Paper DV06

Graph Designer: A R shiny application to create publication quality graphs in minutes

Mukul Mittal, Sanofi USA

Mayank Agarwal, Quantzig India

Andre Couturier, Sanofi USA

ABSTRACT:

The Graph Designer is a data visualization tool developed on R shiny platform. It provides a user-friendly interface for graph generation, where users can click and select to create publication quality graphs from scratch without prior knowledge of any programming language. This application is built around the R package ggplot2.

User can import various types of data files into the application. They can manipulate data before feeding it for graph creation. Then, user can assign variables for x-axis, y-axis, adding colors, title, labels etc. Graphs can be further split by using facet functionality e.g., tumor response by gender. Finally, graph can be published, width and height can be adjusted, and plots can be saved in different formats.

The Graph Designer also generates the R code automatically, which promotes reusability, maximize generalizability and is key to ensure the reproducibility and traceability.

INTRODUCTION:

Data visualization is the presentation of data in a pictorial or graphical format. It enables decision makers to see analytics presented visually, so they can grasp difficult concepts or identify new patterns[1]. Using plots to visualize data is simpler compared with complex spreadsheets and/or long tables/listings. Data visualization can also effectively capture outliers, patterns, trends and in some cases, data/logic issues.

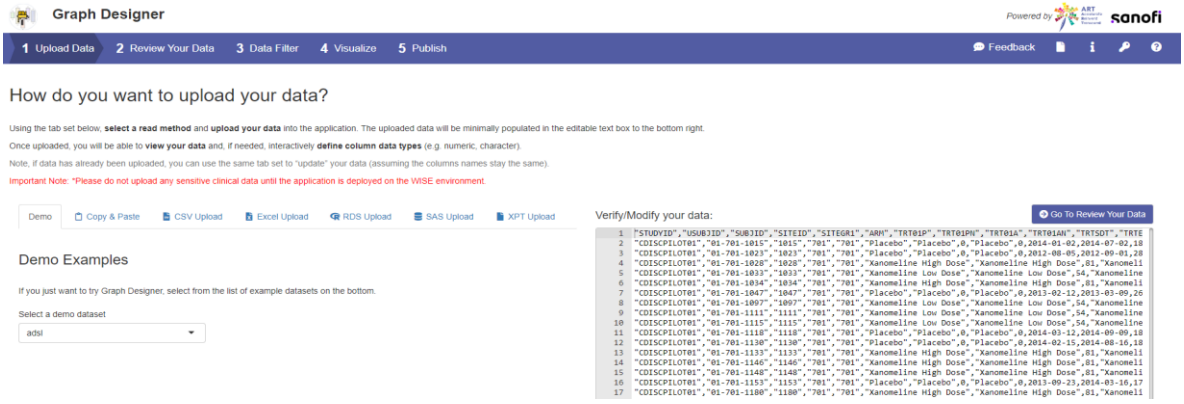
Statistical graphs are the essential component of the clinical trials which are included in the submission package to the regulatory agencies. Clinically sound interpretation of these data, and their clear and concise presentation in regulatory documents, are critical to the success of clinical development programs[2].

The ggplot2 package, created by Hadley Wickham[3] implements the **grammar of graphics**, a coherent system for describing and building graphs. With ggplot2, we can experiment with different scenarios by making slight adjustments. Useful defaults are provided by ggplot2, however understanding of R programming language is required to create graphs.

To address this limitation, we utilized R Shiny framework to create an application based on ggplot2, where user can generate graphs by **"click and select"** without prior knowledge of programming language. Once graph is created, user can download graph along with reproducible code for traceability and validation in future.

DATA IMPORT:

Reading external data into the application is the first step for graph creation. Data can be available in multiple formats e.g., excel, csv, rds, sas and xpt. We designed the app in a way that user can select any type of data and read into the application (Fig.1). We also added few dummy datasets as demo, labelled as demo datasets. We used **rio**[4] package to import data into the application. **rio** allows us to import files in almost any format using one, typically single-argument, function. The **import()** function infers the file format from the file's extension and calls the appropriate underlying data import function for you, returning a simple data.frame.



UI Part:

```
# Excel Upload ----
tabPanel(title = 'Excel Upload', value = 'upload_excel',icon = icon(name = 'file-excel'),
         tags$br(),
         tags$h3('Upload Excel File'),
         tags$br(),
         column(width=12,
                fluidRow(tags$p('Upload a Excel file (including header row/column) from your computer.'),
                         fileInput(inputId = "data_excel_upload_user",
                                   label = NULL,
                                   multiple = FALSE,
                                   width = '500px',
                                   accept = ".xlsx")))
),
```

Server Part:

```
if(isTRUE(tolower(tools::file_ext(input$data_excel_upload_user$datapath)) %in% c("xlsx"))){

  data_excel_load <- try(rio::import(file = input$data_excel_upload_user$datapath), silent = TRUE)
  rv$label_data <- NULL
  message("Checkpoint 1 -> Complete Excel Read -> Initial Reading Data Done...")

  if(inherits(x = data_excel_load, what = 'try-error') || base::NROW(data_excel_load) < 1) {

    output$data_rds_status <- renderUI({
      h4("Status: ERROR!")
    })

  } else {

    output$data_rds_status <- renderUI({
      #h4("Status: SUCCESS!")
      NULL
    })
```

Fig. 1

DATA REVIEW AND CLASS CONVERSION:

Data review is the process to evaluate data to ensure its accuracy and completeness. Once data is read in graph designer, user get a glimpse of overall data, variable types, missing values, and distribution by looking at descriptive statistics (Fig. 2). The application also enables users to go for variable type conversion, if feasible e.g., “integer to character” or “character to date format”.

UI Part:

```
fluidRow(
  column(width = 3,
         h3("Column Types"),
         # Data Structures dynamic mapping
         uiOutput('load_data_str_map')
  ),
  column(width = 7,
         offset = 1,
         fluidRow(
           # show str and summary of data - dynamically
           column(width = 12,
                  style = 'padding: 0px 5px; margin: 5px 0px;',
                  h3("Structure"),
                  verbatimTextOutput(outputId = 'str_data_load', placeholder = FALSE)
           ),
           column(width = 12,
                  style = 'padding: 0px 5px; margin: 5px 0px;',
                  h3("Summary"),
                  verbatimTextOutput(outputId = 'summary_data_load', placeholder = FALSE)
           )
         )
  )
)
```

Server Part:

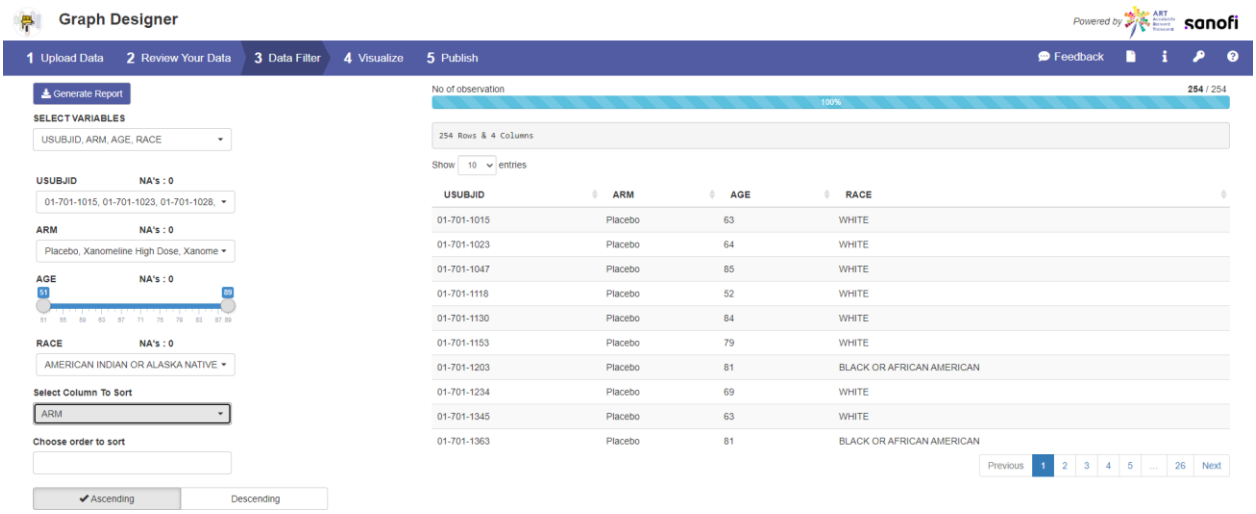
```
if( !all( sapply(data_str_select, is.null) )){
  # error_value <- tryCatch({
  for( i in c('integer', 'numeric', 'character', 'factor', 'ordered', 'Date') ){
    col_nums_change <- grep(i, data_str_select)

    if(length(col_nums_change) != 0){
      if(length(col_nums_change) == 1){
        if(i %in% 'Date')
          data_fetched[[col_nums_change]] <- do.call(what = "as.Date", args = list(data_fetched[[col_nums_change]], origin = "1900-01-01"))
        else
          data_fetched[[col_nums_change]] <- do.call(what = paste0('as.',i), args = list(data_fetched[[col_nums_change]]))
      } else {
        data_fetched[,col_nums_change] <- lapply(data_fetched[,col_nums_change] , paste0('as.',i))
      }
    }
  }
```

Fig.2

DATA SUBSETTING:

Data subsetting in R is the process of selecting specific rows and columns of a dataset to create a new, smaller dataset. **dplyr** package offers a variety of useful functions for subsetting data, such as **filter(), select() and slice()**. In graph designer, we can select variables, which we need to create graphs, subset rows on those variables and sort data automatically in ascending/descending order or sort data manually (Fig.3).



UI Part:

```
column(width = 12,style = "margin-bottom:10rem;",
       column(width = 4,
              DataFilterUI('filters')),
       column(width = 8,
              uiOutput("data_filter_progress_bar_ui"),
              verbatimTextOutput("table_dimension"),
              shiny::dataTableOutput('filter_table'),
              tags$br(),
              uiOutput("filter_header"),
              verbatimTextOutput("summary_data"))
  )
)),
```

Server Part:

```
dataset2 <- reactive({
  if (isFALSE(template)) {
    if (is.null(selected())) {
      dataset
    }
    else if (length(input[["select_variable"]]) == 1) {
      dataset <- data.frame(dataset[selected(), c(input[["select_variable"]])])
      colnames(dataset) <- input[["select_variable"]]
      dataset
    } else {
      dataset[selected(), c(input[["select_variable"]])]
    }
  } else {
    if (is.null(selected())) {
      dataset
    } else {
      dataset[selected(), ]
    }
  }
})
```

Fig.3

GRAPH CREATION:

Variable Assignment and selection of geom layers:

Creating graphs in R is relatively simple process. There are many packages to make graphs in R. ggplot2 is one such package dedicated to data visualization. ggplot2 allows you to build almost any type of graph and it's based on **"The Grammar of Graphics"**. Once data is imported into the application and filtered, user can assign variables to x and y axis, color, and fill options (Fig.4). After that user can pick geom layer based on the data. User can also append multiple geom layers on top of the other.

UI

```
# aes x ----
fluidRow(
  column(width = 3, style = 'padding-top: 7px;', align= 'left',
         tags$b('x')),
  column(width = 8, offset = -1,style = 'padding-top: 0px;',
         selectInput(inputId = "ggplot_aes_x", label = NULL, multiple = FALSE, choices = ""),
         column(width = 12,
                tags$div(id = "x_aes_transform_div",style = "margin-left:-3rem;margin-right:-5rem;",
                         column(width = 2,style = "margin-top:1rem;",tags$b('scale')),
                         column(width = 6,
                                selectInput(inputId = "x_aes_transform", label = NULL,multiple = FALSE,
                                            choices =c("NONE","log10","log2","sqrt"))),
                         column(width = 4,style = "margin-left:-2rem;",
                                radioGroupButtons(inputId = "x_aes_inverse", label = NULL,
                                                  choices = c("+","-")
                                )
```

Server

```
x = if (isTRUE(input$ggplot_aes_x == "" || input$ggplot_aes_x == "NONE")) {
  # will pass null to a list in x aes if nothing is selected
  list(.fn = 'as.null', input$ggplot_aes_x)
```

UI

```
UI for Adding type of plot or geom layer
insertUI(
  selector = "#add_layer",
  where = "beforeBegin",
  immediate = TRUE,
  ui = tags$div(
    id = paste0("add_layer_",layer_id),
    fluidRow(
      column(width = 3, style = 'padding-top: 7px;', align= 'left',
             tags$b(paste('Layer', layer_id))),
      column(width = 9, style = 'padding-top: 0px;',
             selectInput(inputId = paste0("geom_layer_",layer_id),
                         label = NULL,
                         selected = "geom_line",
                         multiple = FALSE,
                         choices = names(geom_layer_args))
      )),
    ## using geomLayer UI module
    geomLayerUI(id = paste0('aes_', layer_id)),
    hr()
  )
)
```

Server

```
observeEvent(input[[paste0("geom_layer_",layer_i)]],{
  rv$geom_layer_list[[layer_i]] <- input[[paste0("geom_layer_",layer_i)]]
  ## using geomLayer Server module
  rv$geom_input[[paste0("aes_map_",layer_i)]] <- geomLayerServer(id = paste0('aes_', layer_i),
                                                                 geom = input[[paste0("geom_layer_",layer_i)]],
                                                                 choices =  isolate(names(rv$configs$data)))
})
```

Fig.4

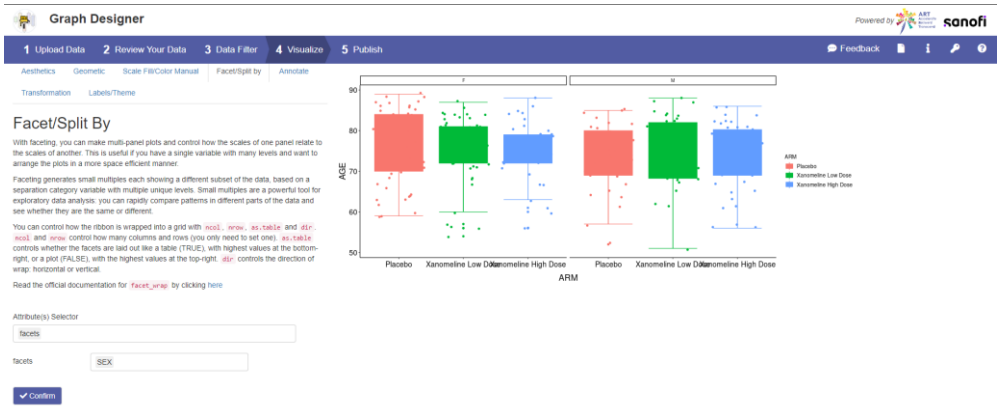
Graph Annotation:

One of the key ideas behind ggplot2 is to build complex graphs, one layer at a time. Thus, if your graph has two annotation layers, the second layer will be applied on first layer. For a given annotation layer, you can start annotating your graph by selecting an annotation type e.g., text, label, point, hline and vline. You can also assign coordinates and change size and shape (Fig.5)

Fig.5

Faceting of the Plots:

When plotting relationship among variables of interest e.g., gender/race, one of the best ways to do that is by partitioning plots into matrix of panels. Each panel shows a different subset of the data. In the application, we can subplot main graph according to any categorial variable. We can also decide to have the subplots on one row or one column (Fig.6).



UI Part:

```
# Facet Tab ----
tabPanel(title = 'Facet/Split by',
         value = "tab3_map_facet",
         # Map facet Section
         column(width = 12,
                style = "margin-bottom: 40px;",
                fluidRow(
                  includeMarkdown(path = file.path('code','facetWrap_writeup.Rmd')),
                  tags$br()
                ),
                fluidRow(
                  tags$div(
                    style = 'padding:0px; padding-top: 10px; padding-bottom: 10px;', align = 'left',
                    facetWrapLayerUI(id = 'facetLayer')
                  )
                ),
                fluidRow(tags$div(
                  style = 'padding:0px; padding-bottom: 10px;', align = 'left',
                  actionButton(inputId = "facetLayer_btn", label = "Confirm", icon = icon("check"), class = "custom-btn-color")
                ))
         ),

         # reset facet selection values ----
         fluidRow(
           column(width = 6, style = 'padding-top: 7px;', align= 'left'),
           column(width = 5, offset = -1, style = 'padding-top: 0px;', align= 'right',
                  actionLink(inputId = 'facet_selection_reset', label = 'Reset Selections',
                             icon = icon("rotate-right"))
           )
         )
),
```

Server Part:

```
if(isTRUE(any(grepl(pattern = "^facet_wrap.*", x = names(rv$graph_initial))))){

  rv$facetWarpCapture <- facetWrapLayerServer(id = "facetLayer", choices = names(rv$configs$data), selected = rv$graph_initial[["facet_wrap"]])

}
```

Fig.6

Title and Labels:

The title is a brief descriptive name given to a plot, which represents its content and subject. We programmed text input in the UI with a placeholder, so that user can write specific title according to the plot (Fig.7). Plot theme also plays important role in graphical display and ggplot2 package provide multiple options for theme types. From a dropdown menu, user can select specific theme as per the need. User can also change X-axis, Y-axis, title, color, and angle of the ticks.

UI Part:

```
fluidRow(
  column(4, class = 'ThemeCustomLabel', 'Plot Title' ),
  column(8,
         textInput('title_text', label = NULL, value = "", placeholder = 'Enter Title of the graph'))
),
```

Server Part:

```
#### Labels ----
labs_custom <- list(.fn = 'labs',
                    title = if(isTRUE(trimws(input$title_text) == "")) c(" ") else list(.fn = 'as.character',
                                                                                        x = input$title_text),
                    y = if(isTRUE(trimws(input$y_lab) == "")) c(" ") else list(.fn = 'as.character',
                                                                               x = input$y_lab),
                    x = if(isTRUE(trimws(input$x_lab) == "")) c(" ") else list(.fn = 'as.character',
                                                                               x = input$x_lab),
                    caption = if(isTRUE(trimws(input$caption_lab) == "")) c(" ") else list(.fn = 'as.character',
                                                                                           x = input$caption_lab)
)
```

Fig.7

Publish the Graph:

Lastly, user can finalize graph by setting width and height of the plot (Fig.8). User can download graph in multiple formats e.g., svg, png, jpeg, pdf etc. Then user also have the option to download code with the graph. Code is regenerative in nature and contain finalized filtered data with ggplot2 code. User can save the code and run in R studio anytime to get the same graph. This validates the results as well as reproducibility of the application.

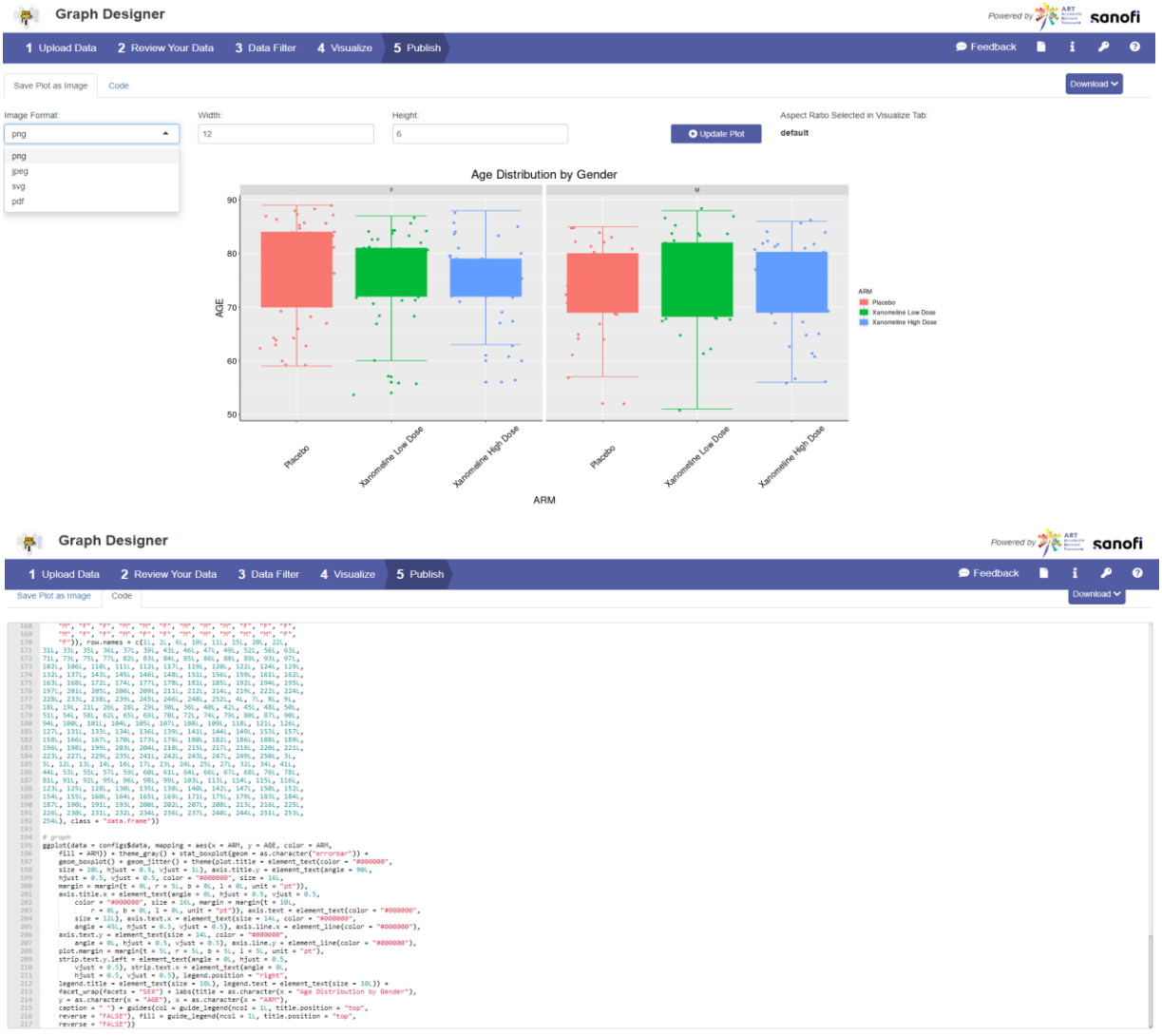


Fig.8

CONCLUSION:

The Graph Designer is a user-friendly interface for graph generation, where users can create publication quality graphs from the scratch without prior knowledge of any programming language. The application is a large wrapper around the R package **ggplot2**, which is a system for declaratively creating graphs, based on **"The Grammar of Graphics".** Graph Designer can also import various types of data files and users can filter and sort data in the application itself (using **dplyr** package on the backend) before feeding the finalized data for graph creation.

Graph Designer is also regenerative. Respective code is generated along with graphs, which covers all the process from data input, data manipulation to visualization. Graphs and respective code with the input data can be downloaded, code can be reused, adapted, validated to ensure reproducibility, and maximize generalizability.

REFERENCES:

1. https://www.sas.com/en_us/insights/big-data/data-visualization.html
2. Masel N, Kendellen M, Ryan R ((2020)), An approach to Dynamic Data Visualization with dplyr, DT and Shiny, DV06, PhUSE US Connect.
3. Wickham, Hadley (July 2010). "ggplot2: Elegant Graphics for Data Analysis". *Journal of Statistical Software*. **35** (1).
4. https://cran.r-project.org/web/packages/rio/vignettes/rio.html

ACKNOWLEDGMENTS:

We gratefully acknowledge our colleagues Xiaodong Luo, Freeman Wang, Benjamin Chiang and Sujeet Das (External) for their contribution in development of this tool.

CONTACT INFORMATION:

Mukul K Mittal
Sanofi, USA
55 Corporate Drive
Bridgewater, NJ, 08807
Cell: 615-554-4560
Email: mukul.mittal@sanofi.com