Georg-August Universität Göttingen

Statistical Programming with R

Gotta Read 'Em All: An RStudio Add-In to visually read different file-formats into R

Author:
Stanislaus Stadlmann,
Student ID: 21144637

Supervisor
Paul Wiemann, M.Sc.

Submitted on August 12, 2016

GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Contents

List of Figures

List of Tables

1 Motivation

R is a statistical software with an almost uncountable number of functions for different statistical methods, procedures and graphs. On the Comprehensive R Archive Network (CRAN) alone, the most popular network for adding new features to R via so called "packages", more than 8000 packages are ready to be downloaded[1]. Each of them provide a variety of functions to solve different tasks. For example, a package called "Vector Generalized Linear and Additive Models" (or short VGAM) can be easily installed via the R command `install.packages("VGAM")` and, once loaded in R, provides functions to estimate a variety of different regression models.

These aforementioned packages with the underlying functions make R the popular statistical package it is today. But most of these additional features require some data to be of any use, most prominently regression model estimation functions. There exist datasets shipped with R, but for most academical purposes, external data will be required to generate new insights.

To read data into R, there are multiple ways, depending on the data type (e.g. .csv, .xlsx) and also the data size (big data, small data). Reading a .csv file, for example, can be achieved via the built-in R function `base::read.table("filename.csv")`. Big .csv files can be read very quick with the function `data.table::fread("bigfile.csv")` from the data.table package. Other packages provide even more functionality, e.g. for dealing with strings or a smaller number of required arguments inside of a function. Most of those packages are also available on CRAN.

The availability of packages to read different filetypes in numerous ways is very helpful for the advanced R user, because there is almost no filetype that cannot be read via an R function. But it also poses a problem: If there are so many ways that a user can read a file into R, how will she/he remember all the necessary packages and functions, and their function arguments? This problem is often encountered by new R users, who want to use R's extended functionalities but fail at importing data into their working environment.

An answer to this problem is provided by Thomas Leeper's R package called "rio", which tries to minimize redundancy by wrapping R reading functions into one import `rio::import()` and one export function `rio::export()`.

[1]There were exactly 8895 packages on CRAN at August 4, 2016.

The R package introduced with this paper takes it one step further. Built on the Shiny Framework and implemented as an RStudio Add-In, “Gotta Read ’Em All” provides a GUI for reading all different file-formats into R.

The general process is the following: In the beginning, the user selects a file on her/his computer. After some adjustments (which are done interactively), the proper function to read the file is pasted into the console, with an object name that can be specified by the user. In between, the user can always head to the preview to see what the parsed file would look like with the current options.

Using this Add-In, the user can now read data into R without remembering any code, but still obtains the correct R code to re-parse the data at a later point.

2 Underlying Frameworks

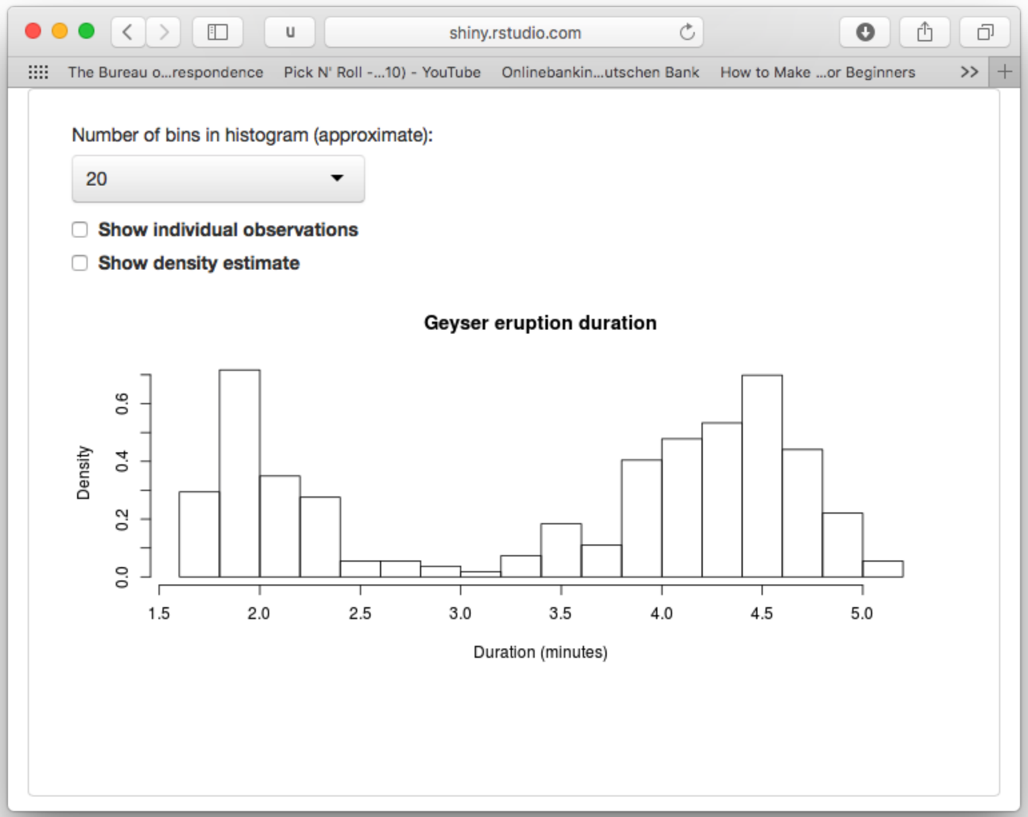
2.1 The Shiny Framework

The Shiny Framework[2] is in itself an R package designed to create interactive visualisations with R functions and HTML code. The author describes the package as “combining the computational power of R with the interactivity of the modern web” (citing a web page blabla).The goal is to create applications with clickable interfaces quickly showcasing different scenarios. This is done via reactive R functions, which are run everytime a user interacts with the GUI.

Figure 1 shows an example Shiny Application, consisting of some user interface (UI) control elements (a select button, and tick boxes) and a graph. The UI is reactive; whenever the user ticks a box or selects a different value in the first box, the graph changes its look. This is built upon reactive functions, which run every time a value inside them changes. The values that are allowed to change are therefore bound to the UI elements.

Shiny Apps consist of two elements: The UI and a “server” side. The UI includes functions which wrap HTML to build the viewable part of the App, for example buttons and placement of graphs. The server side consists of functions that specify the reactive R functions which create dynamic output displayed on the

[2]http://shiny.rstudio.com


shiny.rstudio.com
The Bureau o...respondence
Pick N' Roll -...10) - YouTube
Onlinebankin...utschen Bank
How to Make ...or Beginners

Number of bins in histogram (approximate):
20

Show individual observations

Show density estimate

Geyser eruption duration
Density
0.0
0.2
0.4
0.6
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Duration (minutes)

Figure 1: Screenshot of an example Shiny Application.

UI side, and when the functions should be run. Both sides then are able to automatically communicate with each other to create a smooth interactive experience for the user. The code for an example Shiny app is attached below:

```r
# Define UI
ui <- bootstrapPage(
  numericInput("n", "Number of obs", 100),
  plotOutput("plot")
)
# Define Server
server <- function(input, output) {
  output$plot <- renderPlot({ hist(runif(input$n)) })
}
app <- shinyApp(ui, server)
# Run App
runApp(app)
```

Code-Chunk 1: Example code for a Shiny Application[2].

It is now possible to see in Code-Chunk 1 that both the UI and server elements are R objects, while the server also resembles an R function. `runApp()` then opens up the Application.

2.2 Shiny Gadgets and RStudio Add-Ins

Shiny Applications are useful for displaying interactive visualisations, but they are made for displaying results to the end user. “Shiny Gadgets”[3], an extension of the Shiny framework, are supposed to be part of the programming or analysing process. They are built on the same framework that was introduced with Shiny, but serve the purpose of making programming challenges a little easier. For example, a Shiny Gadget could be used to provide a UI for downloading certain data from complex websites.

“RStudio Add-Ins”[4] are Shiny Gadgets that are built right into RStudio, an Integrated Developer Environment (IDE) for R. Calling the Shiny Gadget is made easier, as the RStudio user only has to press two buttons. Furthermore, Add-Ins have extended access to RStudio itself via a package called “rstudioapi”. For example, Add-Ins are able to paste a string into the console and can modify the currently opened R script.

The combination of the Shiny framework and RStudio add-ins creates the ideal setting for an Add-In that helps the user parse any data format into R. The Shiny Framework provides interactiveness and the RStudio connection makes it easier to call the Application out of an IDE.

3 Implementation

An R-Studio Add-In has to be installed via the R package ecosystem, so GREA is also wrapped up in a package called GREA. Calling the Add-In is done via the main function `GREA::GREA()`. Also, there exist a couple of helper functions, which were necessary to reduce redundant R code. The following functions are implemented:

- `GREA()`
- `GREA_read()`

[3]http://shiny.rstudio.com/articles/gadgets.html
[4]https://rstudio.github.io/rstudioaddins/

- `wd_check()`
- `fileChoose()`

As stated above, `GREA::GREA()` is the function that starts the Add-In. `GREA::GREA_read()` is the most important helper function, which converts any filetype into the data.frame R class inside the global environment. For checking if a file is inside the current R working directory, `GREA::wd_check()` was written. This function arose from my personal frustration with large filepaths. `GREA::fileChoose()` has the same utility as `base::file.choose()` (choosing a file interactively), except that it returns `NULL` when cancelled.

3.1 The Main Function: GREA()

As explained in Chapter 2.2, Add-Ins are built on the Shiny Framework. `GREA()` is therefore also a wrapper for a Shiny Application. In this section, the general structure of the function will be demonstrated.

```
GREA <- function() {
  ui <- miniPage(
    # User Interface Functions #
    # ...
  )
  server <- shinyServer(function(input, output, session) {
    # Reactive Server Functions #
    # ...
    observeEvent(input$done, {
      # Functions that paste the code into the console
    })
  })
  # Functions that start the Add-In
  app <- shinyApp(ui = ui, server = server)
  viewer <- dialogViewer(dialogName = "GREA",
                         height = 350, width = 500)
  runGadget(app, viewer = viewer, stopOnCancel = FALSE)
}
```

Code-Chunk 2: Structure of the `GREA()` function

Code-Chunk 2 shows the structure of GREA's main function. Similarly to Shiny

Applications, we can see that the `ui`(lines 2-5) and `server`(lines 6-12) objects are specified first. For creating the `ui` object, the `miniUI::miniPage()` function from the miniUI package (cite??) is used, which is specifically targeted at small user interfaces in the style of smartphone apps. The `server` object, also resembling an R function, consists of reactive functions that are called every time the user interacts with the UI. This includes the selection of a file to be read in on the user's computer, or the adjustment of reading conditions (e.g. a change in the comma seperator in text-delimited files).

Next, both objects are assembled to a new object called `app`(line 14). In lines 15 and 16, an object named `viewer` is generated via `shiny::dialogViewer()`. This step ensures that GREA is started as an integrated window in RStudio with the right height and width. In the end, the function `runGadget()`(line 17) is called to start the Add-In. In essence, this function has the same functionality as the `shiny::runApp()` function, with the addition of better automatic handling in the event of the user cancelling the app.

The piece of code that sets this Add-In apart from Shiny applications and Shiny Gadgets is resembled by the contents of lines 9-11, a function which is run when the user presses the "done" button.

```r
observeEvent(input$done, {
  # Paste Code into Console
  if (nzchar(fileloc()) && nzchar(input$name_dataset) && !
     is.null(dataset())) {
    # Get code that was used to read dataset
    expr <- attributes(dataset())$GREAcommand
    # Assemble code
    code <- paste0(input$name_dataset, " <- ", expr)
    # Paste into Console
    rstudioapi::insertText(text = code, id = "#console")
  }
  # ... and then stop the app
  stopApp()
})
```

Code-Chunk 3: Contents of reactive function for "Done"-event

Code-Chunk 3 shows the detailed contents of lines 9-11 in Code-Chunk 2. After the user has specified all necessary variables interactively, she/he presses the "done" button. This triggers the event called `input$done`, which leads to the above function being called.

Line 3 checks if three conditions are met:

1. A file location is correctly specified.
2. A dataset name is correctly specified.
3. Reading the dataset with the options provided by the user is successful.

These conditions make sure that the user doesn't obtain code for reading a file which will yield an error. If and only if these conditions are met, the code to read the data is assembled (lines 5-7) and then pasted into the R console (line 9). The procedure to paste the code into the console makes use of the `insertText()` function from the "rstudioapi" package, previously mentioned in Chapter 2.2. After this procedure, the user obtains the code to read the specified file and only has to execute the command to attach the new data to R's global environment.

irgendwo muss hier noch rein was alles dazwischen passiert und wo sozusagen die specifications sind vor allem im server

vielleicht implementation vor das andere packen?

3.2 The File Reading Function: GREA_read()

Chapter 3.1 explained the implementation of the main function, `GREA::GREA()`. Though this function is the cornerstone of the overall Add-In, it relies heavily on a function that reads filetypes into R, `GREA::GREA_read()`. Writing this function was challenging, as it had to fulfill the following requirements at once:

1. Read a filetype and convert it to an R object.
2. Keep the code after a successful reading and attach it to the R object.
3. Omit rarely used arguments in the code that should be pasted into the console (e.g. `NA` values).

Requirement 1 is the goal that many R reading functions want to achieve. Requirement 2 is necessary, because after successful interactive parsing, the right code should be pasted into the active R console. Requirement 3 was included, because only the reading conditions that are not default values should be included in the code that the user obtains in the end.

To fulfill Requirement 1, `GREA_read()` automatically detects the filetype of the selected file. Then, depending on which filetype was detected, it utilizes the following functions from other packages to read data:

- `base::read.table()` for text-delimited files (e.g. .csv, .txt, etc),
- `R.matlab::readMat()` for .mat files (MATLAB),
- and `rio::import()` for all other filetypes.

After detecting the filetype and selecting the right function for reading, a “call” is assembled. In general, an R call is an executed or non-executed command. In this case, the command is created depending on the arguments that the user specified. To show the procedure, an example code snippet is provided below.

```
1 else if (any(filetype == c("xls", "xlsx"))) {
2       expr <- quote(rio::import())
3       expr[c("file", "which")] <- list(filelocation,
           sheetIndex)
4       if (!missing(na.values))
5         expr[c("na")] <- na.values
6       if (skip > 0)
7         expr[c("skip")] <- skip
8     }
```

Code-Chunk 4: Reading procedure of `GREA_read()`

In Line 1 of Code-Chunk 4, the condition for the selected file being of an Excel type (.xls, .xlsx) is examined. Once the condition is met, Line 2 utilizes the `base::quote()` function, which creates a non-executed call of the first argument, in this case `rio::import()`. In Line 3, the arguments `file` (filelocation) and `which` (which of the Excel sheets should be parsed) are specified using the arguments of `GREA::GREA_read()`. Lines 4-7 fulfill Requirement 3 which was specified at the beginning of this chapter: For rare function arguments, these arguments should only be included when they are explicitly specified. In this case, arguments `na` and `skip` are only then added to the call (object `expr`) when they deviate from default values (`na.values` is not a missing argument and `skip` is greater than 0).

After the call is assembled correctly, it should not only be executed (Requirement 1), but it should also be attached to the object that was created during the call (Requirement 2). This is achieved by the following code at the end of the function:

```r
return(structure(eval(expr), GREAcommand = deparse(expr)))
```

Code-Chunk 5: Return Command of the `GREA_read()` function

Code-Chunk 5 shows the execution of said assembled call. The `eval()` command first executes it and the `structure()` function then attaches the non-executed call (`expr`) to the object that was created, thus fulfilling Requirements 1 and 2. Afterwards, the enhanced object is returned (including its attribute).

Whenever a file is now read into the R environment via `GREA_read()`, the code that was used can be accessed.

3.3 Helper Functions

The GREA package features two helper functions: `fileChoose()` and `wd_check()`. As mentioned before, `fileChoose()` is just an enhanced version of `base::file.choose()` that returns `NULL` if the interactive file selection is cancelled. `wd_check` has two tasks. First, it transforms Windows filepaths to a path that can be read by any Operating System. Second, the function checks if a file selected via `fileChoose()` lies inside of the current R working environment, thus being able to shorten the filepath while reading files (that is were it got it's name from). The returned object is just another filepath, which is only a subpath if the file lies inside the current R working directory.

4 Usage

4.1 Requirements

RStudio Add-Ins require the newest release of RStudio. they can be downloaded here and here.

4.2 Installation

The Add-In was uploaded to GitHub (an open-source code sharing platform), so it can be easily installed. Installation is done via the following code:

```
1 if (!require(devtools))
2   install.packages("devtools")
3 devtools::install_github("Stan125/GREA")
```

Code-Chunk 6: Installation of GREA

Lines 1-2 of Code-Chunk 6 check if the devtools package is installed and install it if not. After executing Line 3, GREA is installed on the user's computer.

4.3 Starting the Add-In

To call the Add-In, the user has to click on the Add-In Tab and select "Gotta Read Em All". The Add-In itself then pops up.

4.4 Selecting the Dataset

As seen in Figure ... Once the Add-In is started up, the user has to press the "Select File" button to select a file on your computer. Then, he/she can type in a name for the name of the dataset in R. A suggestion is made via the selected file's filename. Once the file is loaded into the Add-In, additional options for parsing the file on the right are displayed, depending on the filetype. After the user has made his/her adjustments, he/she can click on the previews tab.

4.5 The preview window

hier muss noch schöner geschrieben werden. The previews tab shows a preview of what your dataframe would look like if you parsed it with the current settings. If something looks odd (e.g. your column names fell into the first row of the dataset), head back to the first tab. We can see that in our case, the column and decimal separators are wrongly specified. If everything is right, still head back to the first tab.

4.6 Adjusting reading conditions

If the preview of your dataframe looked off, you now have the chance to adjust some parameters (e.g. Sheet Index for Excel files, or separator for .csv files). Adjust them so your preview looks exactly like you want them to. When you have typed in a name for your newly aquired dataset and are then finished, press "done". Afterwards, the function to read your dataset is pasted into your console. Boom! You're good to go.

Appendix

References

[1] The Comprehensive R Archive Network. (2016) 'Contributed Packages'. *CRAN*. Available: https://cran.r-project.org/index.html [Accessed 4 August 2016].

[2] W. Chang et al (2016). *shiny: Web Application Framework for R*. R package version 0.13.2. https://CRAN.R-project.org/package=shiny