An Overview of TidyGEO and Some Helpful Resources

Avery Bell

June 25, 2019

## Table of Contents

1. [**Introduction to TidyGEO**](#intro-to-tidygeo)

* [Introduction to Gene Expression Omnibus](#intro-to-geo)
* [Motivation for the app](#app-motivation)

1. [**Use Case**](#use-case)

* [Cleaning up a GEO dataset](#geo-cleaning-steps)
* [Possible application](#example)

1. [**Shiny Apps**](#shiny-apps)

* [Basic format](#shiny-apps-overview)
* [Resources](#shiny-apps-resources)

1. [**TidyGEO Code**](#tidygeo-code)

* [Directory structure](#directory-structure)
* [File contents](#file-contents)
* [Style guide](#style-guide)
* [Functions and how to use them](#tidygeo-functions)

1. [**Next Steps**](#next-steps)

* [Known bugs](#known-bugs)
* [TODO items](#todo-items)
* [Possible future features](#future-features)

1. [**Miscellaneous**](#misc)

* [Error log](#error-log)
* [Shiny/R tips and observations](#observations)
* [Tips for time testing](#time-testing)
* [Other resources](#other-resources)
  + [On RMarkdown](#Rmarkdown-resource)

## Introduction

### Gene Expression Omnibus

An online repository of array- and sequence-based data. Scientists can go here for publicly available data to look at genes, proteins, etc. that interest them. One of the largest databases of its kind. Submissions are [MIAME](https://www.ncbi.nlm.nih.gov/geo/info/MIAME.html) compliant but the metadata associated with the submissions (i.e., descriptions of the sample sources, such as patient age and disease assessment) are not regulated, so they are often messy and hard to analyse using computational tools.

GEO data can be accessed via an R package called GEOQuery, but scientists with no R coding experience must learn the language in order to use this tool, and manual data extraction can often be tedious and error-prone, especially with large batches of datasets that are necessary for certain computational tools to perform their analysis.

There are some tools that have attempted to clean up the metadata and to enable scientists to access it, but these tools either perform the analysis for the user (not giving them access to the cleaned-up data) or rely on crowd-curating efforts, which are not able to keep up with the growing number of datasets on GEO.

### TidyGEO

TidyGEO is an app that lets scientists (even those with little coding experience) download and clean up GEO data to use for their own purposes.

[↑ To top](#contents)

## Use case

[GSE112914](https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE112914): Differential gene expression analysis of MYCN-amplified neuroblastoma cells after Doxycycline inducible shRNA knockdown of JMJD6 gene expression

### [BRB Array Tools](https://brb.nci.nih.gov/BRB-ArrayTools/)

A package that helps visualize and perform statistical analysis on Microarray gene expression, copy number, methylation and RNA-Seq data.

### Cleaning up the data

[TidyGEO](https://tidygeo.shinyapps.io/tidygeo/)

1. Filter out unnecessary columns (to be less overwhelming)
2. Split key-value pairs
3. Split multiple values in a column
4. Shift cells
5. Substitute doxy and control in replicate for NA
6. Substitute “repeat” for “”
7. Filter out NAs in replicate, or maybe duplicates in replicate
8. Download clinical data
9. Briefly look at assay and feature data

### Analysis

Now you can look at the effect of replicates on the expression measurements.

[↑ To top](#contents)

## Shiny Apps

Shiny is an R package that makes it easy to build interactive web apps straight from R.

### The basics

* [Structure of a Shiny App](https://shiny.rstudio.com/tutorial/written-tutorial/lesson1/)
* UI
  + The UI lays out what the app looks like. It is made up of [widgets](https://shiny.rstudio.com/gallery/widget-gallery.html) that allow the user to interact with the app and [layouts](https://shiny.rstudio.com/articles/layout-guide.html) that define how the widgets are arranged on the page.
* Server
  + The server is a function made up of
    - Outputs that define how UI objects should behave (e.g., renderText, renderTable, etc.)
    - Listeners that react to a user’s input (e.g., observeEvent)
  + Basically, it does the **work** of the app to actually make it interactive
* Running an app
  + Call to shinyApp function
  + Must have either app.R with UI and server function or ui.R and server.R
  + Builds the app in the directory you name or the working directory, if no name is given
* Other resources
  + reactive values
    - the difference between reactive values and global variables
  + www/ directory
    - must be named www/, recognized by the app as a place where resources (graphics, CSS, JS scripts) are kept
  + CSS file
    - Cascading Style Sheets: an HTML file that defines how the app should look
    - Shiny automatically uses Bootstrap styling but you can customize this in the CSS file
    - generally it is good practice to use a CSS file rather than defining colors, sizes, etc. inline
  + JS scripts
    - for more complicated reactivity, Shiny may not be flexible enough to meet your needs
    - you can write JavaScript that will define how certain objects react to button clicks, etc.
    - this can also be defined inline but it is good practice to put this in a separate file
  + helper functions
    - code that is run in multiple parts of the app can be put in a function to reduce repetition
    - often used by the server function to do work, but also may save some space in the UI part
    - these are often kept in a separate file to clean up code
  + graphics
    - kept in the www/ directory
    - may include browser icon, logos, icons, etc.

### Resources

* [Introduction to Shiny video](https://www.rstudio.com/resources/webinars/introduction-to-shiny/)
* [Shiny Video and Written Tutorials](https://shiny.rstudio.com/tutorial/)
* [R Style Guide](https://style.tidyverse.org/)

[↑ To top](#contents)

## TidyGEO Code

### TidyGEO structure

* Root
  + .gitignore
    - Lists files that should not be updated on GitHub
  + Code\_Review\_Notes.Rmd
    - Some notes from my Code Review–pretty much the same as this document except less detailed
  + geocurate\_repo.Rproj
    - Some settings for the old Rproject (an RStudio thing)
  + README.md
    - The readme for the GitHub repo
  + TidyGEO/
    - Contains all the code
  + TidyGEO\_Directory\_Guide.Rmd
  + This file
* Level 1 - TidyGEO
  + app.R
    - sources all the R scripts from the ui and server directories
    - creates the ui object and the server function
    - creates some lists of reactiveValues that are used in the app
    - contains some instructions for how column navigation should work
    - contains shinyApp function to knit UI and server
  + deploy\_app.R
    - updates the list of available GEO series to choose from
    - updates the list of formatting functions that can be included in the scripts the user can download
    - launches the app to shinyapps.io
  + error\_log.md
    - a list of errors I have encountered and how I fixed them
  + file\_manifest.txt
    - defines which files are necessary when publishing the app to the web
  + generate\_profile.R
    - generates a profile for the app, see the [time testing tips](#time-testing) section
  + generate\_rscript\_functions.R
    - generates a list of functions that can be used in the scripts the user can download
  + tidygeo-install.R
    - contains the packages needed to run the app and use any of the functions in the app (some packages may be extraneous–this hasn’t been updated in a while)
  + tidygeo\_functions.R
    - some functions for the app that don’t really fall under the “ui”, “server”, or “variables” categories; mostly for script writing
  + tidygeo\_variables.R
    - consists of two parts: global variables and naming conventions
    - global variables are constants used by multiple parts of the app
    - naming conventions are patterns for which variables in the app are named; these are useful because the same name will often be referenced by both the server and the ui portions, so naming conventions make it so you don’t have to remember as many variable names; these are also useful because similar variables will often appear for each of the datatypes (clinical, assay, feature, all) and naming conventions make it so you can name these all similarly and reference each of them without having to remember how you named them
  + time\_testing.R
    - basically a sandbox for optimizing various pieces of code; se the [time testing tips](#time-testing) section
  + VERSION
    - defines the version of the app, updates with every commit
  + help\_docs/
    - Markdown files for the help icons in the app
    - (Generally it’s a good idea to separate out large portions of text from code that actually does work.)
  + rsconnect/
    - an automatically-generated directory that I do not touch; necessary to publish the app to shinyapps.io
  + series\_platform/
    - contains a script that populates the dropdown you see on the first tab
  + server/
    - contains all the code for the server function
    - subsectioned by datatype (clinical, assay, feature, all)
  + ui/
    - contains all the code for the ui object
    - subsectioned by datatype (clinical, assay, feature, all)
  + User/
    - pretty much everything in this directory is useless except rscript\_functions.rds–it is the list of the functions that can be included in the script the user can download
  + www/
    - resources for the app, including help gifs, the series and platform lists, the javascript file, and the css styling file
* Level 2 - server
  + Each directory contains R scripts organized by tab (the name of the tab is usually the name of the R script, and the tab is named by what the user does in that tab)
  + assay/
    - feature\_data.R is the code for the modal that comes up when you click the “Replace ID” button
    - side\_panel.R and helper\_functions.R are pretty much useless (the code in helper\_functions.R should all be in formatting\_functions.R now)
  + clinical/
    - helper\_functions.R is pretty much useless (the code in helper\_functions.R should all be in formatting\_functions.R now)
  + formatting\_helper\_functions.R
    - the functions that can be included in the script the user can download; **these same functions** are used in the app to format the data in various ways–usually one function per tab
  + help\_modals.R
    - functions to create the modals that come up when the user clicks a question icon, observers to make the question icons reactive
  + regex\_modal.R
    - the code to create the modal that allows users to test regular expressions
  + tidygeo\_server\_functions.R
    - functions to create various pieces of the server; these often correspond to similarly-named ui functions to create an object the user can interact with
* Level 2 - ui
  + Each directory contains R scripts organized by tab (the name of the tab is usually the name of the R script, and the tab is named by what the user does in that tab)
  + assay/
    - side\_panel.R and side\_panel\_assay.R are useless
  + feature/
    - side\_panel\_feature.R is useless
  + tidygeo\_ui\_functions.R
    - functions to create various pieces of the ui; these often correspond to similarly-named server functions to create an object the user can interact with
* Level 2 - www
  + reactive\_preferences.js
    - when I have a bunch of buttons that do the same thing, I can’t name the buttons the same thing (duplicate IDs not allowed) and I can’t have an observeEvent that listens to multiple buttons; instead, I give the buttons a class and whenever a button from that class is clicked, the JavaScript here puts that button’s ID in an input for me to listen to with an observeEvent; after I’ve used that input, I have to reset it with this JavaScript so the next button click can set off the observeEvent again
    - it’s generally good practice to put any JS in a separate script such as this one so that your R code doesn’t look as cluttered
  + style.css
    - a bunch of HTML that defines how certain elements of the app should look
    - for each chunk, you’ll see a dot followed by the name of a class followed by some attributes for the class
    - it’s generally good practice to put any styling HTML in a separate script such as this one so that your R code doesn’t look as cluttered

### Style guide and naming practices

I tried my best to follow the tidyverse style guide written by Hadley Wickham as it appears in [this article](https://style.tidyverse.org/).

#### Naming conventions

1. Most variable and function names are all undercase with underscores separating the words. If you ever want to change this convention, please change the SEP variable in tidygeo\_variables.R.
2. Much of the app is built using functions, and these functions use specific names to reference certain variable names in the app. Rather than having the programmer memorize how the functions reference the variable names, I created naming patterns in tidygeo\_variables.R that define how certain types of variables are named. To change the naming pattern, one simply changes tidygeo\_variables.R and all the variable names in the app will be updated. In this way, the naming conventions in tidygeo\_variables.R also prevent extensive refactoring.
3. In general, variables are named using three parts: the general name for the variable, the datatype the variable corresponds to (clinical, assay, feature, all), and some kind of extra tag to make that variable unique, in case it is used for the same datatype in multiple sections.

#### Organization

1. Tabs are organized by datatype and by what the user can do to that data in the tab.
2. Code is usually organized by datatype and tab. Code chunks are separated by a comment ending in three or more dashes, which can be easily produced in RStudio using Ctrl + Shift + R.
3. Chunks of text for the app are usually kept in separate files, not in the R code.
4. All global variables are defined in tidygeo\_variables.R, not in app.R or any of the scripts that build the app.
5. All libraries needed for the app to run are loaded in tidygeo\_variables.R.
6. Functions are always defined outside the tab R scripts. Functions that are used in the UI portion are defined in tidygeo\_ui\_functions.R. Functions that make changes to the data are defined in formatting\_functions.R. Functions that are used in the Server portion are defined in tidygeo\_server\_functions.R. Functions that do not fit into a category are defined in tidygeo\_functions.R.

### Functions and how to use them

There are two reasons that I would **strongly encourage** any programmer working on this app to use the existing functions rather than writing things from scratch.

1. Pieces of UI often correspond to similarly-named pieces of Server.

Just like you pair a textOutput in the UI with a renderText in the Server, I have created pieces of UI that correspond to pieces of the Server, and these corresponding pieces have similar names. One example is with the column navigation you might notice above each of the datatables in the app. The UI for the column navigation looks like this:

fluidRow( # A row for all these elements  
 column(4, # The "previous columns" button  
 secondary\_button(id = "prev\_cols\_clinical\_data\_viewer", label = div(PREV\_ICON, "Previous columns"), class = "prev\_cols")  
 ),  
 column(4, # A spot for the text showing what subset the user is viewing  
 div(textOutput("cols\_visible\_clinical\_data\_viewer"), class = "center\_align")  
 ),  
 column(4, # The "next columns" button  
 div(secondary\_button(id = "next\_cols\_clinical\_data\_viewer", label = "Next columns", icon = NEXT\_ICON, class = "next\_cols"), class = "right\_align")  
 )  
 )

and the server looks like this:

# This fills in the spot with the text showing what subset the user is viewing  
output[["cols\_visible\_clinical\_data\_viewer"]] <- renderText({  
 paste("Showing",   
 clinical\_vals$viewing\_subset[1], "to",  
 clinical\_vals$viewing\_subset[2], "of",  
 ncol(clinical\_vals$clinical\_data),  
 "columns")  
})

See how the variable names in the UI and Server look similar? They all have the name of the element (“prev\_cols”, “cols\_visble”, or “next\_cols”), then the datatype, then the section where the element appears (“data\_viewer”), all separated by an underscore. Now, rather than repeating these chunks of UI and Server for every single datatype, I made functions that will create these pieces for me:

# ui functions  
# The "extra tag" specifies the section for the element; it helps make the element unique  
col\_navigation\_set <- function(datatype, extra\_tag = NULL) {  
 fluidRow(  
 column(4,  
 secondary\_button(id = prev\_col(datatype, extra\_tag), label = div(PREV\_ICON, "Previous columns"), class = "prev\_cols")  
 ),  
 column(4,  
 div(textOutput(visible(datatype, extra\_tag)), class = "center\_align")  
 ),  
 column(4,  
 div(secondary\_button(id = next\_col(datatype, extra\_tag), label = "Next columns", icon = NEXT\_ICON, class = "next\_cols"), class = "right\_align")  
 )  
 )  
}  
# server functions  
# Uses the same "extra tag" as above  
col\_navigation\_set\_server <- function(datatype, extra\_tag = NULL) {  
 output[[visible(datatype, extra\_tag)]] <- renderText({  
 paste("Showing",   
 get\_data\_member(datatype, "viewing\_subset")[1], "to",   
 get\_data\_member(datatype, "viewing\_subset")[2], "of",   
 ncol(get\_data\_member(datatype, dataname(datatype))),   
 "columns")  
 })  
}  
# ui  
col\_navigation\_set("clinical", "data\_viewer")  
# server  
col\_navigation\_set\_server("clinical", "data\_viewer")

1. The use of functions makes variable names very important

You might have noticed how the functions above had to know a little bit about how the variables would be named in order to reference them. For example, the server function had to know that the textOutput spot would start with “cols\_visible” and end in the datatype and an extra tag. Rather than forcing the programmer to memorize this naming schema, I created functions that replicate the naming schema–for this example, a function called visible that creates the string cols\_visible\_clinical\_data\_viewer given the datatype clinical and the section data\_viewer. Functions like this are also helpful when the programmer wants to change the naming schema. For example, if the programmer wanted to change the prefix from cols\_visible to view\_subset, they would simply need to change one line intidygeo\_variables.R rather than refactoring all of the ui and server code.

1. Functions have error-checking in place

This app currently only has four datatypes: clinical, assay, feature, and all, and if one references a datatype that does not exist, say they tried to call expression\_vals$viewing\_subset, they will get an error. Referencing the datatypes by hand (i.e. [datatype]\_vals$[datatype]\_data) will throw a generic “variable not found” error that may be difficult to track down, but using functions (i.e. get\_data\_member(datatype, dataname(datatype))) will make use of my built-in error checking that indicates which function caused the error and keeps the app from crashing.

[↑ To top](#contents)

## Next Steps

### Known bugs

1. Loading assay data into the app is ridiculously slow because the loading function must perform the O(n^2) operation of checking whether each cell is a blank string " " and replacing it with NA if so.
2. If the user wants to load the data into BRB array tools, the assay data can’t have been transposed. There is currently nothing to prevent the user from doing this.
3. There’s not really a great way to deal with invalid column names when the user renames a column to something like “#$%^&\*()“. Currently the app just keeps the names, until any formatting is performed on the column, then the name might change to something like”V1….."
4. The wording for that “drop NA” checkbox in the feature modal in the assay data tab is kind of vague and confusing.
5. When the assay data is not numeric, the error message is unclear on which functionality is not supported by the app.

### TODO items

### Possible future features

1. Sometimes, as is the case for GSE20181, datasets will have subsets of rows that should be treated differently from the other rows. Currently the app only supports edits on entire columns. Perhaps we could add a feature to select which rows the user would like to reformat.
2. There are no data analysis options for the assay data. We could look into some common data analysis things like the ones BRB Array Tools expects the data to already have when it is loaded.
3. There are very few formatting options for the feature data.
4. We could add options to create graphs besides bar plots/histograms.
5. We could add standardization options such as those that appear in GoodNomen.
6. We could suggest certain edits based on what we find in the data. For example, if a column has a non-word character, we could suggest that the user split columns on a delimiter.
7. If not all rows in a column contain a delimiter when the user wants to split the column, the entire column is not split. We could make this smarter so it only splits the rows that contain the delimiter and leaves the rest in the first column.
8. There’s probably a better way to summarize the data so the user can see what’s in all the rows than the bar plots. The labels in the bar plots are often cut off so it’s hard to see the unique values.
9. We could add an option to transpose/summarize clinical data like in the assay data tab.
10. We could make it so the series and platform lists don’t have to be udpated manually.
11. The rownames on the clinical data aren’t really necessary since there’s always a geo\_accession column. We could drop the rownames and make the geo\_accession column an ID like the IDs in the assay and feature data tabs.
12. We could add a way to combine two different series.

[↑ To top](#contents)

## Miscellaneous

### Error log

When I run into a tricky error, I make note of it in error\_log.md, so when I encounter the error again, I can know how to fix it quickly. Here are some of the errors I have encountered so far.

## Errors

1. When you try to click the download button after something has been downloaded already (after you click “Undo”… you can check if this is always the case), you get the following error:

* Warning: Error in normalizeChoicesArgs: Please specify a non-empty vector for `choices` (or, alternatively, for both `choiceNames` AND `choiceValues`).  
  observeEventHandler [C:\Users\Avery\Documents\R\_Code\geocurate\_repo\GEOcurate/app.R#371]
* **SOLVED** Kept the platforms module from showing if the platforms do not exist. “Undo” should work normally now.

1. After downloading GSE68849, if you click on the summary tab, it will throw the following error (perhaps it has something to do with the semicolons in the column names?):

* Warning: Error in <Anonymous>: No handler registered for type .clientdata\_output\_agent:ch1\_width  
   [No stack trace available]  
  Error in (function (name, val, shinysession) :   
   No handler registered for type .clientdata\_output\_agent:ch1\_width
* **SOLVED** Added make.names to the plotting code (Summary tab) to get rid of the illegal characters in the column names, but just for the back-end of the plots.

1. GSE10 Expression graphical summary: Error: missing value where TRUE/FALSE needed

* **SOLVED** Added na.rm = TRUE in the renderUI where it’s figuring out the default width of the bars

1. cannot remove prior installation of package 'mime'

* **SOLVED**
  + Screenshot the place where the packages are being installed
  + End RStudio using Task Manager
  + Navigate to the place where the packages are being installed
  + Try to delete mime directory
  + If the directory says it’s being used by another program,
  + Navigate to the innermost level of the directory
  + Attempt to delete the innermost file
  + The message will tell you which program is using the directory
  + Find that program in Task Manager
  + End the program
  + Open RStudio again
  + Reinstall the package

1. Error in value[[3L]](cond) : there is no package called ‘markdown’

* **SOLVED** Added library(rmarkdown) to app.R (<https://community.rstudio.com/t/error-in-deploying/33173/14>)

## Fixes

1. The app does not allow the user to choose which new column name to remove.

* **LEFT** The “remove” button just takes off the last name that was specified, so if the user wants to delete a different one, they must start over.

1. The app allows the user to put multiple of the same column name.

* **SOLVED** Added unique = TRUE to make.names in the renameCols function.

1. When substituting values, the options of which values to substitute do not update after the substitute operation has been evaluated.

* **SOLVED** Updated the options after the “Substitute” button is pressed.

1. The feature data table does not have select inputs for any of the filters, even though those columns should be factors.

* **LEFT**
  + <https://stackoverflow.com/questions/49699991/how-to-change-the-column-filter-control-in-r-shiny-datatables>
  + <https://rstudio.github.io/DT/>

1. The app is taking too long to load because of the series list file.

* **SOLVED** Used feather files instead of RDS files to store the data. [1] "RDS time:" "0.201472043991089" [1] "Feather time:" "0.0897789001464844"

1. The app is still taking too long to load because of the series list file.

* **SOLVED** Prepared the feather files all beforehand (naming the columns correctly, including the correct columns) in another script, because making a dataframe in the list dropdown is very time-expensive.
* [1] "Reading files 0.107712984085083"  
  [1] "Populating dropdown 0.00898313522338867"

### Shiny/R tips and observations

#### On renderUI vs updateInput

renderUIs are, in general, better than observes with update[]Inputs in them. This is because observes are called more often–whenever their dependencies are updated–rather than only when we need to use them in the UI. I have a hypothesis, though, that with computationally-intensive UIs that we need to use often and that update often, an update[]Input might outperform a renderUI since it only has to update a part of the UI rather than render the entire thing again. However, I have never encountered a UI so difficult to render that renderUI lagged down the code. So IDK.

### Tips for time testing

There is an R package called profvis that can perform profiling for Shiny apps, i.e., it watches the app as the app runs and when the app stops, outputs data on which parts of the code took the longest and which were called the most. This package usually didn’t work for me, so I copy-and-pasted sections of code into time\_testing.R, used Sys.time() calls before and after each section to check how long it took, and printed out the time to see how many times it was called. All the Sys.time() calls should be removed but there might be some I forgot about.

start <- Sys.time()  
for (i in 1:10) {  
 Sys.sleep(0.1)  
}  
end <- Sys.time()  
print(end - start)

## Time difference of 1.094794 secs

### Other resources

#### On Rmarkdown

Pimp my RMD: a few tips for R Markdown, by Yan Holtz [(link)](https://holtzy.github.io/Pimp-my-rmd/)

[↑ To top](#contents)