R Shiny

Shiny is a RStudio package that provides a web application framework for R. It can be used to easily build interactive web applications with R. It is meant for R users with little to no experience with web page development, and does not require HTML/CSS/-JavaScript knowledge. With Shiny, we can create an interactive web page that can communicate with R and display R objects such as plots, tables, and calculations. Pretty much anything we can do in R can be done through a Shiny app, and results can be displayed nicely for users. The primary benefit of a Shiny app is the ability to share an R analysis with a non-statistician or someone who has little to no experience with coding.

The purpose of this section is to walk through the steps of building a Shiny app using one of the datasets we've already explored in this course. There are many additional features of Shiny that will not be discussed in this section, but I encourage you to explore for yourself. You can always do a Google search for specific features you want to add to a Shiny app, but there are a number of good tutorials for an overall introduction to the interface. These notes are primarily based on Dean Attali's Shiny tutorial[1], with a little bit of information from Hadley Wickham's "Mastering Shiny" book[2]. Other good resources include the official Shiny tutorial from RStudio[3], a tutorial and examples from Zev Ross[4], and the RStudio cheat sheet[5]. There are many examples of Shiny apps available from Shiny users at http://ShowMeShiny.com. All programs and data sets used for these notes are available from the course website. New files that we have not used before include Cooler.zip, Cooler_conditional.zip, Cooler_tabs.zip,

[1] https://deanattali.com/blog/building-shiny-apps-tutorial/
[2] https://mastering-shiny.org/
[3] https://shiny.rstudio.com/tutorial/
[4] http://zevross.com/blog/2016/04/19/r-powered-web-applications-with-shiny-a-tutorial-and-cheat-sheet-with-40-example-apps/
[5] https://rstudio.com/resources/cheatsheets/

dataset_app.zip, empty_app.zip, emtpy_app_separate.zip, uiOutput_example.zip, and ShinyExamples.R.

Shiny app basics

First, we need to install the `shiny` package in RStudio. Then we'll run one of the demo apps to make sure Shiny was successfully installed.

```
install.packages("shiny")
library(shiny)
runExample("01_hello")
```

Close the internet browser and press ESC to close the app. Now we're ready to use Shiny!

A Shiny app is composed of two parts: a web page that displays the app for the user, and a computer that runs R and powers the app. The computer running the app can be a personal computer (where you run the app in RStudio) or a server somewhere else. For now, we'll run the apps in RStudio from a personal computer, and later, we'll talk about how to share our app to a server. In Shiny, these two parts are called the UI (user interface) and server. The UI is a web document with HTML written by Shiny's functions. It creates the layout of the app and tells Shiny where to put things inside the app. The server tells the app what to show when the user interacts with the app. In my example app, the UI is responsible for creating the user inputs and telling Shiny where to place controls and outputs such as plots and data tables, while the server uses the inputs to actually create the plots or data tables.

Our first Shiny app

All Shiny apps must incorporate the UI and server, and follow a similar template. This template can be found in the app.R file

inside the empty_app folder.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

Notes:

- This template is actually a Shiny app by itself. It initializes an empty UI and an empty server, and runs an app using these parts.
- There should not be any code after the `shinyApp(ui = ui, server = server)` line. This needs to be the last line in the app.R file.
- The file must be saved as "app.R". This allows RStudio to recognize it as a Shiny app. Once this happens, the *Run* botton will change into a *Run App* button. (Note: We will run the entire app and will not run individual lines of code in the R console.) If you don't see the *Run App* button, you either don't have Shiny installed or didn't name the files properly.
- We can also use a keyboard shortcut to run the app: `Cmd/Ctrl` + `Shift` + `Enter`.
- It is best practice to place each app in its own folder without other R scripts or files. Only files that are used by the app should be included in this folder. The directory name is the name of the app.

When we run the empty app, not much will happen because we have not added anything to the UI or server. However, a stop sign will appear in the top right corner of the console and the console will display text like this:

```
> runApp('C:/Users/bhitt/Desktop/STAT 850/Shiny/empty_app')

Listening on http://127.0.0.1:7799
```

The web address in the console will match that of the web browser that opens. 127.0.0.1 is a standard address that means "this computer" and 7799 is a randomly assigned port number. Another copy of this app can be opened by entering this URL into any compatible web browser on the same computer. We will no longer be able to run commands in the console because R is busy running the app and waiting for user interaction (once we add some Shiny functions to the UI and server). To stop the app, click the stop sign button or press ESC. Notice that when we click *Run App*, RStudio runs the function `shiny::runApp()` in the console. We can run that command instead of clicking the button, but **must not** place the `runApp()` function inside the app code.

Another simple app is shown below and is available in the dataset_app folder.

```
library(shiny)
ui <- fluidPage(
  selectInput("dataset", label = "Dataset",
              choices = ls("package:datasets")),
  verbatimTextOutput("summary"),
  tableOutput("table")
)
server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- get(input$dataset, "package:datasets")
    summary(dataset)
  })
  output$table <- renderTable({
    dataset <- get(input$dataset, "package:datasets")
    dataset
  })
}
shinyApp(ui = ui, server = server)
```

This app allows the user to choose a data set from those built in to R and displays a summary of the data set chosen. We'll learn more about the UI and server functions used to create this app later.

Notice that running an app in Shiny is a little different than running other code in R. When writing code in R, one can usually run small portions of the code to debug and make adjustments. This cannot be done with Shiny. The entire app must be run to see if there are errors or changes that need to be made. If there is an error, the app will most likely not run and the errors are not always helpful. This may take some getting used to. The basic process of developing a Shiny app involves writing some code, starting the app, experimenting with the app, writing more code, and so on. I find it is best to add one element at a time and then run the app to check that everything works and appears as you expect it to, at least until you get the hang of Shiny.

Using RStudio to create an app

We can create a Shiny app by creating a project in RStudio.

1. Open RStudio.
2. In the upper right corner of RStudio, you will see Project: (None). Click this drop down arrow and choose New Project.
3. Under Create Project, choose New Directory.
4. Under Project Type, choose Shiny Web Application.
5. Under Create Shiny Web Application, provide a Directory name (we'll use "newapp") and tell R where you would like the app to be saved on your computer. The folder can have any name. It is best to choose a folder that does not include other R scripts or files that will not be used in the app. We will typically not use packrat with these type of projects.
6. Click Create Project to finish the process.
7. The app.R file, located in the newapp directory on your computer, will be created and opened in RStudio. This file contains the template for a Shiny app that displays a histogram

of the Old Faithful Geyser Data and allows the user to change the number of bins for the histogram.

The method above only allows us to create an app with a single file (app.R). Another method in RStudio (below) allows us choose between a single file or multiple file app.

1. In the upper left corner of RStudio, click File > New File > Shiny Web App...
2. Specify an application name, choose whether the app will consist of a single file or multiple files, and specify the directory you want the app created within (if desired).
3. Click Create to finish the process.
4. The ui.R and server.R files, located in the specified directory on your computer, will be created and opened in RStudio. This template app is the same as that created in the first process above.

There is also a shortcut in RStudio that can be used to create an app template. Open a new script and save it as app.R. Then type "shinyapp" and hit Shift + Tab on the keyboard to convert the file to an empty app template.

I personally don't use RStudio to create a template app, because it seems easier to simply type the few lines of code.

Using separate UI and server files

Including code for the UI and server in a single file is easy and makes sense when we have a simple app. But Shiny apps can easily get complex, and a single file can sometimes be difficult to navigate. Another way to outline a Shiny app is by creating separate UI and server files, ui.R and server.R, that each contain their own code. All the code that is assigned to the ui variable is

placed in ui.R and the function assigned to the server variable is placed in server.R, as shown below.

ui.R

```
fluidPage(

)
```

server.R

```
function(input, output, session){

}
```

Notes:

- The code in these files is no longer assigned to a variable.
- The "ui.R" and "server.R" files must be saved in the same directory. This should be a new, isolated folder where there are no other Shiny apps, R scripts, or files other than those that will be included in the app.
- If using this method, we do not include a call to `shinyApp()`.
- We only need to open one of these files for RStudio to recognize the app and change the *Run* button to the *Run App* button.

I personally use the multiple file format for apps, because it seems much easier to keep the app appearance (ui.R) and logic (server.R) separate.

Building the Cooler app

Go ahead and create a single file app inside a folder called "Cooler". We'll be using the Cooler data set for our app. We will need to download the CoolerReduced.csv file and place this file in the same folder as the Shiny app.

Just after the line loading the shiny package, add a line in the app to load the data into a variable called "cooler". Make sure the file path and name are correct, or the app won't run. To make

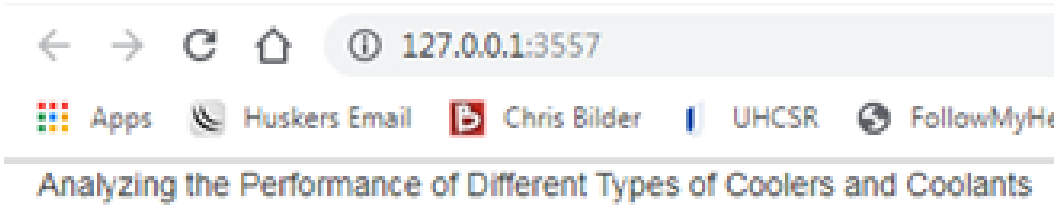

Figure 1: Title Only

sure that the app successfully reads the data, we can add a print statement after reading the data. Our app should look like this:

```
library(shiny)
cooler <- read.csv("CoolerReduced.csv", stringsAsFactors=FALSE)
print(head(cooler))
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

This will print the first six observations of the data set in the console, but will not do anything in the Shiny app itself. Once we confirm that the data is loaded correctly, we can remove this line.

Building the UI

Formatting text

The first step in writing a Shiny app is usually adding the visual elements. We'll do this by adding elements to the UI. We can render text by adding strings inside `fluidPage()`. We'll replace the line in the app that assigns an empty `fluidPage()` in the UI with the line below, and run the app (see Figure 1).

```
fluidPage("Analyzing the Performance of Different Types of
  Coolers", "and Coolants")
```

Our app will be built by passing comma-separated arguments into the `fluidPage()` function. By passing regular text, the web page will just render the strings as unformatted text. Adding additional strings to `fluidPage()` will render the text in a contiguous block.

Shiny uses functions that are wrappers around HTML tags to create the UI. Many of these can be used to format text. We can use `h1()` for a top-level header (`<h1>` in HTML), `h2()` for a secondary header, and so on through `h6()`, where the higher numbers are associated with smaller headers. We can also use `strong()` for bold text and `em()` for italicized text. There are other HTML wrappers such as `p()` for a paragraph, `br()` for a line break, `img()` for an image, `a()` for a hyperlink, and more. Additional HTML tags can be utilized by using the tags object (e.g., `tags$h1()` or `tags$br()`), which you can learn more about by reading the help file on tags[6]. The most used tags are available without the `tags$` notation.

We'll run the app with the following code in our UI (see Figure 2).

```
fluidPage(h2("Cooler Data"),
          br(),
          h4("Analyzing the Performance of Different Types of",
             "Coolers and Coolants to Improve Cold Chain
               Transportation"),
          br(),
          p("The purpose of this", strong("Shiny"),
            "app is to perform and display the analysis of the
              cooler",
            "data obtained from Lowe et al."))
```

Note that the `strong()` function needs to be placed inside the `p()` function in order for "Shiny" to appear in the same line as the rest of the surrounding sentence. If we instead used `strong()`

[6] http://shiny.rstudio.com/articles/tag-glossary.html

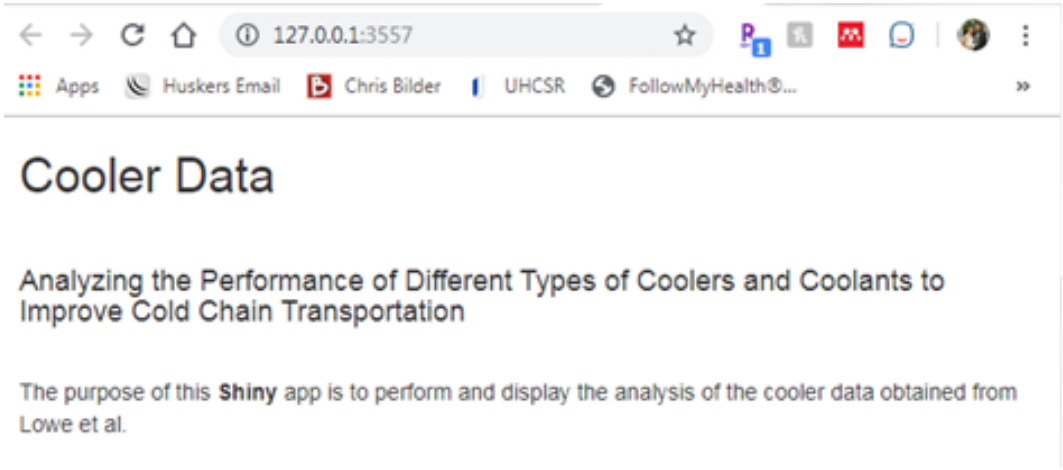

Figure 2: Formatted Text

outside of `p()`, "Shiny" would appear on a separate line. On your own, experiment with different text formatting.

Using a layout

Notice that by default the elements in the UI appear one right after the other, on separate lines. There are many other layouts available for Shiny apps[7]. An "official" title can be specified at the top of the page in large text using `titlePanel()`. We can specify a single string that will act as both the app title and the window title (the name of the browser tab), or we can provide two arguments, one for each. A sidebar layout uses `sidebarPanel()` and `mainPanel()` to create a sidebar for inputs and a larger main space for outputs. A grid layout creates rows through `fluidRow()` and columns through `column()`. The column widths can be specified and should add up to 12 within each row container. Content can be organized into separate tabs using `tabsetPanel()`. Tabs can be located above (default), below, left, or right of the tab content. Another way to organize an app is to use `navlistPanel()` to list navigation controls in a sidebar rather than using tabs. To create distinct pages in the app, one can use `navbarPage()`. The navigation bar appears at the

[7] http://shiny.rstudio.com/articles/layout-guide.html

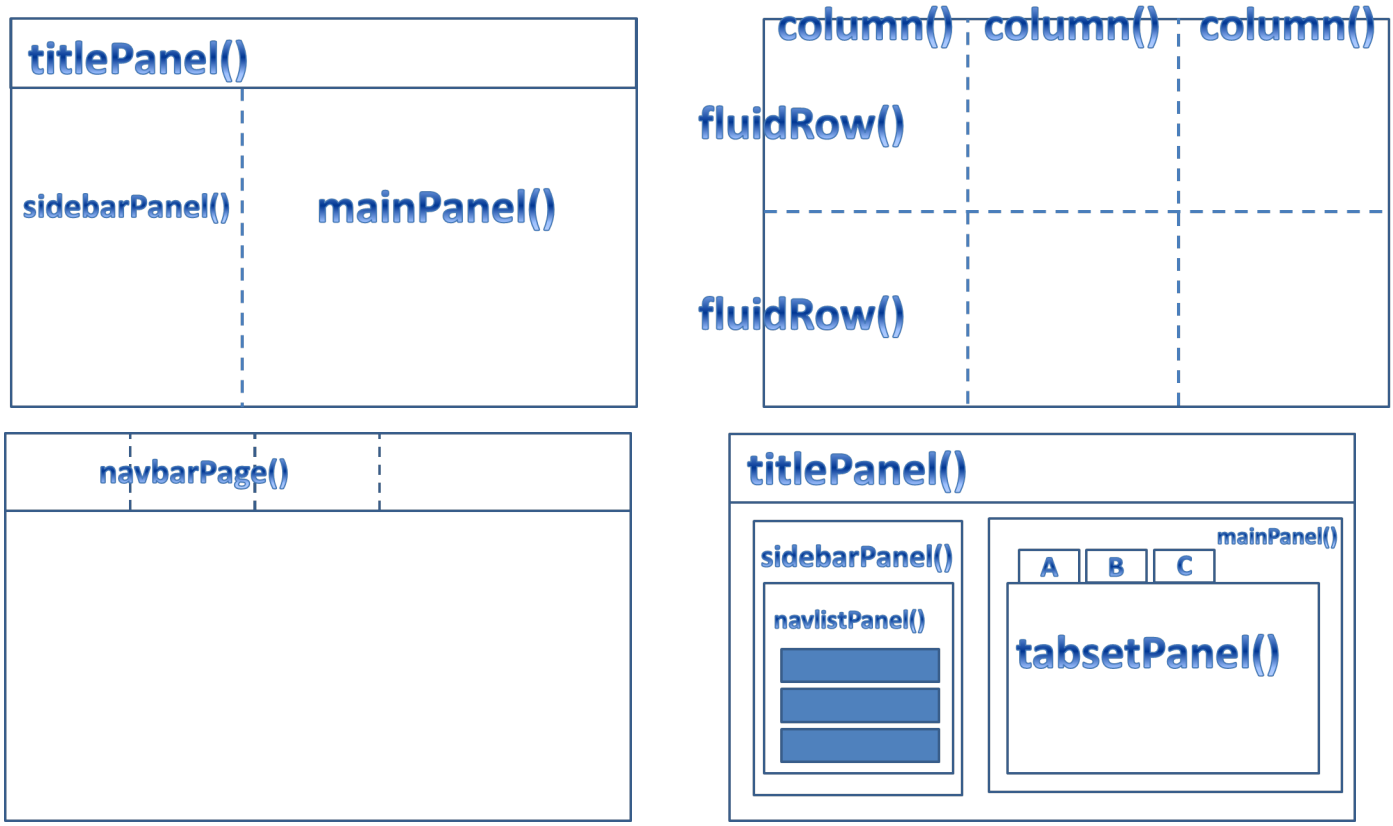

Figure 3: Examples of Layouts

top of the page and each page can have its own sidebar, tabset, or other layout. A second level of navigation can be added to a `navbarPage()` using `navbarMenu()`. These layouts can also be layered to create a unique layout. For example, we could create a sidebar layout, with a navigation list in the side panel and a tabset panel in the main panel. Figure 3 shows examples of a sidebar layout (upper left), a grid layout (upper right), a page with a navigation bar (lower left), and a custom layout with sidebar, navigation list, and tabsets (lower right).

For our Cooler Data app, we'll add an app title and a different window title. We'll keep the `h4()` header we used before and add a sidebar layout below it with some simple text in each part. Our `fluidPage()` function should now look like this (see Figure 4):

```
fluidPage(titlePanel("My STAT850 Shiny app", "Cooler Data"),
          h4("Analyzing the Performance of Different Types of",
             "Coolers and Coolants to Improve Cold Chain
```

```
            Transportation"),
        br(),
        p("The purpose of this", strong("Shiny"),
          "app is to perform and display the analysis of the
            cooler",
          "data obtained from Lowe et al."),
        sidebarLayout(
          sidebarPanel(
            "Our inputs will go here"
          ),
          mainPanel(
            "The results will go here"
          )
        )
)
```

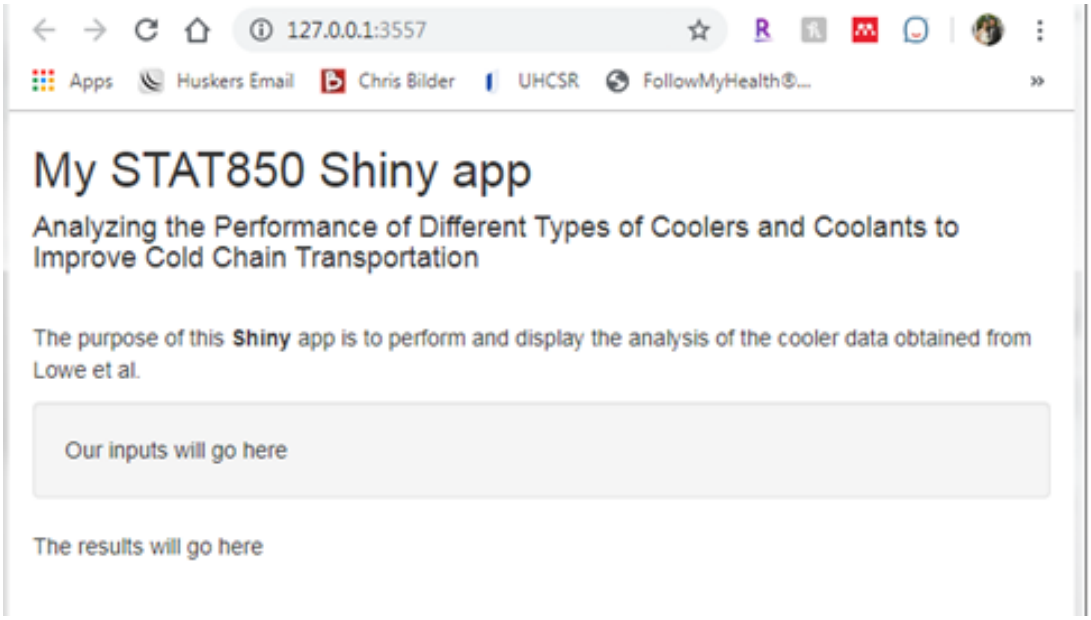

Figure 4: Cooler app with side bar

Remember that all components in the UI need to be separated by commas, and all components within the `sidebarPanel()` or within the `mainPanel()` need to be separated by commas. We can use plain text strings or we can use formatted text (using different levels of headers or paragraphs) in each part of the app.

To once again illustrate that the Shiny functions are simply HTML wrappers, we can use `print(ui)` in our app (outside the

UI and server) to display the background HTML created by our Shiny functions.

```
> runApp('C:/Users/bhitt/Desktop/STAT 850/Shiny/Cooler')
<div class="container-fluid">
 <h2>My STAT850 Shiny app</h2>
 <h4>
   Analyzing the Performance of Different Types of
   Coolers and Coolants to Improve Cold Chain Transportation
 </h4>
 <br/>
 <p>
   The purpose of this
   <strong>Shiny</strong>
   app is to perform and display the analysis of the cooler
   data obtained from Lowe et al.
 </p>
 <div class="row">
   <div class="col-sm-4">
     <form class="well">Our inputs will go here</form>
   </div>
   <div class="col-sm-8">The results will go here</div>
 </div>
</div>
```

Adding inputs

To interact with the user, an app needs inputs and outputs. Shiny provides input functions that support various kinds of interactions between the user and the app. Inputs are added using `*Input()` functions. We can use `textInput()` or `textAreaInput()` to allow the user to enter text, or `numericInput()` to allow the user to select a number. Users can select a date with `dateInput()` or `dateRangeInput()`. Users can select an option from a dropdown menu using `selectInput()` or from a list using `radioButtons()`. An app can provide check box inputs using either `checkboxInput()` or `checkboxGroupInput()`. Another option is a `sliderInput()`

which allows users to use either a one-sided or two-sided numeric slider. Other input options include `actionButton()`, `colourpicker::colourInput()`, `fileInput()`, and `passwordInput`

All input functions use an `inputId`, the name of the input that Shiny uses to retrieve its value, and `label`, the text displayed as the label for the input widget. Each input must have a unique `inputId`. Input functions also have other arguments specific to the input type. For example, you'll need to specify a minimum value, maximum value, and default value for a numeric slider. We can optionally specify a prefix for the slider input too. To learn more about the required arguments for an input, we can simply use `?sliderInput()` or similar.

We'll add a dropdown menu that will allow the user to choose the type of cooler to analyze. We'll need to allow for three types of coolers: injection molded, rotomolded, and PS foam. We'll add the following code inside our `sidebarPanel()` function (see Figure 5):

```
selectInput(inputId="coolertype", label="Cooler Type",
            choices=c("Injection_molded", "Rotomolded",
              "PS_foam"),
            selected=NULL)
```

Note that the `inputId` must be a string. The `selected` option allows us to make a default choice for the input. This code does not provide a default choice for the drop down menu. Use `?selectInput` to find out about other options.

Add placeholders for output

Now that we've created an input, we want to add components to the UI that will display the outputs. Outputs can be any object created by R that we want to display, such as text, a table, or a plot. Since we're still building the UI, we're only adding placeholders for the outputs. These will determine what the output ID is and where it will be located, but no output will

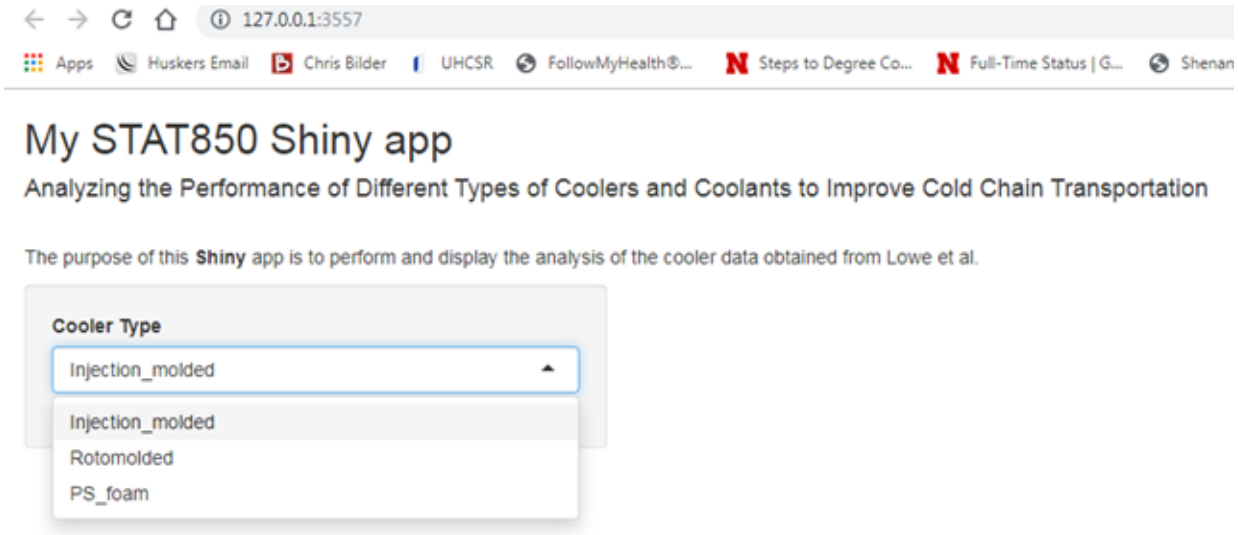

Figure 5: Cooler app with `selectInput()`

be displayed yet. The outputs will be constructed in the server code later.

Shiny provides many output functions, one for each type of output. Most of these mirror the input functions and are added using `*Output()` functions. The `*Output()` functions are used in the UI to hold a place for the output and the matching `render*()` function is used in the server to actually create the output. Similar to the input functions, all the output functions have an `outputId` that identifies the output. This ID must be unique for each output or the app will not behave properly. We can render data tables with `dataTableOutput()` + `DT::renderDataTable()`. Tables can also be added with `renderTable()` + `tableOutput()`. Images can be rendered with `renderImage()` + `imageOutput()` and plots can be rendered with `renderPlot()` + `plotOutput()`. Code output (like what we would see in the R console) can be displayed using `renderPrint()` + `verbatimTextOutput()` and text output can be added using `renderText()` + `textOutput()`. UI output (which we'll mention later) can be contributed using `renderUI()`+ `uiOutput()` or `htmlOutput()`. For more infor-

mation on each type of output, use `?uiOutput` or similar.

In the main panel, we'll display the results. We'll have some text that states the estimated linear regression model, R^2 value, and adjusted R^2 value. Since we want text, the function we use is `textOutput()`. We'll use one text output for the estimated model and one for the r-squared values. Add the following code into the `mainPanel()` (replacing the existing text):

```
textOutput("estmodel"),
br(),
textOutput("rsquared")
```

Below the text output, we will have a plot of the estimated regression model with prediction interval bands. To produce a plot, we'll use the `plotOutput()` function. We'll add `plotOutput("fitplot")` into the `mainPanel()`, just below the text output. We'll also use `br()` to add a line break between the two outputs, so that they don't appear too close to each other. Below the plot, we will have a table of the predicted values and intervals for the number of water bottles. A simple way to create a UI element that will hold a table output is `tableOutput("predictions")`. Add this output to the `mainPanel()` below the plot output. The `mainPanel()` should look like this:

```
mainPanel(
  textOutput("estmodel"),
  br(),
  textOutput("rsquared"),
  br(),
  plotOutput("fitplot"),
  br(),
  tableOutput("predictions")
)
```

Nothing will change when we run the app, but we now have placeholders for all of the outputs. To illustrate the fact that we are still just constructing HTML and not creating the actual results, run

the `textOutput()`, `plotOutput()`, and `tableOutput()` functions in the console to see that it simply creates HTML.

Implementing server logic

So far we have only added code to the `ui` variable. Now we need to write code for the `server` function, which will be responsible for taking into account the inputs and creating outputs to display in the app. That is, the `server` function will tell the server how to render the outputs with R. The `server` function has two primary arguments: `input` and `output`. These arguments must be defined! The `input` argument is a list that we will read values *from* and contains the values of all the different inputs we created in the UI. The `output` argument is a list that we will write values *to* and is where we will save our output objects (such as tables and plots) to display in the app. We will refer to the necessary inputs with `input$<inputId>` and refer to outputs with `output$<outputId>`.

The third (and optional) argument for the server function is `session`. The `session` argument needs to be defined when we want to use functions that need to access the session. For example, `update*Input()` functions can update input values programmatically and would need to access the session to update the inputs. We would also need to access the session to add popover text, text bubbles that pop up to provide more information for the user, either for inputs or for displayed results. You can learn more about this with `?shiny::session`.

Building an output

We created four output placeholders: *estmodel* (text output), *rsquared* (text output), *fitplot* (a plot), and *predictions* (a table). We need to write code in R that will tell Shiny what kind of text, plot, or table to display. To build an output in Shiny, we

will follow three rules:

1. Save the output object to the `output` list (this is the `output` argument in the `server` function).
2. Build the output object using a `render*()` function, where * is the type of output. Text output will be created using `renderText()`, a plot will be created using `renderPlot()`, and a table will be created using `renderTable()`.
3. Access input values using the `input` list (this is the `input` argument in the `server` function). This is only required if we want the output to depend on some input.

Let's first see how to build a very basic text output using only the first two rules. We'll create some text and send it to the `estmodel` output.

```
output$estmodel <- renderText({
  HTML(paste0("The estimated linear regression model for ",
              "coolers is estimated time = intercept +
                beta*water_bottles."))
})
```

This code shows the first two rules: we're creating text inside the `renderText()` function and assigning it to `estmodel` in the `output` list. Now we see why every output created in the UI must have a unique ID. In order to attach an R object to an output with ID x, we assign the R object to `output$x`.

Since `estmodel` was defined as a `textOutput`, we must use the `renderText` function, and we must create a text string inside the `renderText` function. If we add the code above inside the `server` function, we will see the text output in the main panel of the app (see Figure 6).

Making an output react to an input

Now we'll make the text output a little more sophisticated. Instead of always displaying the same text output, let's use the

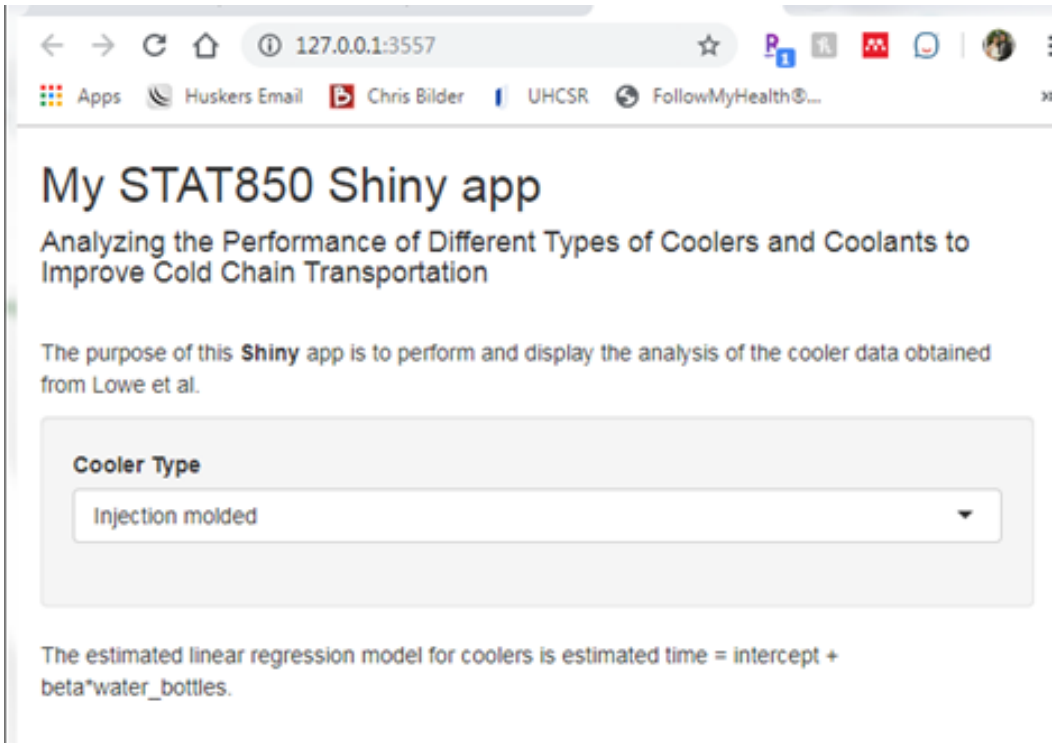

Figure 6: Simple text output

cooler type selected by the user. It still doesn't provide a whole lot of useful information, but this is an example that will help us learn how to make an output depend on an input.

```
output$estmodel <- renderText({
  HTML(paste0("The estimated linear regression model for ",
              input$coolertype,
              " coolers is estimated time = intercept +
                beta*water_bottles."))
})
```

Replace the previous code in the `server` function with the code above and run the app. Whenever we choose a new cooler type in the drop down menu, the text will update with the appropriate cooler type. Notice that the only thing different in the code is the addition of `input$coolertype` in the text output (see Figure 7).

All of the inputs defined in the UI are saved in the variable `input` and `input$coolertype` returns a string containing the name of the cooler type specified by the user. Whenever the user selects a different cooler type from the drop down menu, the value

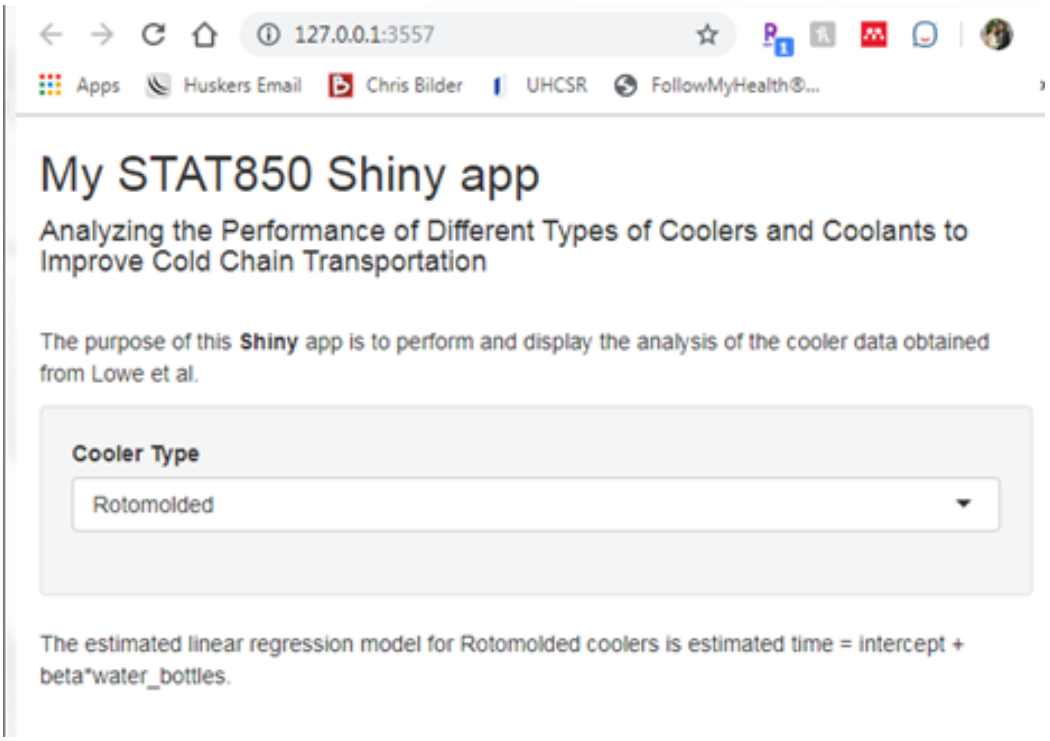

Figure 7: Reactive text output

of `input$coolertype` is updated and whatever code relies on it gets re-evaluated. This is a concept known as **reactivity**, which we will discuss next.

Creating and accessing reactive variables

We don't just want our outputs to refer to an input. Instead, we want to estimate the linear regression model and then refer to those results in our outputs. To accomplish this, we need to take advantage of Shiny's reactivity. The reactive programming in Shiny is what allows the outputs to "react" to changes in the inputs. At a very basic level, this means that when the value of an input changes, anything that relies on that input gets re-evaluated. This is very different from what we are used to in R.

Only reactive variables work this way and all Shiny inputs are automatically reactive variables. This is why we can use `input$x` in render functions and be assured that any output that depends on x will use the updated value of x whenever x is altered. By

accessing the value of a reactive variable (such as an input), the current code block becomes "dependent" on that variable. The text output we created above accesses `input$coolertype`. This means that this code block depends on this input variable, so whenever it is updated, the code gets re-executed with the new input value and `output$estmodel` is updated.

It is important to note that reactive variables can only be accessed inside reactive contexts. Any `render*` function is a reactive context, so we can always access inputs or other reactive variables inside render functions. But if we try to use the cooler type input value in the server function, outside a reactive context, the app won't work properly. Add `print(input$coolertype)` inside the server function outside of a render function. You should get the following error:

```
Error in .getReactiveEnvironment()$currentContext() :
  Operation not allowed without an active reactive context. (You
  tried to do something that can only be done from inside a
  reactive expression or observer.)
```

Shiny tells us that we are trying to use a reactive variable outside a reactive context. To fix this, we can use the `observe({})` or `reactive({})` functions. The `observe({})` function allows us to access the input variable. If we replace `print(input$coolertype)` with `observe({ print(input$coolertype) })`, the app should run properly. Notice that the `observe({})` statement "depends" on `input$coolertype`, so this code will be re-evaluated and the new value will be printed whenever we change the cooler type in the drop down menu. Because we can't run the code in our app line by line, this can be a useful debugging technique to figure out what value is held by a reactive variable.

We can also create our own reactive variables using the `reactive({})` function. It is similar to `observe({})` in that it is a reactive context, but `reactive({})` returns a value. This

will allow us to create new variables in the server function. This can be particularly helpful when you find yourself using the same code in a couple different places (e.g., we would need to create a subset of the cooler data set based on the cooler type and use this for estimating the linear regression model and plotting the data). Let's create a variable called `subset` that will be the subset of the cooler data set, based on the cooler type specified by the user. If we simply define `subset <- cooler[cooler$cooler_type == input$coolertype,]` in the `server` function, we'll see the same error about being outside a reactive context. Since we want to assign a value, we can use the `reactive({})` function to define `subset`. Add the following line to the server:

```
subset <- reactive({
  cooler[cooler$cooler_type == input$coolertype,]
})
```

Now the app will run and we can access the subset of the cooler data set. **To access a reactive variable defined with the `reactive({})` function, we must add parentheses after the variable name, as if it is a function**. Unfortunately, Shiny does not provide specific or helpful errors if we try to access a custom reactive variable without the parentheses, so it is very important to remember this. Shiny's reactivity is a complex concept to comprehend. Once you understand the basics of reactivity, it is a good idea to read more advanced documentation describing reactivity, such as RStudio's tutorial[8]. First, we'll save the subset of the data so that we can easily access it. Next, we'll estimate the linear regression model as shown below.

```
modfit <- reactive({
  lm(formula = time ~ water_bottles,
     data = subset())
})
```

Finally, we'll save the summary of the fitted model so that we can

[8] http://shiny.rstudio.com/articles/execution-scheduling.html

access the r-squared values.

```
sumfit <- reactive({
  summary(modfit())
})
```

Add the code for all three of these reactive variables to the server before any outputs are created.

Creating the text, plot, and table outputs

Now that we have all the necessary reactive variables, we can update the text output in the `server` function using the code below:

```
output$estmodel <- renderText({
  HTML(paste0("The estimated linear regression model for ",
              input$coolertype,
              " coolers is estimated time = ",
              format(round(sumfit()$coefficients[1,1], 2),
                nsmall=2), " + ",
              format(round(sumfit()$coefficients[2,1], 2),
                nsmall=2), "water_bottles."))
})
```

Next we'll create the text output for the r-squared values. Add the code below to the `server` function after `output$estmodel`.

```
output$rsquared <- renderText({
  HTML(paste0("The r-squared value is ",
              format(round(sumfit()$r.squared, 2), nsmall=2),
              " and the adjusted r-squared value is",
              format(round(sumfit()$adj.r.squared, 2),
                nsmall=2), "."))
})
```

Now we want to build a scatter plot for time vs. the number of water bottles, with the estimated regression model and 95% prediction interval bands included. We'll use `renderPlot()` and code from the answer key for assignment #5 to create the scatter plot. Add the following code to the `server` function after `output$rsquared`.

```
output$fitplot <- renderPlot({
  plot(x = subset()$water_bottles,
       y = subset()$time,
       xlab = "Water bottles", ylab = "Time",
       main = "Time vs. Water bottles",
       xlim = c(min(subset()$water_bottles),
         max(subset()$water_bottles)),
       ylim = c(0, max(ceiling(subset()$time/10)*10)*1.1),
       col = "red", pch = 1, cex = 1, lwd = 2,
       panel.first = grid())
curve(expr = predict(object = modfit(), newdata =
  data.frame(water_bottles = x)),
      col = "blue", add = TRUE, lwd = 2,
      xlim = c(min(subset()$water_bottles),
        max(subset()$water_bottles)))
curve(expr = predict(object = modfit(), newdata =
  data.frame(water_bottles = x),
                     interval = "prediction", level = 0.95)[,2],
      col = "blue", add = TRUE, lwd = 2, lty = "dashed",
      xlim = c(min(subset()$water_bottles),
        max(subset()$water_bottles)))
curve(expr = predict(object = modfit(), newdata =
  data.frame(water_bottles = x),
                     interval = "prediction", level = 0.95)[,3],
      col = "blue", add = TRUE, lwd = 2, lty = "dashed",
      xlim = c(min(subset()$water_bottles),
        max(subset()$water_bottles)))
  })
```

Finally, we want to display a table of predicted values and their intervals. We'll use the code from assignment #5 and the `renderTable()` function. Add the following code to the server after `output$fitplot` and run the app.

```
output$predictions <- renderTable({
  data.frame(water_bottles =
    min(subset()$water_bottles):max(subset()$water_bottles),
            round(predict(object = modfit(),
                          newdata = data.frame(water_bottles =
                            min(subset()$water_bottles):max(subse
                          interval = "prediction", levels =
```

```
                                    0.95), 2))
})
```

If we change the cooler type in the drop down menu, we should see that the values in the text output, the plot, and the table all update.

We've now successfully created an interactive app! The results are changing according to user selections. The final app should look like this if you were following along (see Figure 8):

```
library(shiny)
cooler <- read.csv("CoolerReduced.csv", stringsAsFactors=FALSE)
ui <- fluidPage(titlePanel("My STAT850 Shiny app", "Cooler
  Data"),
                h4("Analyzing the Performance of Different Types
                   of",
                    "Coolers and Coolants to Improve Cold Chain
                      Transportation"),
                br(),
                p("The purpose of this", strong("Shiny"),
                  "app is to perform and display the analysis of
                    the cooler",
                  "data obtained from Lowe et al."),
                sidebarLayout(
                  sidebarPanel(
                    selectInput(inputId="coolertype",
                      label="Cooler Type",
                                choices=c("Injection_molded",
                                  "Rotomolded", "PS_foam"),
                                selected=NULL)
                  ),
                  mainPanel(
                    textOutput("estmodel"),
                    br(),
                    textOutput("rsquared"),
                    br(),
                    plotOutput("fitplot"),
                    br(),
                    tableOutput("predictions")
                  )
```

```
                )
)
server <- function(input, output){
  subset <- reactive({
    cooler[cooler$cooler_type == input$coolertype,]
  })
  modfit <- reactive({
    lm(formula = time ~ water_bottles,
       data = subset())
  })
  sumfit <- reactive({
    summary(modfit())
  })
  output$estmodel <- renderText({
    HTML(paste0("The estimated linear regression model for ",
                input$coolertype,
                " coolers is estimated time = ",
                format(round(sumfit()$coefficients[1,1], 2),
                  nsmall=2), " + ",
                format(round(sumfit()$coefficients[2,1], 2),
                  nsmall=2), "water_bottles."))
  })
  output$rsquared <- renderText({
    HTML(paste0("The r-squared value is ",
                format(round(sumfit()$r.squared, 2), nsmall=2),
                " and the adjusted r-squared value is ",
                format(round(sumfit()$adj.r.squared, 2),
                  nsmall=2), "."))
  })
  output$fitplot <- renderPlot({
    plot(x = subset()$water_bottles,
         y = subset()$time,
         xlab = "Water bottles", ylab = "Time",
         main = "Time vs. Water bottles",
         xlim = c(min(subset()$water_bottles),
           max(subset()$water_bottles)),
         ylim = c(0, max(ceiling(subset()$time/10)*10)*1.1),
         col = "red", pch = 1, cex = 1, lwd = 2,
         panel.first = grid())
    curve(expr = predict(object = modfit(), newdata =
```

```
      data.frame(water_bottles = x)),
          col = "blue", add = TRUE, lwd = 2,
          xlim = c(min(subset()$water_bottles),
            max(subset()$water_bottles)))
    curve(expr = predict(object = modfit(), newdata =
      data.frame(water_bottles = x),
                          interval = "prediction", level =
                            0.95)[,2],
          col = "blue", add = TRUE, lwd = 2, lty = "dashed",
          xlim = c(min(subset()$water_bottles),
            max(subset()$water_bottles)))
    curve(expr = predict(object = modfit(), newdata =
      data.frame(water_bottles = x),
                          interval = "prediction", level =
                            0.95)[,3],
          col = "blue", add = TRUE, lwd = 2, lty = "dashed",
          xlim = c(min(subset()$water_bottles),
            max(subset()$water_bottles)))
  })
  output$predictions <- renderTable({
    data.frame(water_bottles =
      min(subset()$water_bottles):max(subset()$water_bottles),
               round(predict(object = modfit(),
                             newdata = data.frame(water_bottles
                               = min(subset()$water_bottles):
                                  max(subset()$water_bottles)),
                             interval = "prediction",
                             levels = 0.95), 2))
  })
}
shinyApp(ui = ui, server = server)
```

Advanced topics

Using uiOutput() to create UI elements dynamically

One of the available output functions we didn't use is `uiOutput()`. This output allows us to render more UI, which can be extremely useful. It is usually utilized to create inputs

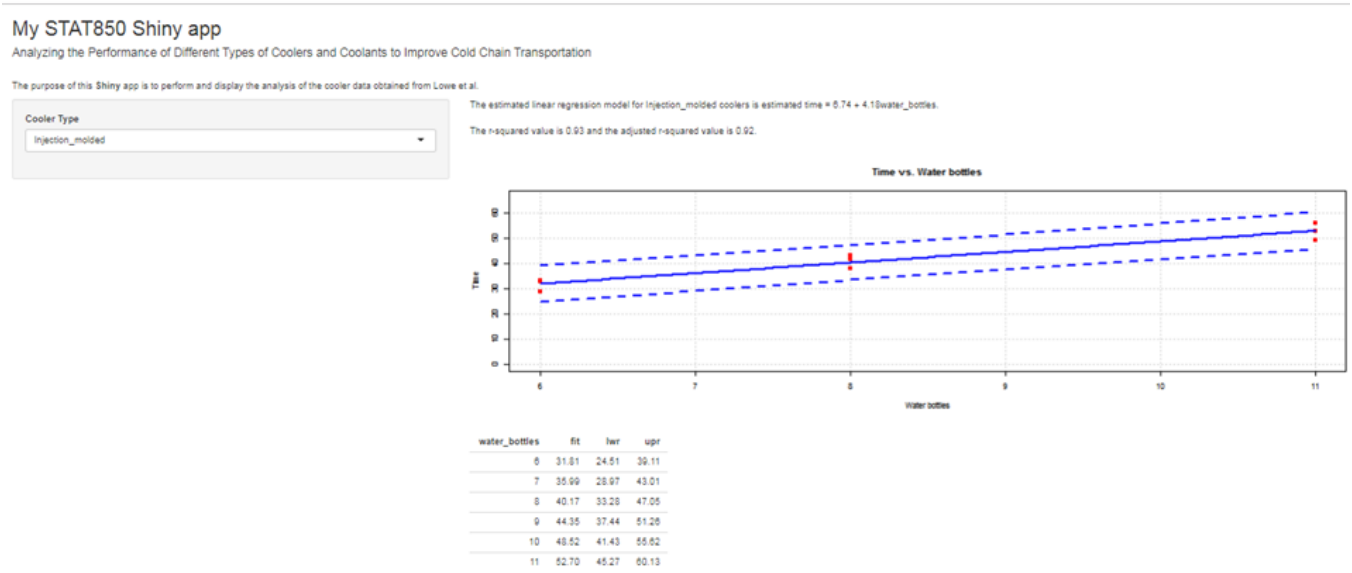

Figure 8: Final cooler app

dynamically from the server, but can be used to create any UI element. Any input defined in the UI is created when the app starts and cannot be changed. But what if one of the inputs depends on another input? We may want to be able to create an input dynamically using `uiOutput()`. The output, which is a UI element in this case, is defined using the `renderUI()` function in the server.

As a basic example, consider the uiOutput_example app. Run the app and you will see that whenever you change the value of the numeric input, the slider input is re-generated with a new maximum value (see Figure 9).

```
library(shiny)
ui <- fluidPage(
  numericInput("num", "Maximum slider value", 5),
  uiOutput("slider")
)
server <- function(input, output){
  output$slider <- renderUI({
    sliderInput("slider", "Slider", min = 0,
                max = input$num, value = 0)
  })
}
```

```
shinyApp(ui = ui, server = server)
```

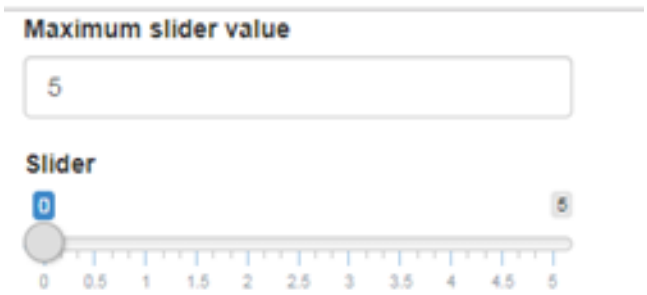

Figure 9: `uiOutput()` example

Conditionally show UI elements

We can use `conditionalPanel()` to display a UI element based on a simple condition. This condition can be as simple as the value of another input. Let's add a check box input to our app, allowing the user to specify which analyses to perform. We'll use a `conditionalPanel()` to display the output depending upon whether the box for that analysis is checked or not. Try adding the check box input to the side bar with the following code:

```
checkboxGroupInput(inputId="analysis", label="Analyses to
  Perform",
                   choiceNames=c("Linear regression", "R-squared
                     values",
                                 "Plot the estimated regression
                                   model",
                                 "Calculate predicted values"),
                   choiceValues=c(1, 2, 3, 4),
                   selected=1, inline=FALSE)
```

Notice that we can provide `choiceNames` to be displayed for each check box and `choiceValues` to be used as the values for each check box within the app code. We could have used the `choices` argument instead, which would provide a single set of labels to be used for both `choiceNames` and `choiceValues`.

Now we can condition on this input. The first argument for

`conditionalPanel()` is the condition to be evaluated. This is specified inside double quotes and inputs are referred to using `input.inputId` rather than `input$inputId`. The other argument is the set of elements to include in the panel if the condition is true. You can learn more with `?conditionalPanel`. We'll use `conditionalPanel()` to display the analyses only if the appropriate box is checked. Because we are conditioning on a `checkboxGroupInput`, we have to use JavaScript (`.includes` to check if the `input.analysis` string contains a certain element) inside the condition. Replace the code in the `mainPanel()` with the code below and run the app.

```
conditionalPanel(condition = "input.analysis.includes('1')",
                 textOutput("estmodel"),
                 br()),
conditionalPanel(condition = "input.analysis.includes('2')",
                 textOutput("rsquared"),
                 br()),
conditionalPanel(condition = "input.analysis.includes('3')",
                 plotOutput("fitplot"),
                 br()),
conditionalPanel(condition = "input.analysis.includes('4')",
                 tableOutput("predictions"))
```

The app now displays only the analyses for which the appropriate box is checked. I have commented out some code for `renderPrint()` in the server and `verbatimTextOutput()` in the UI that I used to figure out how the `input.analysis` variable was formatted before writing the conditions for the `conditionalPanel()` code (see Figure 10).

Multiple tabs in the UI

Because we have three different types of outputs (text, plot, and table), we could remove the conditional panels and organize the outputs into separate tabs. The `mainPanel()` would look like this:

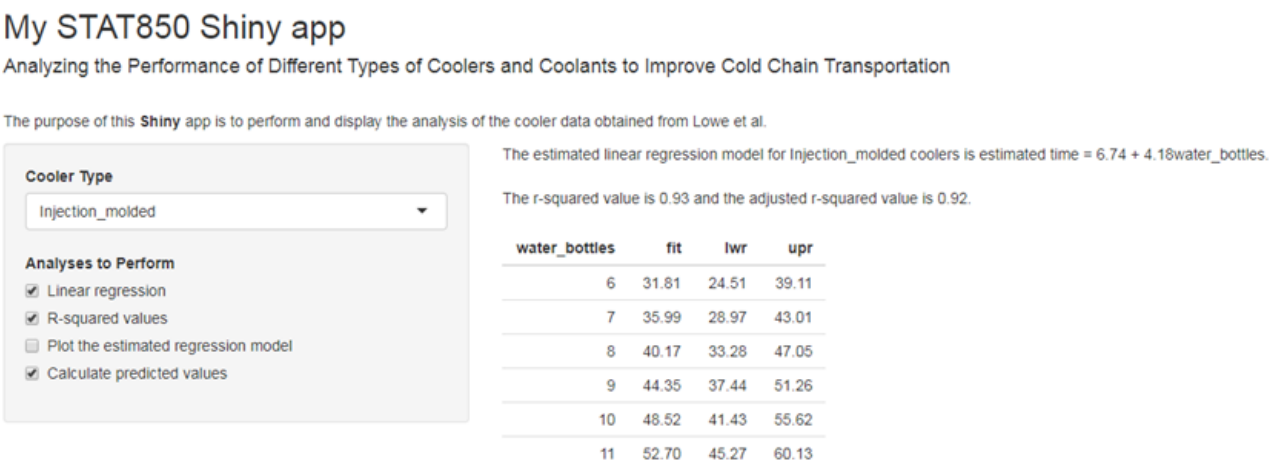

Figure 10: `conditionalPanel()` example

```
mainPanel(
  tabsetPanel(
    tabPanel("Model",
             textOutput("estmodel"),
             br(),
             textOutput("rsquared")),
    tabPanel("Plot",
             plotOutput("fitplot")),
    tabPanel("Table",
             tableOutput("predictions"))
  )
)
```

Try running the app.R file in the Cooler_tabs folder (see Figure 11). Learn more with `?tabsetPanel` or `?navbarPage` (another similar layout).

Remove a dependency on a reactive variable

When we have multiple reactive variables inside a reactive context, the whole code block will get re-executed whenever any of the reactive variables change. If we want to suppress this behavior and make it so that a reactive variable is not a dependency, we can wrap the code that uses that variable inside the `isolate()` function. Any reactive variables inside the `isolate()` function will not result in the code re-executing when their value

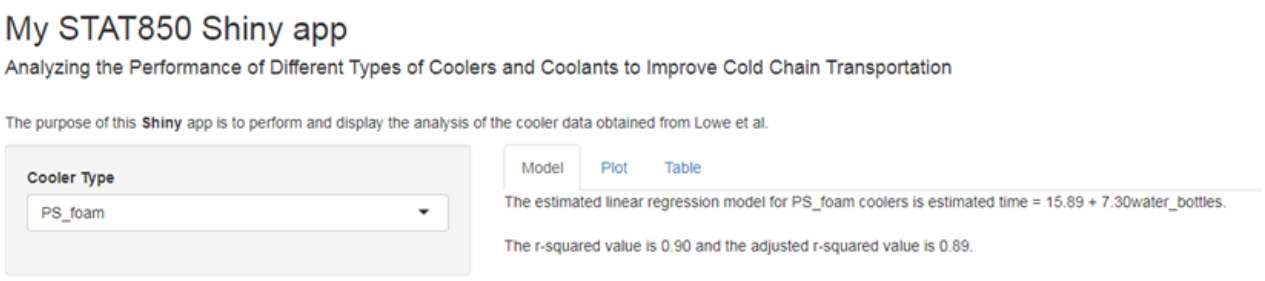

Figure 11: Cooler app with tabsets

is changed. This can also be helpful if we want to add a submit button. The `isolate()` function will allow us to remove dependencies so that a block of code is not re-executed until the submit button is clicked. Learn more about this behavior with `?isolate`.

Validate input values

If we want to ensure that an input has a certain property, we can use the `validate()` function. This function allows us to check that a text input has a keyword included or that a numeric input is between two values. We can also use the `validate()` function to produce error messages in the app if the input does not meet our criterion.

Optional app files

Aside from app.R (or ui.R and server.R for a multiple-file app), there are a few optional files that can be included in the app directory. The global.R file is an optional file that defines objects that we want to be available to both ui.R and server.R. This could include data sets or functions that we want globally defined. In our cooler app, we could save the UI code in ui.R, the server code in server.R, and the code to read in the cooler data set in global.R. We could also write a function to create the plot and save it in global.R so that it is available to the app.

Other files included in the app directory might include data (like the cooler data .csv file) or scripts. A directory of files to be shared with web browsers can also be included. This folder must be named "www" and might include images, CSS, .js, etc. An image can be linked to using img(src = "<file name>") once it is saved in the www directory. JavaScript and CSS files can also be included. If you know JavaScript or CSS, these can help improve the appearance of the app. There are optional DESCRIPTION and README files that can be included in the app directory too.

Add-on packages

The use of the `tableOutput()` + `renderTable()` functions result in a static, plain-looking table. We can download the `DT` package and use `DT::dataTableOutput` + `DT::renderDataTable()` to create nicer looking, interactive tables. Learn more on DT's website[9].

The `shinyjs` package easily improves the user interaction and experience by allowing us to use common JavaScript operations in our app. The `shinyBS` package allows us to add popovers and tooltips, helpful text hints that provide more information to the user when the mouse hovers over an input or displayed results.

The `shinythemes` package allows us to easily alter the appearance of our app by providing more than 15 prefabricated visual themes to choose from. It also allows the use of a `themeSelector()` function, where we can open our app and interactively switch between the available themes to test the appearance in context. The `shinydashboard` package allows us to easily create dashboards, a type of organization/visual layout with a header, sidebar and body.

Other useful packages include `leaflet` for interactive maps and `ggvis` for interactive, web-based plots that are an improvement

[9] https://rstudio.github.io/DT/

over the `ggplot2` package.

Share your app

So far, we've been running Shiny locally on our computers, which means your computer was powering the app. This also means that the app was not available to anyone on the internet. To share your app, you can send all the necessary files via a .zip folder to any interested party, or you need to host it on a server.

1. Host your app on shinyapps.io, a cloud-based service from RStudio.
   (a) You can create a free or professional account at https://shinyapps.io. The main differences in the types of accounts are the number of apps you are allowed to host and the number of times you are allowed to have your app run per month. Even the free account provides some basic stats about the use of your app.
   (b) Click the Publish Application icon in RStudio or run `rsconnect::deployApp("<path to directory>")` and follow the instructions. You might need to login and/or install a couple of packages the first time you try to publish.
   (c) Once the app is successfully published, you'll be redirected to your app in the internet browser and care share the URL with colleagues, family, friends, or whoever.
2. Build or purchase your own private Shiny server. This provides a lot more freedom and flexibility, but requires you to have and administer your own server.