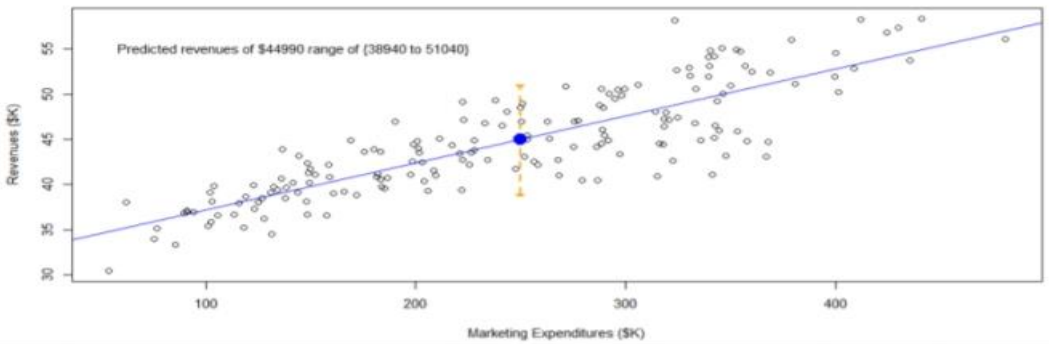
Shiny Web Dashboard

Part II

Revenue Prediction from Marketing Expenditures

Expenditure Level in $K:

54 250 481



```
sliderInput(inputId="spend", label="Expenditure Level in $K:",
            min = 54, max = 481, value = 250)
```

The parameters used in this code are explained as follows:

- The `inputId` parameter is a reference for the value input by the user. A Shiny app might have multiple widgets on the same page and even more than one slider widget. For these reasons, `inputId` is the unique name `spend` to reference this particular widget.
- The `label` parameter can be any text you think is appropriate to inform the user about how to use the widget. In this example, the label informs the user that it establishes the marketing expenditures in thousands of dollars.
- The `min` parameter sets the floor of the allowed input. Your linear regression range (from your earlier work) began at 53.65. You will set the `min = 54` to keep the user input inside the linear regression range.
- The `max` parameter sets the high end of allowed input. Again, from previous experience, you know the maximum range of your model is `481`.
- The `value` parameter allows the Shiny app developer to set a default when the app opens. You will set it around the midpoint at a value of `250`.

Now, you will place the slider widget on the page. The `sidebarLayout()` function is a design theme that includes a `sidebarPanel()` area for widgets on the left-hand side of the page and a larger `mainPanel()` area on the right-hand side for output, such as the regression plot:

```
library(shiny)
shinyUI(fluidPage(
     titlePanel("Revenue Prediction from Marketing Expenditures"),
     sidebarLayout(
          sidebarPanel(
               sliderInput("spend", "Expenditure Level in $K:",
                           min = 54, max = 481, value = 250)
               ),
          mainPanel(
               plotOutput("prediction_plot")
               )
          )
     ))
```

its logic will be above and outside of the shinyServer() function.

```
library(shiny)
revenue <- read.csv("./data/Ch8_marketing.csv")
model <- lm(revenues ~ marketing_total, data = revenue)
```

he following is the code that makes up the remaining parts of the server.R file
plain this code that is broken down into five chunks of logic performed by the
llows:

```
shinyServer(function(input, output) {
      output$prediction_plot <- renderPlot({
            plot(revenue$marketing_total, revenue$revenues,
                  xlab = "Marketing Expenditures ($K)",
                  ylab = "Revenues ($K)")
            abline(model, col = "blue")
            newdata <- data.frame(marketing_total = input$spend)
            pred <- predict.lm(model, newdata, interval = "predict")
            points(c(rep(input$spend, 2)), c(pred[2], pred[3]),
                  pch = "-", cex = 2, col = "orange")
            segments(input$spend, pred[2], input$spend, pred[3],
                  col = "orange", lty = 2, lwd = 2)
            points(input$spend, pred[1], pch = 19, col = "blue",
                  cex = 2)
            text(54, 55, pos = 4, cex = 1.0,
                  paste0("Predicted revenues of $",
                        round(pred[1], 2) * 1000,
                        " range of {", round(pred[2], 2) * 1000,
                        " to ", round(pred[3], 2) * 1000, "}"))
            })
      })
```

he following is an explanation of the previous code:

- The server logic is initiated by `shinyServer(function(input, output) {...})` so that it can receive the slider input value coming from the UI and send the rendered plot back.
- The `output$prediction_plot <- renderPlot({...})` line of the code contains the `renderPlot{}` function. It will wrap any R plot so that you can pass it as the output variable back to the UI for the user to view. All outputs back to the UI will follow the `output${name}` naming convention. Look back at the main panel in the `ui.R` file. It expects an output object called `prediction_plot` to the `plotOutput()` function.
- The third logic chunk consists of the `plot()` and `abline()` functions that create a scatter plot of the data and a fitted line from the regression model.
- The fourth chunk is similar to work you did in linear regression, but with one Shiny element included: it depends on input from the UI. This chunk takes the `input$spend` slider input value to form a data frame called `newdata`. This input passes to the `predict.lm()` function to retrieve that prediction from our model. Notice how this slider input value is pulled from the UI using the `input$spend` variable. Refer back to the `ui.R` file and see that the `inputId` slider is `spend`. It passes to the `server.R` file and is identified on the server side by its `inputId`.
- The final chunk creates the accent points, line segments, and text to enhance and label the outputted plot. Once again, you will see the use of `input$spend`, which is used in this instance to specify the x-axis location in the `points()` and `segments()` functions.

Hello, this application allows you to specify different clusters of customers based on age and income to determine more targeted marketing segments. Simply, move the slider at right for more or less clusters and chose a clustering method.

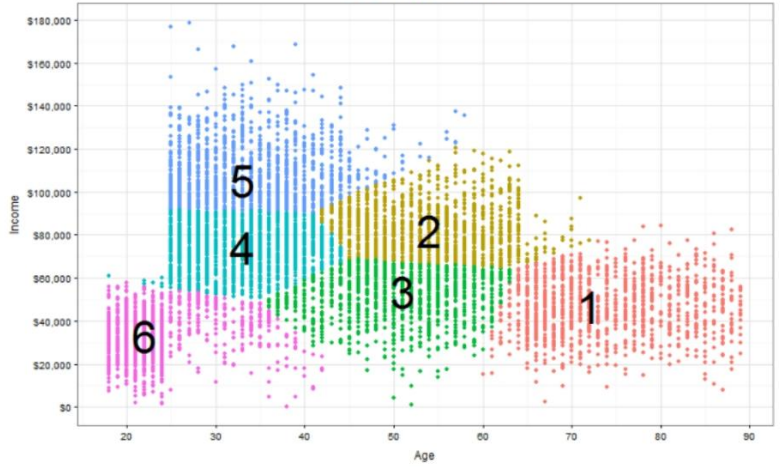
After choosing a cluster scheme you can filter and download customer data based on those clusters to run a campaign.

How Many Clusters?

2 6 10

Clustering Method?

- (•) K-means
- ( ) Hierarchical



Cluster Summary Table

Median Age	Median Income	Cluster Id	Min. Age	Max. Age	Min. Income	Max. Income
22	$32,980	6	18	42	$234	$57,863
72	$46,575	1	60	89	$2,319	$84,301
51	$53,666	3	36	63	$973	$69,136
33	$74,137	4	18	44	$50,703	$91,706
54	$82,258	2	42	72	$65,050	$120,686
33	$105,502	5	25	58	$90,700	$178,676

Using a grid layout

Designing the layout of a web application page can be difficult. One challenge is aligning the elements containing information that you would like to convey. Fortunately, Shiny includes a grid-based layout framework based on the Bootstrap project to help you tackle this challenge.

Learn more: If you have experience in web development, you may have used Bootstrap. It is helpful in standardizing the appearance of web pages. You can learn more about the Bootstrap project and what it can do at `http://getbootstrap.com/`.

There are two main functions in the Shiny implementation of the Bootstrap framework: `fluidRow()` and `column()`. These functions separate the page layout into rows of content that are fluid with the page width. These rows are further divided into columns to create a grid. The following chunk of code creates a row with a column element containing text. Again, there is no need to type this into your console:

```
fluidRow(
     class = "vertical-align",
     column(12, "Hello, this is my Shiny App")
)
```

UI components of the marketing-campaign app

Here are some of the components of the marketing-campaign app broken down into implementation code to show you how to use and tailor Shiny widgets.

One design element will be a radio button to toggle between k-means and hierarchical clustering methods as options:

```
radioButtons("cluster_method", label = "Clustering Method?",
             choices = c("K-means", "Hierarchical"),
             selected = "K-means", inline = FALSE)
```

Another design element will be a second slider input that allows the user to select the number of clusters between a minimum of two and a maximum of ten:

```
sliderInput("cluster_count", label = "How Many Clusters?",
            min = 2, max = 10, value = 6, step = 1)
```

There should also be an area devoted to displaying a plot of the clusters created from the selected method and a table to summarize the average values in the dataset for the membership of each cluster. Finally, there should be an area to display a table of potential customers based on their cluster membership with a button to download the data into a spreadsheet for the marketing team to use:

```
fluidRow(
      column(7, plotOutput("cluster_viz", height = "500px")),
      column(5, h3("Cluster Summary Table"),
             DT::dataTableOutput("campaign_summary_table",
                                 width = "100%"))
      ),
    fluidRow(
      column(12,
          downloadButton("downloadDataFromTable","Download Table Data"))
    )
    fluidRow(

      column(12,
             DT::dataTableOutput("campaign_table", width = "100%"))
    )
```

Server components of the marketing-campaign app

After you design and code the core UI components of the marketing-campaign app, you can put your knowledge about scoping and reactivity to practice inside the `server.R` file. First, you know that you will want to load the dataset from a .csv file above and outside the `shinyServer()` environment to create a globally-scoped market data frame:

```
setwd("Ch8-CampaignCreatorApp")
market <- read.csv("./data/Ch8_global_market_data.csv")
shinyServer(function(input, output, session) {...
```

Next, we will create a reactive component that responds to the `cluster_method` and `cluster_count` inputs from the UI, builds a model, and assigns membership based on this user configuration:

```
clustered_dataset <- reactive({
      result_dat <- market
      if (input$cluster_method == "K-means") {
            kmeans_model <- kmeans(x = market[, c("age_scale",
                                                  "inc_scale")],
                                   centers = input$cluster_count)
            result_dat$cluster_id <- as.factor(kmeans_model$cluster)
      } else {
            hierarchical_model <-
                  hclust(dist(market[, c("age_scale", "inc_scale")]),
                         method = "ward.D2")
            result_dat$cluster_id <-
                  as.factor(cutree(hierarchical_model,
                                   input$cluster_count))
      }
      return(result_dat)
})
```