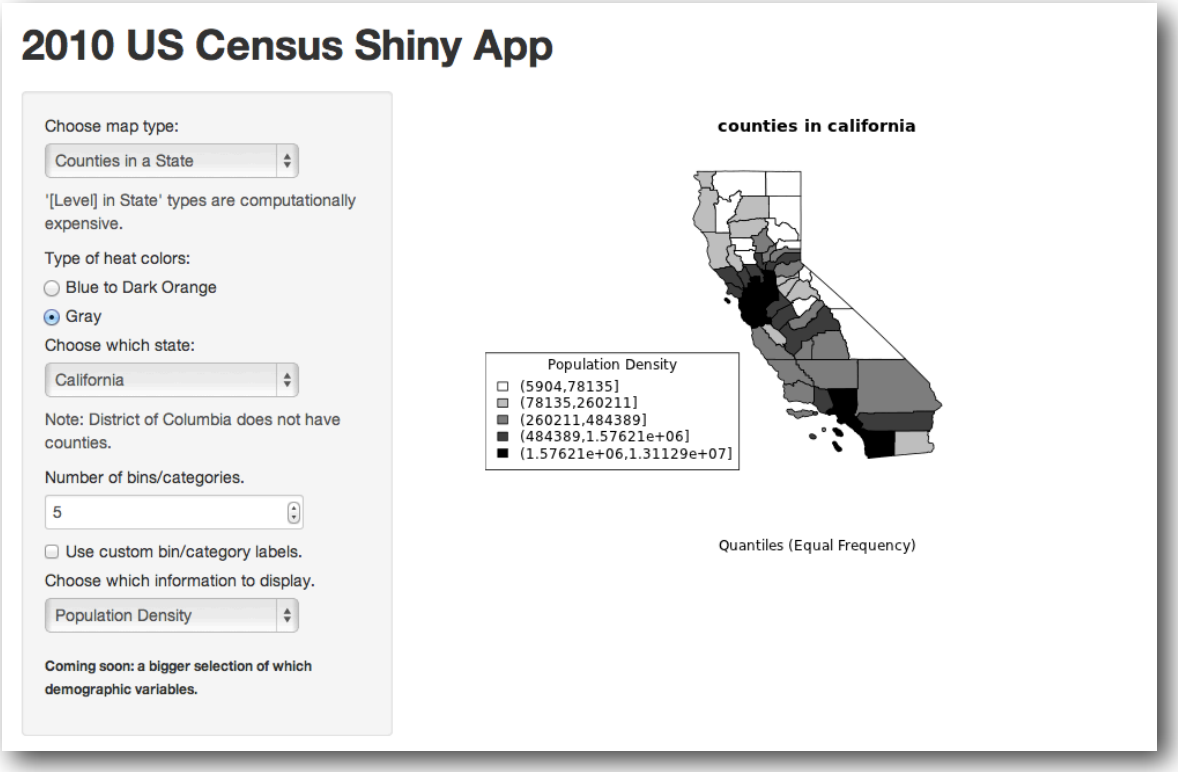
> What is Shiny?

Shiny is an R package developed by the RStudio team that lets you develop an interactive web application in R. You do this with two files:

- ui.R (where you define the user interface)
- server.R (where you define the server logic)



Demo: shiny.mpopov.com:3838/census **Full code**: github.com/bearloga/2010-US-Census-Shiny-App

> Building a Shiny app

When starting the development of a Shiny app, it is useful to break down the problem as

What we want to do
Which informs us about
What our UI elements should be
Which informs us about
What our server logic must do
Which informs us about
What our UI elements should be
Which informs us about...
What our server logic must do
Which informs us...

Install: install.packages("shiny",repos="http://cran.rstudio.com/")

Tutorial: http://rstudio.github.io/shiny/tutorial/

Demos: shiny::runExample(example="name")
Valid examples are "01_hello", "02_text", "03_reactivity", "04_mpg", "05_sliders", "06_tabsets", "07_widgets", "08_html", "09_upload", "10_download", "11_timer"

User Interface - ui.R	Server Logic - server.R

`library(shiny)`
 	
`shinyUI(pageWithSidebar(`
`  headerPanel("Application Title"),`
`  sidebarPanel(`
`  # Input UI elements go here.`
`  ),`
`  mainPanel(`
`  # Output UI elements go here.`
`  )`
`))`
	`library(shiny)`
	
	`shinyServer(function(input,output){`
	`  # Server logic goes here.`
	`})`

Run: shiny::runApp("path/to/folder/with/server-and-ui-R-files")

Sidebar Panel Input UI Elements (Some)	
numericInput	textInput
sliderInput	checkboxGroupInput, checkboxInput
selectInput	submitButton

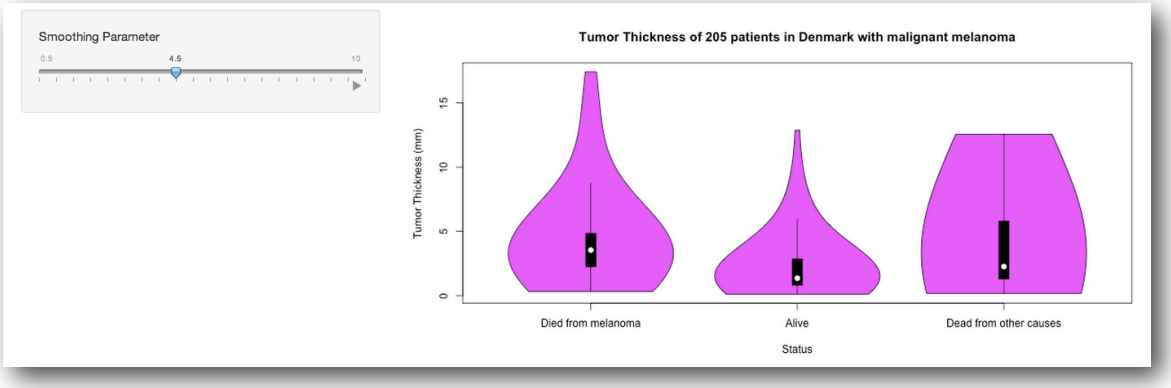
Sidebar Panel Miscellaneous UI Elements (Some)	
helpText	HTML Tags: code, h1–h6, p, strong, div...

Main Panel Output UI Elements (Some)	**Server Output (Some)**
textOutput, verbatimTextOutput	renderText
plotOutput	renderPlot
tableOutput	renderTable
uiOutput	renderUI
tabsetPanel, tabPanel	

Malignant Melanoma Violin Plots

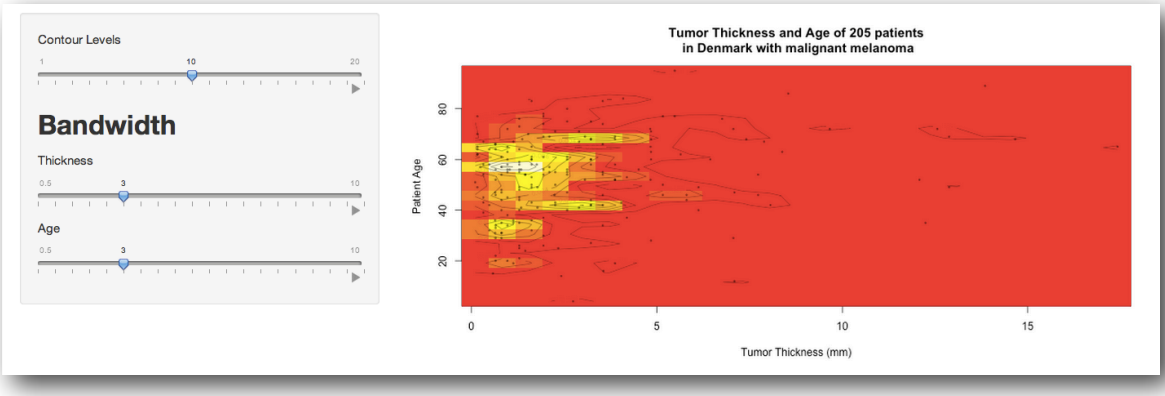
wvioplot has been removed from CRAN. Download: http://cran.r-project.org/src/contrib/Archive/wvioplot/

```
shiny::runApp("lectureapps/violin")
```



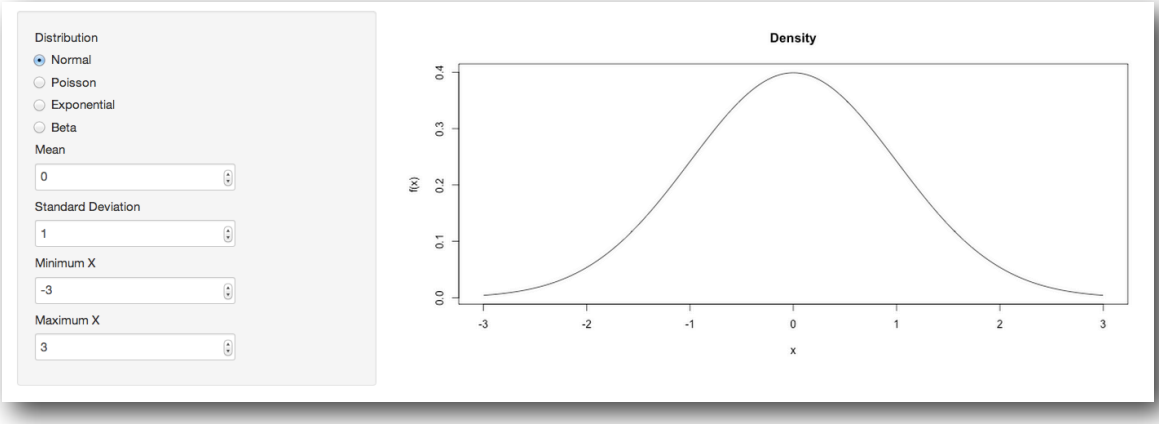
Visualizing Density with Heat Map and Contour Plot

```
shiny::runApp("lectureapps/density")
```



Distribution Visualizer

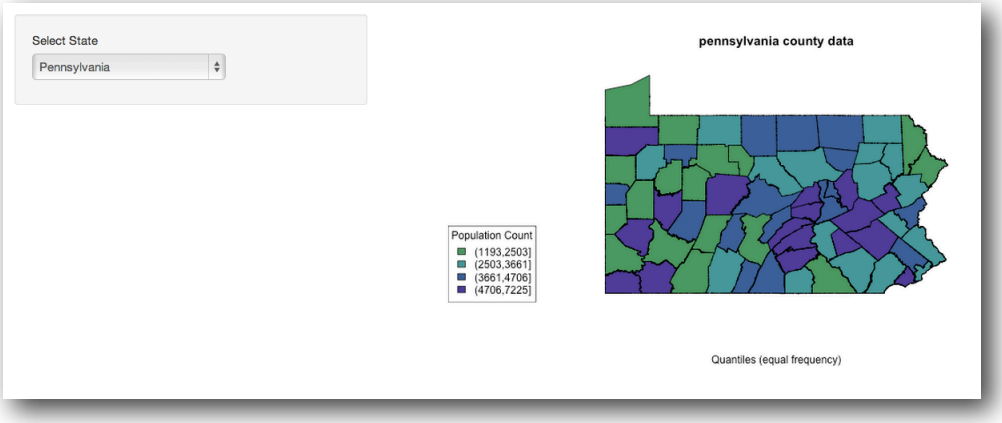
```
shiny::runApp("lectureapps/random")
```



2010 US Census App Example: State Selection

Requires: UScensus2010 (on CRAN), UScensus2010county (not on CRAN)

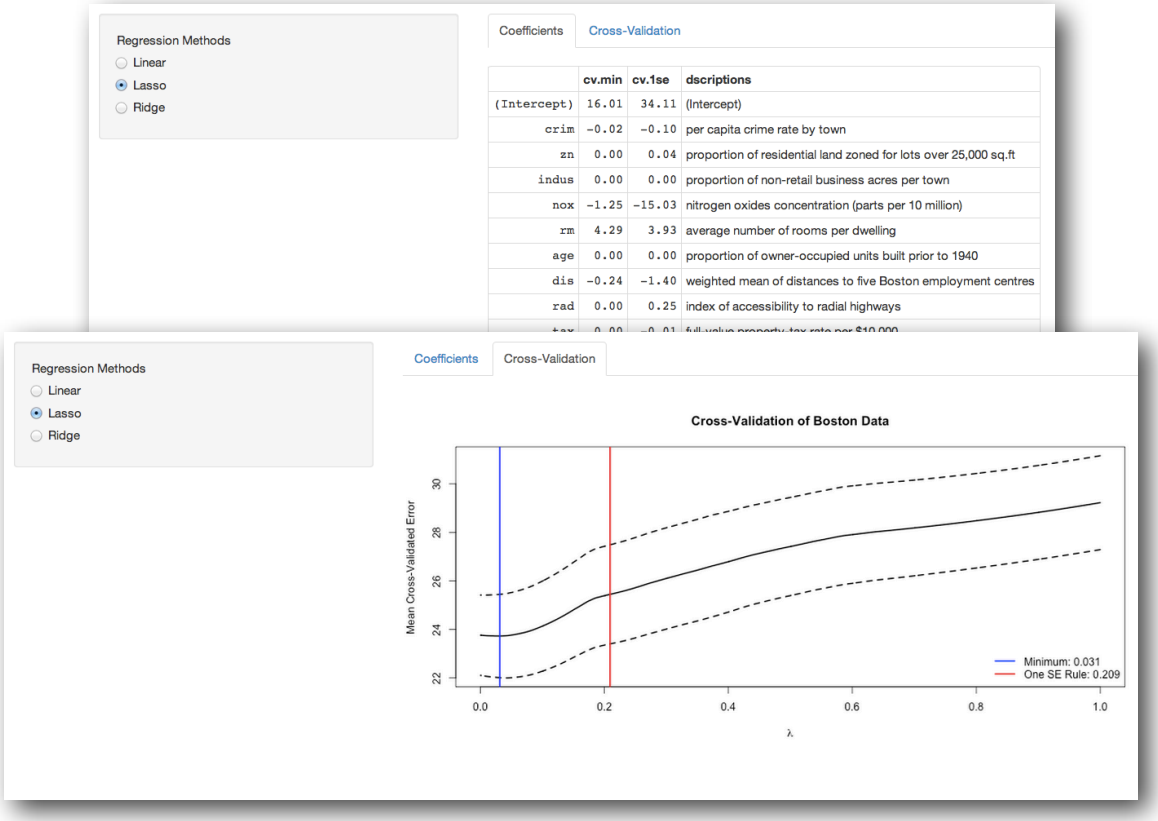
```
shiny::runApp("lectureapps/counties")
```



Median value of owner-occupied homes in Boston in $1000s

Requires: glmnet package (on CRAN)

```
shiny::runApp("lectureapps/cv")
```



Example 1: Malignant Melanoma Violin Plots

ui.R

```
library(shiny)
shinyUI(pageWithSidebar(
  headerPanel("Malignant Melanoma Violin Plots"),
  sidebarPanel(
```

Slider Input is the most intuitive way to tune the smoothing parameter.

```
    sliderInput("adjust",label="Smoothing Parameter",
      min=0.5, max=10, value=3, step=0.5,animate=T)
  ),
  mainPanel(plotOutput("violins"))
))
```

server.R

```
library(shiny); library(MASS); install.packages("Hmisc")

download.file(url="http://cran.r-project.org/src/contrib/Archive/
wvioplot/wvioplot_0.1.tar.gz", destfile="wvioplot_0.1.tar.gz")

install.packages("wvioplot_0.1.tar.gz", repos = NULL, type = "source")
unlink("wvioplot_0.1.tar.gz") # Delete the downloaded file.
library(wvioplot)
thickness <- Melanoma$thickness # tumour thickness in mm
status <- Melanoma$status
shinyServer(function(input,output){
```

Our only output is the 'violins' plot which we want to be reactive to the 'adjust' input. Therefore we use renderPlot and create the violin plot with the appropriate titles inside.

```
  output$violins <- renderPlot({
    wvioplot(thickness[status==1], thickness[status==2],
thickness[status==3], names=c('Died from melanoma','Alive','Dead from
other causes'), col="magenta",

    adjust=input$adjust) # Here we pull the value from the adjust sliderInput

    title(main='Tumor Thickness of 205 patients in Denmark with
malignant melanoma',ylab="Tumor Thickness (mm)",xlab="Status")

  })
})
```

Example 2: Visualizing Density with Heat Map and Contour Plot

ui.R

```
library(shiny)
shinyUI(pageWithSidebar(
  headerPanel("Visualizing Density with Heat Map and Contour Plot"),
  sidebarPanel(
```

Alternatively, we could have chosen to use numericInput since that lets us input any number between the minimum and maximum allowed values. However, we don't want the user to be able to type in non-whole number values and the animate=T feature of sliderInput is very useful in this application.

```
    sliderInput("nlevels",label="Contour Levels",
          min=1,max=20,value=10,step=1,animate=T),
    h2("Bandwidth"),
```

Slider Input is the most intuitive way to tune the smoothing parameter. We need to remember that X refers to Thickness and Y refers to Age (refer to server.R) and not mix them up.

```
    sliderInput("adjustX",label="Thickness",
          min=0.5,max=10,value=3,step=0.5,animate=T),
    sliderInput("adjustY",label="Age",
          min=0.5,max=10,value=3,step=0.5,animate=T)
  ),
  mainPanel(
    plotOutput("density")
  )
))
```

server.R

```
library(shiny); library(MASS)
thickness <- Melanoma$thickness
age <- Melanoma$age
shinyServer(function(input,output){
```

Our only output is the 'density' plot which we want to be reactive to the 'adjustX,' 'adjustY,' and 'nlevels' inputs. Therefore we use renderPlot and create the heat map and the contour plots with the appropriate titles inside:

```
  output$density <- renderPlot({
```

Estimate the two-dimensional kernel density using the values of the adjustX and adjustY slider inputs:

```
    kde <- kde2d(thickness,age,h=c(input$adjustX,input$adjustY))
    image(kde)
```

We add the titles separately:

```
    title(main="Tumor Thickness and Age of 205 patients\n in Denmark
with malignant melanoma",ylab="Patient Age",xlab="Tumor Thickness
(mm)")
```

We allowed the user to specify the number of levels of the contour plot. So we pull the value from the nlevels sliderInput. We also want to plot a 50% transparent contour plot instead of completely opaque one.

```
    contour(kde,add=T,lwd=1,col=rgb(0,0,0,0.5),nlevels=input$nlevels)
```

We want to see the heatmap and the contour overlay, so we plot 50% transparent points instead of 100% opaque ones. We also decrease their size.

```
    points(thickness,age,pch=16,cex=0.5,col=rgb(0,0,0,0.5))
  })
})
```

Example of Dynamic (Conditional) Panels: Distribution Visualizer

ui.R

```
library(shiny)
dists <- list("Normal"="norm","Poisson"="pois",
                "Exponential"="exp","Beta"="beta")
shinyUI(pageWithSidebar(
  headerPanel("Distribution Visualizer"),
  sidebarPanel(
    # Input UI elements go here.
    radioButtons("distribution",label="Distribution",choices=dists),
    conditionalPanel(
      condition = "input.distribution == 'norm'",
      numericInput("mu","Mean",value=0),
      numericInput("sigma","Standard Deviation",min=0.01,value=1)
    ),
    conditionalPanel(
      condition = "input.distribution == 'pois'",
      numericInput("lambda","Lambda",min=1,value=10)
    ),
    conditionalPanel(
      condition = "input.distribution == 'exp'",
      numericInput("rate","Rate",min=1,value=10)
    ),
    conditionalPanel(
      condition = "input.distribution == 'beta'",
      sliderInput("alpha","Alpha",min=0.1,value=1,
                  step=0.1,max=5,animate=T),
      sliderInput("beta","Beta",min=0.1,value=1,
                  step=0.1,max=5,animate=T)
    ),
    conditionalPanel(
      condition = "input.distribution != 'beta'",
      conditionalPanel(
        condition = "input.distribution == 'norm'",
        numericInput("min","Minimum X",value=-3)
      ),
      numericInput("max","Maximum X",value=3)
    )
  ),
  mainPanel(
    # Output UI elements go here.
    plotOutput("density")
  )
))
```

server.R

```
library(shiny)

shinyServer(function(input,output){

  output$density <- renderPlot({

    plotType <- "l"

    if ( input$distribution == "norm" ) {
      x <- seq(input$min,input$max,length.out=1000)
      y <- dnorm(x,input$mu,input$sigma)
    }

    if ( input$distribution == "pois" ) {
      x <- 0:round(input$max)
      y <- dpois(x,input$lambda)
      plotType <- "h" # since discrete, not continuous
    }

    if ( input$distribution == "exp" ) {
      x <- seq(0,input$max,length.out=1000)
      y <- dexp(x,input$rate)
    }

    if ( input$distribution == "beta" ) {
      x <- seq(0,1,0.001)
      y <- dbeta(x,input$alpha,input$beta)
    }

    plot(x,y,type=plotType,main="Density",xlab="x",ylab="f(x)")

  })

})
```

Example: 2010 US Census App Example: State Selection

ui.R

```
library(shiny)
library(UScensus2010) # If not installed: install.packages("UScensus2010")

data(states.names)
data(states.names.cap)
```

Here we create a list out of the state names in UScensus2010 package and then use the capitalized versions to name those values. The capitalized names will show up inside the drop-down menu while the lowercase names will be the actual values we pass to the server script.

```
states.list = as.list(states.names)
names(states.list) <- states.names.cap

shinyUI(pageWithSidebar(
  headerPanel("2010 US Census App Example: State Selection"),
  sidebarPanel(

  # selectInput creates a drop-down menu for selecting from a list of values

    selectInput("state",label="Select
State",choices=states.list,selected="Pennsylvania")
  ),
  mainPanel(
    plotOutput("choropleth")
  )
))
```

Example: 2010 US Census App Example: State Selection

server.R

```
library(shiny)
library(UScensus2010) # If not installed: install.packages("UScensus2010")
```

server.R uses UScensus2010county which is NOT on CRAN.
If not installed, run: `install.county("windows")` or `"osx"` or `"linux"`

```
library(UScensus2010county)
```

The R code in this server script features some very advanced concepts. Take your time to read through the code step-by-step and the accompanying comments to understand what exactly the code is doing and why I did what I did.

```
shinyServer(function(input,output){
```

The only output is the 'choropleth' plot which we want to be reactive to the 'state' selectInput, so we use a renderPlot.

```
output$choropleth <- renderPlot({
```

Whenever we select a different state, that state's data needs to be loaded in. However, the previously selected state's data remains in memory. Although we're only dealing with the data at county level (as opposed to the heavier tract and block group levels), we still don't want to end up with 49 unnecessary objects in memory. So the first thing we want to do is clean up and get rid of the previously loaded data. Unfortunately, we don't actually know the name of the object. This presents a small complication but nothing we can't solve.

```
loaded = unname(sapply(ls(envir=.GlobalEnv),function(x){
  substr(x,nchar(x)-8,nchar(x)) == '.county10'
}))
if ( sum(loaded) > 0 )
   do.call(rm,args=list(ls(envir=.GlobalEnv)[loaded]),envir=.GlobalEnv)
rm(loaded)
```

We use sapply to cycle through the elements returned to us by calling ls(envir=.GlobalEnv) function. Recall that ls() returns a vector of names of objects in memory. We're only interested in the one that ends with '.county10,' so we use substr() and nchar() functions to check if the last 9 characters of each object name are '.county10' and return either TRUE or FALSE.

Oh the joys of dynamically loading in data sets with variable names... If we call rm(loaded) directly then we'll just delete the thing we just created. That's why we have

to use do.call which "constructs and executes a function call." In other words, if ls() [loaded] = "pennsylvania.county10" then do.call(rm,args=list(ls()[loaded])) will write:

> rm(pennsylvania.county10)

and then run it! Then we get rid of loaded.
Next, we need to dynamically load the data for the chosen state. This is done by:
1) constructing a character vector of the form 'data(statename.county10)' via the paste() function
2) passing it to parse() function which returns an unevaluated expression
3) using eval() to evaulate it.
loaded.data will be the character vector "statename.county10"

```
loaded.data <- eval(parse(text=paste("data(",input
$state,".county10)",sep="")))
```

Since we don't know the name of the SpatialPolygonsDataFrame we loaded, we need to use the do.call function again to construct our choropleth command dynamically. Recall that loaded.date is the name of our dataset and recall that if we just type the name of an object and hit ENTER then we'll see the printout of the object. Sooooo...we can use parse() and eval() again to pass the SPDF object to the sp argument of the choropleth function:

```
do.call(choropleth,args=list(
  sp=eval(parse(text=loaded.data))
))
title(main=paste(input$state,"county data")) # constructs the title

  })
})
```

Example: Median value of owner-occupied homes in Boston in $1000s

(Just pure source code without comments.)

ui.R

```
library(shiny)

shinyUI(pageWithSidebar(
  headerPanel("Median value of owner-occupied homes in Boston in $1000s"),
  sidebarPanel(
    radioButtons("method","Regression Methods",
                 choices=list("Linear"="lm","Lasso"="lasso", "Ridge"="ridge"),
                       selected="Linear")
  ),
  mainPanel(
    tabsetPanel(
      tabPanel("Coefficients",tableOutput("coef")),
      tabPanel("Cross-Validation",plotOutput("cv"))
    )
  )
))
```

server.R

```
library(shiny)

data(Boston,package="MASS")

library(glmnet)
library(boot)
```

We'll use this later:

```
descriptions = as.character(c(
  "(Intercept)",
  "per capita crime rate by town",
  "proportion of residential land zoned for lots over 25,000 sq.ft",
  "proportion of non-retail business acres per town",
  "nitrogen oxides concentration (parts per 10 million)",
  "average number of rooms per dwelling",
  "proportion of owner-occupied units built prior to 1940",
  "weighted mean of distances to five Boston employment centres",
  "index of accessibility to radial highways",
  "full-value property-tax rate per $10,000",
  "pupil-teacher ratio by town",
  "1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town",
  "lower status of the population (percent)"))

fit <- glm(medv ~ .,data=Boston[,-4])

cv <- cv.glmnet(x=as.matrix(Boston[,-c(4,14)]),y=Boston[,14],
        lambda=seq(0,1,0.001),nfolds=10)
```

```
shinyServer(function(input,output){

  output$coef <- renderTable({
    if (input$method=="lm") {
      foo = as.data.frame(as.matrix(coef(fit)))
      names(foo) <- "est"
      foo$dscriptions <- descriptions
      foo
    } else {
      temp = glmnet(x=as.matrix(Boston[,-c(4,14)]),y=Boston[,14],
           family="gaussian",
           alpha=as.numeric(input$method=="lasso"),
           nlambda=2,standardize=T,
           lambda=c(cv$lambda.min,cv$lambda.1se))
      foo = as.data.frame(as.matrix(coef(temp)))
      names(foo) <- c("cv.min","cv.1se")
      foo$dscriptions <- descriptions
      foo
    }
  })

  output$cv <- renderPlot({
    if (input$method=="lm") {
      temp5 = cv.glm(Boston,fit,K=5)$delta
      temp10 = cv.glm(Boston,fit,K=10)$delta
      plot(x=1:2,y=c(temp5[1],temp10[1]),
         ylim=range(c(temp5,temp10)),
         xlim=c(0.5,2.5),xaxt="n",
         ylab="Cross-Validation Error Estimate",col="red",
         main="Cross-Validation of Boston Data",pch=16,
         xlab="Number of Folds")
      points(x=1:2,y=c(temp5[2],temp10[2]),pch=16,col="blue")
      axis(1,at=1:2,labels=c("K = 5","K = 10"))
      legend("bottomleft",c("Raw","Adjusted"),pch=16,
col=c("red","blue"),bty="n")
    } else {
      plot(x=cv$lambda,y=cv$cvm,type="l",
         ylim=range(c(cv$cvup,cv$cvlo)),
         main="Cross-Validation of Boston Data",
         xlab=expression(lambda),
         ylab="Mean Cross-Validated Error",lwd=2)
      lines(x=cv$lambda,y=cv$cvup,lty="dashed",lwd=2)
      lines(x=cv$lambda,y=cv$cvlo,lty="dashed",lwd=2)
      abline(v=cv$lambda.min,col="blue",lwd=2)
      abline(v=cv$lambda.1se,col="red",lwd=2)
      legend("bottomright",lty="solid",col=c("blue","red"),bty="n",lwd=2,
               c(paste("Minimum:",cv$lambda.min),
                 paste("One SE Rule:",cv$lambda.1se)))
    }
  })

})
```