R PROJECT

Libraries

shiny:

Purpose: This library is used for creating interactive web applications in R without needing to learn HTML, CSS, or JavaScript.

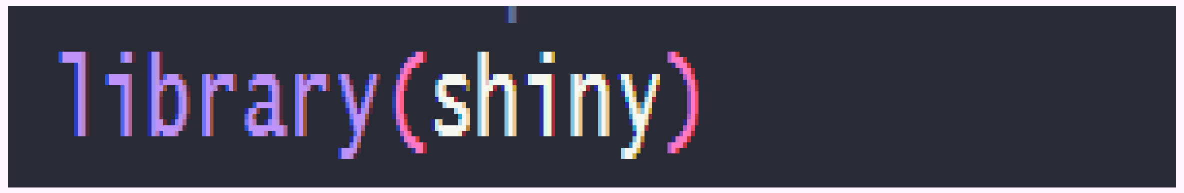
Functionality: It allows you to build user interfaces with various elements like sliders, buttons, dropdowns, plots, and tables. Users can interact with these elements to explore and analyze data within a web browser.

Common Use Cases:

Data visualization: Create interactive plots and dashboards for data exploration.

Data analysis tools: Build custom analytical tools and workflows for statistical analysis

Reporting: Develop interactive reports and documents that dynamically update based on user interactions or data changes.



```
library(shiny)
```

Libraries

ggplot2:

Purpose: ggplot2 is a popular library for creating elegant and customizable statistical graphics in R.

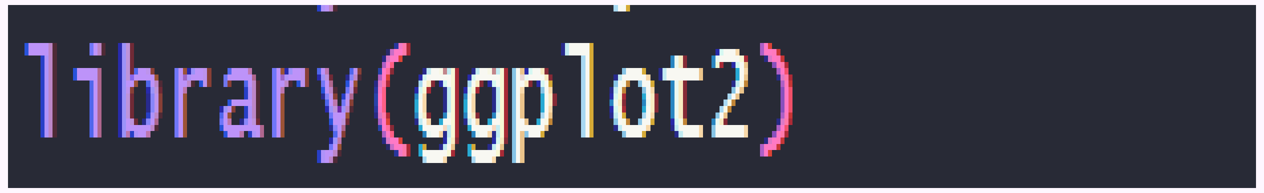
Functionality: It offers a grammar-based approach to plotting, allowing to define visual elements like data aesthetics and plot types (e.g., scatter plots, bar charts, line graphs).

Common Use Cases:

Exploratory data analysis: Quickly visualize distributions, trends, and relationships in datasets.

Publication-quality graphics: Generate high-resolution pleasing plots for inclusion in reports and presentations.

Customization: Fine-tune the appearance and style of plots to meet specific design requirements .



```
library(ggplot2)
```

Libraries

arules:

Purpose: This library is specifically designed for working with association rule learning (ARL) in R.

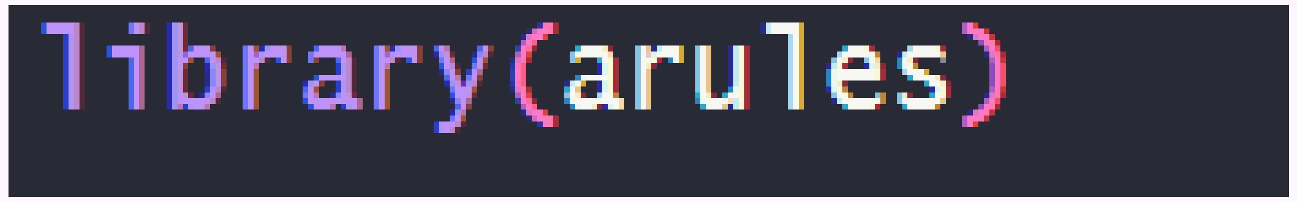
Functionality: It provides functionalities for reading transaction data ,calculating support, confidence, and lift for association rules, and exploring frequent item sets within a dataset.

Common Use Cases:

Market basket analysis: Identify associations between products frequently purchased together in retail transactions.

Cross-selling and recommendation systems: Generate rules to suggest related items to customers based on their purchase history.

Web usage mining: Analyze patterns in website navigation or user behavior to improve website.



```
library(arules)
```

Codes

fluidPage() ->

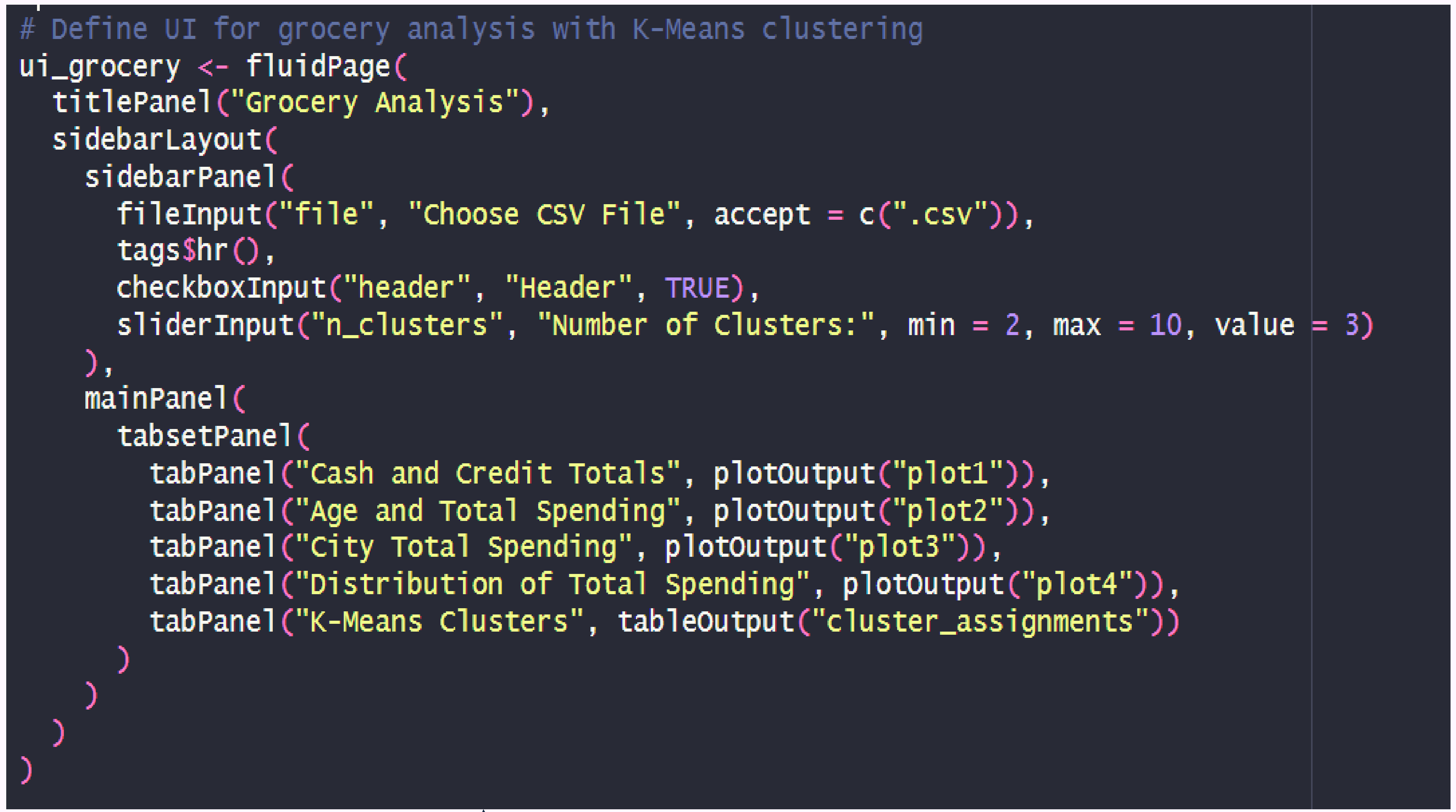
This function from the **shiny** library creates the overall layout of the web application. It's designed to be responsive and adapt to different screen sizes.

titlePanel("Grocery Analysis") ->

This sets the title displayed at the top of the application, in this case, "Grocery Analysis.“

sidebarLayout() ->

This function defines a layout with two main sections: a sidebar and a main panel.



```
# Define UI for grocery analysis with K-Means clustering
ui_grocery <- fluidPage(
  titlePanel("Grocery Analysis"),
  sidebarLayout(
    sidebarPanel(
      fileInput("file", "Choose CSV File", accept = c(".csv")),
      tags$hr(),
      checkboxInput("header", "Header", TRUE),
      sliderInput("n_clusters", "Number of Clusters:", min = 2, max = 10, value = 3)
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("Cash and Credit Totals", plotOutput("plot1")),
        tabPanel("Age and Total Spending", plotOutput("plot2")),
        tabPanel("City Total Spending", plotOutput("plot3")),
        tabPanel("Distribution of Total Spending", plotOutput("plot4")),
        tabPanel("K-Means Clusters", tableOutput("cluster_assignments"))
      )
    )
  )
)
```

This code creates a Shiny application with a user-friendly interface. Users can upload a CSV file, number specify whether it has a header row, and choose the number of clusters for K-Means analysis. The application then displays various visualizations and tables in different tabs based on the selected options and uploaded data.

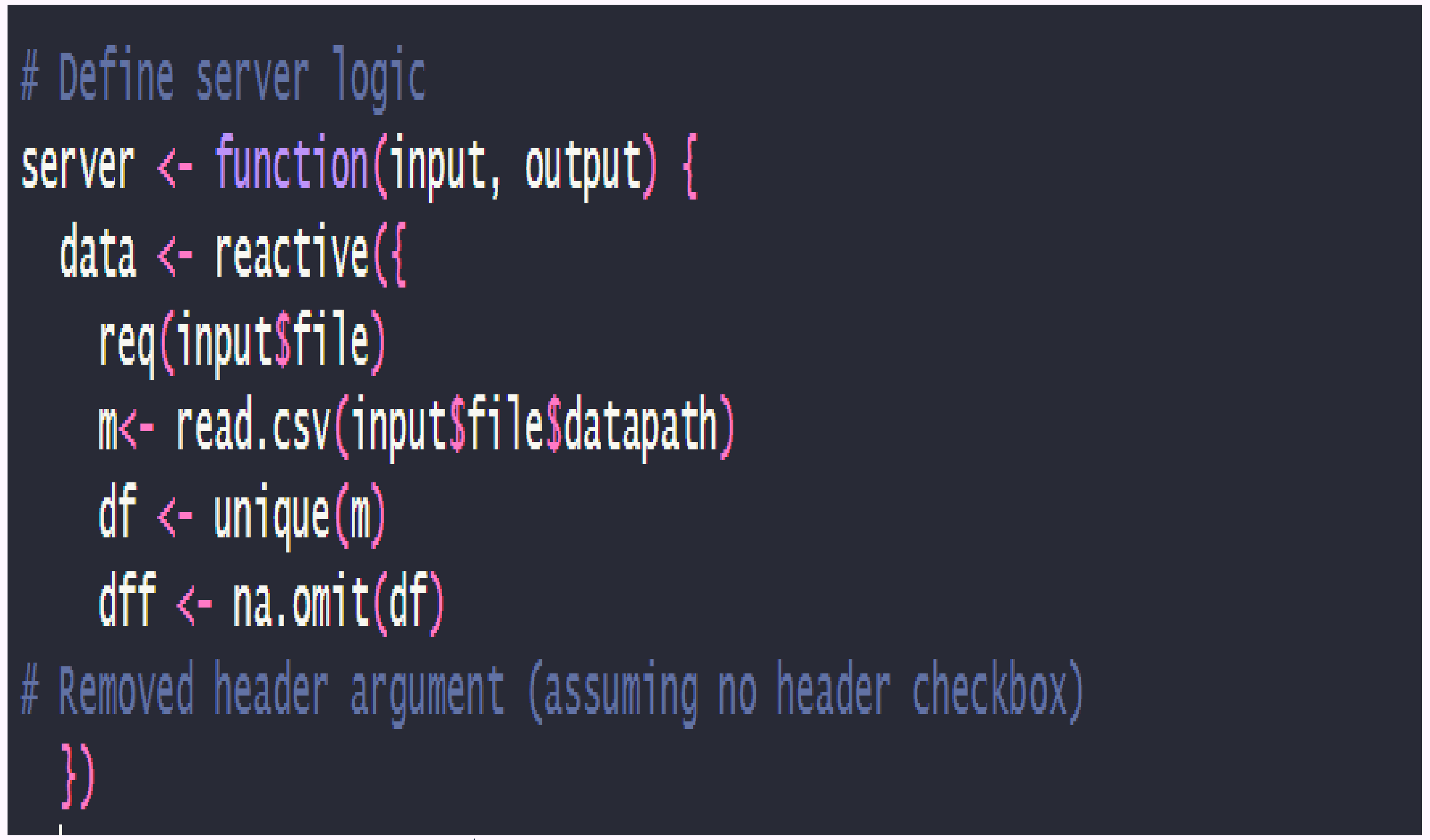
Codes

server ->
takes two arguments:
input: This object captures user input from the UI.
output: This object allows the server to send data or visualizations back to the UI for display.

reactive() ->
This function from the **shiny** library creates the overall layout of the web application. It's designed to be responsive and adapt to different screen sizes.

req(input$file) - >
This function ensures that the file input widget is not empty before proceeding. It throws an error message if the file input is empty.

read.csv(input$file$datapath): This function reads the CSV file uploaded by the user.



```
# Define server logic
server <- function(input, output) {
  data <- reactive({
    req(input$file)
    m<- read.csv(input$file$datapath)
    df <- unique(m)
    dff <- na.omit(df)
# Removed header argument (assuming no header checkbox)
  })
```

this code defines a reactive element named **data** that reads and prepares the uploaded CSV file for further analysis in the server logic.

Codes

req(data()) ->

ensures that the plot is only rendered if the **data** reactive element has been evaluated and contains valid data.

x <- table(data()$paymentType)->

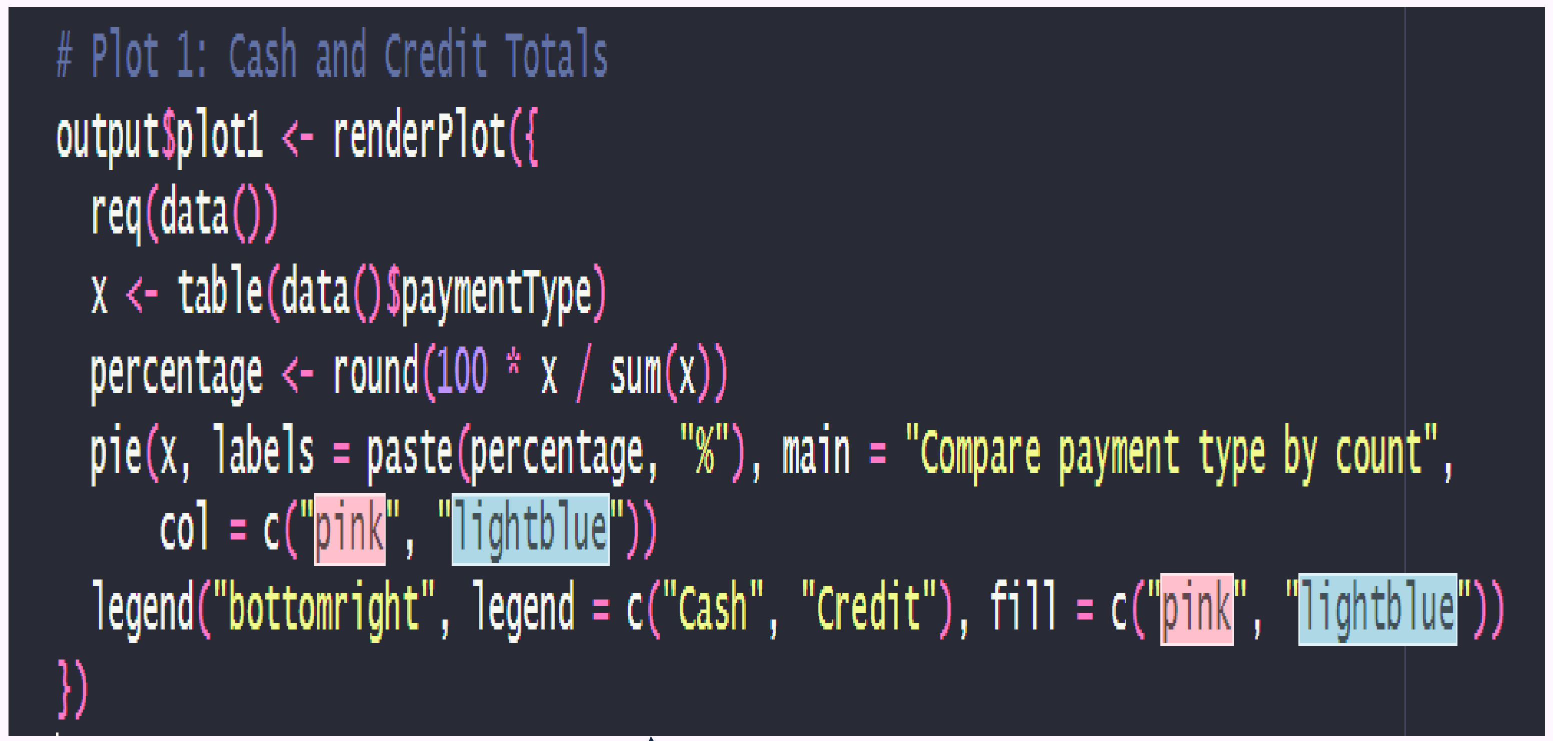
creates a frequency table using **table**(). It counts the occurrences of each unique value in the paymentType column ,The result is stored in the variable x.

percentage <- round(100 * x / sum(x))->

calculates the percentage of each payment type. It divides each value in x by the total sum of all values in x and multiplies by 100. The result is stored in the variable percentage.

legend("bottomright", legend = c("Cash", "Credit"), fill = c("pink", "lightblue")):

This line adds a legend to the plot.



```
# Plot 1: Cash and Credit Totals
output$plot1 <- renderPlot({
  req(data())
  x <- table(data()$paymentType)
  percentage <- round(100 * x / sum(x))
  pie(x, labels = paste(percentage, "%"), main = "Compare payment type by count",
      col = c("pink", "lightblue"))
  legend("bottomright", legend = c("Cash", "Credit"), fill = c("pink", "lightblue"))
})
```

this code effectively generates a pie chart that visualizes the distribution of payment types (cash vs. credit) along with their percentages in your data.

Codes

output$plot2 <- renderPlot({ ... })->

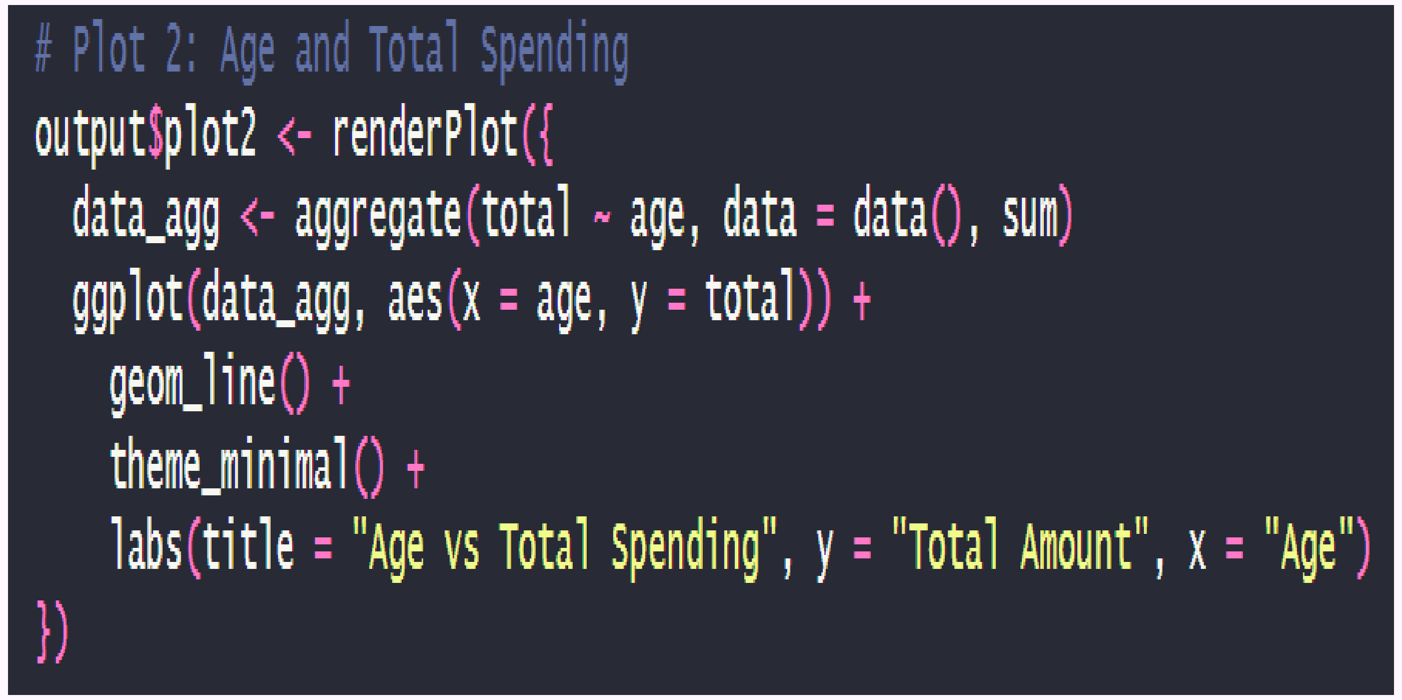
This line defines a render function for the plot2 output element in the UI. Similar to output$plot1, this function generates the content for this specific plot.

req(data())->

This line ensures that the plot is only rendered if the data reactive element has been evaluated and contains valid data.

data_agg <- aggregate(total ~ age, data = data(), sum)->

This line uses the aggregate function to summarize the data by age



```
# Plot 2: Age and Total Spending
output$plot2 <- renderPlot({
  data_agg <- aggregate(total ~ age, data = data(), sum)
  ggplot(data_agg, aes(x = age, y = total)) +
    geom_line() +
    theme_minimal() +
    labs(title = "Age vs Total Spending", y = "Total Amount", x = "Age")
})
```

this code creates a line plot using ggplot2 to visualize the relationship between age and total spending in your data. The aggregated data simplifies the plot by showing average total spending for each age group.

Codes

ggplot(data_agg, aes(x = age, y = total)) +...- >

This line creates a ggplot object using the ggplot function.

geom_line()->

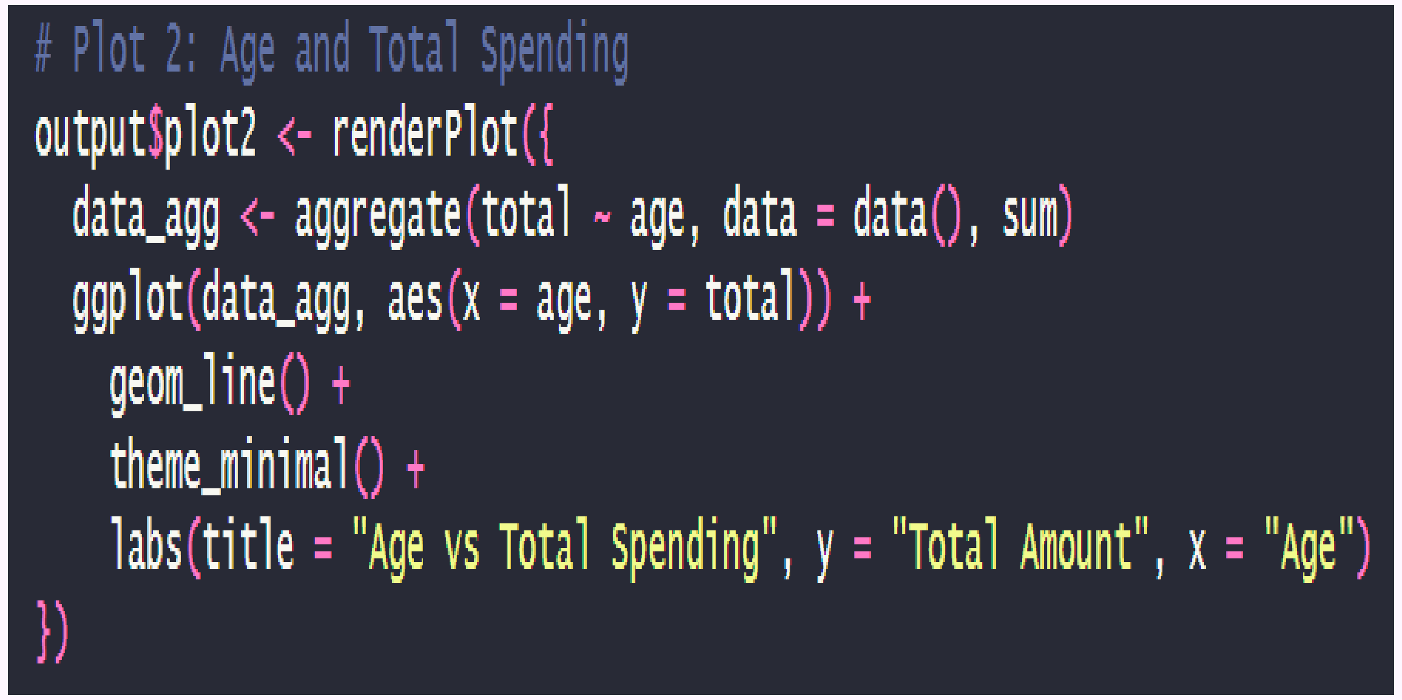
This line adds a line geometry to the plot. This creates a line connecting the average total spending for each age group.

theme_minimal()->

This line applies a minimalist theme to the plot using theme_minimal(). This removes unnecessary decorative elements, making the plot cleaner and easier to focus on the data.

labs(title = "Age vs Total Spending", y = "Total Amount", x = "Age")->

This line sets labels for the plot title, y-axis, and x-axis using the labs function.



```r
# Plot 2: Age and Total Spending
output$plot2 <- renderPlot({
  data_agg <- aggregate(total ~ age, data = data(), sum)
  ggplot(data_agg, aes(x = age, y = total)) +
    geom_line() +
    theme_minimal() +
    labs(title = "Age vs Total Spending", y = "Total Amount", x = "Age")
})
```

this code creates a line plot using ggplot2 to visualize the relationship between age and total spending in your data. The aggregated data simplifies the plot by showing average total spending for each age group.

Codes

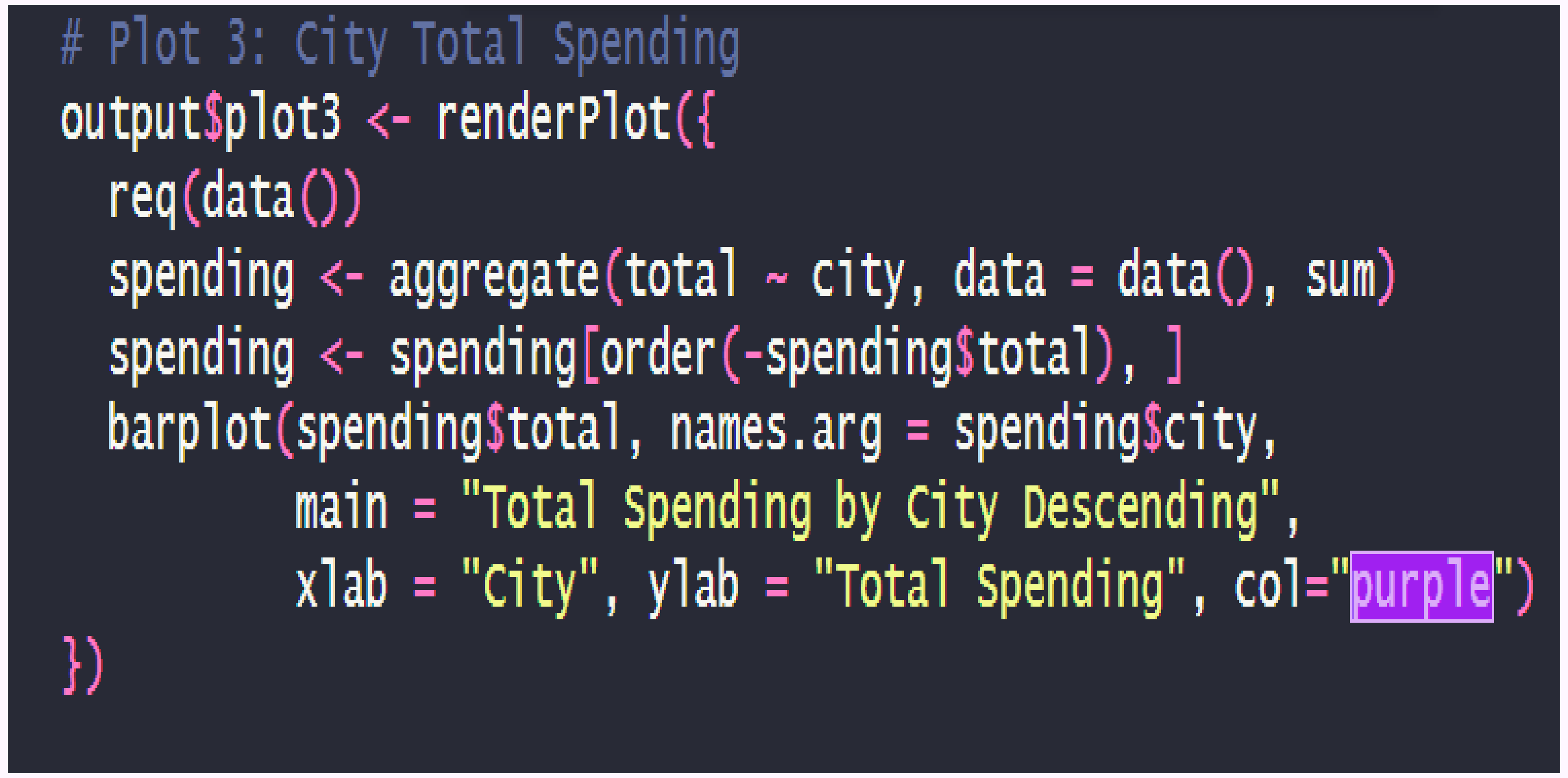
output$plot3 <- renderPlot({ ... })->
Similar to previous plots, this line defines a render function for the plot3 output element in the UI.

req(data())->
Ensures the plot is only rendered if the data reactive element has been evaluated and contains valid data.

spending <- aggregate(total ~ city, data = data(), sum)->
This line uses aggregate to summarize the data by city.

spending <- spending[order(-spending$total),]->
This line sorts the spending data frame by the total column in descending order (highest spending first).

barplot(spending$total, names.arg = spending$city, ...)->
This line creates a bar chart using barplot.



```
# Plot 3: City Total Spending
output$plot3 <- renderPlot({
  req(data())
  spending <- aggregate(total ~ city, data = data(), sum)
  spending <- spending[order(-spending$total), ]
  barplot(spending$total, names.arg = spending$city,
          main = "Total Spending by City Descending",
          xlab = "City", ylab = "Total Spending", col="purple")
})
```

this code snippet effectively creates a bar chart showing the total spending for each city in your data, ordered by the highest spending city first.

Codes

output$plot4 <- renderPlot({ ... })->

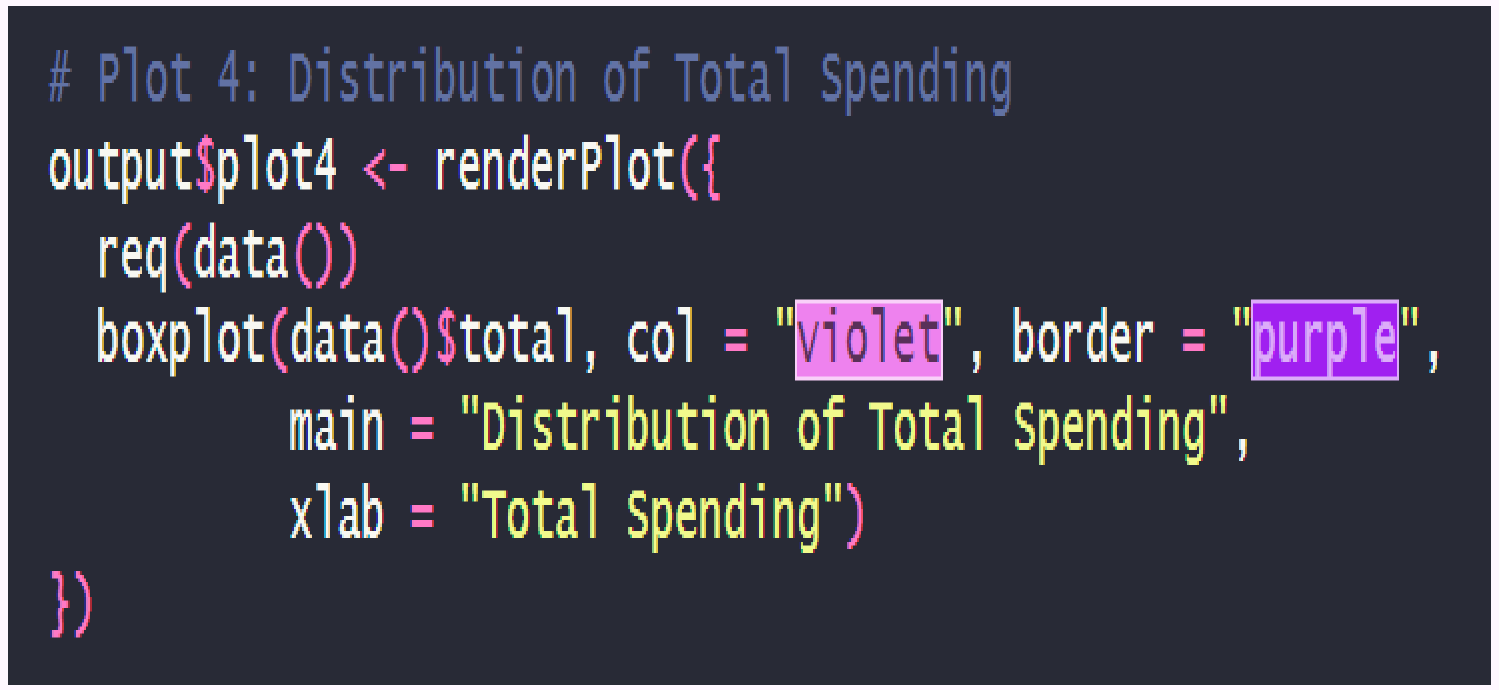
Similar to previous plots, this line defines a render function for the **plot4** output element in the UI.

req(data())->

Ensures the plot is only rendered if the **data** reactive element has been evaluated and contains valid data.

boxplot(data()$total, ...)->

This line creates a boxplot using the boxplot function.



```r
# Plot 4: Distribution of Total Spending
output$plot4 <- renderPlot({
  req(data())
  boxplot(data()$total, col = "violet", border = "purple",
          main = "Distribution of Total Spending",
          xlab = "Total Spending")
})
```

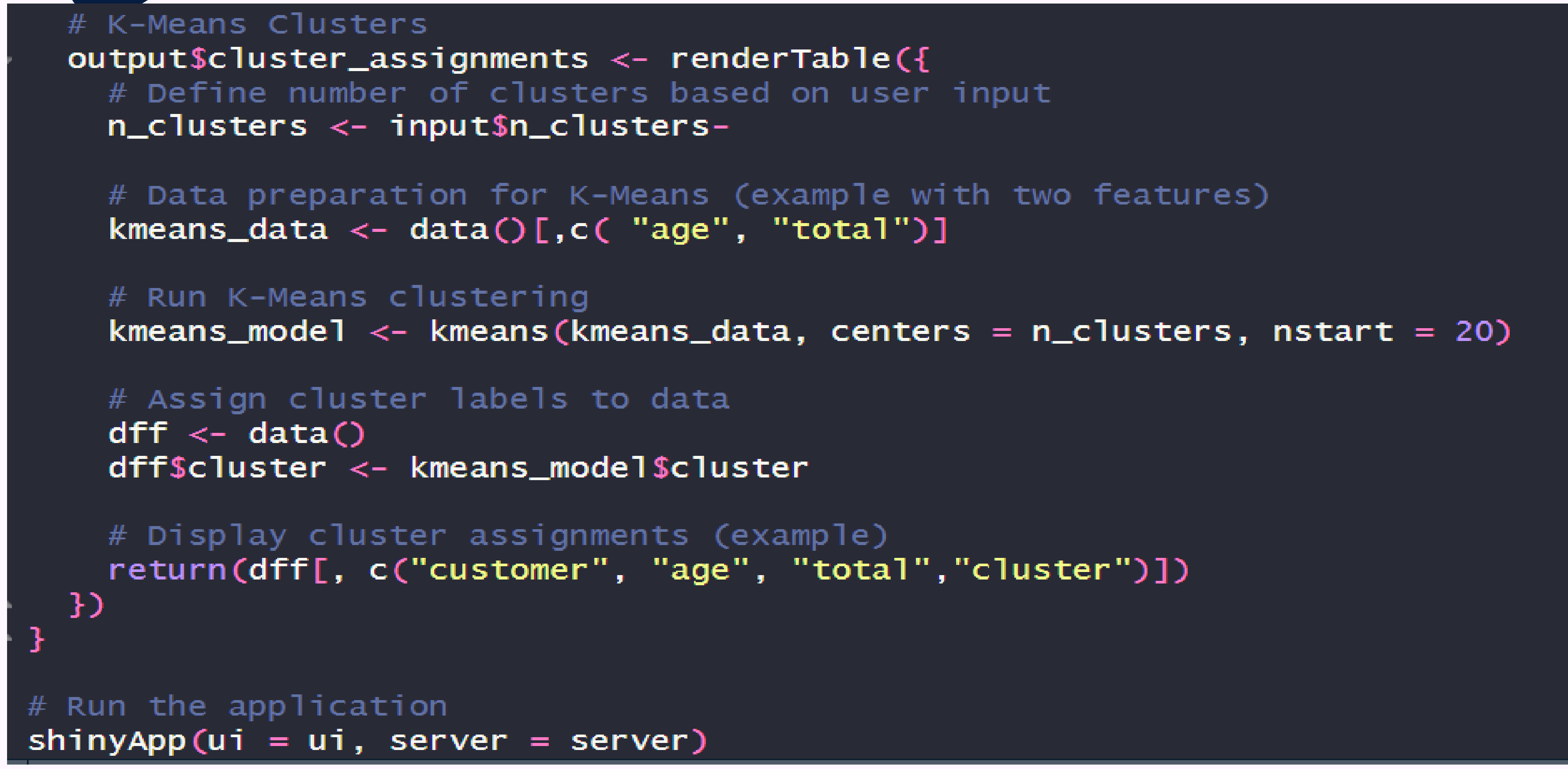
this code creates a boxplot that visualizes the distribution of total spending in your data, highlighting potential outliers and the spread of the data.

Codes

output$cluster_assignments <- renderTable({ ... })->
This line defines a render function for the cluster_assignments output element, which will display a table in the UI.

n_clusters <- input$n_clusters->
Retrieves the number of clusters chosen by the user from the n_clusters slider input.

kmeans_data <- data()[,c("age", "total")]->
This line selects specific features for K-Means clustering. It assumes you want to cluster based on age and total spending. Replace "age", "total" with the actual names of your desired features if different.



```
  # K-Means Clusters
  output$cluster_assignments <- renderTable({
    # Define number of clusters based on user input
    n_clusters <- input$n_clusters-

    # Data preparation for K-Means (example with two features)
    kmeans_data <- data()[,c( "age", "total")]

    # Run K-Means clustering
    kmeans_model <- kmeans(kmeans_data, centers = n_clusters, nstart = 20)

    # Assign cluster labels to data
    dff <- data()
    dff$cluster <- kmeans_model$cluster

    # Display cluster assignments (example)
    return(dff[, c("customer", "age", "total","cluster")])
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

this code performs K-Means clustering based on user-defined features and number of clusters. then assigns cluster labels to each data point and displays these assignments along with potentially relevant data (like customer ID, age, and total spending) in a table within the Shiny application.

Codes

kmeans_model <- kmeans(kmeans_data, centers = n_clusters, nstart = 20)->

This line performs K-Means clustering using the kmeans function.

dff <- data()->

Retrieves the entire data frame from the data() reactive element and stores it in dff.

dff$cluster <- kmeans_model$cluster->

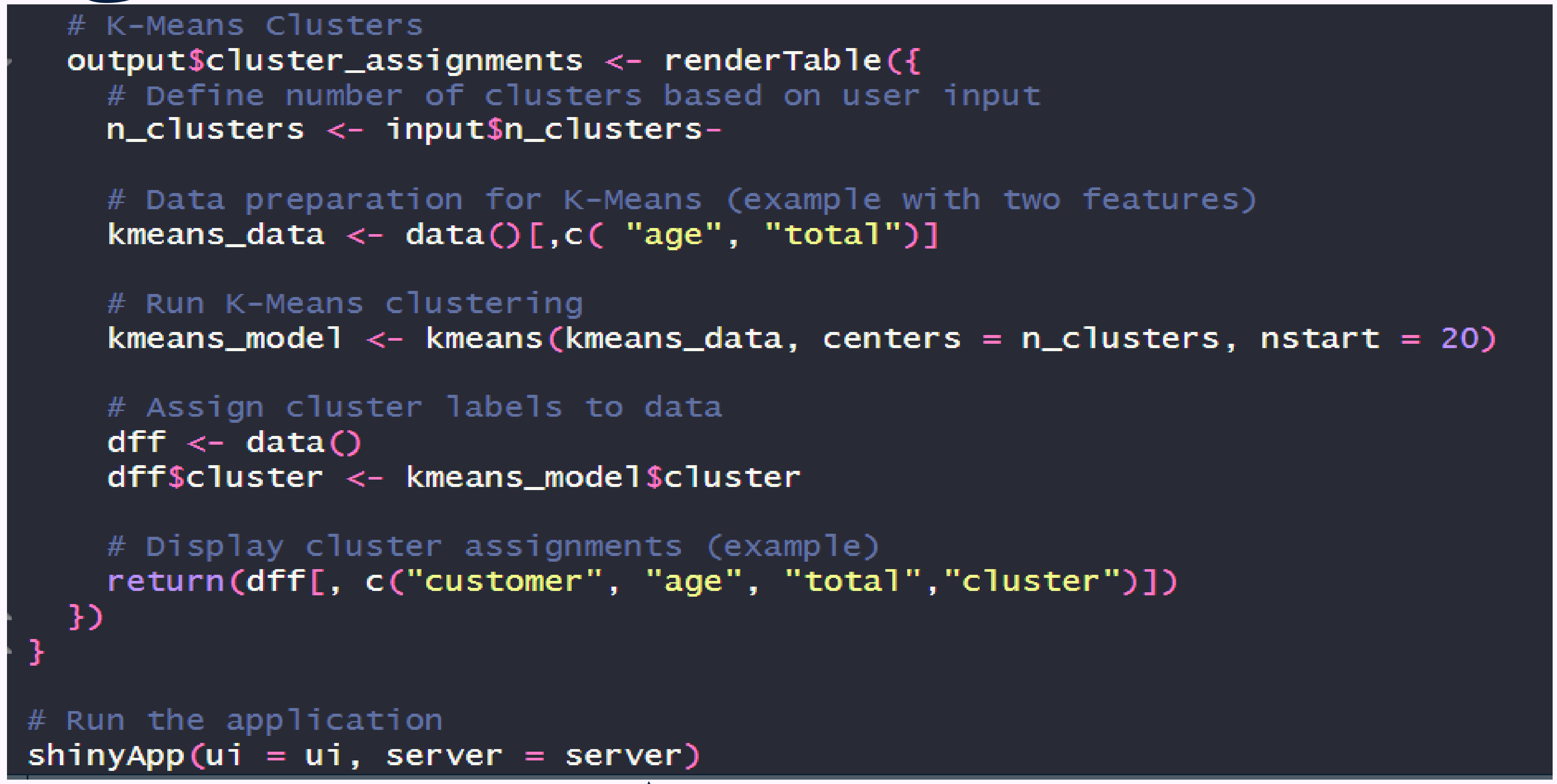
This line assigns cluster labels to each data point in dff.

return(dff[, c("customer", "age", "total","cluster")])->

This line defines what the render function should return (the data to be displayed in the table).

shinyApp(ui = ui, server = server)->

This line launches the Shiny application



```
  # K-Means Clusters
  output$cluster_assignments <- renderTable({
    # Define number of clusters based on user input
    n_clusters <- input$n_clusters-

    # Data preparation for K-Means (example with two features)
    kmeans_data <- data()[,c( "age", "total")]

    # Run K-Means clustering
    kmeans_model <- kmeans(kmeans_data, centers = n_clusters, nstart = 20)

    # Assign cluster labels to data
    dff <- data()
    dff$cluster <- kmeans_model$cluster

    # Display cluster assignments (example)
    return(dff[, c("customer", "age", "total","cluster")])
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

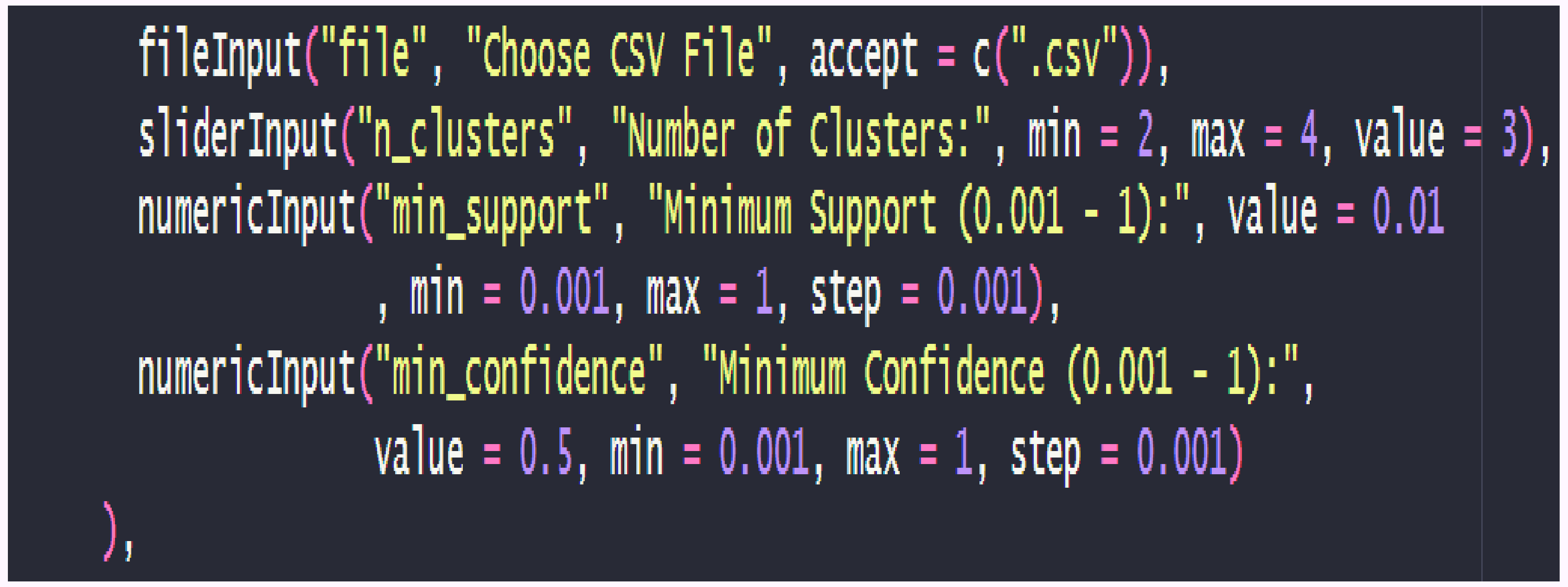
this code performs K-Means clustering based on user-defined features and number of clusters. then assigns cluster labels to each data point and displays these assignments along with potentially relevant data (like customer ID, age, and total spending) in a table within the Shiny application.

Codes

sliderInput("n_clusters", "Number of Clusters:", min = 2, max = 4, value = 3)->
This creates a slider labeled "Number of Clusters:" that allows users to choose a value between 2 and 4 . The default value is set to 3.

numericInput("min_support", "Minimum Support (0.001-1):", value = 0.1, min = 0.001, max = 1, step = 0.001)->
This creates a numeric input field labeled "Minimum Support" where users can enter a value between 0.001 and 1 in steps of 0.001. The default value is set to 0.1.

numericInput("min_confidence", "Minimum Confidence (0.001-1):", value = 0.5, min = 0.001, max = 1, step = 0.001)->
This creates another numeric input field labeled "Minimum Confidence" with similar properties to the minimum support input, but the default value is 0.5.



```
    fileInput("file", "Choose CSV File", accept = c(".csv")),
    sliderInput("n_clusters", "Number of Clusters:", min = 2, max = 4, value = 3),
    numericInput("min_support", "Minimum Support (0.001 - 1):", value = 0.01
                 , min = 0.001, max = 1, step = 0.001),
    numericInput("min_confidence", "Minimum Confidence (0.001 - 1):",
                 value = 0.5, min = 0.001, max = 1, step = 0.001)
  ),
```

This code snippet suggests the application might be related to data analysis, possibly for clustering or association rule mining. The minimum support and confidence values are likely used in algorithms like Apriori for frequent itemset generation.

Codes

output$rules_output <- renderPrint({ ... }):->

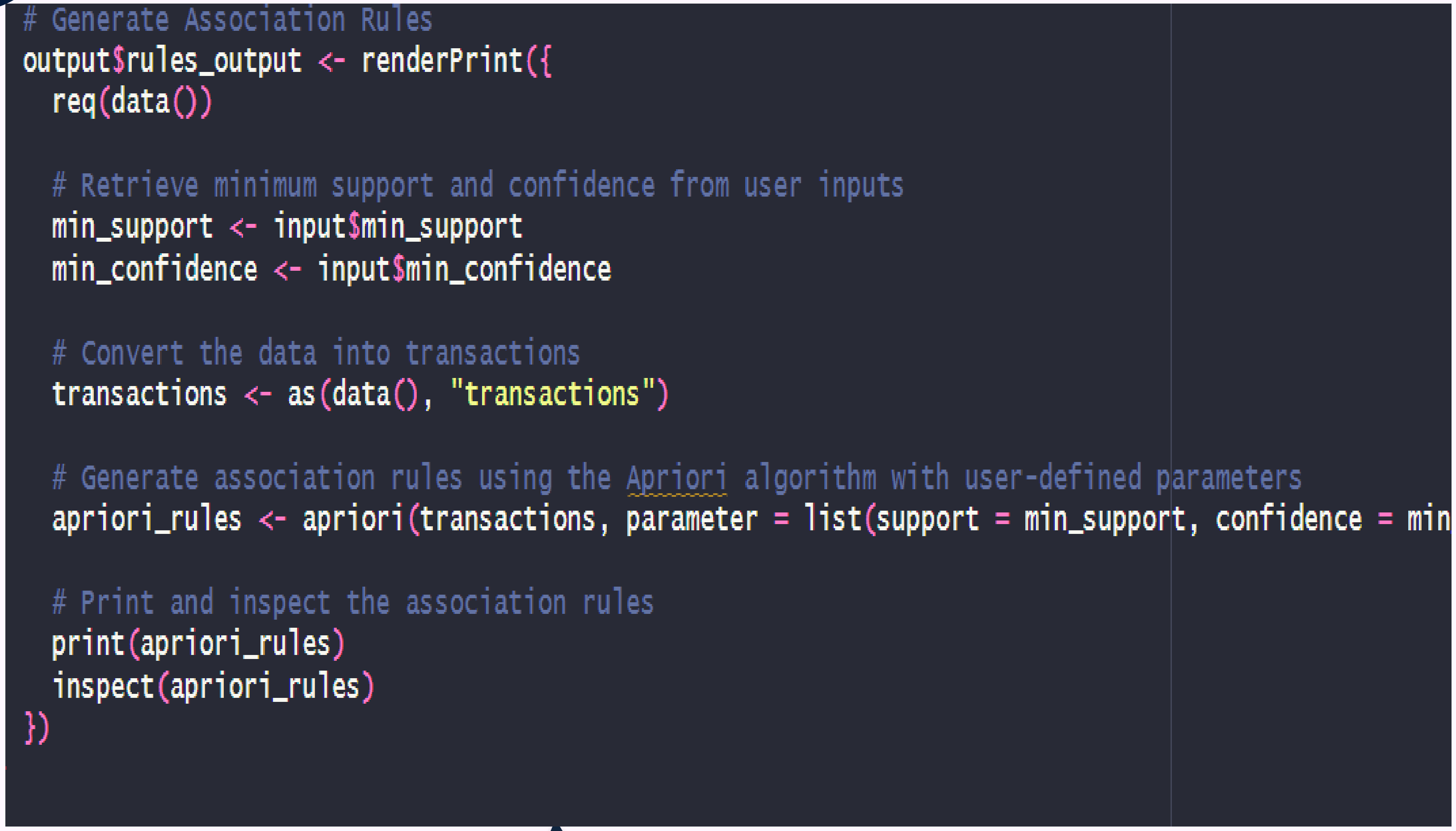
This defines a reactive function named rules_output that uses the renderPrint output binding.
This line ensures that the data() function (presumably defined elsewhere) is called before proceeding. This function likely retrieves the dataset used for analysis.

min_support <- input$min_support->

This retrieves the value entered by the user for the minimum support threshold from a UI element named min_support.

min_confidence <- input$min_confidence->

Similar to the previous line, this retrieves the user-defined minimum confidence threshold from the min_confidence UI element. Confidence determines the strength of the association between items in a rule.



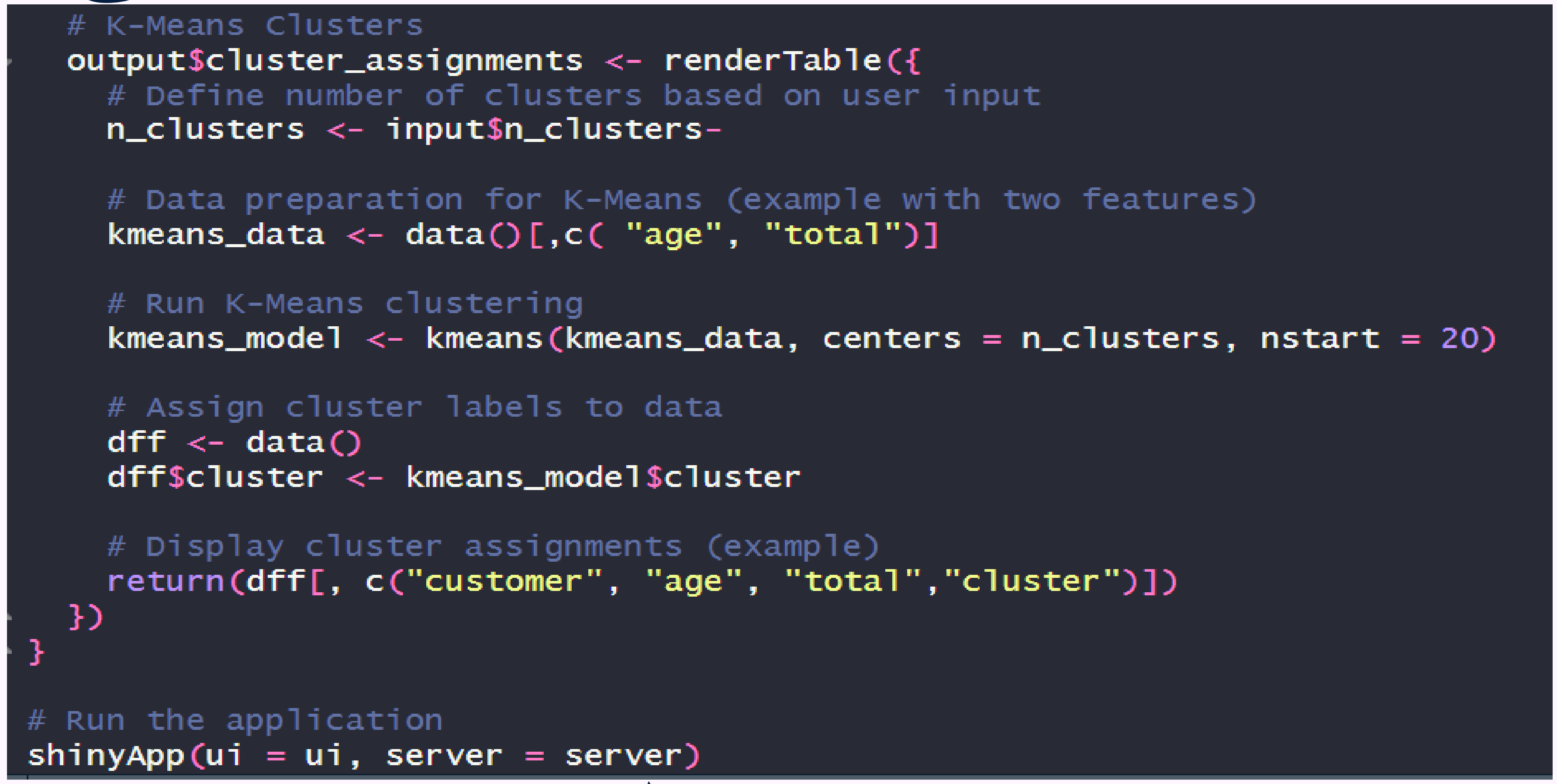
```r
# Generate Association Rules
output$rules_output <- renderPrint({
  req(data())

  # Retrieve minimum support and confidence from user inputs
  min_support <- input$min_support
  min_confidence <- input$min_confidence

  # Convert the data into transactions
  transactions <- as(data(), "transactions")

  # Generate association rules using the Apriori algorithm with user-defined parameters
  apriori_rules <- apriori(transactions, parameter = list(support = min_support, confidence = min

  # Print and inspect the association rules
  print(apriori_rules)
  inspect(apriori_rules)
})
```

this code utilizes Shiny's reactive framework to generate association rules based on user-defined parameters and the uploaded data. The printed and inspected outputs might be used for debugging or further processing within the server logic, but they wouldn't be directly displayed in the Shiny application's UI by default.

Codes

transactions <- as(data(), "transactions")-> This line converts the retrieved data (data()) into a format suitable for the Apriori algorithm. The as function likely performs any necessary transformations on the data to represent it as transactions (sets of items bought together).

Generating Association Rules:apriori_rules <- apriori(transactions, parameter = list(support = min_support, confidence = min_confidence))-> This is the core functionality. It utilizes the apriori function (likely from a package like arules) to generate association rules from the transactions data. The function takes two arguments:transactions.



```
  # K-Means Clusters
  output$cluster_assignments <- renderTable({
    # Define number of clusters based on user input
    n_clusters <- input$n_clusters-

    # Data preparation for K-Means (example with two features)
    kmeans_data <- data()[,c( "age", "total")]

    # Run K-Means clustering
    kmeans_model <- kmeans(kmeans_data, centers = n_clusters, nstart = 20)

    # Assign cluster labels to data
    dff <- data()
    dff$cluster <- kmeans_model$cluster

    # Display cluster assignments (example)
    return(dff[, c("customer", "age", "total","cluster")])
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

this code performs K-Means clustering based on user-defined features and number of clusters. then assigns cluster labels to each data point and displays these assignments along with potentially relevant data (like customer ID, age, and total spending) in a table within the Shiny application.

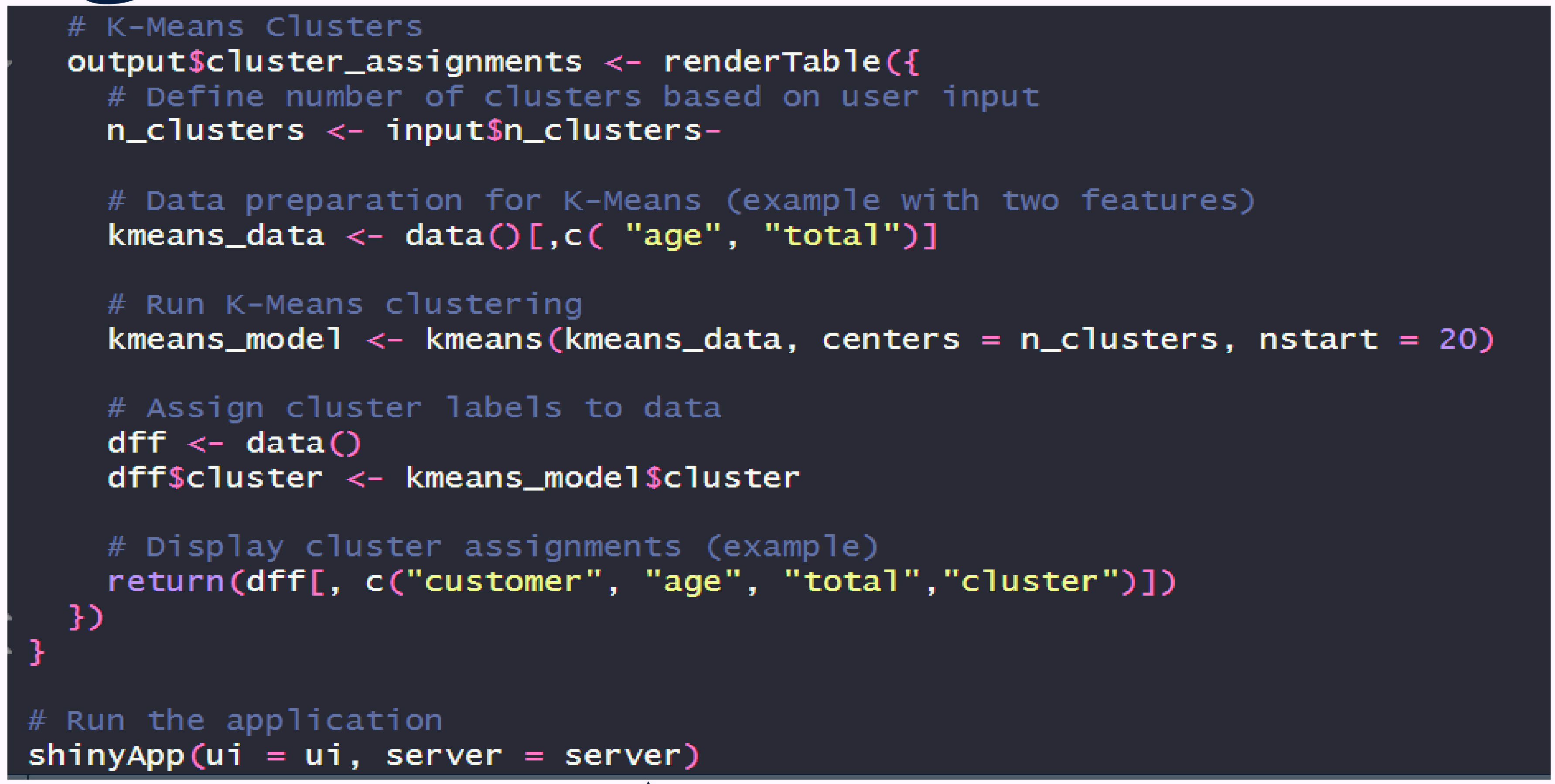
Codes

Printing and Inspecting Results:print(apriori_rules)->

This simply prints the generated association rules to the console (likely for debugging purposes).

inspect(apriori_rules)->

This function, likely from a development package, provides a more detailed inspection of the generated association rules, possibly opening them in a separate window for examination



```
  # K-Means Clusters
  output$cluster_assignments <- renderTable({
    # Define number of clusters based on user input
    n_clusters <- input$n_clusters-

    # Data preparation for K-Means (example with two features)
    kmeans_data <- data()[,c( "age", "total")]

    # Run K-Means clustering
    kmeans_model <- kmeans(kmeans_data, centers = n_clusters, nstart = 20)

    # Assign cluster labels to data
    dff <- data()
    dff$cluster <- kmeans_model$cluster

    # Display cluster assignments (example)
    return(dff[, c("customer", "age", "total","cluster")])
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

this code performs K-Means clustering based on user-defined features and number of clusters. then assigns cluster labels to each data point and displays these assignments along with potentially relevant data (like customer ID, age, and total spending) in a table within the Shiny application.

Thank You