We have been working on a Shiny application which allows the user to upload data, do some analysis and processing on each variable in the data, and finally use the processed variables to build a statistical model. As there may be hundreds of variables in the data, the user may want to process only a few variables in one sitting and later continue the work from where they left off. They may also want to pass on their work to another user who can then continue processing the variables. The Shiny application is also divided into modules – so there are different modules for data upload and exploration, processing each variables, building the statistical model and run further analysis on the model.

One option to allow this functionality would be to store the partially processed data and variables into an *RDS* file, which can then be loaded onto another session allowing an user to continue the work. This post describes a simple application to implement this functionality. This framework can also be extended to larger Shiny applications.

The example application has three tabs. The first tab is the *Data* tab. This allows the user to upload a CSV file. Once uploaded, the names of the variables in the data along with their data types are displayed in a table below. The data is read using the *read\_csv* function from the *readr* package, and the data types detected automatically. The second tab is the *Distribution* tab. This provides a dropdown of the numeric (class *numeric* or *integer*) variables available in the data. On choosing a variable, it is grouped into deciles and the volume for each group is computed (of course, by definition, the volume will be approximately 10% of the total for each group) and displayed in a table. The third tab allows you to save the state of the application. A download button will store the data, variable details and the computed deciles into an *RDS* file. The *RDS* file can be re-uploaded back to the application to retrieve the data and the deciles which were already computed.

There are two Shiny modules. The first module is the *data\_module* which allows you to upload the data and then view the variables and their types. The second module is the *distribution\_module* which provides a dropdown of the numeric variables available in the data and decile them.

The overall UI of the application is created using the code below.

ui <- shinyUI(

navbarPage(

"Data Distribution",

id = "nav\_top",

tabPanel(

"Data",

data\_ui("data\_module")

),

tabPanel(

"Distribution",

distribution\_ui("distribution\_module")

),

tabPanel(

"Save/Restore",

downloadButton("save\_state", "Save to file"),

br(),

br(),

br(),

fileInput("restore\_state", "Restore from file",

placeholder = ".rds file")

)

)

)

The *data\_ui* and *distribution\_ui* are the UI functions defined by the data and distribution module respectively. The third tab is the “Save/Restore” tab, which comprises a *downloadButton* to download the *RDS* and a *fileInput* to upload the *RDS*.

The *data\_ui* and *distribution\_ui* functions are really simple. The only thing to note is the use of the *NS* function to create a namespaced ID.

data\_ui <- function(id) {

ns <- NS(id)

tagList(

fileInput(ns("in\_data"), "Upload data", accept = "text/csv",

placeholder = "CSV file"),

tableOutput(ns("out\_data"))

)

}

distribution\_ui <- function(id) {

ns <- NS(id)

tagList(

selectInput(ns("varlist\_distribution"), "Select variable", choices = NULL),

tableOutput(ns("binned\_distribution"))

)

}

The *data\_ui* consists of a *fileInput* to upload the data and a *tableOutput* to view the variables in the data along with their types. The *distribution\_ui* consists of a *selectInput* to choose the variable for which the deciles need to be computed and a *tableOutput* to view the results.

The most important functions in this application are the server functions. This includes the overall *server* function for the application as well as those of the individual modules.

The *data\_server* function is shown below.

data\_server <- function(input, output, session, restored\_ds) {

# Reactive uploaded file

data\_file <- reactive({

validate(need(input$in\_data, message = FALSE))

input$in\_data

})

# Reactive to store data if uploaded or restored

data\_ds <- reactive({

if (!is.null(restored\_ds())) {

ds <- restored\_ds()

} else {

ds <- read\_csv(data\_file()$datapath, guess\_max = 50000)

ds <- as.data.frame(ds)

}

ds

})

# Name and data types of variables for uploaded or restored data

data\_vars <- reactive({

ds <- data\_ds()

vnames <- gsub("\\.", "\_", make.names(colnames(ds), unique = TRUE))

vtypes <- vapply(ds, class, character(1))

data.frame(vnames = vnames, vtypes = vtypes, stringsAsFactors = FALSE)

})

# Display name and data types

output$out\_data <- renderTable({

data\_vars()

})

# Return

list(

data\_ds = data\_ds,

data\_vars = data\_vars

)

}

The arguments to the function include the standard Shiny boilerplate – *input*, *output* and *session*. In addition, we have one more argument called *restored\_ds*. When the application runs, the global *server* function will call this module with *restored\_ds* set to *NULL*. However, if the user restores the application state by uploading a *RDS* file, then the data stored in the *RDS* will be passed as an argument to this function. In the definition of the reactive *data\_ds*, we first check if *restored\_ds* is *NULL*; if yes, we read the CSV data uploaded by the user and if not, we return the restored dataset passed to the server function. As you might guess, the variable which is passed by the global *server* function is itself a reactive (a *reactiveValue* object in Shiny). The use of reactivity at each stage allows for the outputs to be refreshed on any change. Finally, this module returns two data frames – one containing the uploaded or restored data and the other containing the variables along with their data types.

The *distribution\_server* is shown below.

distribution\_server <- function(input, output, session, data\_ds\_vars,

restored\_decile) {

# Global reactive values to store deciles

deciles\_g <- reactiveValues()

# Numeric variables

numeric\_vars <- reactive({

validate(need(data\_ds\_vars, message = FALSE))

data\_vars <- data\_ds\_vars$data\_vars()

data\_vars <- data\_vars[data\_vars$vtypes == "numeric" |

data\_vars$vtypes == "integer", ]

vnames <- data\_vars$vnames

vnames

})

# Update choice of variables for which deciles can be created

observe({

updateSelectInput(session, "varlist\_distribution", "Select variable",

choices = numeric\_vars())

}, priority = 20)

# If the list of variables change, reset everything to NULL

observeEvent(numeric\_vars(), {

for (name in names(deciles\_g)) {

deciles\_g[[name]] <<- NULL

}

}, priority = 10)

# Calculate deciles whenever a new variable selection is made and display

observeEvent(input$varlist\_distribution, {

validate(need(data\_ds\_vars, message = FALSE))

req(input$varlist\_distribution)

if (is.null(deciles\_g[[input$varlist\_distribution]])) {

if (!is.null(restored\_decile()) &&

!is.null(restored\_decile()[[input$varlist\_distribution]])) {

deciles\_g[[input$varlist\_distribution]] <-

restored\_decile()[[input$varlist\_distribution]]

} else {

deciles\_g[[input$varlist\_distribution]] <<-

bin\_numeric(data\_ds\_vars$data\_ds(), input$varlist\_distribution)

}

}

output$binned\_distribution <- renderTable({

deciles\_g[[input$varlist\_distribution]]

})

})

# Store deciles as a list

deciles <- reactive({

Filter(Negate(is.null), reactiveValuesToList(deciles\_g))

})

# Return

deciles

}

This is slightly more complex than the *data\_server*. Consider what happens when a new CSV file is uploaded by the user. We first find the numeric variables in the data and display it in the dropdown. This is accomplished using the reactive *numeric\_vars* which refreshes whenever the underlying data changes. We also store the deciles created for any variable in the *reactiveValues* object *deciles\_g*. If the underlying data changes, we set all the computed deciles to *NULL* – this is accomplished using a high-priority *observeEvent* block.

Another *observeEvent* block observes any changes in the choice of the input variable. It first checks if the deciles have already been computed by checking if the corresponding entry in *deciles\_g* is NULL. If not, it calls the function *bin\_numeric* defined in this module to compute the deciles. Otherwise, it simply retrieves the already saved value.

In order to enable a user to restore previously computed deciles, the function accepts an additional argument called *restored\_decile*. This is similar to the *restored\_ds* argument we have seen in *data\_server* above; a NULL value is passed when the application is run but the user may choose to pass previously computed deciles from the *RDS* file. In the *observeEvent* block described in the previous paragraph, we also check if *restored\_decile* is not *NULL* and whether it has the deciles for the variable stored inside it; if yes, it does not recompute the deciles.

This function returns the *deciles* reactive, which converts the *deciles\_g* object to a list, removes any *NULL* values and returns it back to the global server.

Finally, the global *server* function is shown below.

server <- function(input, output, session) {

restored\_ds <- reactiveVal()

restored\_decile <- reactiveVal()

# Call modules

data\_ds\_vars <- callModule(data\_server, "data\_module", restored\_ds)

distributions <- callModule(distribution\_server, "distribution\_module",

data\_ds\_vars, restored\_decile)

# Save state

output$save\_state <- downloadHandler(

filename = function() {

paste0("dd-", Sys.Date(), ".rds")

},

content = function(file) {

data\_ds <- isolate(data\_ds\_vars$data\_ds())

data\_vars <- isolate(data\_ds\_vars$data\_vars())

deciles <- isolate(distributions())

save\_list <- list(

data\_ds = data\_ds,

data\_vars = data\_vars,

deciles = deciles

)

saveRDS(save\_list, file)

}

)

# Reactive restore file

restore\_file <- reactive({

validate(need(input$restore\_state, message = FALSE))

input$restore\_state

})

# Reactive to store restored information

restored\_state <- reactive({

rs <- readRDS(restore\_file()$datapath)

rs

})

# Restore state

observeEvent(restored\_state(), {

rs <- restored\_state()

restored\_ds(rs$data\_ds)

restored\_decile(rs$deciles)

})

}

*restored\_ds* and *restored\_decile* are objects of type *reactiveValue* which are initially *NULL*. They can be restored by uploading an *RDS* file, which causes the execution of the *observeEvent(restored\_state())…* block and sets the value for these two reactive variables. The *save\_state* button defines a function to store the data, details of the variables and any computed deciles into an *RDS* file. The modules are called using the *callModule* function with appropriate arguments.

Note that the entire behaviour is possible due to reactivity. For example, when the user starts the application, uploads a CSV file and starts creating deciles, the *restored\_ds* and *restored\_decile* values are *NULL* at this stage. Now, if the user uploads a previously saved *RDS*, then the reactivity of *restored\_ds* ensures that the *data\_ds\_vars* returned by the data module is recomputed. This in turn ensures that the *numeric\_vars* reactive in the distribution module is recomputed leading to a change in the dropdown. Also, the *restored\_decile* argument ensures that any previously computed deciles are read from the *RDS* and the *bin\_numeric* function is not called on these variables.