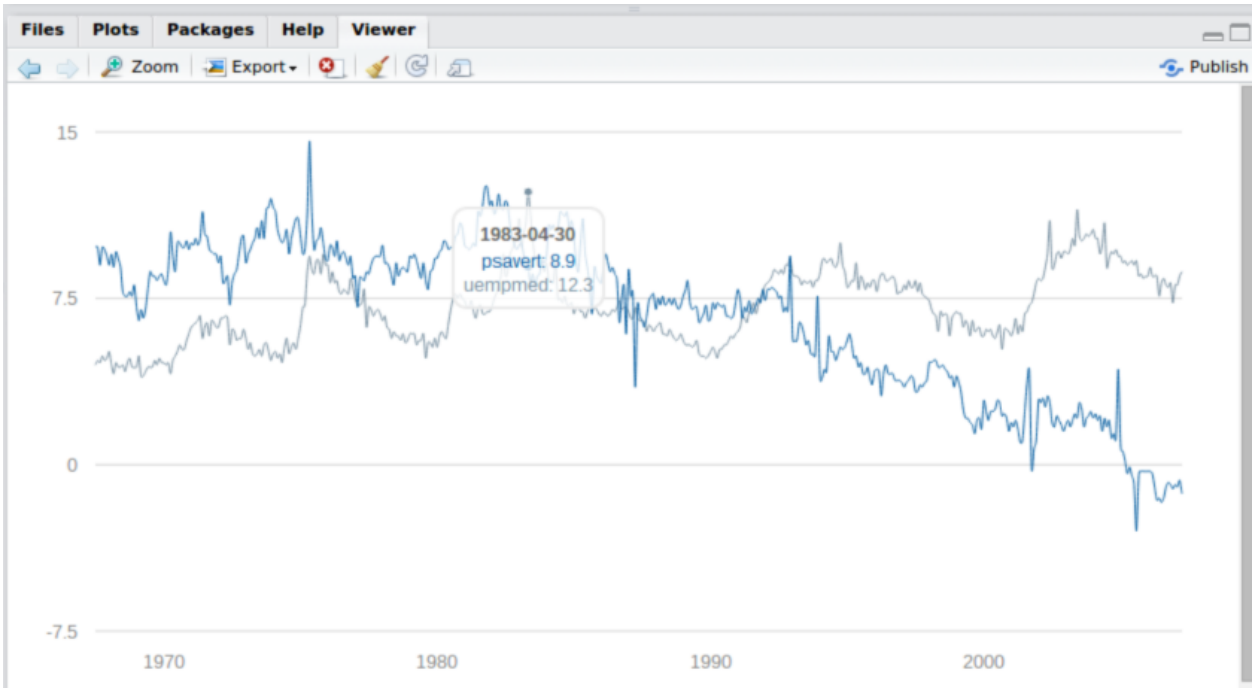
Output of first RCharts example.

Play around with the controls to see the interactivity of the plot. It can switch automatically from a grouped to a stacked bar plot. Groups can be included or omitted by selecting on the factor variables.

The object `n1` contains the plot. You can save the plot with `n1$save(destfile = <FILENAME.HTML>)`. Furthermore, `n1$html()` will display the `html` for the plot. Let's deconstruct another example from Ramnath's page.

```
data(economics, package = "ggplot2")
econ = transform(economics, date = as.character(date))
m1 = mPlot(x = "date", y = c("psavert", "uempmed"), type = "Line", data = econ)
m1$set(pointSize = 0, lineWidth = 1)
m1
```

The output can be seen below.



Output of the second rcharts example

The `data` statement loads the data. The `transform` is used to change the `date` variable to a character variable. The `mplot` statement creates the plot. `date` is the horizontal axis variable while two horizontal variables are shown simultaneously.

The output of an `rCharts` call is an S4 object. The object has slots for a few functions that can change or save the plot. For example, the function `m1$set` changes characteristics about the plot. In this case, it sets the `pointSize = 0` and the `lineWidth` to be small. Other than `set`, here's a collection of other operartions that can be done on to save and publis the plot.

```
m1$print("chart1") # print out the js
m1$save('myPlot.html') #save as html file
m1$publish('myPlot', host = 'gist') # save to gist, rjson required
m1$publish('myPlot', host = 'rpubs') # save to rpubs
```

Now that you have a few examples under your belt, work through the remainder of the examples here[38]. There's a different plot name for each JS library (`mplot`, `nPlot`, `rPlot`).

Ultimately, if you're not willing to get into the weeds, `rCharts`, is most useful for modifying the large bank of available examples. Fortunately, there's an ever expanding list of examples with code that you can rely on.

[38] http://ramnathv.github.io/rCharts/

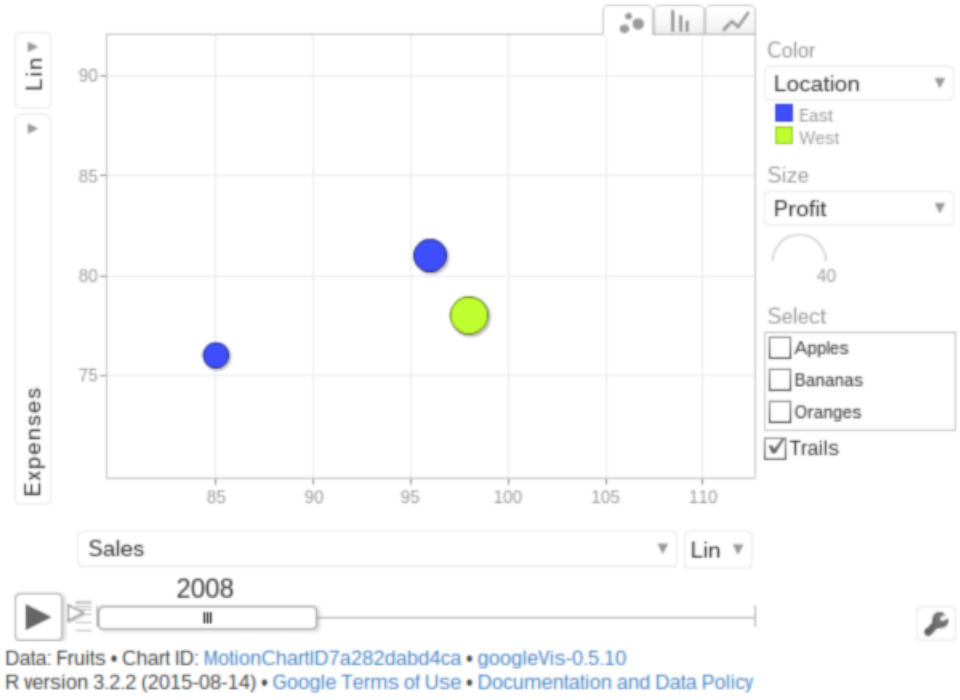
googleVis

Acknowledgement to Jeff Leek for assistance with these notes.

Google has some nice visualization tools built into their products. These include maps and interactive graphs. `googleVis` is an R package to use R as a front end to create google visualizations. You have to do `install.packages("googleVis")` and load the library first. Then try

```
M = gvisMotionChart(Fruits, "Fruit", "Year", options = list(width = 600, height \
= 400))
plot(m)
```

The `gvisMotionChart` creates a motion chart (ala Hans Rosling[39].) The function, `print` for a `googleVis` object displays the html. The result of the `plot` command is shown below.The



Example of `googleVis` motion chart.

Here are some different types of plots and the function to create them:

- Motion charts: gvisMotionChart
- Interactive maps: gvisGeoChart
- Interactive tables: gvisTable

[39] https://www.youtube.com/watch?v=hVimVzgtD6w

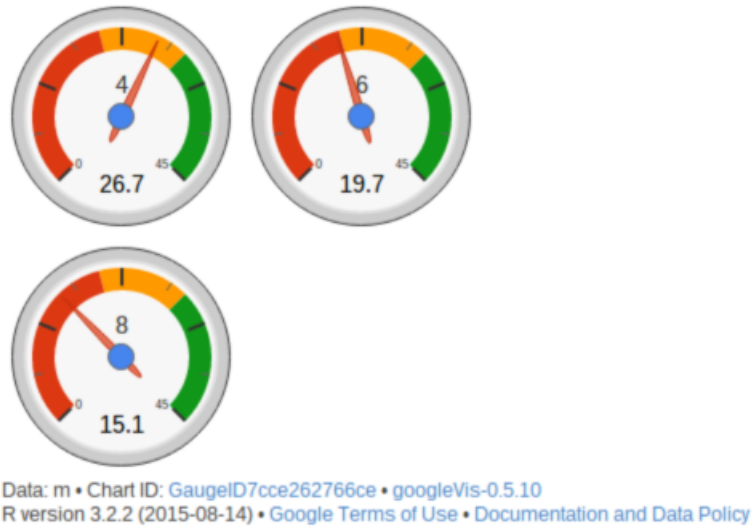
- Line charts: gvisLineChart
- Bar charts: gvisColumnChart
- Tree maps: gvisTreeMap

Like all of these interactive graphing libraries, it's best to work from a similar example. A thorough list of examples can be found here[40].

For fun, let's do a gauge chart. This seems reasonable for the `mtcars` dataset. Let's create gauges for average `mpg` for each `cly`. The code

```
m = aggregate(mpg ~ cyl, mean, data = mtcars) %>%
  transform(cyl = as.character(cyl))
g = gvisGauge(m,
        labelvar = "cyl",
        options=list(min=0, max=45,
                 greenFrom=30, greenTo=45,
                 yellowFrom=20, yellowTo=30,
                 redFrom=0, redTo=20,
                 width=400, height=300)
        )
plot(g)
```

Here's the output:



Output of the `gvisGauge` example.

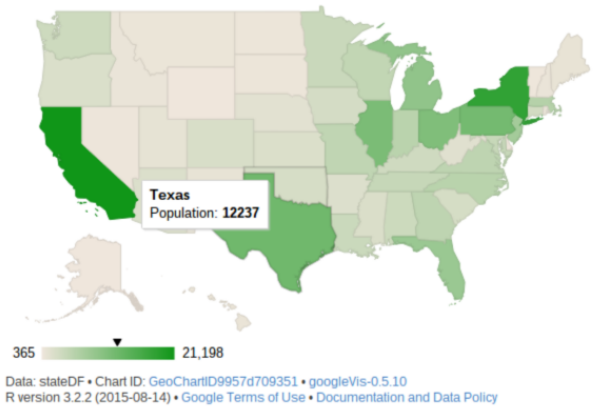
Let's try creating a map. In this example, (expanding on the one here[41]) we create a map with state populations as hover over variables.

[40] https://cran.r-project.org/web/packages/googleVis/vignettes/googleVis_examples.html
[41] https://cran.r-project.org/web/packages/googleVis/vignettes/googleVis_examples.html

```
stateDF = data.frame(State = state.name, state.x77)
gchart = gvisGeoChart(data = stateDF,
                          locationvar = "State",
                          colorvar = "Population",
                          options = list(region="US",
                                         displayMode="regions",
                                         resolution="provinces",
                                         width=600, height=400))

plot(gchart)
```

Do `head(stateDF)` to see the format of the dataframe. The `locationvar` is the variable defining the locations. The `colorvar` variable defines the colorbar and hover over variable. An optional `hovervar` will give the label heading for hovering over. By default, it appears to be taken from `locationvar`. Here's the output with the mouse over Texas:



Example of a Google Chart fromg `googleVis`

That's enough `googleVis` to get you started. It's worth noting that `googleVis` is tremendously well documented. See the collection of vignettes here[42].

[42] https://cran.r-project.org/web/packages/googleVis/index.html

leaflet

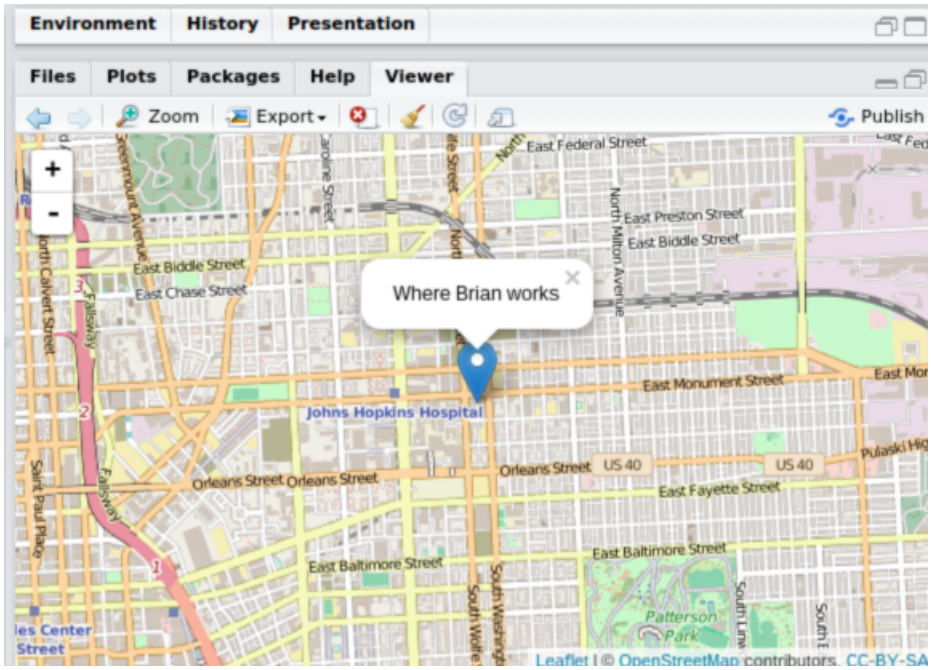
There's quite a few ways to make really slick maps in R. `googleVis` for example leverages Google Maps. Nonetheless, `leaflet` seems to be emerging as the most popular R package for creating interactive maps. We'll go through an example here and hopefully then you can build on it. There's great documentation for `leaflet` here[43].

Let's work through a simple example just to get you started. First, install the `leaflet` package (`install.package`, it's on `cran`) and load it up (`require` or `library`).

Let's work through a simple example. Let's create a map with the Bloomberg School of Public Health at Johns Hopkins as a marker. (That's where your author works at. It's really a neat place. Read more about the JHU Bloomberg School of Public Health here[44])

```
m = leaflet() %>%
        addTiles() %>%
        addMarkers(lat=39.298113, lng=-76.590248, popup="Where Brian works")
m
```

The plot is shown below. Note that it opens in the RStudio display window by default.



Output of simple `leaflet` example.

The map widget (the `leaflet()` command) starts out a map and then you add elements or modify the map by passing it as arguments to mapping functions. So, our code could have been:

[43] https://rstudio.github.io/leaflet/

[44] https://en.wikipedia.org/wiki/Johns_Hopkins_Bloomberg_School_of_Public_Health

```
m = leaflet()
m =         addTiles(m)
m =         addMarkers(m, lat=39.298113, lng=-76.590248, popup="Where Brian works")
```

The `magrittr` code is just useful to clean it up a bit. There's several functions to annotate and markers of different kinds. They all work functionally equivalent to `addMarkers`. The `lat` and `lng` variables can take vectors. In addition, the `leaflet` function can take in a data frame and then `addMarkers` can be composed without arguments.

```
hopkins = data.frame(
        lat = c(39.298113,39.299838),
        lng=c(-76.590248,-76.593143),
        labels=c("BSPH", "KKI")
        )
m = leaflet(hopkins) %>% addTiles %>% addMarkers(popup=~labels)
```

One could use `addMarkers(lat=~<LATITUDE LABEL>, lng = ~ <LONG LABEL>)` if `leaflet` can't figure out the variable names.

The `addTiles` function adds the layer so that you can see the map. Other layers, such as markers and popups can be added later. By default, `addTiles` will just add a basic streetview. Custom tiles can be added as described here[45].

That's enough to get your started on `leaflet`. It's super simple and extremely well documented[46].

plot.ly

Watch this video.[47]

`plotly` relies on the platform/website `plot.ly` for creating interactive graphics. Fortunately, they've made the R coding to get plots into their system trivially easy. They have free plans (for public graphs) and paid plans (for private graphs). Before beginning, sign up for `ploty` at their site.[48]. You can install `plotly` with:

```
require(devtools)
install_github("ropensci/plotly")
```

You must have the `viridis` and `devtools` packages installed. After doing this, set your credentials with

[45] https://rstudio.github.io/leaflet/map_widget.html
[46] https://rstudio.github.io/leaflet/
[47] https://www.youtube.com/watch?v=6ddBAUzIfmw&index=6&list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90
[48] https://plot.ly/how-to-sign-up-to-plotly/

```
Sys.setenv("plotly_username"="your_plotly_username")
Sys.setenv("plotly_api_key"="your_api_key")
```

Your environment variables can be found when you login to `plotly` on their site. Your username is the same one that you used to log onto the site and the `api_key` is shown when you do. You can reset your `api_key` there as well. You can test whether you've done it right with:

```
plotly:::verify("username")
plotly:::verify("api_key")
```

To avoid having to set your environment variables every time, you can put the `Sys.setenv` statements in your `.Rprofile`. Or, if you know how, directly edit your `bashrc`.

Let's try a simple example using the `mtcars` dataset.

```
plot_ly(mtcars, x = hp, y = mpg,
        mode = "markers",
                color = wt,
                text=paste("Weight:", wt))
```

The result is shown below:

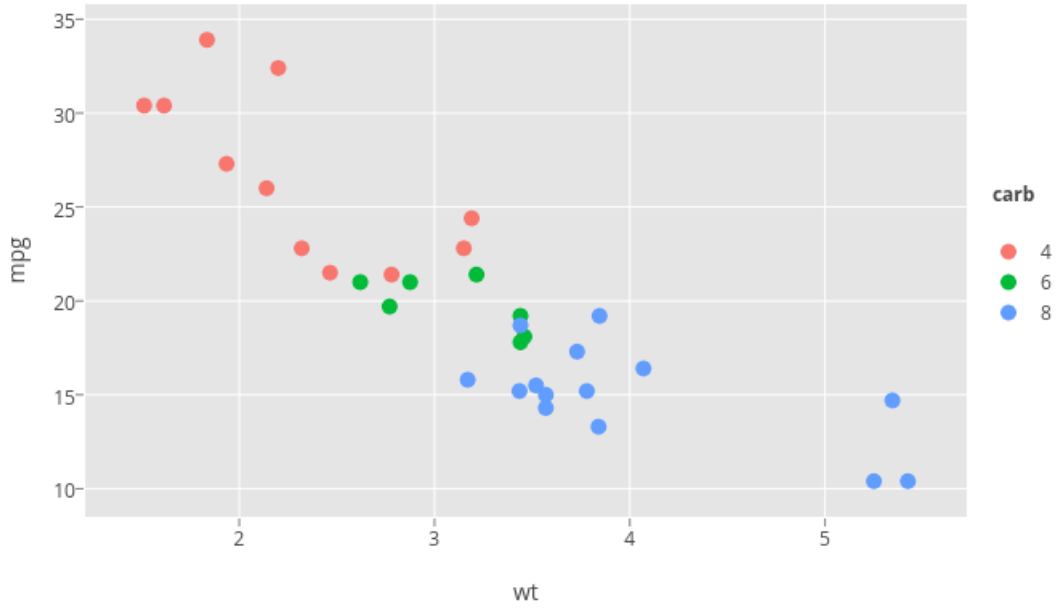
Result of simple plotly example.

Clicking on `Edit Graph` gives a host of options for changing the interactive graph. Plotly will give you the json data and gives a tremendous number of options for publishing the graph.

Notably, `plotly` allows for integration with `ggplot2`. Let's work out our simple example now using ggplot2.

```
library(ggplot2)
g = ggplot(mtcars, aes(x=wt,y=mpg))
g = g + geom_point(aes(color=factor(cyl)))
ggplotly(g)
```

The result is shown below. Here I've omitted the `plotly` controls. However, they remain the same as always.

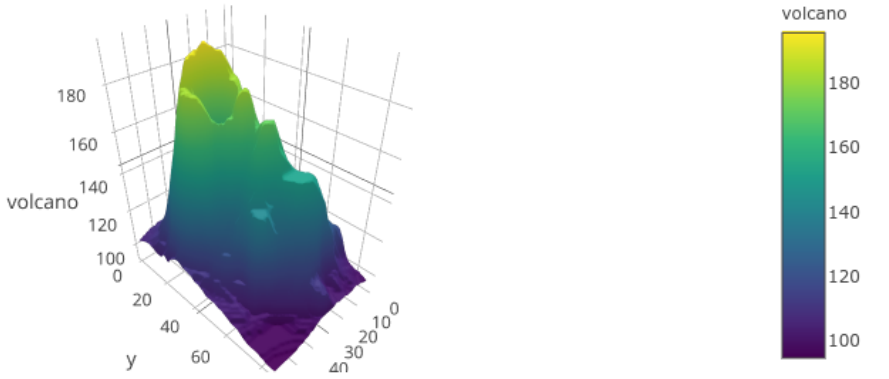


Result of ggplotly.

Finally, let's try an interactive 3D plot.

```
data(volcano)
## Show a static 3D plot in R
persp(z = volcano)
## Show the dynamic 3D plot in plotly
plot_ly(z = volcano, type = "surface")
```

The result is shown below:



The result of a surface plot in plotly/R

Try playing around with the controls. In the next section, we'll show how to create interactive 3D graphics with R. The plotly web site has tons of cool examples[49] that you can use to build off of for creating interactive graphics. Furthermore, the publishing features of `plotly` make it easy to share your graphs on social media or embed on web sites.

[49] https://plot.ly/r/

Summary

As we reach the end of the course notes, I hope that you've taken the time to code and work through the examples in the book. For next steps, you should try creating your own app, and perhaps taking the Developing Data Products Coursera class.

In the terms of next steps, for app development a good place to start would be to dive more deeply into client/server setups and perhaps venture into languages beyond R and Shiny. For interactive graphics, learning some javascript and following that up with D3 would be the logical next step.

Regardless, I am delighted that you made it this far in the text, and hope that you found the whirlwind introduction to the tools useful. We'll hopefully see you in the Coursera Data Science Specialization classes!