Developing Data Products Course Notes

Contents

GitHub

- **Workflow**
  1. make edits in workspace
  2. update index/add files
  3. commit to local repo
  4. push to remote repository
- `git add .` = add all new files to be tracked
- `git add -u` = updates tracking for files that are renamed or deleted
- `git add -A` = both of the above
  - ***Note**: `add` is performed before committing*
- `git commit -m "message"` = commit the changes you want to be saved to the local copy
- `git checkout -b branchname` = create new branch
- `git branch` = tells you what branch you are on
- `git checkout master` = move back to the master branch
- `git pull` = merge you changes into other branch/repo (pull request, sent to owner of the repo)
- `git push` = commit local changes to remote (GitHub)

https://help.github.com/en/github/importing-your-projects-to-github/adding-an-existing-project-to-github-using-the-comm

Adding project to github: (tutorial)

Shiny

- Shiny = platform for creating interactive R program embedded on web pages (made by RStudio)
- knowledge of these helpful: HTML (web page structure), css (style), JavaScript (interactivity)
- **OpenCPU** = project created by Jerom Ooms providing API for creating more complex R/web apps
- `install.packages("shiny"); library(shiny)` = install/load shiny package
- capabilities
  - upload or download files
  - tabbed main panels
  - editable data tables
  - dynamic UI
  - user defined inputs/outputs
  - submit button to control when calculations/tasks are processed

Structure of Shiny App

- **two** scripts (in one directory) make up a Shiny project
  - `ui.R` - controls appearance/all style elements
    - alternatively, a `www` directory with an `index.html` file enclosed can be used instead of `ui.R`
      - ***Note**: output is rendered into HTML elements based on matching their id attribute to an output slot and by specifying the requisite CSS class for the element (in this case either `shiny-text-output`, `shiny-plot-output`, or `shiny-html-output`)*
      - it is possible to create highly customized user-interfaces using user-defined HTML/CSS/JavaScript
  - `server.R` - controls functions
- `runApp()` executes the Shiny application
  - `runApp(display.mode = 'showcase')` = displays the code from `ui.R` and `server.R` and highlights what is being executed depending on the inputs
- ***Note**: `","` must be included **ONLY INBETWEEN** objects/functions on the same level*

ui.R

- `library(shiny)` = first line, loads the shiny package
- `shinyUI()` = shiny UI wrapper, contains sub-methods to create panels/parts/viewable object
- `pageWithSideBar()` = creates page with main/side bar division
- `headerPanel("title")` = specifies header of the page
- `sideBarPanel()` = specifies parameters/objects in the side bar (on the **left**)
- `mainPanel()` = specifies parameters/objects in the main panel (on the **right**)
- for better control over style, use `shinyUI(fluidpage())` (tutorial) <– produces responsive web pages
  - `fluidRow()` = creates row of content with width 12 that can be subdivided into columns
    - `column(4, ...)` = creates a column of width 4 within the fluid row
    - `style = "CSS"` = can be used as the last element of the column to specify additional style
- `absolutePanel(top=0, left=0, right=0)` = used to produce floating panels on top of the page (documentation)
  - `fixed = TRUE` = panel will not scroll with page, which means the panel will always stay in the same position as you scroll through the page
  - `draggable = TRUE` = make panel movable by the user
  - `top = 40` / `bottom = 50` = position from the top/bottom edge of the browser window

    * `top = 0, bottom = 0` = creates panel that spans the entire vertical length of window
  - `left = 40` / `right = 50` = position from the left/right edge of the browser window
    * `top = 0, bottom = 0` = creates panel that spans the entire horizontal length of window
  - `height = 30` / `width = 40` = specifies the height/width of the panel
  - `style = "opacity:0.92; z-index = 100"` = makes panel transparent and ensures the panel is always the top-most element

- **content objects/functions**
  - ***Note**: more HTML tags can be found here*
  - ***Note**: most of the content objects (h1, p, code, etc) can use **both** double and single quotes to specify values, just be careful to be consistent*
  - `h1/2/3/4('heading')` = creates heading for the panel
  - `p('pargraph')` = creates regular text/paragraph
  - `code('code')` = renders code format on the page
  - `br()` = inserts line break
  - `tags$hr()` = inserts horizontal line
  - `tags$ol()`/ `tags$ul()` = initiates ordered/unordered list
  - `div( ... , style = "CSS Code")` / `span( ... , style = "CSS Code")` = used to add additional style to particular parts of the app
    * `div` should be used for a section/block, `span` should be used for a specific part/inline
  - `withMathJax()` = add this element to allow Shiny to process LaTeX
    * inline LaTex must be wrapped like this: `\\(LaTeX\\)`
    * block equations are still wrapped by: `$$LaTeX$$`

- **inputs**
  - `textInput(inputId = "id", label = "textLabel")` = creates a plain text input field
    * `inputId` = field identifier
    * `label` = text that appear above/before a field
  - `numericInput('HTMLlabel', 'printedLabel', value = 0, min = 0, max = 10, step = 1)` = create a number input field with incrementer (up/down arrows)
    * `'HTMLLabel'` = name given to the field, not printed, and can be called
    * `'printedLabel'` = text that shows up above the input box explaining the field
    * `value` = default numeric value that the field should take; 0 is an example
    * `min` = minimum value that can be set in the field (if a smaller value is manually entered, then the value becomes the minimum specified once user clicks away from the field)
    * `max` = max value that can be set in the field
    * `step` = increments for the up/down arrows
    * more arguments can be found in `?numericInput`
  - `checkboxGroupInput("id2", "Checkbox",choices = c("Value 1" = "1", ...), selected = "1", inline = TRUE)` = creates a series of checkboxes
    * `"id2", "Checkbox"` = field identifier/label
    * `choices` = list of checkboxes and their labels
      · format = `"checkboxName" = "fieldIdentifier"`
      · ***Note**: `fieldIdentifier` should generally be different from checkbox to checkbox, so we can properly identify the responses*
    * `selected` = specifies the checkboxes that should be selected by default; uses `fieldIndentifier` values
    * `inline` = whether the options should be displayed inline
  - `dateInput("fieldID", "fieldLabel")` = creates a selectable date field (dropdown calendar/date picker automatically generated)
    * `"fieldID"` = field identifier

    - * `"fieldLabel"` = text/name displayed above fields
    - * more arguments can be found in `?dateInput`
  - `submitButton("Submit")` = creates a submit button that updates the output/calculations only when the user submits the new inputs (default behavior = all changes update reactively/in real time)
  - `actionButton(inputId = "goButton", label = "test")` = creates a button with the specified label and id
    - * output can be specified for when the button is clicked
  - `sliderInput("id", "label", value = 70, min = 62, max = 74, 0.05)` = creates a slider for input
    - * arguments similar to `numericInput` and more information can be found `?sliderInput`

- **outputs**
  - ***Note**: every variable called here must have a corresponding method corresponding method from the `output` element in `server.R` to render their value*
  - `textOutput("fieldId", inline = FALSE)` = prints the value of the variable/field in text format
    - * `inline = TRUE` = inserts the result inline with the HTML element
    - * `inline = FALSE` = inserts the result in block code format
  - `verbatimTextOutput("fieldId")` = prints out the value of the specified field defined in `server.R`
  - `plotOutput('fieldId')` = plots the output ('sampleHist' for example) created from `server.R` script
  - `output$test <- renderText({input$goButton}); isolate(paste(input$t1, input$2))})` = `isolate` action executes when the button is pressed
    - * `if (input$goButton == 1){  Conditional statements }` = create different behavior depending on the number of times the button is pressed

ui.R Example

- below is part of the ui.R code for a project on Shiny

```
# load shiny package
library(shiny)
# begin shiny UI
shinyUI(navbarPage("Shiny Project",
    # create first tab
    tabPanel("Documentation",
        # load MathJax library so LaTeX can be used for math equations
        withMathJax(), h3("Why is the Variance Estimator \\(S^2\\) divided by \\(n-1?\\)"),
        # paragraph and bold text
        p("The ", strong("sample variance"),"" can be calculated in ", strong(em("two")),
          " different ways:",
          "$$S^2 \\mbox{(unbiased)} = \\frac{\\sum_{i=1}^n (X_i - \\bar X)^2}{n-1}
          ~~~\\mbox{and}~~S^2\\mbox{(biased)}=\\frac{\\sum_{i=1}^n (X_i-\\bar X)^2}{n}$$",
          "The unbiased calculation is most often used, as it provides a ",
          strong(em("more accurate")), " estimate of population variance"),
        # break used to space sections
        br(), p("To show this empirically, we simulated the following in the ",
                strong("Simulation Experiment"), " tab: "), br(),
        # ordered list
        tags$ol(
            tags$li("Create population by drawing observations from values 1 to 20."),
            tags$li("Draw a number of samples of specified size from the population"),
```

```
            tags$li("Plot difference between sample and true population variance"),
            tags$li("Show the effects of sample size vs accuracy of variance estimated")
        )),
    # second tab
    tabPanel("Simulation Experiment",
        # fluid row for space holders
        fluidRow(
            # fluid columns
            column(4, div(style = "height: 150px")),
            column(4, div(style = "height: 150px")),
            column(4, div(style = "height: 150px"))),
        # main content
        fluidRow(
            column(12,h4("We start by generating a population of ",
                         span(textOutput("population", inline = TRUE),
                         style = "color: red; font-size: 20px"),
                         " observations from values 1 to 20:"),
                   tags$hr(),htmlOutput("popHist"),
                   # additional style
                   style = "padding-left: 20px"
            )
        ),
        # absolute panel
        absolutePanel(
            # position attributes
            top = 50, left = 0, right =0,
            fixed = TRUE,
            # panel with predefined background
            wellPanel(
                fluidRow(
                    # sliders
                    column(4, sliderInput("population", "Size of Population:",
                                          min = 100, max = 500, value = 250),
                           p(strong("Population Variance: "),
                           textOutput("popVar", inline = TRUE))),
                    column(4, sliderInput("numSample", "Number of Samples:",
                                          min = 100, max = 500, value = 300),
                           p(strong("Sample Variance (biased): "),
                           textOutput("biaVar", inline = TRUE))),
                    column(4, sliderInput("sampleSize", "Size of Samples:",
                                          min = 2, max = 15, value = 10),
                           p(strong("Sample Variance (unbiased): "),
                           textOutput("unbiaVar", inline = TRUE)))),
                style = "opacity: 0.92; z-index: 100;"
            ))
    )
))
```

server.R

- preamble/code to set up environment (executed only ***once***)
  - start with `library()` calls to load packages/data

- define/initiate variables and relevant default values
  - `<<-` operator should be used to assign values to variables in the parent environment
  - `x <<- x + 1` will define x to be the sum of 1 and the value of x (defined in the parent environment/working environment)
- any other code that you would like to only run once

- `shinyServer()` = initiates the server function
  - `function(input, output){}` = defines a function that performs actions on the inputs user makes and produces an output object
  - **non-reactive** statements/code will be executed ***once for each page refresh/submit***
  - **reactive** functions/code are ***run repeatedly*** as values are updated (i.e. render)
    - ***Note****: Shiny only runs what is needed for reactive statements, in other words, the rest of the code is left alone*
    - `reactive(function)` = can be used to wrap functions/expressions to create reactive expressions
      - `renderText({x()})` = returns value of x, "()" must be included (syntax)

- **reactive function example**

```
# start shinyServer
shinyServer(
    # specify input/output function
    function(input, output) {
        # set x as a reactive function that adds 100 to input1
        x <- reactive({as.numeric(input$text1)+100})
        # set value of x to output object text1
        output$text1 <- renderText({x()                          })
        # set value of x plus value of input object text2 to output object text1
        output$text2 <- renderText({x() + as.numeric(input$text2)})
    }
)
```

- **functions/output objects in `shinyServer()`**
  - `output$oid1 <- renderPrint({input$id1})` = stores the user input value in field `id1` and stores the rendered, printed text in the `oid1` variable of the `output` object
    - `renderPrint({expression})` = reactive function to render the specified expression
    - `{}` is used to ensure the value is an expression
    - `oid1` = variable in the output object that stores the result from the subsequent command
  - `output$sampleHist <- renderPlot({code})` = stores plot generated by code into `sampleHist` variable
    - `renderPlot({code})` = renders a plot generated by the enclosed R code
  - `output$sampleGVisPlot <- renderGvis({code})` = renders Google Visualization object

server.R Example

- below is part of the server.R code for a project on Shiny that uses googleVis

```
# load libraries
library(shiny)
require(googleVis)
# begin shiny server
```

```
shinyServer(function(input, output) {
    # define reactive parameters
    pop<- reactive({sample(1:20, input$population, replace = TRUE)})
    bootstrapSample<-reactive({sample(pop(),input$sampleSize*input$numSample,
        replace = TRUE)})
    popVar<- reactive({round(var(pop()),2)})
    # print text through reactive funtion
    output$biaVar <- renderText({
        sample<- as.data.frame(matrix(bootstrapSample(), nrow = input$numSample,
            ncol =input$sampleSize))
        return(round(mean(rowSums((sample-rowMeans(sample))^2)/input$sampleSize), 2))
    })
    # google visualization histogram
    output$popHist <- renderGvis({
        popHist <- gvisHistogram(data.frame(pop()), options = list(
            height = "300px",
            legend = "{position: 'none'}", title = "Population Distribution",
            subtitle = "samples randomly drawn (with replacement) from values 1 to 20",
            histogram = "{ hideBucketItems: true, bucketSize: 2 }",
            hAxis = "{ title: 'Values', maxAlternation: 1, showTextEvery: 1}",
            vAxis = "{ title: 'Frequency'}"
        ))
        return(popHist)
    })
})
```

Distributing Shiny Application

- running code locally = running local server and browser routing through local host
- quickest way = send application directory
- possible to create R package and create a wrapper that calls `runApp` (requires R knowledge)
- another option = run shiny server (link)

Debugging

- `runApp(display.mode = 'showcase')` = highlights execution while running a shiny application
- `cat` = can be used to display output to stdout/R console
- `browser()` = interrupts execution (tutorial)

Tutorial

For more info: (tutorial)

ShinyApps.io (link)

- platform created by RStudio to share Shiny apps on the web
- log in through GitHub/Google, and set up access in R
    1. Make sure you have `devtools` installed in R (`install.packages("devtools")`)
    2. enter `devtools::install_github('rstudio/shinyapps')`, which installs the `shinyapps` package from GitHub

3. follow the instructions to authenticate your shiny apps account in R through the generated token
4. publish your app through `deployApp()` command

- the apps your deploy will be hosted on ShinyApps.io under your account

Tutorial

For more info: (tutorial)

Presentations

Slidify

- create data-centric presentations created by Ramnath Vaidyanathan
- amalgamation of knitr, Markdown, JavaScript libraries for HTML5 presentations
- easily extendable/customizable
- allows embedded code chunks and mathematical formulas (MathJax JS library) to be rendered correctly
- final products are HTML files, which can be viewed with any web browser and shared easily
- **installation**

  1. make sure you have `devtools` package installed in R
  2. enter `install_github('ramnathv/slidify'); install_github(''ramnathv/slidifyLibraries'')` to install the slidify packages
  3. load slidify package with `library(slidify)`
  4. set the working directory to the project you are working on with `setwd("~/project")`

- `author("title")` = sets up initial files for a new slidify project (performs the following things)

  1. `title` (or any name you typed) directory is created inside the current working directory
  2. `assets` subdirectory and a file named `index.Rmd` are created inside `title` directory
  3. `assets` subdirectory is populated with the following empty folders:
     - `css`
     - `img`
     - `js`
     - `layouts`
     - ***Note**: any custom CSS/images/JavaScript you want to use should be put into the above folders correspondingly*
  4. `index.Rmd` R Markdown file will open up in RStudio

- `slidify("index.Rmd")` = processes the R Markdown file into a HTML page and imports all necessary libraries
- `library(knitr); browseURL("index.html")` = opens up the built-in web browser in R Studio and displays the slidify presentation
  - ***Note**: this is only necessary the first time; you can refresh the page to reflect any changes after saving the HTML file*

YAML (YAML Ain't Markup Language/Yet Another Markup Language)

- used to specify options for the R Markdown/slidify at the beginning of the file
- format: `field : value # comment`
  - `title` = title of document
  - `subtitle` = subtitle of document
  - `author` = author of document
  - `job` = occupation of author (can be left blank)
  - `framework` = controls formatting, usually the name of a library is used (i.e. `io2012`)
    - io2012
    - html5slides
    - deck.js
    - dzslides
    - landslide
    - Slidy
  - `highlighter` = controls effects for presentation (i.e `highlight.js`)
  - `hitheme` = specifies theme of code (i.e. `tomorrow`)
  - `widgets` = loads additional libraries to display LaTeX math equations(`mathjax`), quiz-styles components (quiz), and additional style (`bootstrap` = Twitter-created style)
    - for math expressions, the code should be enclosed in `$expresion$` for inline expressions, and `$$expression$$` for block equations
  - `mode = selfcontained/standalone/draft` = depending whether the presentation will be given with Internet access or not
    - `standalone` = all the JavaScript libraries will be save locally so that the presentation can be executed without Internet access
    - `selfcontained` = load all JavaScript library at time of presentation
  - `logo` = displays a logo in title slide
  - `url` = specify path to assets/other folders that are used in the presentation
    - ***Note:*** *`../` signifies the parent directory*
- **example**

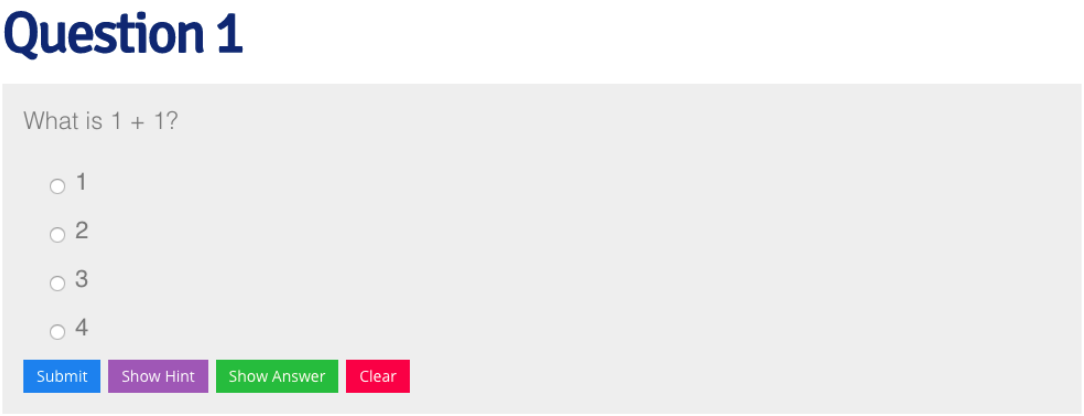
```
---
title       : Slidify
subtitle    : Data meets presentation
author      : Jeffrey Leek, Assistant Professor of Biostatistics
job         : Johns Hopkins Bloomberg School of Public Health
logo        : bloomberg_shield.png
framework   : io2012        # {io2012, html5slides, shower, dzslides, ...}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

Slides

- `##` = signifies the title of the slide → equivalent of `h1` element in HTML
- `---` = marks the end of a slide

- `.class #id` = assigns `class` and `id` attributes (CSS) to the slide and can be used to customize the style of the page
- ***Note**: make sure to leave space between each component of the slidify document (title, code, text, etc) to avoid errors*
- advanced HTML can be added directly to the `index.Rmd` file and most of the time it should function correctly
- interactive element (quiz questions, rCharts, shiny apps) can be embedded into slidify documents (demos)
  - quiz elements
    - `--- &radio` before slide content for multiple choice (make sure quiz is included in widgets)
    - `##` = signifies title of questions
    - the question can be type in plain text format
    - the multiple choice options are listed by number (`1. a`, `2. b`, etc.)
      - wrap the correct answer in underscores (`2. _b_`)
    - `*** .hint` = denotes the hint that will be displayed when the user clicks on ***Show Hint*** button
    - `*** .explanation` = denotes the explanation that will be displayed when the user clicks on ***Show Answer*** button
    - a page like the one below will be generated when processed with `slidify`

```
--- &radio

## Question 1

What is 1 + 1?

1. 1
2. _2_
3. 3
4. 4

*** .hint
This is a hint

*** .explanation
This is an explanation
```



- **`knit HTML`** button can be used to generate previews for the presentation as well

Publishing

- first, you will need to create a new repository on GitHub
- `publish_github("user", "repo")` can be used to publish the slidify document on to your on-line repo

RStudio Presentation

- presentation authoring tool within the RStudio IDE (tutorial)
- output = html5 presentation
- `.Rpres` file → converted to `.md` file → `.html` file
- uses R Markdown format as slidify/knitr
  - **mathjax** JS library is loaded by default
- RStudio format/runs the code when the document is saved

Creating Presentation

- file → New File → R Presentation (`alt-f` + `f` + `p`)
- `class: classname` = specify slide-specific control from CSS
- `css: file.css` = can be used to import an external CSS file
  - alternatively, a css file that has the same name as the presentation will be automatically loaded
- knowledge of CSS/HTML/JavaScript useful to customize presentation more granularly
  - ***Note**: though the end HTML file can be edited directly, it should be used as a last resort as it defeats the purpose of reproducible presentations*
- clicking on **Preview** button brings up ***Presentation*** viewer in RStudio
  - *navigation controls* (left and right arrows) are located in right bottom corner

  - the *Notepad* icon on the menu bar above displays the section of code that corresponds with the current slide in the main window
  - the *More* button has four options
    * "Clear Knitr Cache" = clears cache for the generated presentation previews
    * "View in Browser" = creates temporary HTML file and opens in default web browser (does not create a local file)
    * "Save as Web Page" = creates a copy of the presentation as a web page
    * "Publish to RPubs" = publishes presentation on RPubs
  - the *Refresh* button refreshes the page
  - the *Zoom* button opens a new window to display the presentation

- **transitions between slides**
  - just after the beginning of each slide, the `transition` property (similar to YAML) can be specified to control the transition between the previous and current slides
  - `transition: linear` = creates 2D linear transition (html5) between slides
  - `transition: rotate` = creates 3D rotating transition (html5) between slides
  - more transition options are found here

- **hierarchical organization**
  - attribute `type` can be added to specify the appearance of the slide ("slide type")
  - `type: section` and `type: sub-section` = distinct background and font colors, slightly larger heading text, appear at a different indent level within the slide navigation menu
  - `type: prompt` and `type: alert` = distinct background color to communicate to viewers that the slide has different intent

- **columns**
  - simply place `***` in between two sections of content on a slide to separate it into two columns
  - `left: 70%` can be used to specify the proportions of each column
  - `right: 30%` works similarly

- **change slide font (guide)**
  - `font-family: fontname` = changes the font of slide (specified in the same way as HTML)
  - `font-import: http://fonts.googleapis.com/css?family=Risque` = imports font
    * ***Note**: fonts must be present on the system for presentation (or have Internet), or default fonts will be used*
  - ***Note**: CSS selectors for class and IDs must be preceded by `.reveal` to work (`.reveal section del` applies to any text enclosed by `~~text~~`)*

Slidify vs RStudio Presenter

- **Slidify**
  - flexible control from the `.Rmd` file
  - under constant development
  - large user base, more likely to get answer on *StackOverflow*
  - lots of styles and options by default
  - steeper learning curve
  - more command-line oriented

- **R Studio Presenter**
  - embedded in R Studio
  - more GUI oriented
  - very easy to get started

- smaller set of easy styles and options
- default styles look nice
- as flexible as Slidify with CSS/HTML knowledge

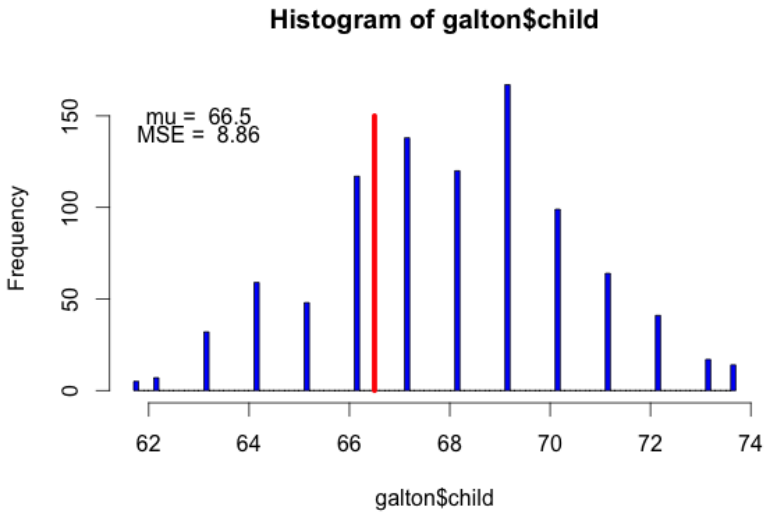
Nice Viz Packages

`manipulate` Package

- `manipulate` = package/function can be leveraged to create quick interactive graphics by allowing the user to vary the different variables to a model/calculation
- creates sliders/checkbox/picker for the user (documentation)

Example

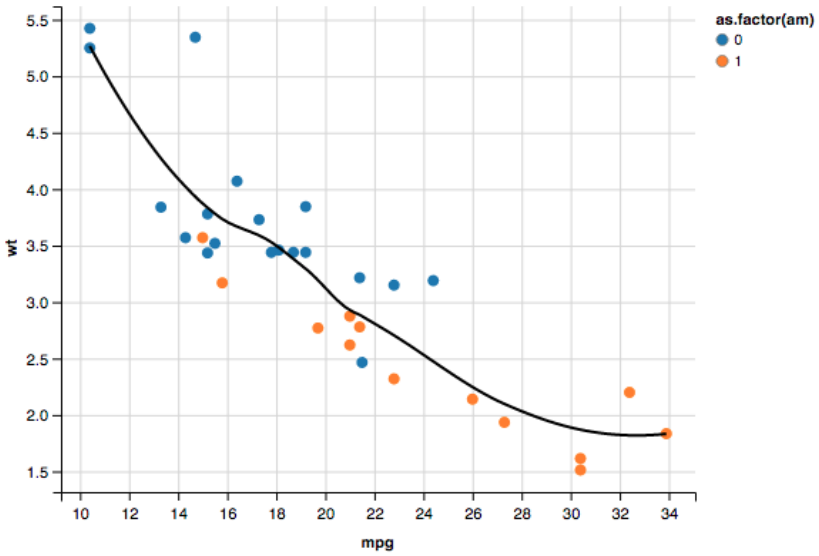
```
# load data and manipulate package
library(UsingR)
library(manipulate)
# plotting function
myHist <- function(mu){
    # histogram
    hist(galton$child,col="blue",breaks=100)
    # vertical line to highlight the mean
    lines(c(mu, mu), c(0, 150),col="red",lwd=5)
    # calculate mean squared error
    mse <- mean((galton$child - mu)^2)
    # updates the mean value as the mean is changed by the user
    text(63, 150, paste("mu = ", mu))
    # updates the mean squared error value as the mean is changed by the user
    text(63, 140, paste("MSE = ", round(mse, 2)))
}
# creates a slider to vary the mean for the histogram
manipulate(myHist(mu), mu = slider(62, 74, step = 0.5))
```



ggvis package

- `ggvis` is a data visualization package for R that lets you:
  - declaratively describe data graphics with a syntax similar in spirit to ggplot2
  - create rich interactive graphics that you can play with locally in Rstudio or in your browser
  - leverage `shiny`'s infrastructure to publish interactive graphics usable from any browser (either within your company or to the world).
- the goal is to combine the best of R and the best of the web
  - data manipulation and transformation are done in R
  - graphics are rendered in a web browser, using Vega
  - for RStudio users, ggvis graphics display in a viewer panel, which is possible because RStudio is a web browser
- can use the **pipe operator**, `%>%`, to chain graphing functions
  - `set_options(renderer = "canvas")` = can be used to control what renderer the graphics is produced with
  - ***example***: `mtcars %>% ggvis(~mpg, ~wt, fill = ~ as.factor(am)) %>% layer_points() %>% layer_smooths()`

```
# for pdf version
library(ggvis)
mtcars %>% ggvis(~mpg, ~wt, fill = ~ as.factor(am)) %>% layer_points() %>% layer_smooths()
```



GoogleVis API

- GoogleVis allows R to create interactive HTML graphics (Google Charts)
- **chart types** (format = gvis+ChartType)
  - Motion charts: `gvisMotionChart`
  - Interactive maps: `gvisGeoChart`
  - Interactive tables: `gvisTable`
  - Line charts: `gvisLineChart`

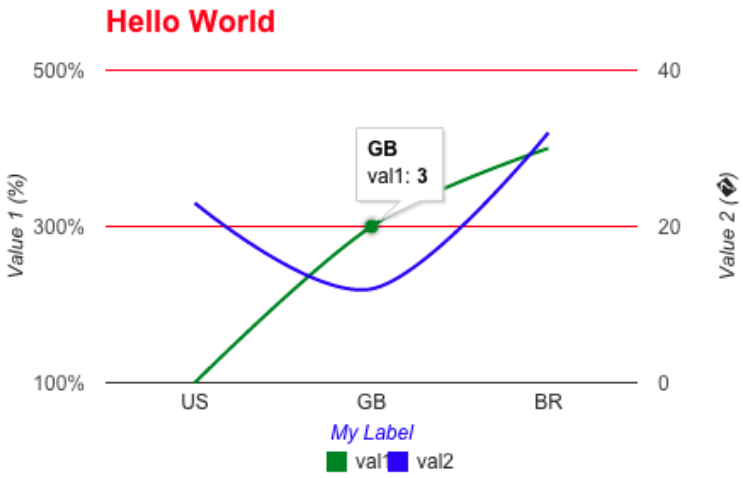
  - Bar charts: `gvisColumnChart`
  - Tree maps: `gvisTreeMap`
  - more charts can be found here

- configuration options and default values for arguments for each of the plot types can be found here
- `print(chart, "chart")` = prints the JavaScript for creating the interactive plot so it can be embedded in slidify/HTML document
  - `print(chart)` = prints HTML + JavaScript directly
- alternatively, to print the charts on a HTML page, you can use `op <- options(gvis.plot.tag='chart')`
  - this sets the googleVis options first to change the behaviour of `plot.gvis`, so that ***only the chart component*** of the HTML file is written into the output file
  - `plot(chart)` can then be called to print the plots to HTML
- `gvisMerge(chart1, chart2, horizontal = TRUE, tableOptions = "bgcolor = \"#CCCCCC\" cellspacing = 10)` = combines the two plots into one horizontally (1 x 2 panel)
  - ***Note:*** *`gvisMerge()` can only combine* ***TWO*** *plots at a time*
  - `horizontal = FALSE` = combines plots vertically (TRUE for horizontal combination)
  - `tableOptions = ...` = used to specify attributes of the combined plot
- `demo(googleVis)` = demos how each of the plot works
- **resources**
  - vignette
  - documentation
  - plot gallery
  - FAQ

Example (line chart)

```
# load googleVis package
library(googleVis)
# set gvis.plot options to only return the chart
op <- options(gvis.plot.tag='chart')
# create initial data with x variable as "label" and y variable as "var1/var2"
df <- data.frame(label=c("US", "GB", "BR"), val1=c(1,3,4), val2=c(23,12,32))
# set up a gvisLineChart with x and y
Line <- gvisLineChart(df, xvar="label", yvar=c("val1","val2"),
        # set options for the graph (list) - title and location of legend
        options=list(title="Hello World", legend="bottom",
                # set title text style
                titleTextStyle="{color:'red', fontSize:18}",
                # set vertical gridlines
                vAxis="{gridlines:{color:'red', count:3}}",
                # set horizontal axis title and style
                hAxis="{title:'My Label', titleTextStyle:{color:'blue'}}",
                # set plotting style of the data
                series="[{color:'green', targetAxisIndex: 0},
                         {color: 'blue',targetAxisIndex:1}]",
                # set vertical axis labels and formats
                vAxes="[{title:'Value 1 (%)', format:'##,######%'},
                                 {title:'Value 2 (\U00A3)'}]",
                # set line plot to be smoothed and set width and height of the plot
```

```
                curveType="function", width=500, height=300
                ))
# print the chart in JavaScript
plot(Line)
```



Example (merging graphs)

```
G <- gvisGeoChart(Exports, "Country", "Profit",options=list(width=200, height=100))
T1 <- gvisTable(Exports,options=list(width=200, height=270))
M <- gvisMotionChart(Fruits, "Fruit", "Year", options=list(width=400, height=370))
GT <- gvisMerge(G,T1, horizontal=FALSE)
GTM <- gvisMerge(GT, M, horizontal=TRUE,tableOptions="bgcolor=\"#CCCCCC\" cellspacing=10")
plot(GTM)
```

- ***Note**: the motion chart only displays when it is hosted on a server or a trusted Macromedia source, see googlVis vignette for more details*

plot.ly (link)

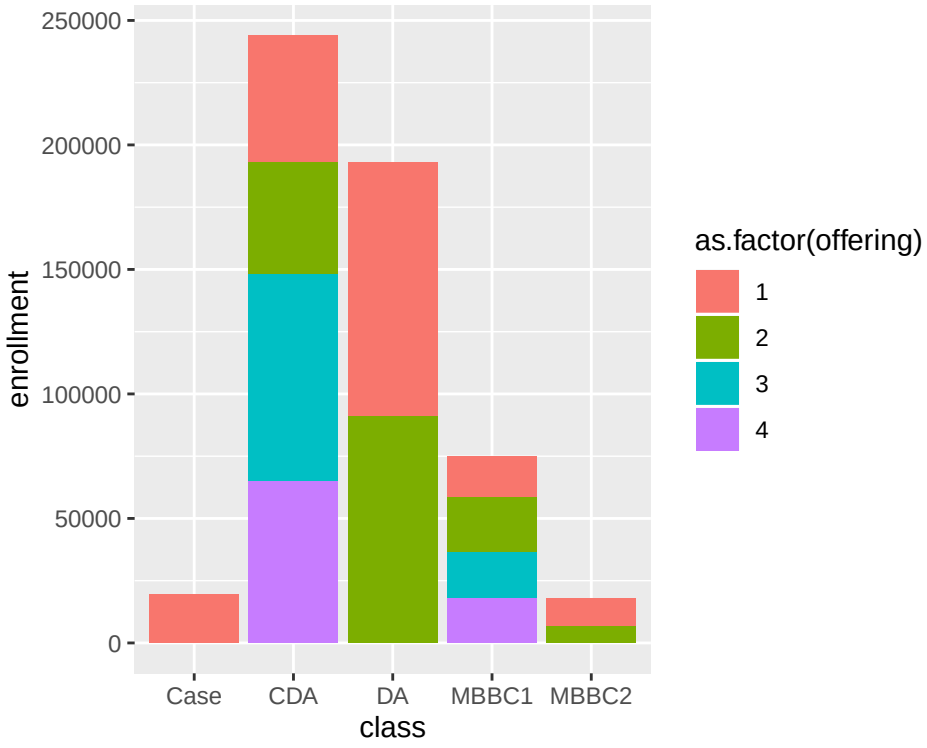
- platform share and edit plots modularly on the web (examples)
  - every part of the plot can be customized and modified
  - graphs can be converted from one language to another
- you can choose to log in through Facebook/Twitter/Google/GitHub

1. make sure you have `devtools` installed in R
2. enter `devtools::install_github("ropensci/plotly")`, which installs `plotly` package from GitHub
3. go to https://plot.ly/r/getting-started/ and follow the instructions
4. enter `library(plotly); set_credentials_file("<username>", "<token>")` with the appropriate username and token filled in
5. use `plotly()` methods to upload plots to your account
6. modify any part of the plot as you like once uploaded
7. share the plot

Example

```
# load packages
library(plotly); library(ggplot2)
# make sure your plot.ly credentials are set correctly using the following command
```

```
#   set_credentials_file(username=<FILL IN>, api_key=<FILL IN>)

# load data
load("courseraData.rda")
# bar plot using ggplot2
g <- ggplot(myData, aes(y = enrollment, x = class, fill = as.factor(offering)))
g <- g + geom_bar(stat = "identity")
g
```




```
# interface with plot.ly and ggplot2 to upload the plot to plot.ly under your credentials
ggplotly(g)
```

- the above is the response URL for the plot created on plot.ly
  - *__Note__: this particular URL corresponds to the plot on my personal account