Building another Machine Learning Web App

- This Web App aims to build an Iris Predictor which deploys a machine learning model of the Iris dataset using again the Shiny Web App framework package
- **FOCUS:** how we can develop an Iris predictor using a ML model in the background
    - The web app allows the user to select the input values for the 4 input parameters
    - Similar to the previous app, I will create another "*Submit*" button to make a prediction

Iris Dataset

- Multivariate dataset
- This dataset contains 3 classes of 50 instances each, where each class refers to a type of Iris plant
    - One class is linearly separable from the other 2; the latter are NOT separable from each other
- Predicted attribute: class of Iris plant
- Attribute information:
    - 1. Sepal Length in cm
    - 2. Sepal Width in cm
    - 3. Petal Length in cm
    - 4. Petal Width in cm
    - 5. Class:
        - > Iris Setosa
        - > Iris Versicolour
        - > Iris Virginica

File: *model.R*

- This file creates the model and other csv files that will be part of the dataset used by the web app itself
- This code pre-builds the Random Forest Model
    - The '*app-numeric*' file loads it in
- **RECALL:** Initially deploy your predictive model in an RDS file
- **Want:** Develop the model in this file and save it as the RDS file,
    - Deploying this, and read it in using the following code (this is within the '*app-numeric'* file):
    - **model <- readRDS("model.rds")**

```
# Performs stratified random split of the data set
# This helps organize the dataset
TrainingIndex <- createDataPartition(iris$Species, p=0.8, list = FALSE)
# The 'p=0.8' represents the 80% split stated above (having to do with the
#caret package); this will go into the 'TrainingIndex'
TrainingSet <- iris[TrainingIndex,]
# Training Set
TestingSet <- iris[-TrainingIndex,]
# Test Set
```

- We will subsequently use the training index to create a training set
  - In which it will perform slicing using the below:
    - **TrainingSet <- iris[TrainingIndex,]**

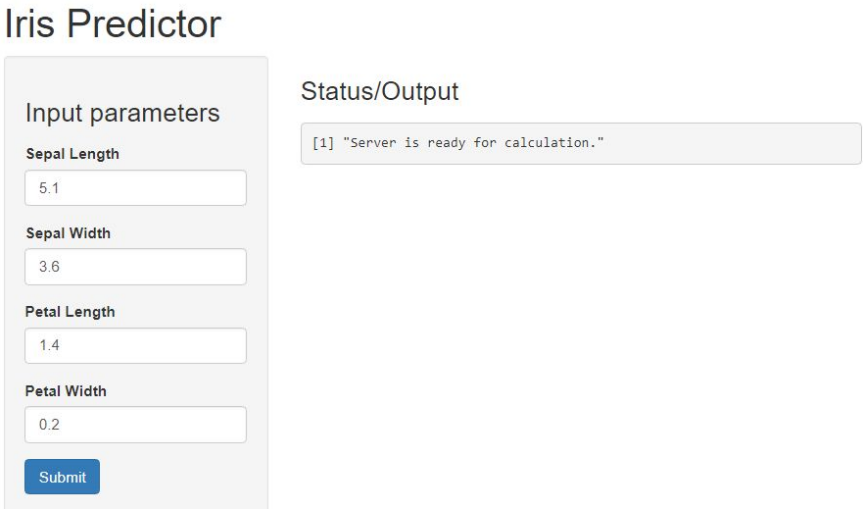
```
TrainingIndex <- createDataPartition(iris$Species, p=0.8, list = FALSE)
# The 'p=0.8' represents the 80% split stated above (having to do with the
#caret package); this will go into the 'TrainingIndex'
TrainingSet <- iris[TrainingIndex,]
# Training Set
# We will subsequently use the training index to create a training set
# In which it will perform slicing of the original iris data frame
# And then the remaining 20% will go to the 'TestingSet' below
TestingSet <- iris[-TrainingIndex,]
# Test Set
```

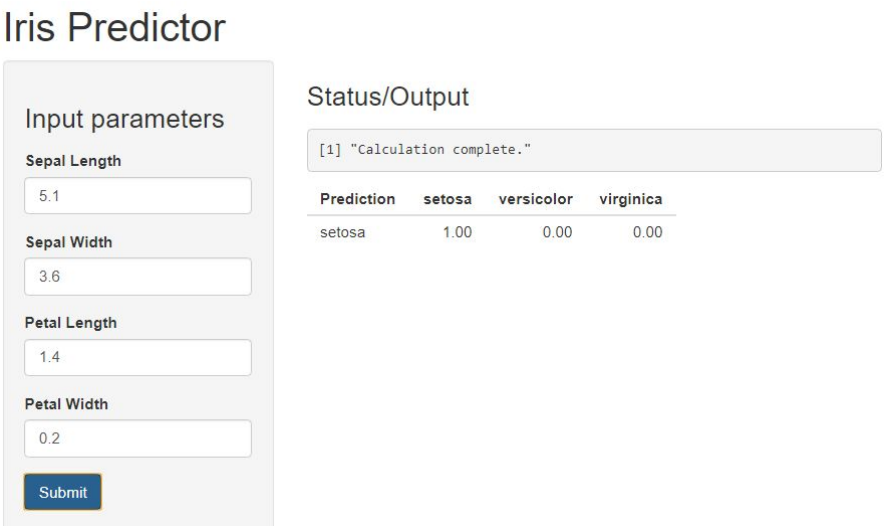
```
# This will write the files within the folder / location of which the
#'model', 'app-numeric', and 'app-slider' are placed
# IOW, we want all app files to be located in the SAME directory
write.csv(TrainingSet, "training.csv", file = 'WebAppsInR_Part2/training.csv')
write.csv(TestingSet, "testing.csv", file = 'WebAppsInR_Part2/testing.csv')
```

- The **'file = ...'** code will vary; it all depends on where the R code files of this app are located

- '*Ctrl+a*' & '*Ctrl+enter*' (to run full code)

Running 'app-numeric' file





- **Basic overview:**
  - It allows you to put the 4 input parameters (Sepal Length, Sepal Width, Petal Length, Petal Width); can adjust them to your choice
  - After selecting the '*Submit*' button, a prediction will be made
  - The example / image of the app & prediction above shows that the input parameters predict the plant to be a *sertosa*
    - Basically, the input parameter will be sent to the predictive model, (random forest model), and then this model will perform the classification. It has classified this input parameter as an iris *sertosa*.

```
28 # After initially deploying the predictive model
29 #in an RDS file (in 'model' file), the line below
30 #reads-in the Random Forest model
31 # This model will be used for making the prediction
32 model <- readRDS("model.rds")
```

  - The advantage of using this model / general method, is that the model is already built, and so there is no additional workload on the shiny application.
    - So it can just readily read / load in the model & perform the classification.
    - This approach / method of building a predictive model saves time.

User Interface

```
sidebarPanel(
  HTML("<h3>Input parameters</h3>"),
  tags$label(h3('Input parameters')),
  numericInput("Sepal.Length",
               label = "Sepal Length",
               value = 5.1),
  numericInput("Sepal.Width",
               label = "Sepal Width",
               value = 3.6),
  numericInput("Petal.Length",
               label = "Petal Length",
               value = 1.4),
  numericInput("Petal.Width",
               label = "Petal Width",
               value = 0.2),

  actionButton("submitbutton", "Submit",
               class = "btn btn-primary")
),
```

- Recall: HTML tag; size of heading will be *<h3>*
  - Within **numericInput("Sepal.Length",...)** etc.
    - Notice the S & L are capitalized, and this is the ID of this input parameter
    - (as these are case-sensitive)
    - Recall the following format: **input$Sepal.Length** etc.
      - This will be the input parameter which the server will be using as the data to be sent into the Random Forest Model
    - Recall: **value = 5.1** etc. is the default value that appears within the app
- **actionButton("submitbutton",...)**
  - This will overwrite the *reactive* function (in which when there is no *submit* button, every time we modify the numbers in here, a prediction will be made)

```
82   mainPanel(
83     tags$label(h3('Status/Output')), # Status/Output Text Box
84     verbatimTextOutput('contents'),
85     tableOutput('tabledata') # Prediction results table
86     # The results will be displayed in the 'tabledata'
87     #(below the text message in the app)
88
```

- The following code: **tags$label(h3('Status/Output'))** and
  - **HTML("<h3>Status/Output<h3>")**
  - Do the exact same thing

Server

```
server<- function(input, output, session) {


  # Input Data / Input Parameters
  datasetInput <- reactive({

    df <- data.frame(
      Name = c("Sepal Length",
               "Sepal Width",
               "Petal Length",
               "Petal Width"),
      Value = as.character(c(input$Sepal.Length,
                             input$Sepal.Width,
                             input$Petal.Length,
                             input$Petal.Width)),
      stringsAsFactors = FALSE)
```

- These are obtained from the UI component; where the user will input the input parameters and click on the submit button
  - And upon doing that, all the input parameters will be sent

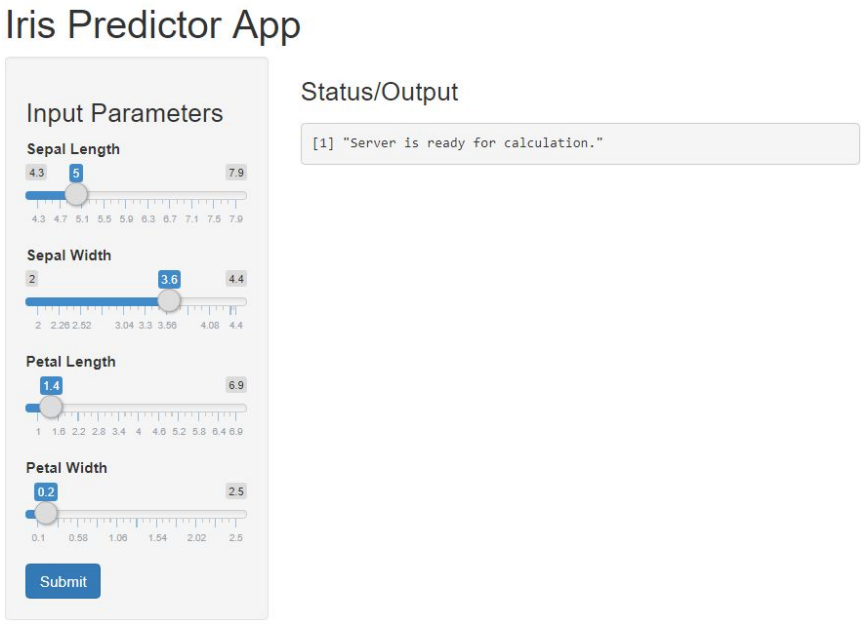
```

  Species <- 0
  df <- rbind(df, Species)
  input <- transpose(df)
  write.table(input,"input.csv", sep=",", quote = FALSE, row.names = FALSE, col.names = FALSE)

  test <- read.csv(paste("input", ".csv", sep=""), header = TRUE)

  Output <- data.frame(Prediction=predict(model,test), round(predict(model,test,type="prob"), 3))
  print(Output)

})
```

- This will generate the csv file as an output
  - Which will be read into the test object / variable created
  - Then apply the model to make a prediction on this test object (this ensures if the code & model works)

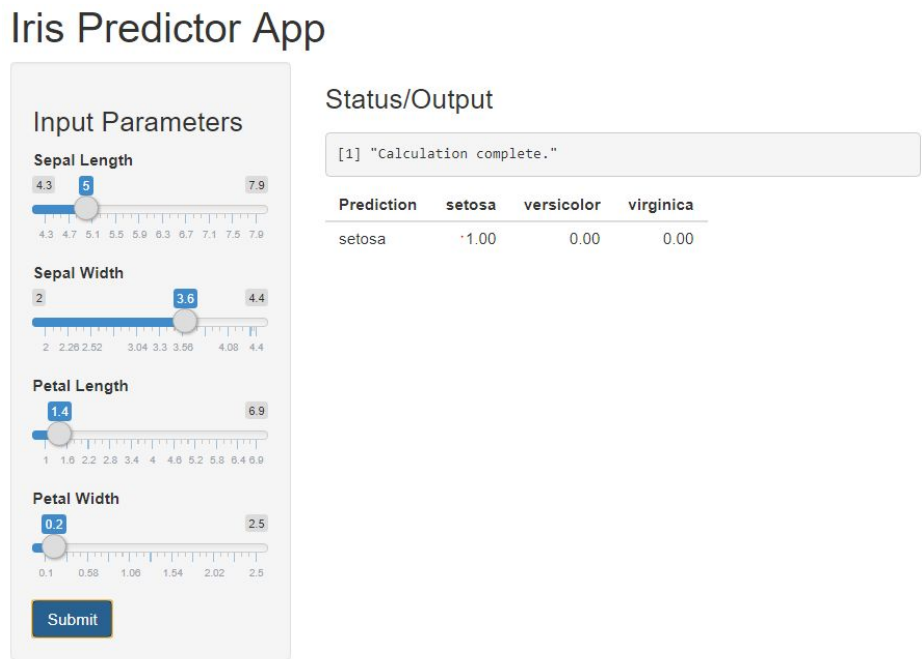
```
  # Status/Output Text Box
  output$contents <- renderPrint({
    if (input$submitbutton>0) {
      isolate("Calculation complete.")
    } else {
      return("Server is ready for calculation.")
    }
  })


  # Prediction results table
  output$tabledata <- renderTable({
    if (input$submitbutton>0) {
      isolate(datasetInput())
    }
  })

}
```

- Once the prediction has been made, it will be sent to **output$tabledata**

Running 'app-slider'



-



-

- Basically, the only difference between the two apps is that the input parameters now have a slider instead of the numeric text-box
- Main differences in terms of code (These do not appear in 'app-numeric')

-

```
39 # Training set
40 TrainSet <- read.csv("training.csv", header = TRUE)
41 TrainSet <- TrainSet[,-1]
42
```

```
> head(TrainSet)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.6         3.1          1.5         0.2  setosa
4          5.4         3.9          1.7         0.4  setosa
5          4.6         3.4          1.4         0.3  setosa
6          5.0         3.4          1.5         0.2  setosa
>
```

- This is the data after the first column (the index) has been deleted

```
57    sidebarPanel(
58      HTML("<h3>Input Parameters</h3>"),
59      sliderInput("Sepal.Length", label = "Sepal Length", value = 5.0,
60                  min = min(TrainSet$Sepal.Length),
61                  max = max(TrainSet$Sepal.Length)
```

- Specifically **min = min(TrainSet$Sepal.Length)** &
- **max = max(TrainSet$Sepal.Length)**
- **TrainSet$SepalLength**

```
> TrainSet$Sepal.Length
  [1] 5.1 4.9 4.6 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5.1 4.6 5.1 4.8 5.0
 [23] 5.0 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.5 4.9 4.4 5.1 5.0 4.5 5.0 5.1 4.8 4.6 7.0 6.4 6.9 5.5
 [45] 6.5 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 5.6 5.8 5.6 5.9 6.1 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5
 [67] 5.5 5.8 6.0 5.4 6.7 5.6 5.5 6.1 5.8 5.0 5.7 5.7 6.2 5.7 6.3 5.8 7.1 6.5 7.6 4.9 6.7 6.5
 [89] 6.4 6.8 5.7 5.8 6.4 7.7 7.7 6.0 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1
[111] 7.7 6.3 6.0 6.9 5.8 6.8 6.7 6.7 6.3 6.5
>
```

- This shows values within the ‘Sepal.Length’ column

```
> min(TrainSet$Sepal.Length)
[1] 4.3
> max(TrainSet$Sepal.Length)
[1] 7.9
>
```

- Also, notice the name change **numericInput** vs. **sliderInput**

References:

https://archive.ics.uci.edu/ml/datasets/iris

https://en.wikipedia.org/wiki/Iris_flower_data_set

https://www.youtube.com/watch?v=ceg7MMQNIn8&list=PLtqF5YXg7GLkxx_GGXDI_EiAvkhY9olbe&index=4