Co-lab Shiny Workshop

Integrating Shiny and `ggplot`

October 29, 2020

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 controls and review promotes idea generation and validation, making a well designed Shiny app a resource for exploratory research of data. In this session, we will further our use of `ggplot` "geoms" and controls for faceting (creating multi-panel plots), controlling aesthetics (defining which variables are colored, sized, etc., based on their levels), configuring axes, and adjusting overall appearance using themes. Each control is specifically designed to improve the viewer's understanding of various distributions observable in the sample data, including flights by origin and destination city; flights by weekday and month; and departure delay by carrier, weekday, and month. Because we are using `ggplot` in a Shiny context, our emphasis will be on binding on-screen controls to R script variables that are used to initialize parameters of various `ggplot` functions.

1 Overview

- Preliminaries
  - What can Shiny and `ggplot` do for you?
  - What are your expectations of this workshop?
- Examples
  - Example `ggplot` visualizations
  - Example Shiny apps
- Resources
- Anatomy of a Shiny app
- Workshop material
  - From RStudio Cloud
  - From github (execute locally)
- Review previous app, data tables and the OPM Central Personnel Data File, V3
- U.S. commercial flight app using `ggplot` and `ggridges`
  - `ui.r`
  - `server.r`
  - U.S. domestic flight map (`ggplot`)
  - Flight departure delay distribution (`ggridges`)
- Debugging

2 Examples

2.1 Visualizations

- `ggplot` gallery: https://www.r-graph-gallery.com/all-graphs.html
- `ggplot` extensions: https://mode.com/blog/r-ggplot-extension-packages

2.2 Shiny apps

- Duke Data+ project, *Big Data for Reproductive Health*, http://bd4rh.rc.duke.edu:3838
- Duke Data+ project, *Water Quality Explorer*, http://WaterQualityExplorer.rc.duke.edu:3838
- Duke Med H2P2 Genome Wide Association Study, http://h2p2.oit.duke.edu
- Duke Nicholas School, *Health Effects of Airborne Pollutants*, http://shindellgroup.rc.duke.edu
- Duke Nursing School, *Urea Cycle Disorder SNOMED/RxNorm Graph Associations*
- Shiny gallery: https://shiny.rstudio.com/gallery/
- NCSU Institute for Advanced Analytics intro: https://www.csc2.ncsu.edu/faculty/healey/msa-17/shiny/

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
- `ggplot`
  - `?ggplot2` from the R command line
  - Click `ggplot2` in the `Packages` tab of RStudio
  - https://cran.r-project.org/web/packages/ggplot2/ggplot2.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

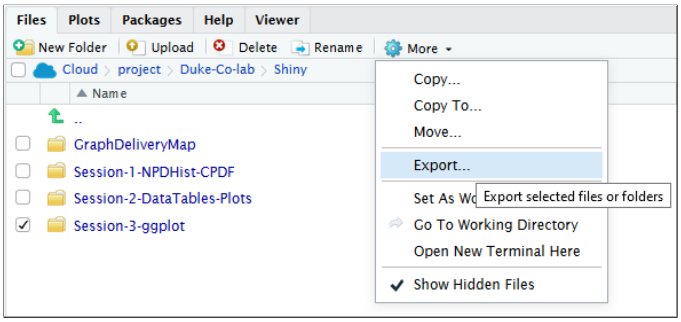
5 Workshop material

5.1 RStudio Cloud

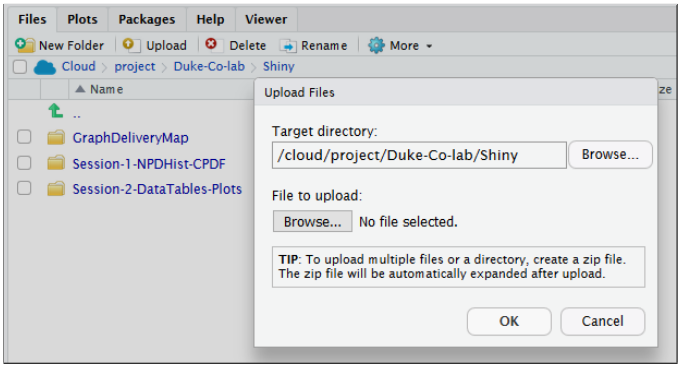
- What is RStudio Cloud?
  - *We* [RStudio] *created RStudio Cloud to make it easy for professionals, hobbyists, trainers, teachers and students to do, share, teach and learn data science using R.*

- With RStudio Cloud
  - You do not need RStudio installed locally
  - Packages, example scripts, and data sets needed for the Shiny workshop are available in the Cloud environment

- Access workshop material
  - Create an account: https://rstudio.cloud
  - Create a convenient directory structure in your project
  - Copy workshop materials
    * Go to https://rstudio.cloud/project/1768881 (Duke-Co-lab/Shiny directory)
    * Export a sub-directory



    * Save to your computer
    * Upload saved zip file to your RStudio Cloud project



  - Install packages:
    * Shiny: `install.packages("shiny")`
    * ggplot: `install.packages("ggplot2")`
    * Data Tables: `install.packages("maps")`
    * Data Tables: `install.packages("ggridges")`
    * Data Tables: `install.packages("ggrepel")`

- All workshop scripts should function in RStudio Cloud, except OS shells that are used to automate execution

5.2 Execute locally (copy workshop material from github repo)

- Copy scripts and data from https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3/Session-3-ggplot.zip
- Install packages:
  - Shiny: `install.packages("shiny")`
  - ggplot: `install.packages("ggplot2")`
  - Data Tables: `install.packages("maps")`
  - Data Tables: `install.packages("ggridges")`
  - Data Tables: `install.packages("ggrepel")`
- All workshop scripts should function locally

6 Review data tables app, V3

Material:

- RStudio Cloud
  - https://rstudio.cloud/project/1768881 (Duke-Co-lab/Shiny/Session-2-DataTables-Plots/CourseMaterial/Ou
  - https://rstudio.cloud/project/1768881 (Duke-Co-lab/Shiny/Session-2-DataTables-Plots/App/V3)
- git
  - https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-2/CourseOutline
  - https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-2/App/V3

Discussion points: *Section 8.3 of Co-lab-Session-2-DataTables-Plots.pdf*

7 U.S. commercial flight app using `ggplot` and `ggridges`

The apps presented in section 2 and in previous sessions of this workshop demonstrate the ability of `ggplot` to effectively represent patterns in data using a wide variety of graphs. Combining the rich visual features of `ggplot` with the interactive features of Shiny gives a platform for developing versatile applications for exploring data in a self-directed manner, opening new avenues of understanding of the system being studied. To demonstrate the power of `ggplot` with Shiny, we will review an app that explores U.S. domestic flight data. Data consist of a fifty percent random sample of flights that originated and terminated in the continental U.S. in January, April, July, and October 2018.[1] The `ggridges` extension package for `ggplot` will be used to produce density plots of flight delay time for visual comparative analysis. Typical questions that the app helps to answer include:

- Which weekdays are heavy or light air travel days?
- Are there seasonal differences in the volume of flights?
- Which carriers service which regions, states, and cities of interest?
- Are there differences in the distributions of departure delay by weekday, season, or carrier?

Material:

- RStudio Cloud
  - Outline: https://rstudio.cloud/project/1768881 (Duke-Co-lab/Shiny/Session-3-ggplot/CourseMaterial)
  - Scripts: https://rstudio.cloud/project/1768881 (Duke-Co-lab/Shiny/Session-3-ggplot/App)
- git
  - Project: https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3
  - Outline: https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3/Docs
  - Scripts: https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3/App
  - Data: https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3/SampleData
  - Outline, scripts, and data: https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-3/Session-3-ggplot.zip

Shiny scripts for this app consist of three files: `ui.r`, `server.r`, and `FlightEvaluation-Functions.r`.

`ui.r` features of interest include (approximate line numbers appear in parentheses):

- (52) `shinyUI()` is defined. This is one of the three required elements of a Shiny app.
- (56) A browser tab title is specified
- (59-63) The notification window position and appearance is modified
- (70, 71, 195) Use of `div()` and a CSS style specification to provide a slight left margin to the fluid row containing main contents
- (72, 75, 131, 184) Use of `TabsetPanel()` and `TabPanel()` for separation of flight map app, delay density app, and file location controls
- (78, 99, 133, 150) Use of two `fluidRow()` elements for input controls
- (79-96, 100-117) Use of `column()` elements for horizontal placement of input controls (the number of columns in each row is limited to twelve; to present an aligned appearance, identical widths are specified for positionally corresponding columns in each row)
- (82, 87, 92, 103, 109, 114) Use of `div()` and style elements with `vertical-align:top` and `margin-top` specifications to align controls vertically

[1] As reported by the U.S. Bureau of Transportation Statistics on their Carrier On-Time Performance Data download site https://www.transtats.bts.gov/DL_SelectFields.asp.

- (134-138, 151-170) Use of `margin-left` style elements in place of columns to horizontally position input controls (this overcomes the twelve column limitation and provides much greater control over positioning of elements)
- Input controls include `sliderInput()`, `selectInput()`, `radioButtons()`, and `checkBoxInput()`
  - They are identified by the value in their `inputId` parameter (the first, if parameter names are not used)
  - Labels appear in the `label` (second, if not named) parameter
  - Note the specifications for `choices`, `choiceValues`, `choiceNames`, `selected` (default values applied when transitioning from NULL during initialization - causing event triggers to fire), `multiple`, and `inline`
  - *Discuss using a list for `choices` in a `selectInput()` specification*
- (123-127, 176-180) Declaration of `ggplot` `plotOutput()` elements to display rendered plots (note the use of a `fluidRow()` for placement on a separate vertical section and use of `HTML("<center>")` for horizontal centering within the section)

`server.r` features of interest include (approximate line numbers appear in parentheses):

- `server.r` has no `ggplot` or `ggridges` function calls. In this implementation, the purpose of `server.r` is to define global variables, read data to be treated, and to define functions for intercepting control when user input triggers action.
- (19) Functions to implement `ggplot` and `ggridges` features are imported from `FlightEvaluation-Functions.r`. Use of a separate script to implement application functions, keeps Shiny scripts simple, basically providing the shell needed to coordinate user input and presentation of results.
- (23-25) Global vectors are defined for month and weekday labels. Note that, although these are declared outside (seemingly above in environment-space hierarchy) of the `shinyServer()` function, they are not recognized within `shinyServer()` unless declared as globals.
- (29) `shinyServer()`, another of the three required Shiny app elements, is defined. `input` and `output` parameter values are used to reference input controls (`input$pthreshFlight`, line 79), and output presentation objects (`output$plotUSFlights`, line 125). `session` is used, for instance, to update current values and appearance of input controls during program operation (`updateSelectInput()`).
- (35, 83) `graphFlightMap()` and `graphFlightDelayDensity()` functions are defined to call plot composition functions (from `FlightEvaluation-Functions.r`) and render the resulting graphs, when user input prompts plot regeneration
- (42-62, 88-108) Displayed plot height, width, and facet labels are adjusted based on the specified facet variable
- (39, 72, 85, 118) Use of `showNotification()` to display a warning message when no observations correspond to specified filtering values
- (130, 156, 167, 180) `observeEvent()` functions are defined that monitors the state of input controls and triggers action when they are changed. Data are read and aggregated and plot configuration functions are called with parameter values corresponding to user input values.
- (149) Observe events are executed when the reactive variables being monitored undergo a change in value. During program initialization, reactive variables (including action buttons) transition from NULL to their beginning values. The `ignoreInit` parameter of `observeEvent()` controls whether an event function is executed after the initial reactive variable transition from NULL, with `ignoreInit=TRUE` preventing execution, which is what we desire for the `retrieveData` button. The default value of `ignoreInit` is `FALSE`.

- (156-160, 167-173, 180-188) Data aggregation and plot generation functions (`aggfdat()`, `graphFlightMap()`, and `graphFlightDelayDensity()`) are called strictly from within `obserEvent()` functions. `ignoreInit=F` instructs to execute these monitor functions during program initialization, without which the viewer would have to alter an on-screen control before observing a plot.
- (158) Aggregated data are created in the global environment to be available for plotting functions
- (202) The `readData()` function is called to read flight data from source records into the global environment
- 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

U.S. domestic flight map (`ggplot`) features of interest include (approximate line numbers in `FlightEvaluation-Functions.r` appear in parentheses):

- (11-60, `readData()` function) Observations are read from source files to the global environment
- (27, 38) A progress meter advances as each file is read
- (41-56) Commercial airline and airport identying data are read into the global environment
- (67-120, `aggfdat()` function) Flight frequencies are aggregated by origin and destination within a third aggregation variable (for faceting). Parameter values control whether flights are limited to those with delayed departure due tto the carrier and whether to include cancellations.
- (129-131, `composePlotMap()` function) Parameter values specify thresholds for fligh inclusion, faceting, and plot appearance. They correspond to on-screen reactive values and are passed by corresponding `observeEvent()` function execution triggered by a change in value.
- (141) Flights are filtered to those having a proportion of total above a specified threshold (pthresh-Flight)
- (144-166) Airport labels are composed for routes having a proportion of flights below a specified threshold (`pthreshAirportLabel`)
- (175) A map of the continental U.S. is generated, `using geom_polygon()`, each polygon tracing the outline of a single state
- (179) An arc is drawn, using `geom_curve()`, from each flight origin (`x, xend`) to destination (`y, yend`). Color, size (line weight), and transparency are a function of the proportion of all routes that a given route represents (specification of appearance variables within the aesthetic function causes treatment of individual levels of p)
- (183) Specific color, line weight, and transparency values are supplied in corresponding function parameters

- (190-191) Airport labels are appended to the plot object
- (196) Individual plots are constructed for each level of the specified faceting variable
- (200-222) A theme is appended
- (227) the `ggplot` object (essentially, a list containing instructions for rendering the plot) is returned to the calling function to be rendered

Flight departure delay distribution (`ggridges`) features of interest include (approximate line numbers in `FlightEvaluation-Functions.r` appear in parentheses):

- (247-254) Observations are filtered based on specified delay style and cancellation styles
- (257-269) Data for the specified independent (`x`), dependent (`y`), and faceting variables are selected
- (272-278) Specified dependent data factor order is imposed
- (281-290) Specified fill colors are applied
- (293-298) Axis labels are composed based on specified independent and dependent variables
- (301-337) The plot object is constructed
- (339-382) Vertical lines are included at mean, median, or zero, as specified
- (387) the `ggplot` object is returned

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

8 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