Co-lab Shiny Workshop

Integrating Shiny and `plotly`

November 14, 2019

thomas.balmat@duke.edu
rescomputing@duke.edu

In previous sessions of the series, we used features of Shiny and `ggplot` to accept input from a user and, in the context of analyses implemented, present informative results intended to guide the analyst to further exploration of the data. One strength of a Shiny app is that analyses, tables, and graphs are prepared with no requirement from the user other than to "fill out" the input form and wait a moment in anticipation. The ease of iterative adjustment of on-screen parameters and review promotes idea generation and validation, making a well designed Shiny app a resource for exploratory research. In this session, we will use `plotly` to make a `ggplot` more dynamic, by adding hover labels and clickable geoms that enable filtering, highlighting, and modifying a graph as the user probes visible features and relationships in the data. Because we are using `plotly` in a Shiny context, our emphasis will be on the interaction of `plotly` features and functions with your Shiny script, particularly within the `server()` function.

1 Overview

- Preliminaries
  - What can `Shiny and ggplot` do for you?
  - What are your expectations of this workshop?
  - Interest in R Day?
- Examples
- Resources
- Anatomy of a Shiny app
- Workshop material
  - From github (execute locally)
  - From RStudio Cloud
- Review previous app, `ggplot()` U.S. domestic flight map
- Hello `plotly`
- `plotly` app: GWAS pleiotropy
- Debugging

2 Examples

- `plotly` Visualizations
  - `plotly` gallery: https://plot.ly/r/shiny-gallery/
  - Frank Harrell
    - More with less: https://www.fharrell.com/post/interactive-graphics-less/
    - Hmisc package: https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf
    - Recent presentation: http://hbiostat.org/talks/rmedicine19.html
- Shiny Apps
  - Duke Data+ project: *Big Data for Reproductive Health*, http://bd4rh.rc.duke.edu:3838
  - Duke Med H2P2 Genome Wide Association Study: http://h2p2.oit.duke.edu

3 Resources

- R
  - Books
    - Norm Matloff, *The Art of R Programming*, No Starch Press
    - Wickham and Grolemund, *R for Data Science*, O'Reilly
    - Andrews and Wainer, *The Great Migration: A Graphics Novel*, https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1740-9713.2017.01070.x
    - Friendly, *A Brief History of Data Visualization*, http://datavis.ca/papers/hbook.pdf
  - Reference cards
    - R reference card: https://cran.r-project.org/doc/contrib/Short-refcard.pdf
    - Base R: https://rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf
    - Shiny, `ggplot`, `markdown`, `dplyr`, `tidy`: https://rstudio.com/resources/cheatsheets/
- Shiny
  - `?shiny` from the R command line
  - Click `shiny` in the `Packages` tab of RStudio
  - https://cran.r-project.org/web/packages/shiny/shiny.pdf
- `plotly`
  - `?plotly` from the R command line
  - Click `plotly` in the `Packages` tab of RStudio
  - https://cran.r-project.org/web/packages/plotly/plotly.pdf
- Workshop materials
  - https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3

4 Anatomy of a Shiny App

A Shiny app is an R script executing in an active R environment that uses functions available in the Shiny package to interact with a web browser. The basic components of a Shiny script are

- `ui()` function
  - Contains your web page layout and screen objects for inputs (prompt fields) and outputs (graphs, tables, etc.)
  - Is specified in a combination of Shiny function calls and raw HTML
  - Defines variables that bind web objects to the execution portion of the app
- `server()` function
  - The execution portion of the app
  - Contains a combination of standard R statements and function calls, such as to `apply()`, `lm()`, `ggplot()`, etc., along with calls to functions from the Shiny package that enable reading of on-screen values and rendering of results
- `runApp()` function
  - Creates a process listening on a tcp port, launches a browser (optional), renders a screen by calling the specified `ui()` function, then executes the R commands in the specified server() function

5 Access Workshop Material

This document:

https://github.com/tbalmat/Duke-Co-lab/blob/master/Session-4/CourseOutline/Co-lab-Session-4-plotly.pdf

5.1 Execute Locally (copy workshop material from github repo)

- Copy scripts and data from https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3
- In a separate directory, copy scripts and data from https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-4
- Install Shiny package: R command `install.packages("shiny")`
- Install data tables package: R command `install.packages("shinythemes")`
- Install plotting package: R command `install.packages("ggplot2")`
- Install data tables package: R command `install.packages("maps")`
- Install data tables package: R command `install.packages("ggridges")`
- Install data tables package: R command `install.packages("ggrepel")`
- Install data tables package: R command `install.packages("plotly")`
- All workshop scripts should function locally

5.2 RStudio Cloud

- What is RStudio Cloud?
  - *We* [RStudio] *created RStudio Cloud to make it easy for professionals, hobbyists, trainers, teachers and students to do, share, teach and learn data science using R.*
- With RStudio Cloud
  - You do not need RStudio installed locally
  - Packages and data are available without installation and transfer
- Access workshop material
  - Create an Account: `https://rstudio.cloud`
  - Workshop project link: `https://rstudio.cloud/project/580472`
  - Files are in the directories `Duke-Co-lab/Shiny/Session-2-DataTables-Plots` and `Duke-Co-lab/Shiny/Session-3-ggplot`
- All workshop scripts should function on RStudio Cloud, except OS shells that are used to automate execution

6 Review previous app, `ggplot()` U.S. Domestic Flight Map

Overview: (git or RStudio Cloud directory) Session-3/Docs/FlightEvaluationShinyAppOverview-01.pdf

Discussion points:

- User interface (`ui.r`) features of interest
  - Use of `col()` for aligned placement of selection objects above plots
  - Use of `fluidRow()` for column within row formatting
  - Use of `div()` for CSS styles (width, alignment, margins)
- Server (`server.r`) features of interest
  - Use of a separate source file for functions (FlightEvaluation-Functions.r)
  - Functions for retrieving data, aggregating data, and composing plots
  - Reactive elements bound to R variables within `observe()` events, so that modification of input objects (selection lists, slider bars, etc) cause immediate update of plots
  - Observation and label filtering from change of on-screen p selectors
  - Arc size, color, and transparency set by on screen controls (they supply values to `aes()` parameters in `ggplot()` calls)
  - Faceting controlled by on-screen radio buttons
  - Facet labels are composed for use as a labeler in `facet_wrap()`
  - Descriptive labels are joined to codes in the source data using `merge()` and displayed instead of actual codes
  - Overall rendered plot size is adjusted for facet panels displayed
- `ggplot` features of interest
  - Flight map (tab 1)
    - `geom_polygon()` used to draw a map from coordinates contained in a U.S. state map data frame

    - `geom_curve()` is used to draw arcs between flight origin and destination x-y (lat-long) coordinates (with color, size, and alpha aesthetics and fixed curvature and arrowheads)
    - Custom scales are defined for arc size, color, and alpha using on-screen values (these control arc and legend appearance)
    - Airport labels are added with `geom_label_repel()` to avoid overlay
  - Density ridges (tab 2)
    - x-axis (measure of time), y-axis (categories forming ridges), and faceting variables specified on-screen
    - y and color orders are reversible
    - x-axis limits and distribution transparency are controllable
    - `geom_density_ridges()` is used to compose ridges (x-axis = time) for each level of the selected y variable, with fill color specified as an aesthetic (each level of y receives a distinct color)
    - `scale_fill_manual()` is used to define a fixed set of fill colors composed from a call to `colorRampPalette()`, requesting a range of colors from the low and high range specified on-screen, with a number of gradations equal to the number of distinct levels of y
    - Vertical lines at mean or median time values are computed by using the density distributions composed by `geom_density_ridges()`. `ggplot_build()` is used to extract the densities for all ridges and x, y coordinates are used for mean and median computation. Distribution height is then used as an endpoint and individual line segments are plotted using `geom_segment()`.
  - Internals
    - `map_data()` function from the maps package: https://cran.r-project.org/web/packages/maps/maps.pdf
    - `ggplot_build()`: https://www.rdocumentation.org/packages/ggplot2/versions/3.2.1/topics/ggplot_build
    - `ggplot` innards: https://cran.r-project.org/web/packages/gginnards/vignettes/user-guide-2.html
    - `gg` list editing: gglisteditingg:https://cran.r-project.org/web/packages/gginnards/vignettes/user-guide-2.html

7 Hello `plotly`

```
library(ggplot2)
library(plotly)

x <- sample(1:100, 100, replace=T)
y <- 25 + 5*x + rnorm(length(x), sd=25)

g <- ggplot() +

geom_point(aes(x=x, y=y, msg="hi"))
gp <- ggplotly(g)
```

8 `plotly` App: GWAS Pleiotropy

Within a single genome wide association study (GWAS), a researcher may identify single nucleotide polymorphisms (SNP)s (positions on a chromosome), such that the configuration of genotype at a SNP appears correlated with phenotypic (disease, trait) response. With two independent GWAS studies, typically with disjoint sets of phenotypes, a researcher might find phenotypes in one GWAS that appear to be associated, by SNP, with phenotypes in the other GWAS. This process is termed "pleiotropy."

Workshop file: App/Pleiotropy/Pleiotropy.r

Figure 1 is an example screen shot of tab 1 of the pleiotropy app. Clicking phenotype/SNP points in the left hand (GWAS 1) plot identifies, with contrasting color and informative labels, phenotype points in the right hand plot (GWAS 2) that are associated by SNP.

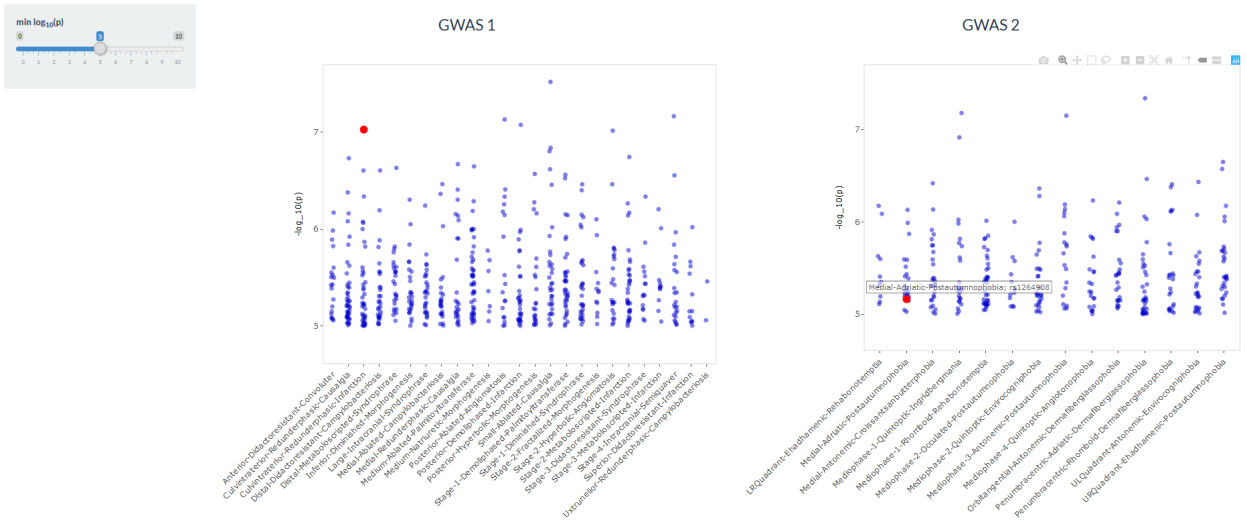

Figure 1: `plotly` pleiotropy app, tab 1, cross-GWAS phenotype SNP associations.

Important `plotly` features (numbers are lines in `Pleiotropy.r` source file):

- (69) `geom_jitter()`
- (85) Axis text angle
- (69) User defined variable in `aes()` for appearance in hover labels
- (151) Use of `plotly()` %>% to string function calls
- (380, 425) Use of `ggplotly()` to convert a `ggplot()` object to `plotly()`
- Hover label configuration (position and ID of aesthetics)
- (380, 388) Point selection and control of events
- (396) Subsetting observations for selected point
- (102) Coloring selected points using two geoms and data subsets in `t1ComposePlot2()`
- (411) Adding custom labels with `add_annotations`, since `ggplotly()` does not convert them
- (420-425) Nested rendering of plots

Figure 2 is an example screen shot of tab 2 of the pleiotropy app. After filtering within-GWAS associations by significance (p), groups of inter-GWAS phenotypes are identified by clicking points in either set (left and right vertical blue dots). Hovering over blue dots causes display of SNPs associated with a phenotype (within GWAS). Hovering over a green dot (on an edge, or line connecting the GWAS sets) causes display of SNPs that are associated with the connected phenotypes (between GWAS).

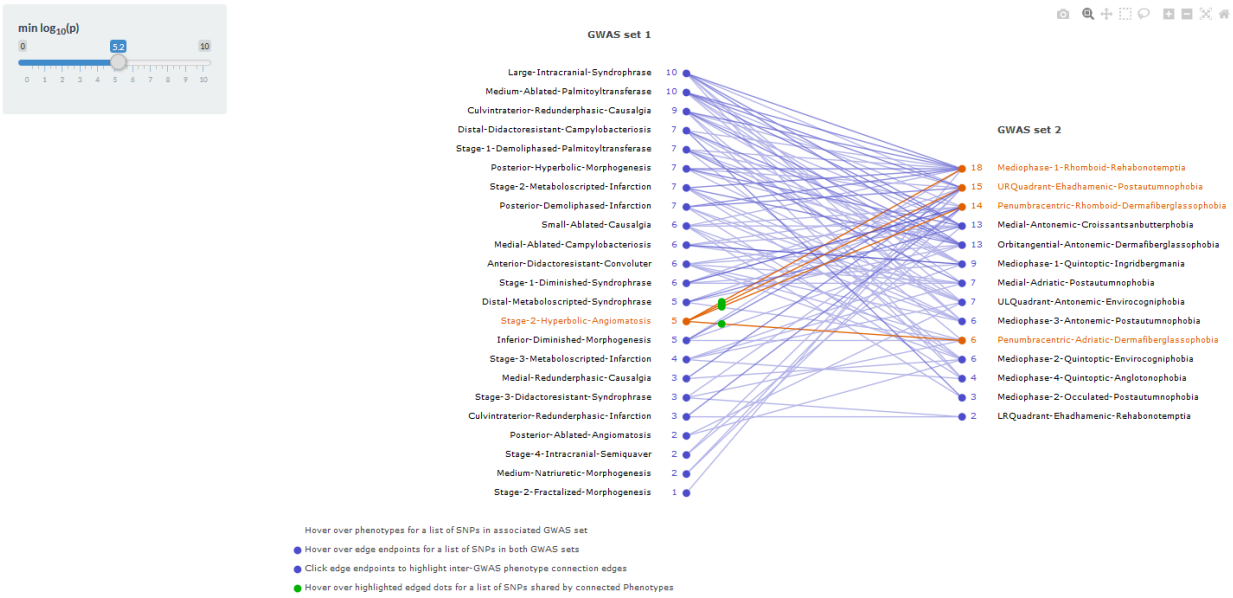

Figure 2: `plotly` pleiotropy app, tab 2, identification of phenotype association groups. Associated SNPs appear as points are hovered over.

Important `plotly` features (numbers are lines in `Pleiotropy.r` source file):

- Subsetting plotted points by p-filter
- Phenotype ordering by number of associated SNPs
- Display of number of SNPs (`geom_text()`)
- Phenotype point hover labels
- Coloring vertices (phenotype) and edges (lines) of selected point (left and right)
- Display of green intersection point
- SNP intersection of lines, hover labels

9 Debugging

It is important that you have a means of communicating with your app during execution. Unlike a typical R script, that can be executed one line at a time, with interactive review of variables, once a Shiny script launches, it executes without the console prompt. Upon termination, some global variables may be available for examination, but you may not have reliable information on when they were last updated. Error and warning messages are displayed in the console (and the terminal session when executed in a shell) and, fortunately, so are the results of `print()` and `cat()`. When executed in RStudio, Shiny offers sophisticated debugger features (more info at https://shiny.rstudio.com/articles/debugging.html). However, one of the simplest methods of communicating with your app during execution is to use `print()` (for a formatted or multi-element object, such as a data frame) or `cat(, file=stderr())` for "small" objects. The `file=stderr()` causes displayed items to appear in red. Output may also be written to an error log, depending on your OS. Considerations include

- Shiny reports line numbers in error messages relative to the related function (`ui()` or `server()`) and, although not always exact, reported lines are usually in the proximity of the one which was executed at the time of error
- `cat("your message here")` displays in RStudio console (generally, consider Shiny Server)
- `cat("your message here", file=stderr())` is treated as an error (red in console, logged by OS)
- Messages appear in RStudio console when Shiny app launched from within RStudio
- Messages appear in terminal window when Shiny app launched with the `rscript` command in shell
- There exists a "showcase" mode (`runApp(display.mode="showcase")`) that is intended to highlight each line of your script as it is executing
- The reactivity log may be helpful in debugging reactive sequencing issues (`options(shiny.reactlog=T)`, https://shiny.rstudio.com/reference/shiny/0.14/showReactLog.html It may be helpful to initially format an apps appearance with an empty `server()` function, then include executable statements once the screen objects are available and configured
- Although not strictly related to debugging, the use of `gc()` to clear defunct memory (from R's recycling) may reduce total memory in use at a given time