Tutorial: implementing and visualizing machine learning (ML) clinical prediction models into web-accessible calculators using Shiny R

Hyrum S. Eddington[1]^, Amber W. Trickey[1]^, Vaibhavi Shah[1]^, Alex H. S. Harris[1,2]^

[1]Stanford-Surgery Policy, Improvement Research, and Education Center, Department of Surgery, Stanford School of Medicine, Stanford, CA, USA; [2]Center for Innovation to Implementation, Veterans Affairs Palo Alto Healthcare System, Palo Alto, CA, USA

Correspondence to: Hyrum S. Eddington, BS. 3145 Porter Drive MC5552, Palo Alto, CA 94304, USA. Email: hyrumedd@stanford.edu.

Submitted Feb 17, 2022. Accepted for publication Aug 16, 2022.
doi: 10.21037/atm-22-847
View this article at: https://dx.doi.org/10.21037/atm-22-847

Introduction

As applications of computation and machine learning (ML) become ubiquitous in many areas of science, user interfaces are being developed to allow users to draw meaningful interpretations without extensive expertise in computational methods (1-5). In medicine, clinical prediction calculators are increasingly becoming a popular tool for decision making, allowing clinicians and patients to utilize predictive models in a way that is user-friendly and accessible (6-8). Although many different software environments can be used to create web-accessible user interfaces (UI) for these prediction models, the R Shiny package, along with auxiliary packages such as *shinydashboard* (9) and *flexdashboard* (10), provides easy methods for researchers who are familiar with R to build interactive online interfaces without extensive web development knowledge. Our goal in this paper is to provide a brief tutorial with examples to aid developers of prediction models in making web-accessible interfaces.

An excellent tutorial on creating basic risk calculators using the R Shiny package (11) has already been published in this journal (12). We highly recommend that tutorial to get started making calculators using Shiny. The purpose of this tutorial is to build on that foundation for researchers aiming to implement more complicated predictive models, especially those developed with ML methods, and/or more complex interfaces or visualizations. To achieve these goals, we explain how to use .rds files to store R objects containing complex statistic models that can be directly used by Shiny apps to produce predictions. We also introduce the R packages *shinydashboard* and *flexdashboard* which allow for quick creation of engaging visual displays of clinical predictions, as well as multi-tabular calculator applications.

Step 1: set-up the ui.R file to design the collection of user supplied values

As prerequisites for this tutorial, we assume basic knowledge of R and RStudio (13,14). For readers with limited exposure, we suggest referencing https://www.rstudio.com/training (15) prior to replicating the code in this paper. We also suggest updating R and RStudio to their most recent versions. Similarly to the previously published tutorial by Ji and Kattan (12), we begin by creating a ui.R and server.R file (Appendixes 1,2). The ui.R file, short for 'user interface' file, will contain code that dictates the layout of the application's web-interface, i.e., how input fields, graphics, and application components appear to the user. The server.R file, on the other hand, contains functions that determine how the calculator functions, including using the user inputs to create new predictions with the ML models.

Create a new file directory on your local computer with the title of your project (e.g., 'Tutorial Paper') and place both the server.R and ui.R files within it. You should

^ ORCID: Hyrum S. Eddington, 0000-0003-4508-073X; Amber W. Trickey, 0000-0001-9993-3860; Vaibhavi Shah, 0000-0001-5576-9878; Alex H. S. Harris, 0000-0001-7267-3077.

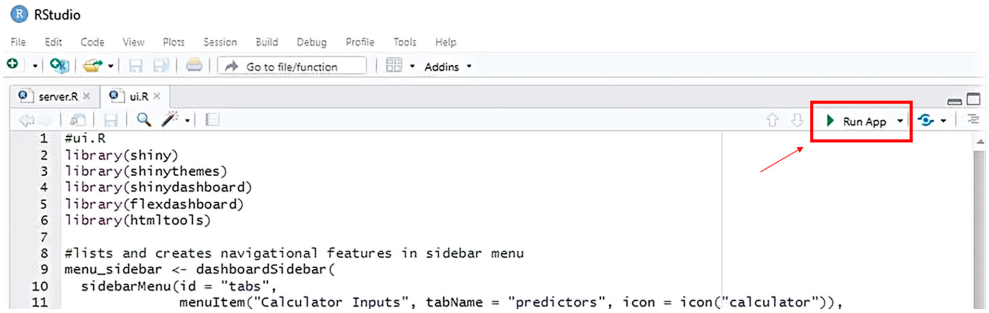

Figure 1 The 'Run' button in RStudio becomes a 'Run App' play button when the server.R and ui.R files are correctly configured.

then paste the code in the example documents into their respective files. While it is possible to construct a Shiny application using just one file with distinct sections for your server and ui code, we highly recommend the two-file approach, especially for calculators with many inputs and complex dashboard schemas. When both files (*must* be titled 'server.R' and 'ui.R') are placed in the same directory and opened in R Studio, in the top righthand corner of the console the user will see a 'Run' icon denoted by a green play button that can be used in either file to start the application (*Figure 1*).

Although a basic app layout is easily achievable with the default functions provided in Shiny, the package *shinydashboard* implements an intuitive framework for creating apps that utilize multiple tabs for inputs, results, and other information. For this tutorial, we will implement a calculator which uses a truncated version of a model from our previous work (16) with only five input variables: age (continuous), sex (M/F), sepsis (None/SIRS/Sepsis/Septic shock), dyspnea (None/Moderate exertion/At rest), and function health status (Independent/Partially dependent/Totally dependent). This calculator is for didactic purposes only and *is not* for clinical use.

Open the ui.R file. Lines 2–6 of ui.R list the packages required for the user interface portion of the application. Running these 5 lines will load the packages if they have already been installed. If not already installed, you can use the code install.packages ("shiny","shinythemes", "shinydashboard", "flexdashboard", "htmltools") before loading the packages.

The Shiny user interface using *shinydashboard* centers around three functions: dashboardSidebar (lines 8–14), dashBoardBody (lines 29–33), and dashBoardPage (lines 35–41). Once lines 8–14 are run, the dashboardSidebar object named "menu_sidebar" will contain the tabs (menuItems) that will be used to organize the app's content. Once lines 29–33 are run, the dashBoardBody object named "body" will contain tabItems (e.g., "predictors" that will be displayed on that tab. Both the menu_sidebar and body are contained within dashboardPage. *Figure 2* illustrates visual aspects associated with these code segments in a live application.

Two additional objects are contained in ui.R: input_box and display_gauge (lines 16–27). These objects contain the code for patient characteristics input fields and the display of the model results in a gauge-style visual which is referenced and integrated into the body of the app (lines 31–32).

The numericInput and selectInput lines in the code [19–23] are associated with the variables to be entered by the user of the app. For categorical variables, the "choices" parameter in the selectInput function specifies the different categories of the variable to be displayed as a drop-down menu. For consistency and simplicity, we will use selectInput for all categorical input variables. As an alternative to selectInput, readers may choose other options for displaying variable inputs, such as checkBoxInput. For a more comprehensive overview on this topic see the "Inputs and outputs" tutorial at https://shiny.rstudio.com/tutorial/ (17).

Step 2: store statistical model objects as .rds files

When you create a prediction model in R, it is contained in a R model object. Prediction models in R can take the form of a simple linear regression or more complex models such as LASSO and boosted regression. This R model object contains information about how the model calculates predictions based on new underlying data, which is then used to make predictions on new datasets. For less complicated models with a small number of inputs, it is reasonable to extract this information, which often takes the form of variable coefficients, from the R model object

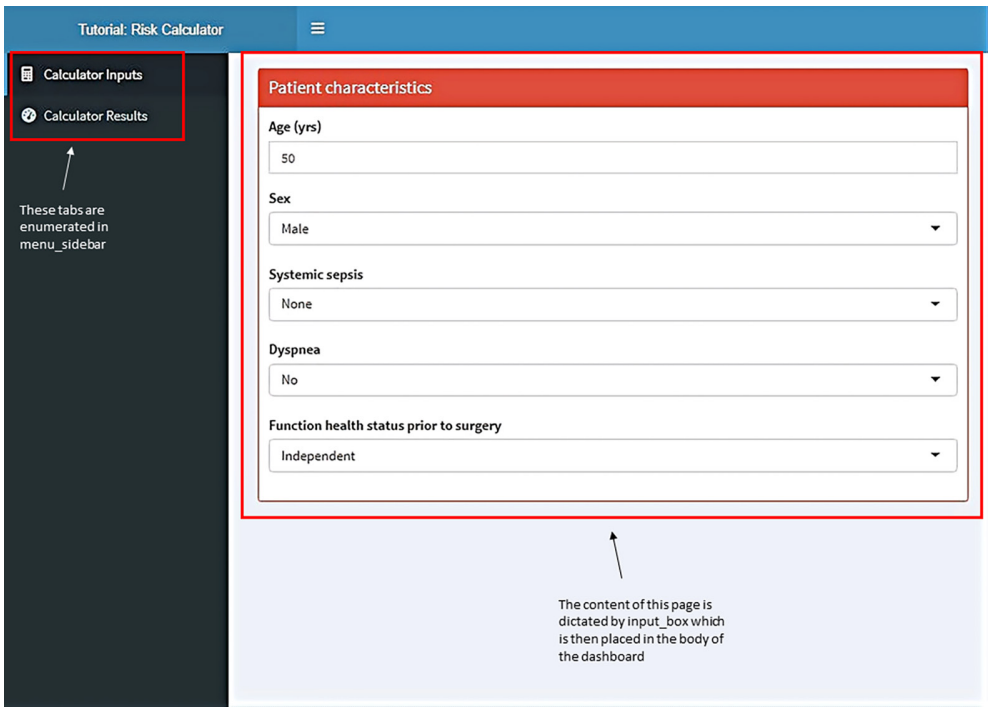

Figure 2 The layout of a Shiny app using shinydashboard is dictated by the sidebar and the content in the body associated with each sidebar tab.

and hard code the formula into the server.R file of the Shiny calculator. However, this quickly becomes unwieldy with many inputs and/or if the underlying model has a complex method for translating inputs into predictions. Using the R model object directly to produce predictions has several advantages over manually copying model coefficients into the Shiny server.R file. First, using the object directly reduces a source of error created when coefficients are copied manually. Second, many complex ML models, such as neural nets and boosted regression, are impossible to render with a simple formula. Therefore, we will demonstrate how to export your models from R and reference them appropriately within the application so they can be deployed with the other project files in Step 4.

After generating an R model object using a function such as lm() or glmnet(), you can save the model to your local computer with the saveRDS() function:

saveRDS(your_model, "your_path/demo_model.rds")

Following this, you should place the R model object file in the same directory as your server.R and ui.R files. For this calculator, you can download the 'tutorial.rds' file at the following link: https://github.com/S-SPIRE/clinical_calculators/blob/main/tutorial_model.rds. This file is then referenced in the following code, and will be deployed with the other application files at a later step:

mort_model <- readRDS('./tutorial_model.rds')

If you are using GitHub (18) to store your project files, it is also possible to reference the GitHub location where your .rds file is stored from within the application (see commented code in server.R lines (9-12). This may be useful in instances where you have many applications that reference the same models. In this case, each application would reference the same repository link, and updating the model at that GitHub location would effectively update each calculator without the need to redeploy each calculator individually (as long as calculator inputs have not changed). However, for most users, placing the .rds file within the project directory where it can be deployed with the other project files is the simplest and most effective way to reference your R model object.

Step 3: design the server.R file to translate user inputs into predictions and display the results

This calculator implements a logistic regression model for 30-day mortality following total hip replacement utilizing

5 variables. While not all methods presented here will be directly applicable to all models, the principles of choosing variables corresponding to user input, and adjusting these variables based on user input, are the same regardless of model type. For additional resources regarding user input and Shiny interfaces see https://shiny.rstudio.com/tutorial/.

The server.R code is mostly contained within the shinyServer function (line 14). The import line [6–7] directly precedes shinyServer such that the model is imported only once and saved.

Model inputs and reactive functions

In order for the application to be dynamic and refresh its displays whenever user input is changed, the logic that determines what values are passed to the R model object must be contained within a reactive{()} function, which in our calculator is inputData().

The model used in this application uses 0/1 encoding for binary variables, so each categorical user input must be translated programmatically to 0 if absent, and 1 if present. The code from lines 19–33 assigns each model input variable an initial value of 0. Following this is code implementing logic to change the value to 1 if the user input indicates the presence of the associated patient characteristic [35–70]. After placing these inputs and the corresponding logic, the variables to be passed to the model will change dynamically based on the selectInput objects contained in ui.R. These variables are then used to create a dataframe called userPredictData (lines 72–79) which the function then returns. Dynamic variable input is enabled by containing the assignment of user input in a 'reactive' function (line 16), whereby the expressions are frequently re-evaluated by the server and values are updated based on new user input.

When constructing this data.frame() code segment for your own app, you must use the original variable names of the model. Remember that because you are giving new user input to your source model, the table this code segment creates must match the format of an entry in the original data from which the source model was created. If done correctly, this code will return a single row dataframe containing the user information used in the generate_model function (lines 88–86) to obtain an accurate prediction. Note that the function inputData is called within generate_model to obtain the relevant patient information. In addition, generate_model also accepts as a parameter the R model object needed to calculate the patient's risk of post-surgical mortality.

Risk gauge visual

Gauges are a useful, compelling visual to display information associated with risk. The R package *flexdashboard* provides a built-out gauge object which can be configured to dynamically display the risk associated with the patient characteristics input by the user. Code for constructing the gauge visual is located on lines 88–92 of server.R, and execution of the visual on lines 26–27 of ui.R.

The call to generate_model within the renderGauge function (line 90) computes the risk prediction from the indicated model, and post-calculation formatting is provided with round(). The "symbol" parameter appends "%" to end of the calculated value. The cut-offs for the gauge itself are dictated through the min, max, and gaugeSector parameters, the latter of which are used to determine the percentage thresholds for which the gauge progresses from green to yellow to red. Note the max for the gauge does not *have* to be set to 100; in fact, in risk models that tend towards smaller percentages (e.g., mortality) it could be advantageous to set the max to a lower level, for example, three times the population mortality occurrence. In our test gauge, however, we have set the max to a default of 100 and set the benchmarks for "warning" and "danger" at 5% and 10%. Upon completion the gauge display tab of the app should appear as in *Figure 3*. The app generated from this code can also be accessed here with the addition of the infoBox visual discussed next (https://s-spire-clintools.shinyapps.io/tutorial/).

Other options to display risk predictions

Dynamic box

One alternative to the gauge visual is a *shinydashboard* box object which displays a single number with formatted text in a simple but engaging manner. Code for this visual is located on lines 94–98 of server.r and line 32 (infoBoxOutput("death_box", width = 6) of ui.r. This will display the predictions for the same mortality model but as an infoBox (*Figure 4*).

The "tags" statement within the value parameter allows the user to adjust the appearance of the display using html. For more information on html tags in Shiny see https://shiny.rstudio.com/articles/tag-glossary.html (19). To explore valid icons for the "icon" parameter see https://fontawesome.com/ (20). Please note that fontawesome is an actively updated collection and that there is a small chance for icons used within your calculator to be altered in name

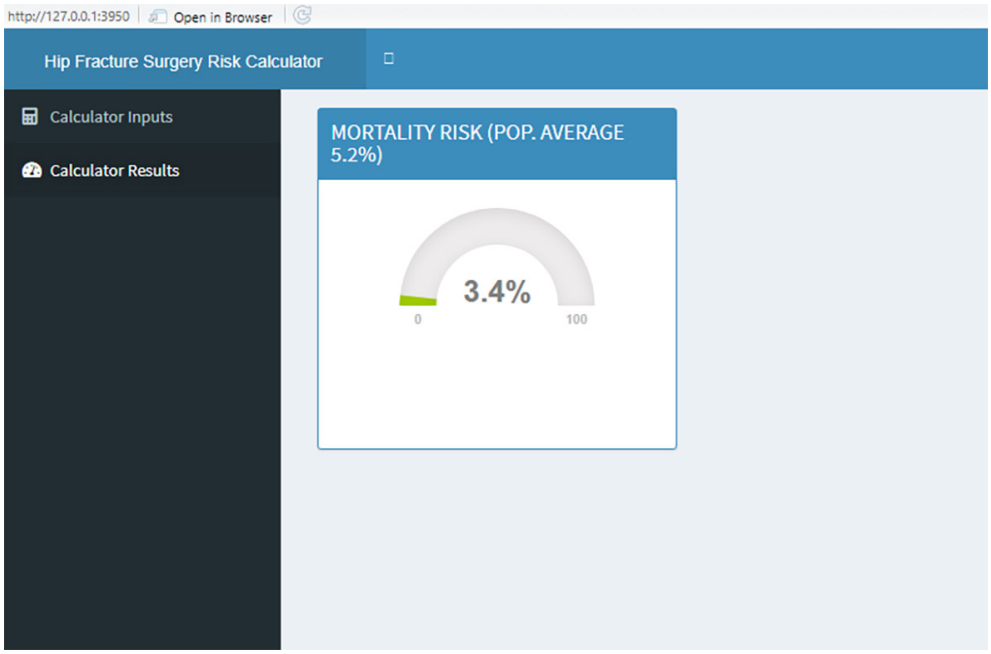

Figure 3 Landing page of tab "Calculator Results" once the app is configured (without optional dynamic box described below).

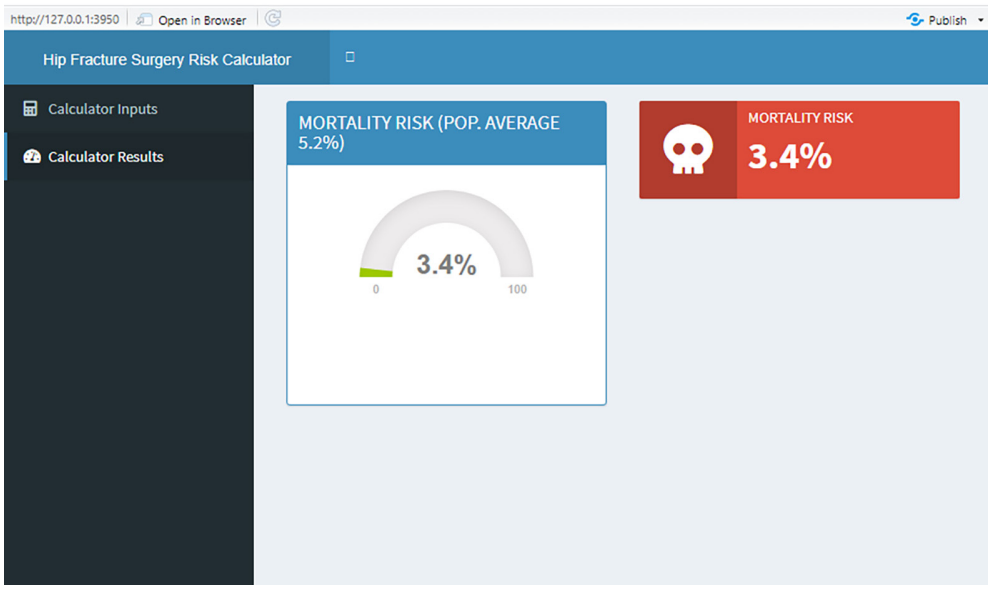

Figure 4 The Shiny dashboard infobox displays the same information as the flexdashboard gauge in a simpler format.

or appearance in future updates. Dynamic info boxes can be especially useful in conveying dashboard information when several models are included and gauge visuals for all of them would be visually overwhelming. An example of a more complicated calculator using a mix of gauges and infoBoxes can be found in our hip fracture prediction model (16) (https://s-spire-clintools.shinyapps.io/hip_deploy/).

Step 4: deployment

When deploying a Shiny application, we recommend hosting the app at shinyapps.io, although other alternatives

are available. In brief, these alternatives include options to host the app on-premises with either open source (Shiny Server) or commercial software (RStudio Connect). While any of these options might suit your individual needs, shinyapps.io is the easiest option for quick deployment, debugging, and user testing. See https://shiny.rstudio.com/deploy/ (21) for more information on app deployment.

Before you deploy, double check to confirm that your .rds files are located *within* your project directory and referenced appropriately in your application per server line 7. To deploy your application to shinyapps.io, you must create a shinyapps.io account (https://www.shinyapps.io/) and install the *rsconnect* package in RStudio. To link *rsconnect* and your shinyapps.io account, use the following function: rsconnect::setAccountInfo(name="<ACCOUNT>", token="<TOKEN>", secret="<SECRET>"). The token and secret are obtained from your shinyapps.io account by accessing Profile->Tokens. Once this is done, you can deploy your application by setting your working directory using setwd() to the directory your server.R and ui.R files are located and running the deployApp() function from *rsconnect* in the RStudio console. For more detailed instructions on shinyapps.io deployment see https://docs.rstudio.com/shinyapps.io/index.html (22).

Discussion

The principles outlined in this work are derived from our own experience in orthopedic procedures but can also be applied broadly to other practices. If readers desire an additional exercise beyond orthopedic surgery, we recommend using the 'heart' dataset from R's kmed package (23) to create a model of heart disease using chest pain, age, ECG, and gender as predictor variables. Using the steps outlined in this paper, these predictor variables would then map to calculator inputs, and the same methods used in this hip fracture calculator could be applied to visualize predicted risk of heart disease.

Conclusions

As risk calculators become increasingly more popular as decision tools, attention must be given to not only creating clinically accurate models, but also designing tools for interfacing with these models that are both professional and accessible for physicians and patients. By using the variety of tools that Shiny has to offer, including accessory packages such as *shinydashboard* and *flexdashboard* offer, user interfaces with these models can become compelling aids to clinical decision making.

Acknowledgments

Funding: This work was funded in part by a grant from the VA HSR&D Service (RCS14-232; AHSH) and support from the Stanford–Surgical Policy Improvement Research and Education Center (S-SPIRE).

Footnote

Peer Review File: Available at https://atm.amegroups.com/article/view/10.21037/atm-22-847/prf

Conflicts of Interest: All authors have completed the ICMJE uniform disclosure form (available at https://atm.amegroups.com/article/view/10.21037/atm-22-847/coif). The authors have no conflicts of interest to declare.

Ethical Statement: The authors are accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved.

References

1. Ekiz HA, Conley CJ, Stephens WZ, et al. CIPR: a web-based R/shiny app and R package to annotate cell clusters in single cell RNA sequencing experiments. BMC Bioinformatics 2020;21:191.
2. Danger R, Moiteaux Q, Feseha Y, et al. FaDA: A web application for regular laboratory data analyses. PLoS One 2021;16:e0261083.
3. Ge SX, Son EW, Yao R. iDEP: an integrated web application for differential expression and pathway analysis of RNA-Seq data. BMC Bioinformatics 2018;19:534.
4. Jain S, Tumkur KR, Kuo TT, et al. Weakly supervised

learning of biomedical information extraction from curated data. BMC Bioinformatics 2016;17 Suppl 1:1.
5. Samir J, Rizzetto S, Gupta M, et al. Exploring and analysing single cell multi-omics data with VDJView. BMC Med Genomics 2020;13:29.
6. Shipe ME, Deppen SA, Farjah F, et al. Developing prediction models for clinical use using logistic regression: an overview. J Thorac Dis 2019;11:S574-84.
7. Goff DC, Lloyd-Jones DM, Bennett G, et al. 2013 ACC/AHA Guideline on the Assessment of Cardiovascular Risk. Circulation 2014;129:S49-73.
8. Collins GS, Reitsma JB, Altman DG, et al. Transparent reporting of a multivariable prediction model for individual prognosis or diagnosis (TRIPOD): the TRIPOD statement. BMJ 2015;350:g7594.
9. Chang W, Ribeiro BB, RStudio, Bootstrap) AS (AdminLTE theme for, font) ASI (Source SP. shinydashboard: Create Dashboards with "Shiny" [Internet]. 2021 [cited 2022 Feb 16]. Available online: https://CRAN.R-project.org/package=shinydashboard
10. Iannone R, Allaire JJ, Borges B, RStudio, CSS) KI (Dashboard, CSS) AA (Dashboard, et al. flexdashboard: R Markdown Format for Flexible Dashboards [Internet]. 2020 [cited 2022 Feb 16]. Available online: https://CRAN.R-project.org/package=flexdashboard
11. Chang W, Cheng J, Allaire JJ, Sievert C, Schloerke B, Xie Y, et al. shiny: Web Application Framework for R [Internet]. 2021 [cited 2022 Feb 15]. Available online: https://CRAN.R-project.org/package=shiny
12. Ji X, Kattan MW. Tutorial: development of an online risk calculator platform. Ann Transl Med 2018;6:46.
13. R Development Core Team. R: A language and environment for statistical computing. R Found Stat Comput Vienna Austria. 2018.
14. RStudio Team. RStudio: Integrated Development Environment for R. RStudio Inc. 2016.
15. Training [Internet]. [cited 2022 Feb 15]. Available online: https://www.rstudio.com/resources/training/
16. Harris AHS, Trickey AW, Eddington HS, et al. A Tool to Estimate Risk of 30-day Mortality and Complications After Hip Fracture Surgery: Accurate Enough for Some but Not All Purposes? A Study From the ACS-NSQIP Database. Clin Orthop Relat Res 2022. [Epub ahead of print]. doi: 10.1097/CORR.0000000000002294.
17. Shiny - Tutorial [Internet]. [cited 2022 Feb 15]. Available online: https://shiny.rstudio.com/tutorial/
18. GitHub. Home [Internet]. GitHub Resources. [cited 2022 Feb 15]. Available online: https://resources.github.com/
19. Shiny - Shiny HTML Tags Glossary [Internet]. [cited 2022 Feb 15]. Available online: https://shiny.rstudio.com/articles/tag-glossary.html
20. Font Awesome [Internet]. [cited 2022 Feb 15]. Available online: https://fontawesome.com
21. Shiny - Hosting and deployment [Internet]. [cited 2022 Feb 15]. Available online: https://shiny.rstudio.com/deploy/
22. Team shinyapps io. shinyapps.io user guide [Internet]. [cited 2022 Feb 15]. Available online: https://docs.rstudio.com/shinyapps.io/index.html
23. Budiaji W. kmed: Distance-Based k-Medoids [Internet]. 2021 [cited 2022 Jun 2]. Available online: https://CRAN.R-project.org/package=kmed

Cite this article as: Eddington HS, Trickey AW, Shah V, Harris AHS. Tutorial: implementing and visualizing machine learning (ML) clinical prediction models into web-accessible calculators using Shiny R. Ann Transl Med 2022;10(24):1414. doi: 10.21037/atm-22-847

Supplementary

Appendix 1

```
#Supplemental File 1: Shiny R file 'server.R'
library(shiny)
library(shinydashboard)
library(glmnet)

#load model .rds packaged with shiny app deployment
mort_model <- readRDS('./tutorial_model.rds')

#alternate .rds file storage option; upload to github repository of choice and reference the download
link as follows:
#mort_model <- readRDS(gzcon(url("https://github.com/S-
SPIRE/clinical_calculators/raw/main/tutorial_model.rds")))

shinyServer(function(input, output, session){
 #reactive function assigns user input to variable table used in predictions
 inputData <- reactive({
  female <- 0

  age_74 <- 0
  age_79 <- 0
  age_84 <- 0
  age_89 <- 0
  age_90 <- 0

  fnstatus_partd <- 0
  fnstatus_totd <- 0

  dyspnea_atrest <- 0
  dyspnea_modexe <- 0

  prsepis_sirs <- 0
  prsepis_sepsis <- 0
  prsepis_shock <- 0

  if (input$Sex == "Female"){
   female <- 1
  }


  if (input$Age < 75 & input$Age >= 74){
   age_74 <- 1
  } else if (input$Age < 80 & input$Age >= 75){
   age_79 <- 1
  } else if (input$Age < 85 & input$Age >= 80){
   age_84 <- 1
  } else if (input$Age < 90 & input$Age >= 85){
   age_89 <- 1
  } else if (input$Age >= 90){
```


https://dx.doi.org/10.21037/atm-22-847

```
    age_90 <- 1
   }

   if (input$healthStatus == "Partially Dependent") {
    fnstatus_partd <- 1
   } else if (input$healthStatus == "Totally Dependent") {
    fnstatus_totd <- 1
   }

   if (input$Dyspnea == "At rest") {
    dyspnea_atrest <- 1
   } else if (input$Dyspnea == "Moderate exertion") {
    dyspnea_modexe <- 1
   }

   if (input$Sepsis == "SIRS") {
    prsepis_sirs <- 1
   } else if (input$Sepsis == "Sepsis") {
    prsepis_sepsis <- 1
   } else if (input$Sepsis == "Septic Shock") {
    prsepis_shock <- 1
   }

   #function returns a single row data frame containing variables needed for model prediction
   #variable names in this data frame must match those of the original dataset used to train the model
   userPredictData <- data.frame("female" = female, "age_74" = age_74, "age_79" = age_79, "age_84" =
age_84, "age_89" = age_89, "age_90" = age_90, "fnstatus_partd" = fnstatus_partd, "fnstatus_totd" =
fnstatus_totd, "dyspnea_atrest" = dyspnea_atrest, "dyspnea_modexe" = dyspnea_modexe,
"prsepis_sirs" = prsepis_sirs, "prsepis_sepsis" = prsepis_sepsis, "prsepis_shock" = prsepis_shock)
  })

  #function takes an as argument the imported model object and returns the model prediction based on
data returned by inputData()
  generate_model <- function(m){
   dat <- as.matrix(inputData())
   p <- predict(m, newx = dat, type = "response", s = "lambda.1se")
   p
  }

  #render gauge with predicted value for display in ui.R
  output$mortality_gauge <-
flexdashboard::renderGauge(flexdashboard::gauge(round(generate_model(mort_model)*100, 1),
symbol = "%",min = 0, max = 100, sectors = gaugeSectors(success = c(0, 5), warning = c(5, 5*2), danger =
c(5*2, 100)), abbreviate = TRUE, abbreviateDecimals = 0))

  #render display box with predicted value for display in ui.R
```


https://dx.doi.org/10.21037/atm-22-847

```
  output$death_box <- renderInfoBox({infoBox("Mortality Risk", value = tags$p(style = "font-size:
35px;",paste0(round(generate_model(mort_model)*100, 1), "%")),  icon = icon("skull"), color = "red", fill
= TRUE)})
})
```


https://dx.doi.org/10.21037/atm-22-847

Appendix 2

```
#Supplementary File 2: Shiny R file 'ui.R'
library(shiny)
library(shinythemes)
library(shinydashboard)
library(flexdashboard)
library(htmltools)

#lists and creates navigational features in sidebar menu
menu_sidebar <- dashboardSidebar(
  sidebarMenu(id = "tabs",
          menuItem("Calculator Inputs", tabName = "predictors", icon = icon("calculator")),
          menuItem("Calculator Results", tabName = "risk", icon = icon("gauge-high"))
  )
)

#input_box and display_gauge contain application features organized into task-relevant components
input_box <- box(title ="Patient characteristics", status = "danger", header = T, solidHeader = T,  width =
12,
             numericInput("Age", "Age (yrs)", 50, min = 0, max = 120),
             selectInput("Sex", "Sex", choices = c("Male", "Female")),
             selectInput("Sepsis", "Systemic sepsis", choices = c("None", "SIRS", "Sepsis", "Septic Shock")),
             selectInput("Dyspnea", "Dyspnea", choices = c("No", "Moderate exertion", "At rest")),
             selectInput("healthStatus", "Function health status prior to surgery", choices =
c("Independent", "Partially Dependent", "Totally Dependent")))

display_gauge <- box(title = "MORTALITY RISK", align = "center", status = "primary", solidHeader = TRUE,
flexdashboard::gaugeOutput("mortality_gauge"))

#body takes the components created in previous items and assigns them to proper tab windows
body <- dashboardBody(tabItems(
  tabItem(tabName = "predictors", input_box),
  tabItem(tabName = "risk", display_gauge, infoBoxOutput("death_box", width = 6))
))

#dashboardPage combines all application content including the sidebar and body into one coherent UI
dashboardPage(
  dashboardHeader(title = span("Tutorial: Risk Calculator", style = "font-size:14px"),  titleWidth = 300),

  menu_sidebar,
  body
)
```


https://dx.doi.org/10.21037/atm-22-847