**Exp-1**

**Downloading and installing Hadoop on Ubuntu, Understanding different Hadoop modes, Startup scripts, Configuration files**

**Aim:**

To successfully install, configure, and run Hadoop on a local system using a single-node setup.

**Procedure:**

1. **Install Java and SSH:**
   * Update your package lists and install OpenJDK 8 and SSH.

sudo apt update

sudo apt install openjdk-8-jdk

java -version # Verify Java installation

sudo apt install ssh

1. **Create Hadoop User:**
   * Add a dedicated user for Hadoop and generate SSH keys for passwordless SSH.

sudo adduser hadoop

su - hadoop # Switch to Hadoop user

ssh-keygen -t rsa

cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys

chmod 640 ~/.ssh/authorized\_keys

ssh localhost # Test SSH connection to localhost

1. **Download and Install Hadoop:**
   * Download the latest Hadoop version (3.3.6), extract the tarball, and move it to the desired location.

wget https://downloads.apache.org/hadoop/common/hadoop-3.3.6/hadoop-3.3.6.tar.gz

tar -xvzf hadoop-3.3.6.tar.gz

mv hadoop-3.3.6 hadoop

1. **Configure Environment Variables:**
   * **Update. bashrc to include Hadoop and Java paths.**

nano ~/.bashrc

# Add the following lines at the end

export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64

export HADOOP\_HOME=$HOME/hadoop

export PATH=$PATH:$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin

source ~/.bashrc # Apply changes

1. **Edit Hadoop Configuration Files:**
   * Modify configuration files to set up the necessary Hadoop directories and services.
   * **core-site.xml:**

nano $HADOOP\_HOME/etc/hadoop/core-site.xml

# Add between <configuration></configuration>:

<property>

<name>fs.defaultFS</name>

<value>hdfs://localhost:9000</value>

</property>

* + **hdfs-site.xml:**

nano $HADOOP\_HOME/etc/hadoop/hdfs-site.xml

**Add:**

<property>

<name>dfs.replication</name>

<value>1</value>

</property>

<property>

<name>dfs.namenode.name.dir</name>

<value>file:///home/hadoop/hadoopdata/hdfs/namenode</value>

</property>

<property>

<name>dfs.datanode.data.dir</name>

<value>file:///home/hadoop/hadoopdata/hdfs/datanode</value>

</property>

* + **mapred-site.xml:**

cp $HADOOP\_HOME/etc/hadoop/mapred-site.xml.template $HADOOP\_HOME/etc/hadoop/mapred-site.xml

nano $HADOOP\_HOME/etc/hadoop/mapred-site.xml

**Add:**

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

* + **yarn-site.xml:**

nano $HADOOP\_HOME/etc/hadoop/yarn-site.xml

**Add:**

<property>

<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

</property>

1. **Format the NameNode:**
   * Format the HDFS NameNode.

hdfs namenode -format

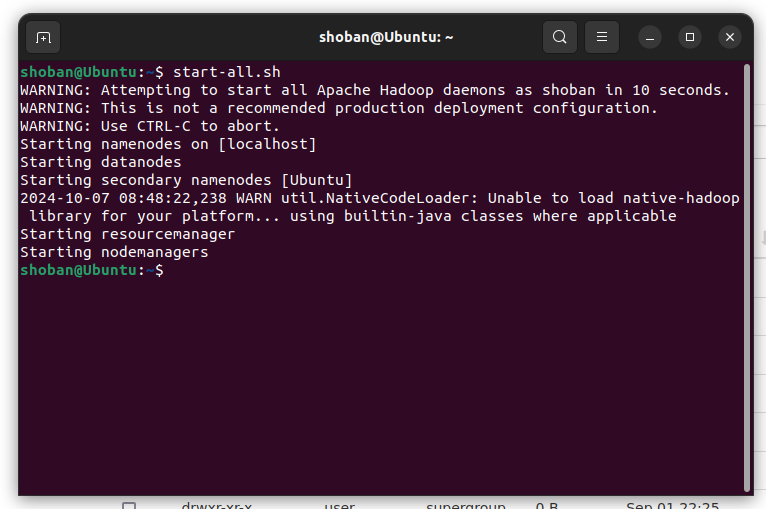
1. **Start Hadoop:**
   * Start Hadoop services (NameNode, DataNode, ResourceManager, and NodeManager).

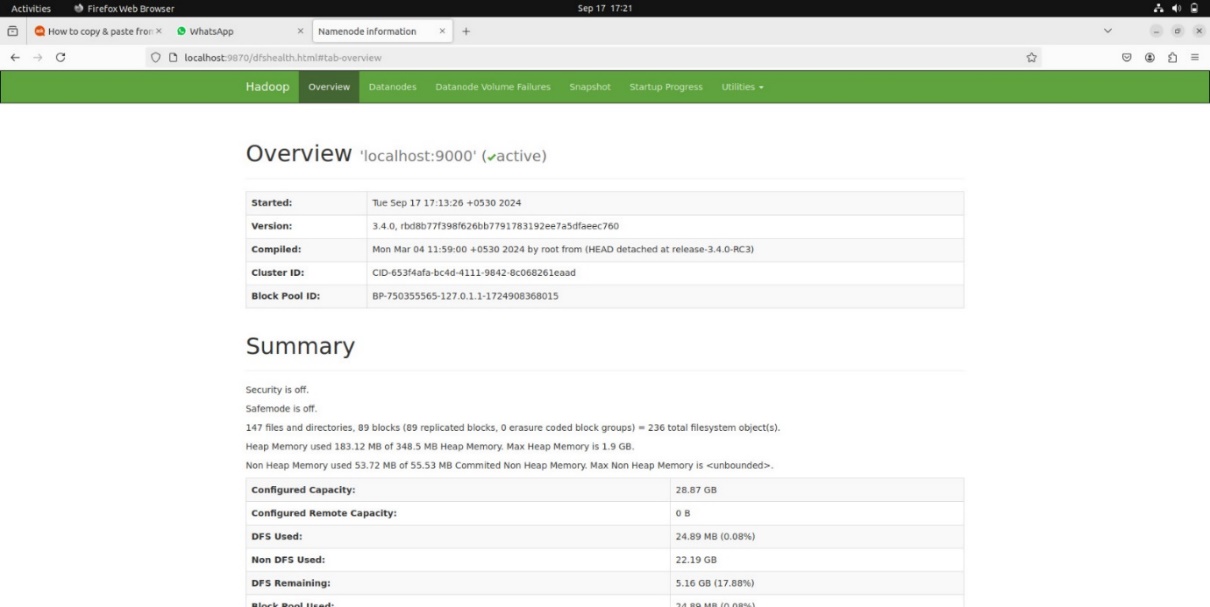
start-all.sh

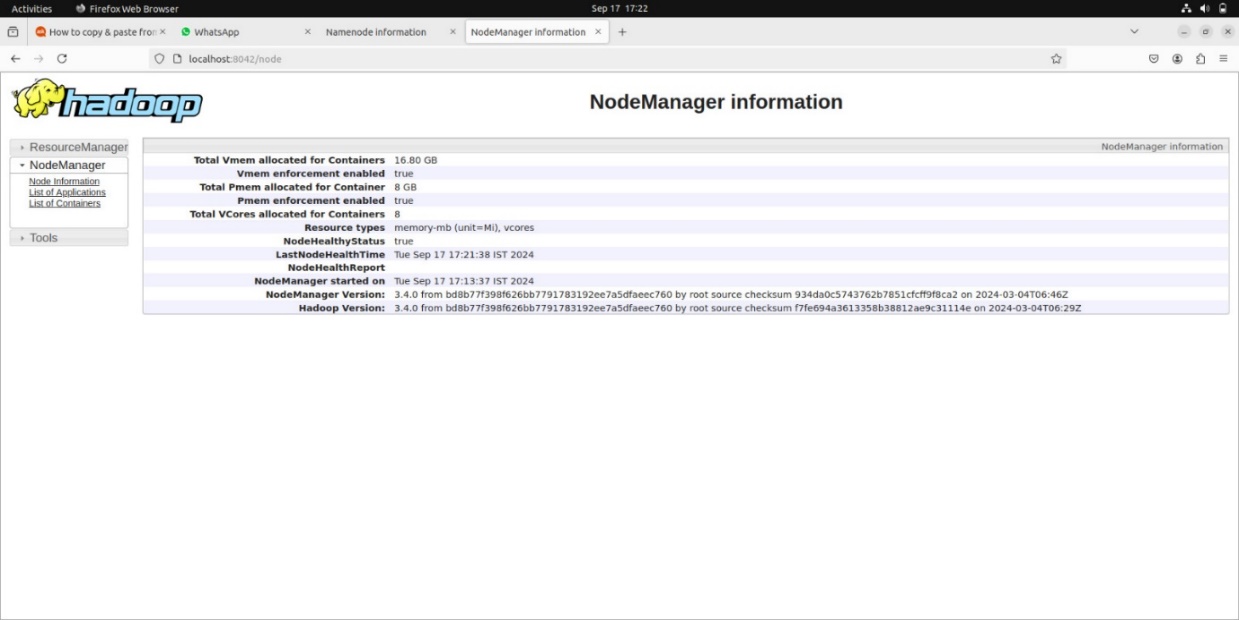
jps # Verify running services

1. **Access Web Interfaces:**
   * Verify that Hadoop is running by accessing the following URLs:
     + **NameNode:** <http://localhost:9870>
     + **Resource Manager:** <http://localhost:8088>
2. **Stop Hadoop Cluster:**
   * Stop all Hadoop services.

stop-all.sh







**RESULT:**

The step-by-step installation and configuration of Hadoop on Ubuntu system have been successfully completed.

**Expt-2**

**Run a basic Word Count Map Reduce program to understand Map Reduce Paradigm.**

**AIM:**

To run a basic Word Count MapReduce program using Hadoop.

**PROCEDURE:**

1. **Create Data File**:

nano word\_count\_data.txt

**Example content for word\_count\_data.txt:**

Hadoop is a framework that allows for distributed processing of large data sets.

1. **Mapper Program (mapper.py)**:

import sys

for line in sys.stdin:

line = line.strip()

words = line.split()

for word in words:

print(f'{word}\t1')

1. **Reducer Program (reducer.py)**:

import sys

current\_word = None

current\_count = 0

word = None

for line in sys.stdin:

line = line.strip()

word, count = line.split('\t', 1)

try:

count = int(count)

except ValueError:

continue

if current\_word == word:

current\_count += count

else:

if current\_word:

print(f'{current\_word}\t{current\_count}')

current\_count = count

current\_word = word

if current\_word == word:

print(f'{current\_word}\t{current\_count}')

1. **Set Hadoop Environment**:

hdfs dfs -mkdir /word\_count\_input

hdfs dfs -copyFromLocal word\_count\_data.txt /word\_count\_input

1. **Run Word Count Program**:

hadoop jar $HADOOP\_HOME/share/hadoop/tools/lib/hadoop-streaming-\*.jar \

-input /word\_count\_input/word\_count\_data.txt \

-output /word\_count\_output \

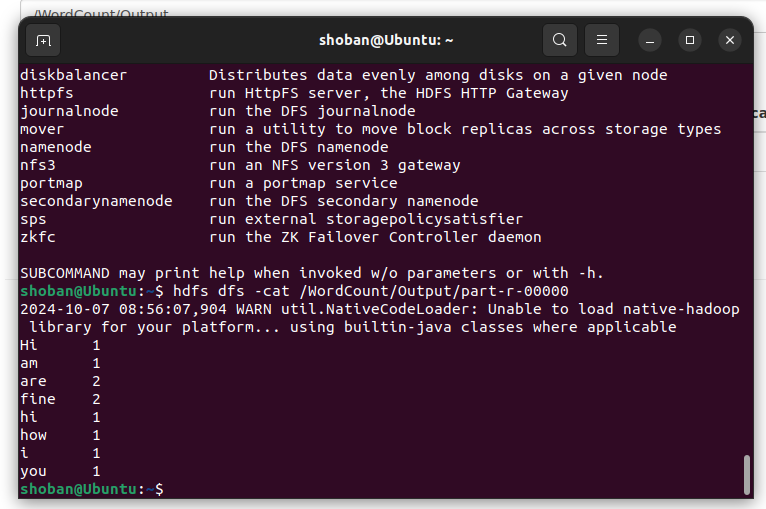
-mapper mapper.py \

-reducer reducer.py

1. **Check Output**:

hdfs dfs -cat /word\_count\_output/part-00000

**OUTPUT:**



**RESULT:**

Thus, the program for basic Word Count Map Reduce has been executed successfully.

**Expt-3**

**Map Reduce program to process a weather dataset.**

**AIM:**

To implement MapReduce program to process a weather dataset.

**PROCEDURE:**

1. **Create Weather Dataset**:

nano weather\_data.txt

**Example content:**

20220101 30.5

20220102 29.8

1. **Mapper Program (mapper.py)**:

#!/usr/bin/env python3

import sys

for line in sys.stdin:

line = line.strip()

month = line[4:6] # Extracting month

temp = line[7:11] # Extracting temperature

print(f'{month}\t{temp}')

1. **Reducer Program (reducer.py)**:

#!/usr/bin/env python3

import sys

current\_month = None

current\_max\_temp = -float('inf')

for line in sys.stdin:

line = line.strip()

month, temp = line.split('\t')

try:

temp = float(temp)

except ValueError:

continue

if current\_month == month:

current\_max\_temp = max(current\_max\_temp, temp)

else:

if current\_month:

print(f'{current\_month}\t{current\_max\_temp}')

current\_month = month

current\_max\_temp = temp

if current\_month == month:

print(f'{current\_month}\t{current\_max\_temp}')

1. **Run the Program**:

hdfs dfs -mkdir /weatherdata

hdfs dfs -copyFromLocal weather\_data.txt /weatherdata

hadoop jar $HADOOP\_HOME/share/hadoop/tools/lib/hadoop-streaming-\*.jar \

-input /weatherdata/weather\_data.txt \

-output /weatherdata/output \

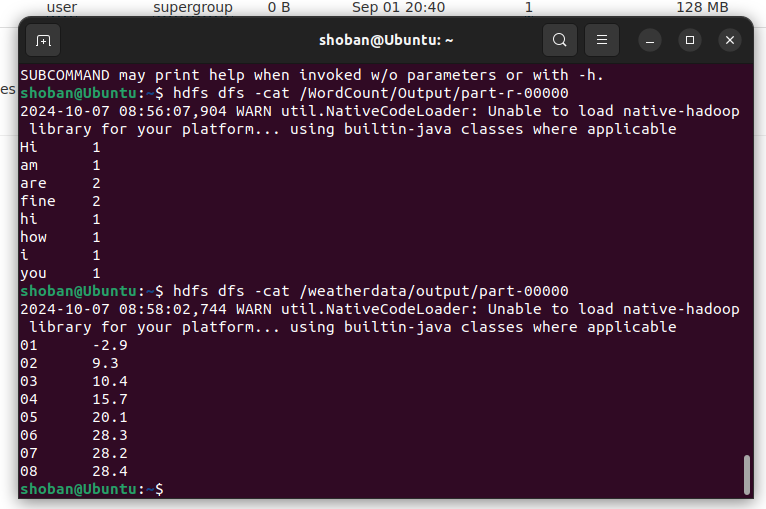
-mapper mapper.py \

-reducer reducer.py

1. **Check Output**:

hdfs dfs -cat /weatherdata/output/part-00000

**OUTPUT:**



**RESULT:**

Thus, the program for weather dataset using Map Reduce has been executed successfully.

**Expt-4**

**Create UDF (User Defined Functions) in Apache Pig and execute it in MapReduce / HDFS mode**

**AIM:**

To create UDF in Apache Pig and execute it in MapReduce/HDFS mode.

**Procedure:**

**Step 1: Install and Configure Apache Pig**

1. **Download Apache Pig**:

**Download the latest version of Pig from the official website:**

wget https://dlcdn.apache.org/pig/pig-0.16.0/pig-0.16.0.tar.gz

1. **Extract Pig**:

tar xvzf pig-0.16.0.tar.gz

1. **Move Pig Directory**:

**Move the extracted Pig files to a dedicated folder:**

sudo mv pig-0.16.0 /usr/local/pig

1. **Set Environment Variables**:

**Edit the .bashrc file to set up Pig environment variables:**

nano ~/.bashrc

**Append the following lines:**

export PIG\_HOME=/usr/local/pig

export PATH=$PATH:$PIG\_HOME/bin

export PIG\_CLASSPATH=$HADOOP\_HOME/conf

**Apply the changes:**

source ~/.bashrc

1. **Verify Pig Installation**:

**Run the following command to verify if Pig has been installed correctly:**

pig -version

**Step 2: Create Sample Data for the Pig Job**

1. **Create a Sample Data File**: Create a sample text file (sample.txt) with some dummy data:

nano sample.txt

**Add the following content:**

1,John

2,Jane

3,Joe

4,Emma

1. **Upload the Data File to HDFS**: Upload the sample file to Hadoop's distributed file system (HDFS):

hdfs dfs -mkdir /piginput

hdfs dfs -put sample.txt /piginput

**Step 3: Write Pig Script for the UDF**

1. **Create the Pig Script**:

**Create a new Pig script (demo\_pig.pig):**

nano demo\_pig.pig

Write the following code in the script to load and display the data:

pig

**-- Load data from HDFS**

data = LOAD '/piginput/sample.txt' USING PigStorage(',') AS (id:int, name:chararray);

**-- Display the loaded data**

DUMP data;

**Step 4: Write the UDF in Python**

1. **Create the Python UDF**:

**Create a Python file (uppercase\_udf.py) to convert text to uppercase:**

nano uppercase\_udf.py

def uppercase(text):

return text.upper()

if \_\_name\_\_ == "\_\_main\_\_":

import sys

for line in sys.stdin:

line = line.strip()

print(uppercase(line))

1. **Upload the Python UDF to HDFS**:

**Upload the UDF to HDFS:**

hdfs dfs -mkdir /udfs

hdfs dfs -put uppercase\_udf.py /udfs

**Step 5: Update Pig Script to Use UDF**

1. **Modify the Pig Script to Include UDF**:

**Edit the demo\_pig.pig script to register the UDF and process the data:**

nano demo\_pig.pig

**Modify the script as follows:**

pig

**-- Register the Python UDF script**

REGISTER '/udfs/uppercase\_udf.py' USING jython AS myudf;

**-- Load data from HDFS**

data = LOAD '/piginput/sample.txt' USING PigStorage(',') AS (id:int, name:chararray)

**-- Apply UDF to convert names to uppercase**

uppercased\_data = FOREACH data GENERATE myudf.uppercase(name);

**-- Display the transformed data**

DUMP uppercased\_data;

**Step 6: Run the Pig Script**

1. **Run the Pig Script**:

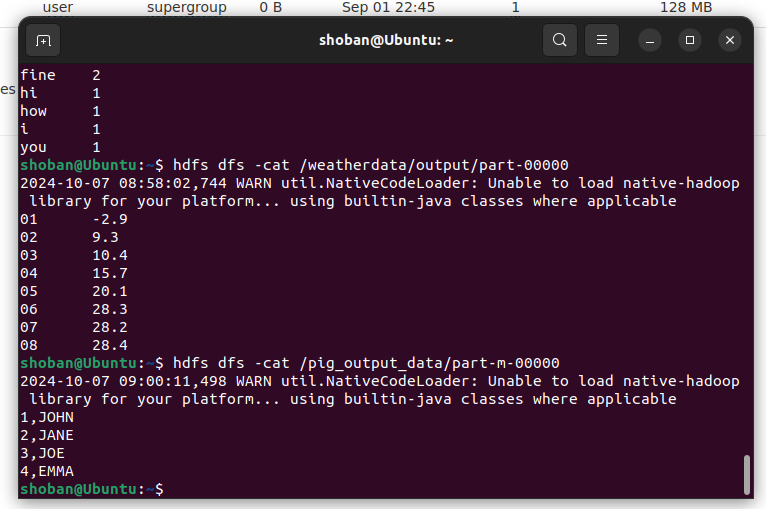
**Run the Pig script using the following command:**

pig -x mapreduce demo\_pig.pig

1. **View Output**

hdfs dfs -cat /pigoutput/part-m-00000

**OUTPUT:**



**RESULT:**

Thus, UDF in Apache Pig has been created and executed in MapReduce/HDFS mode successfully.

**Ex No 5**

**Create tables in Hive and write queries to access the data in the table**

**AIM:**

To create tables in Hive and write queries to access the data in the table.

**PROCEDURE:**

**Step 1: Download and Install Hive**

1. **Download Hive**:

**Download Hive from the official website:**

wget https://downloads.apache.org/hive/hive-3.1.2/apache-hive-3.1.2-bin.tar.gz

1. **Extract Hive**:

tar -xvf apache-hive-3.1.2-bin.tar.gz

1. **Move Hive Directory**:

sudo mv apache-hive-3.1.2-bin /usr/local/hive

1. **Set Hive Environment Variables**:

**Edit .bashrc to configure Hive:**

nano ~/.bashrc

**Add the following lines:**

export HIVE\_HOME=/usr/local/hive

export PATH=$PATH:$HIVE\_HOME/bin

**Apply the changes:**

source ~/.bashrc

1. **Configure Hive**:

**Configure Hive to use MySQL as its metastore by editing the Hive configuration file (hive-site.xml):**

nano $HIVE\_HOME/conf/hive-site.xml

**Add the following configuration for MySQL connection:**

<property>

<name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:mysql://localhost/metastore</value>

</property>

<property>

<name>javax.jdo.option.ConnectionDriverName</name>

<value>com.mysql.jdbc.Driver</value>

</property>

<property>

<name>javax.jdo.option.ConnectionUserName</name>

<value>root</value>

</property>

<property>

<name>javax.jdo.option.ConnectionPassword</name>

<value>password</value>

</property>

1. **Start Hive**:

**Once everything is configured, start Hive by simply typing:**

hive

**Step 2: Create a Database and Table in Hive**

1. **Create a Database**:

**In the Hive terminal, create a new database:**

CREATE DATABASE financials;

1. **Use the Database**:

USE financials;

1. **Create a Table**:

**Create a table to store financial data:**

CREATE TABLE finance\_table (

id INT,

name STRING

)

1. **Insert Data into the Table**:

**Insert sample data into the finance\_table:**

INSERT INTO TABLE finance\_table VALUES (1, 'Alice'), (2, 'Bob'), (3, 'Charlie');

**Step 3: Store the Output in HDFS**

1. **Create a Partitioned Table**:

**For optimized storage, create a partitioned table by year:**

CREATE TABLE partitioned\_finance\_table (

id INT,

name STRING

)

PARTITIONED BY (year INT)

1. **Insert Data into the Partitioned Table**:

INSERT INTO partitioned\_finance\_table PARTITION (year=2023) VALUES (1, 'Alice'), (2, 'Bob');

INSERT INTO partitioned\_finance\_table PARTITION (year=2024) VALUES (3, 'Charlie');

1. **Create a Bucketed Table**:

**Create a bucketed table to improve query performance:**

CREATE TABLE bucketed\_finance\_table (

id INT,

name STRING

)

CLUSTERED BY (id) INTO 4 BUCKETS

1. **Insert Data into the Bucketed Table**:

INSERT INTO TABLE bucketed\_finance\_table VALUES (1, 'Alice'), (2, 'Bob'), (3, 'Charlie');

**Step 4: View the Output in HDFS**

1. **Create an ORC Table**:

**Use ORC (Optimized Row Columnar) format for efficient storage:**

CREATE TABLE orc\_finance\_table (

id INT,

name STRING

)

1. **Insert Data into the ORC Table**:

INSERT INTO TABLE orc\_finance\_table SELECT \* FROM finance\_table;

1. **View the Output in HDFS**:

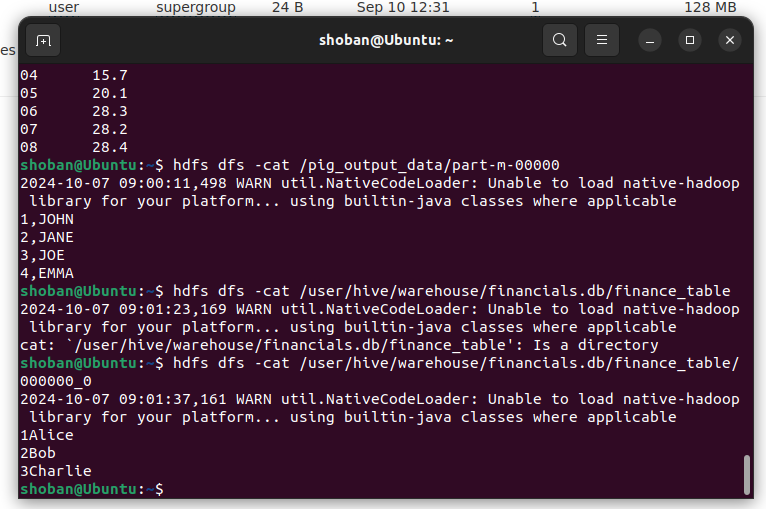
**You can view the output by navigating to the HDFS directory where Hive stores the data. Use the following command to view the stored data:**

hdfs dfs -ls /user/hive/warehouse/financials.db/finance\_table

**To view the contents of the ORC table:**

hdfs dfs -cat /user/hive/warehouse/financials.db/orc\_finance\_table/000000\_0

**OUTPUT:**



**RESULT:**

Thus, to create tables in Hive and write queries to access the data in the table was completed successfully.

**Ex No 6**

**Import a JSON file from the command line. Apply the following actions with the data present in the JSON file where, projection, aggregation, remove, count, limit, skip and sort**

**AIM:**

To import a JSON file from the command line and apply the following actions with the data present in the JSON file where, projection, aggregation, remove, count, limit, skip and sort using jq tool.

**PROCEDURE:**

* Create a json file ‘employees.json’ and provide data in it.
* Open the command prompt.
* Navigate to the folder where employees.json is stored.
* Load and view the JSON data with jq.
* Use the jq commands for projection, aggregation, removal, counting, limiting, and sorting operations.

**employees.json:**

[

{

"id": 1,

"name": "Alice Johnson",

"department": "Engineering",

"age": 29,

"salary": 70000

},

{

"id": 2,

"name": "Bob Smith",

"department": "Marketing",

"age": 35,

"salary": 55000

},

{

"id": 3,

"name": "Charlie Davis",

"department": "Engineering",

"age": 25,

"salary": 60000

},

{

"id": 4,

"name": "Dana Lee",

"department": "Human Resources",

"age": 40,

"salary": 65000

},

{

"id": 5,

"name": "Eve Martinez",

"department": "Finance",

"age": 45,

"salary": 75000

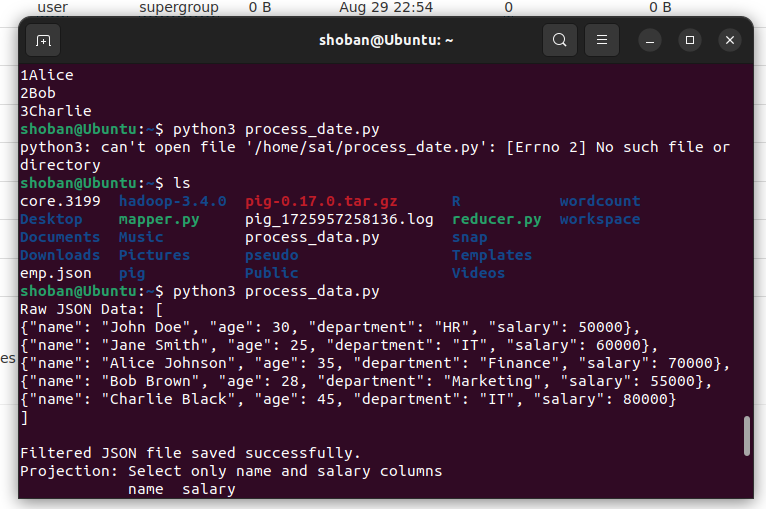
}

]

**OUTPUT:**

**Running jq queries:**

1. **Projection:**



**II. Aggregation:**

A purple screen with white text

Description automatically generated

**III. Count:**

A screenshot of a computer

Description automatically generated

**IV. Remove:**

A black and white screen with white text

Description automatically generated

1. **Limit:**

A screenshot of a computer

Description automatically generated

1. **Skip:**

A screenshot of a computer

Description automatically generated

1. **Sort:**

A screenshot of a computer

Description automatically generated

**RESULT:**

Thus to import a JSON file from the command line and apply the following actions with the data present in the JSON file where, projection, aggregation, remove, count, limit, skip and sort using jq tool is completed successfully.

**Ex No 7**

**Implement Linear and Logistic Regression in R**

**AIM:**

To Implement Linear and Logistic Regression using R

**PROCEDURE:**

* Collect and load the dataset from sources like CSV files or databases.
* Clean and preprocess the data, including handling missing values and encoding categorical variables.
* Split the dataset into training and testing sets to evaluate model performance.
* Normalize or standardize the features to ensure consistent scaling. 5.Choose the appropriate model: Linear Regression for continuous outcomes.
* Train the model on the training data using the `fit` method.
* Make predictions on the testing data using the `predict` method.
* Evaluate the model using metrics like Mean Squared Error (MSE) for Linear Regression or accuracy and confusion matrix for Logistic Regression.
* Visualize the results with plots, such as scatter plots for Linear Regression or decision boundaries for Logistic Regression.
* Fine-tune the model by adjusting hyperparameters or applying regularization Techniques.

**CODE:**

**LinearRegression.R:**

# Sample data

heights <- c(150, 160, 165, 170, 175, 180, 185)

weights <- c(55, 60, 62, 68, 70, 75, 80)

# Create a data frame

data <- data.frame(heights, weights)

# Fit a linear regression model

linear\_model <- lm(weights ~ heights, data = data)

# Print the summary of the model

print(summary(linear\_model))

# Plotting the data and regression line

plot(data$heights, data$weights,

main = "Linear Regression: Weight vs. Height",

xlab = "Height (cm)",

ylab = "Weight (kg)",

pch = 19, col = "blue")

# Add regression line

abline(linear\_model, col = "red", lwd = 2)

**LogisticRegression.R:**

# Load the dataset

data(mtcars)

# Convert 'am' to a factor (categorical variable)

mtcars$am <- factor(mtcars$am, levels = c(0, 1), labels = c("Automatic", "Manual"))

# Fit a logistic regression model

logistic\_model <- glm(am ~ mpg, data = mtcars, family = binomial)

# Print the summary of the model

print(summary(logistic\_model))

# Predict probabilities for the logistic model

predicted\_probs <- predict(logistic\_model, type = "response")

# Display the predicted probabilities

print(predicted\_probs)

# Plotting the data and logistic regression curve

plot(mtcars$mpg, as.numeric(mtcars$am) - 1,

main = "Logistic Regression: Transmission vs. MPG",

xlab = "Miles Per Gallon (mpg)",

ylab = "Probability of Manual Transmission",

pch = 19, col = "blue")

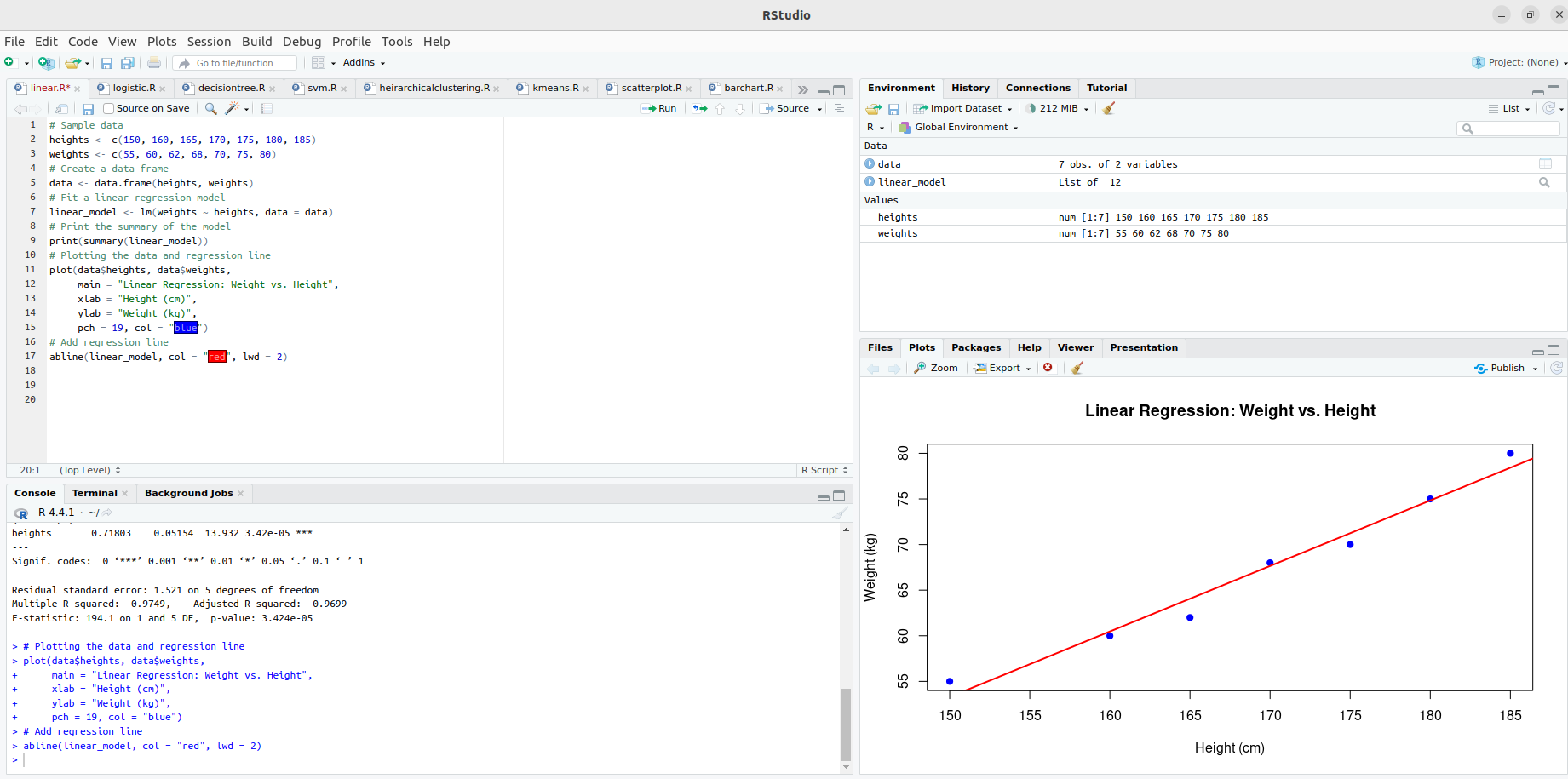
# Add the logistic regression curve

curve(predict(logistic\_model, data.frame(mpg = x), type = "response"),

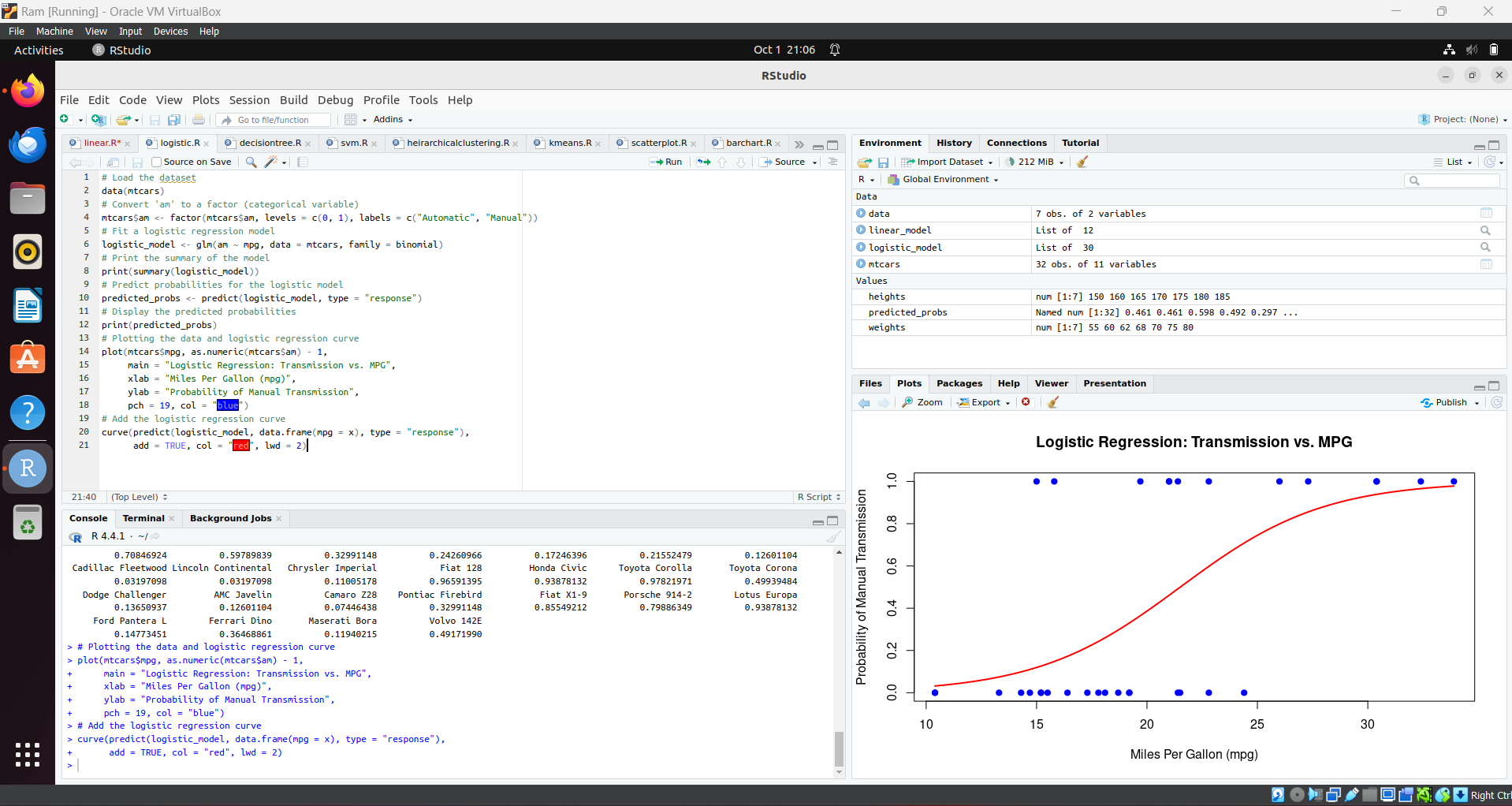
add = TRUE, col = "red", lwd = 2)

**OUTPUT:**

**Linear Regression:**



**Logistic Regression:**

****

**RESULT:**

Thus to Implement Linear and Logistic Regression using R has been successfully executed.

**Ex No 8**

**Implement SVM/Decision tree classification techniques**

**AIM:**

To Implement SVM/Decision tree classification techniques using R.

**PROCEDURE:**

* Collect and load the dataset from sources like CSV files or databases.
* Clean and preprocess the data, including handling missing values and encoding categorical variables.
* Split the dataset into training and testing sets to evaluate model performance.
* Normalize or standardize the features, especially for SVM, to ensure consistent scaling.
* Choose the appropriate model: SVM for margin-based classification, Decision Tree for rule-based classification.
* Train the model on the training data using the ‘fit’ method.
* Make predictions on the testing data using the ’predict’ method.
* Evaluate the model using metrics like accuracy, confusion matrix, precision, and recall.
* Visualize the results with plots, such as decision boundaries for SVM or tree structures for Decision Trees.
* Fine-tune the model by adjusting hyperparameters like `C` for SVM or

`max\_depth` for Decision Trees.

**CODE:**

**SVM.R:**

# Install and load the e1071 package (if not already installed)

install.packages("e1071")

library(e1071)

# Load the iris dataset

data(iris)

# Inspect the first few rows of the dataset

head(iris)

# Split the data into training (70%) and testing (30%) sets

set.seed(123) # For reproducibility

sample\_indices <- sample(1:nrow(iris), 0.7 \* nrow(iris))

train\_data <- iris[sample\_indices, ]

test\_data <- iris[-sample\_indices, ]

# Fit the SVM model

svm\_model <- svm(Species ~ ., data = train\_data, kernel = "radial")

# Print the summary of the model

summary(svm\_model)

# Predict the test set

predictions <- predict(svm\_model, newdata = test\_data)

# Evaluate the model's performance

confusion\_matrix <- table(Predicted = predictions, Actual = test\_data$Species)

print(confusion\_matrix)

# Calculate accuracy

accuracy <- sum(diag(confusion\_matrix)) / sum(confusion\_matrix)

cat("Accuracy:", accuracy \* 100, "%\n")

**Decision Tree.R:**

# Install and load the rpart package (if not already installed)

install.packages("rpart")

library(rpart)

# Load the iris dataset

data(iris)

# Split the data into training (70%) and testing (30%) sets

set.seed(123) # For reproducibility

sample\_indices <- sample(1:nrow(iris), 0.7 \* nrow(iris))

train\_data <- iris[sample\_indices, ]

test\_data <- iris[-sample\_indices, ]

# Fit the Decision Tree model

tree\_model <- rpart(Species ~ ., data = train\_data, method = "class")

# Print the summary of the model

summary(tree\_model)

# Plot the Decision Tree

plot(tree\_model)

text(tree\_model, pretty = 0)

# Predict the test set

predictions <- predict(tree\_model, newdata = test\_data, type = "class")

# Evaluate the model's performance

confusion\_matrix <- table(Predicted = predictions, Actual = test\_data$Species)

print(confusion\_matrix)

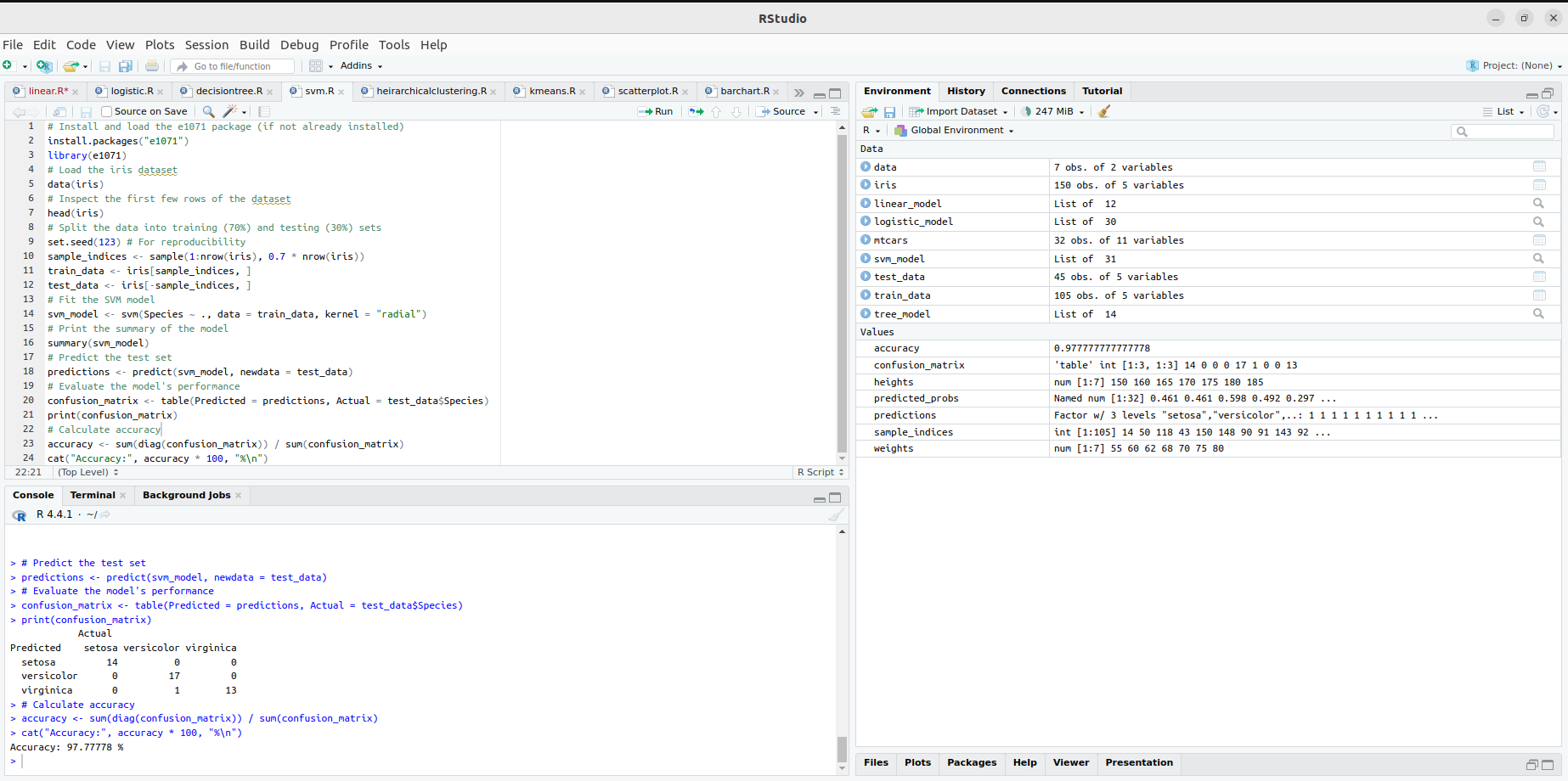
# Calculate accuracy

accuracy <- sum(diag(confusion\_matrix)) / sum(confusion\_matrix)

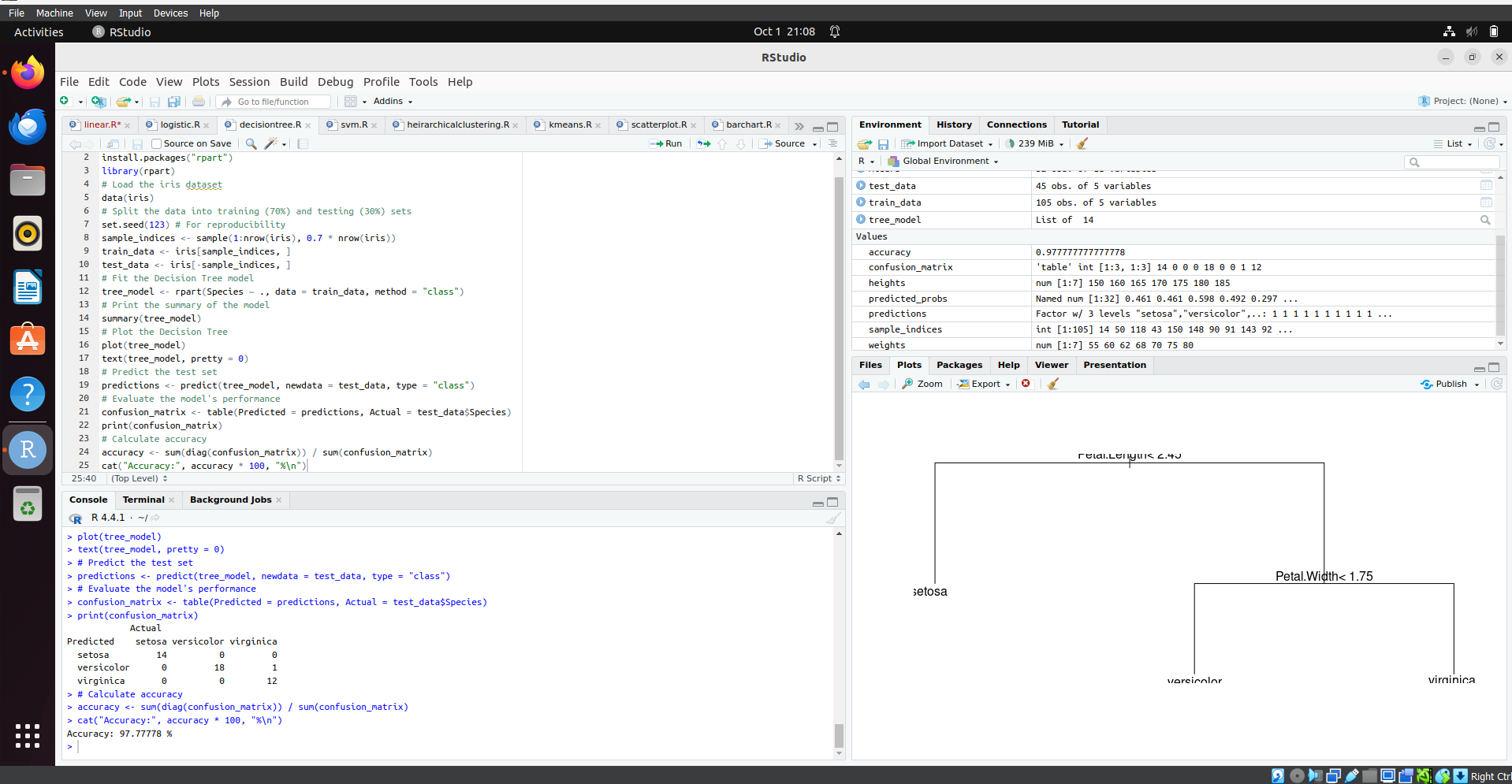
cat("Accuracy:", accuracy \* 100, "%\n")

**OUTPUT:**

**SVM in R:**

****

**Decision tree:**

****

**RESULT:**

Thus, Implement SVM and Decision tree classification techniques has been successfully executed.

**Ex No 9**

**Implement clustering techniques – Hierarchical and K-Means**

**AIM:**

To Implement clustering techniques – Hierarchical and K-Means using R.

**PROCEDURE:**

* Collect and load the dataset from sources like CSV files or databases.
* Clean and preprocess the data, including handling missing values and scaling features.
* Determine the number of clusters (K) for K-Means, or decide on the stopping criterion for Hierarchical Clustering.
* Choose the appropriate clustering algorithm: K-Means for partitioning, Hierarchical for nested clustering.
* Apply the K-Means algorithm using fit\_predict to assign data points to clusters.
* Apply the Hierarchical Clustering algorithm using Agglomerative Clustering for hierarchical clusters.
* Visualize the clusters with scatter plots for K-Means, and dendrograms for Hierarchical Clustering.
* Evaluate clustering performance using metrics like silhouette score or inertia (for K-Means).
* Fine-tune the clustering by adjusting the number of clusters or linkage criteria.
* Interpret the results to understand the structure and relationships within the data.

**CODE:**

**Hierarchical Clustering.R:**

# Load the iris dataset

data(iris)

# Use only the numeric columns for clustering (exclude the Species column)

iris\_data <- iris[, -5]

# Standardize the data

iris\_scaled <- scale(iris\_data)

# Compute the distance matrix

distance\_matrix <- dist(iris\_scaled, method = "euclidean")

# Perform hierarchical clustering using the "complete" linkage method

hc\_complete <- hclust(distance\_matrix, method = "complete")

# Plot the dendrogram

plot(hc\_complete, main = "Hierarchical Clustering Dendrogram", xlab = "", sub = "", cex = 0.6)

# Cut the tree to form 3 clusters

clusters <- cutree(hc\_complete, k = 3)

# Print the cluster memberships

print(clusters)

# Add the clusters to the original dataset

iris$Cluster <- as.factor(clusters)

# Display the first few rows of the updated dataset

head(iris)

**K-Means Clustering.R:**

# Load the iris dataset

data(iris)

# Use only the numeric columns for clustering (exclude the Species column)

iris\_data <- iris[, -5]

# Standardize the data

iris\_scaled <- scale(iris\_data)

# Set the number of clusters

set.seed(123) # For reproducibility

k <- 3 # Number of clusters

# Perform K-Means clustering

kmeans\_result <- kmeans(iris\_scaled, centers = k, nstart = 25)

# Print the K-Means result

print(kmeans\_result)

# Print the cluster centers

print(kmeans\_result$centers)

# Add the cluster assignments to the original dataset

iris$Cluster <- as.factor(kmeans\_result$cluster)

# Display the first few rows of the updated dataset

head(iris)

# Plot the clusters

library(ggplot2)

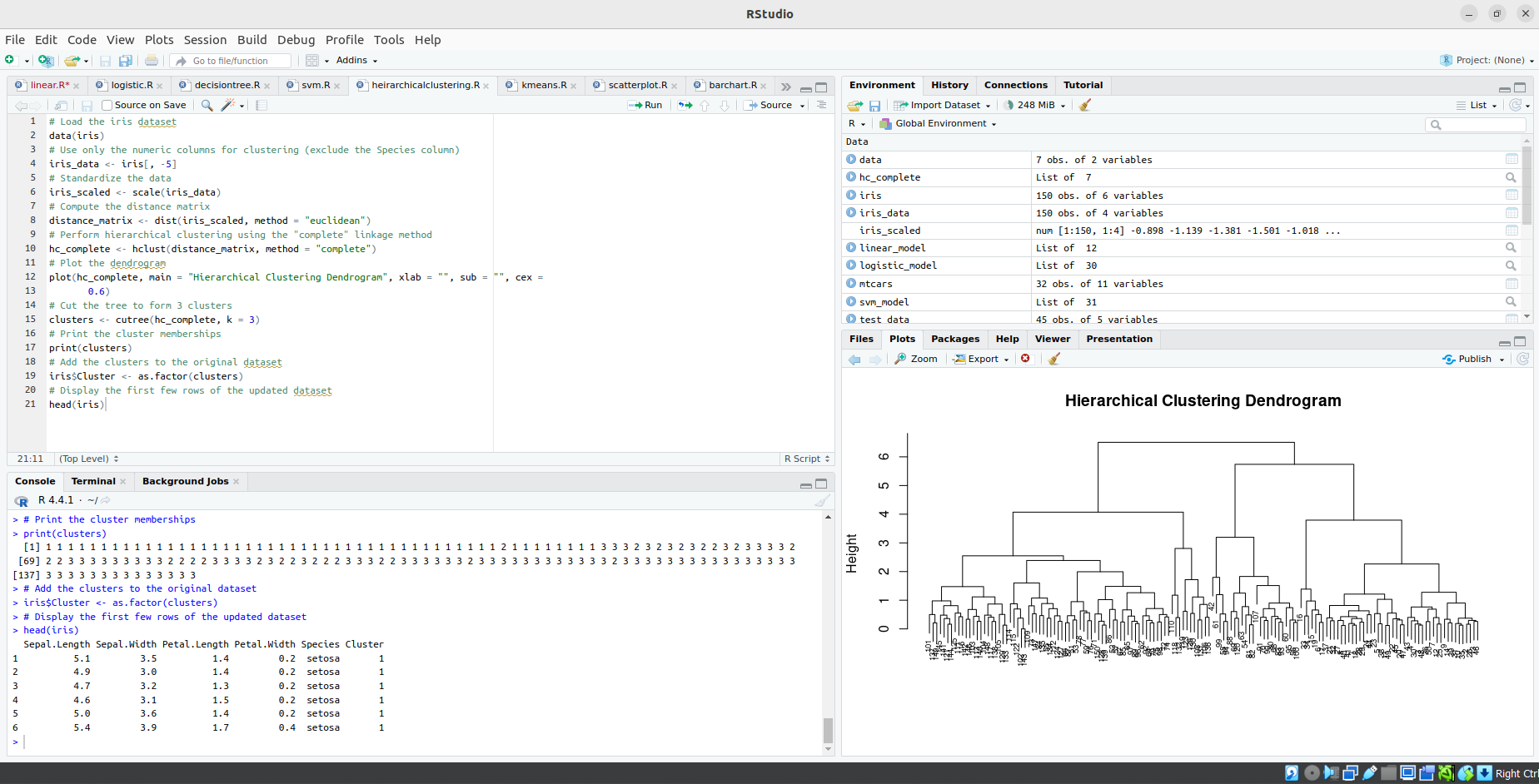
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Cluster)) +

geom\_point(size = 3) +

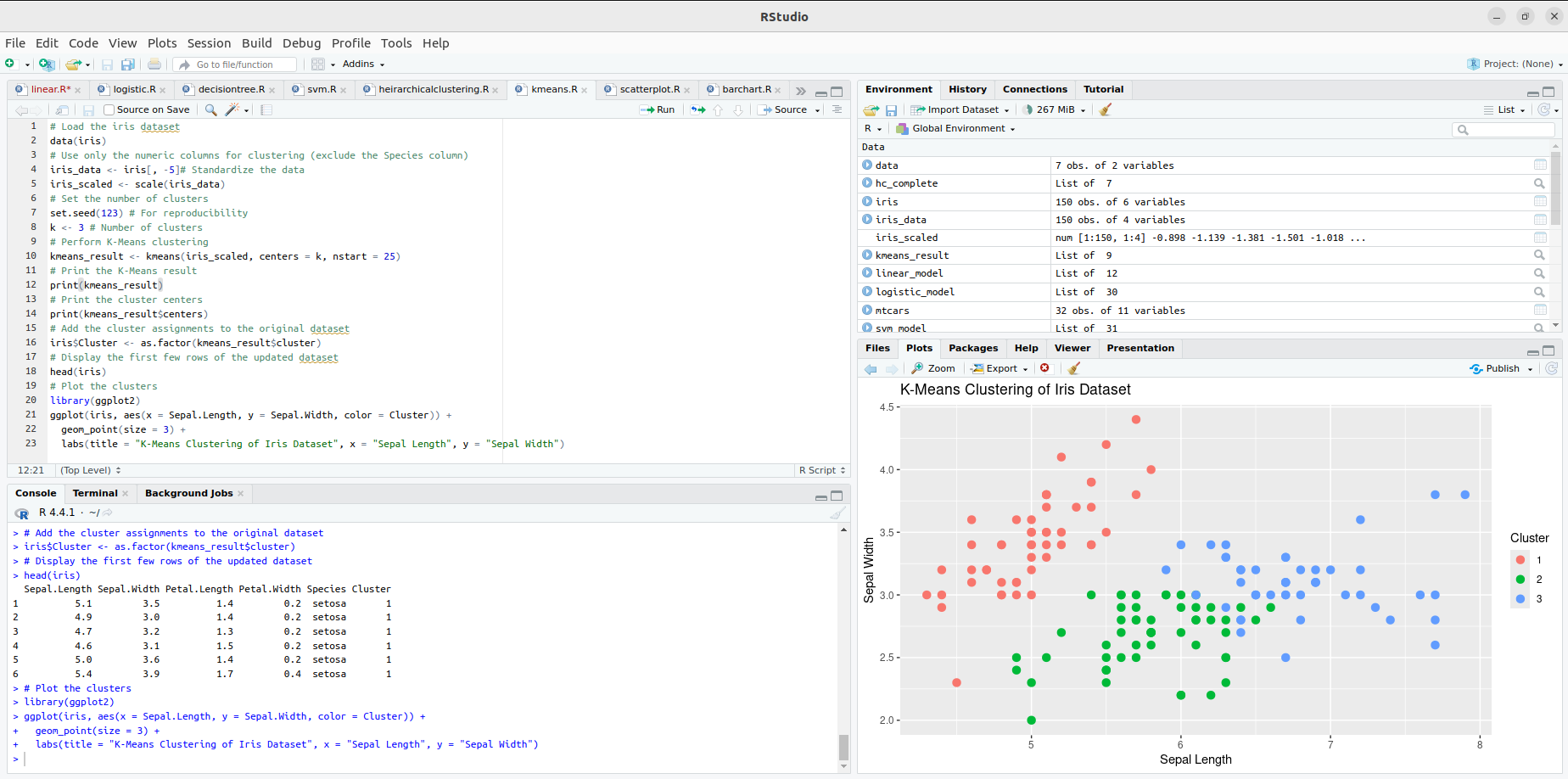
labs(title = "K-Means Clustering of Iris Dataset", x = "Sepal Length", y = "Sepal Width")

**OUTPUT:**

**Hierarchical Clustering:**

****

**K-Means Clustering:**

****

**RESULT:**

Thus, to implement clustering techniques – Hierarchical and K-Means using R has been successfully executed.

**Ex No 10**

**Visualize Data using Any plotting Framework**

**AIM:**

To Visualize Data using Any plotting Frame work using R programming.

**PROCEDURE:**

* Install Plotly using pip install plotly if it’s not already installed.
* Import the necessary libraries: import plotly.express as px and import pandas as pd.
* Load your dataset into a DataFrame using pd.read\_csv() or other data loading methods.
* Explore the dataset to understand its structure, variables, and potential visualizations.
* Choose the appropriate Plotly function (e.g., px.scatter, px.bar,px.line) based on the type of data and the desired plot.
* Define the x and y axes by specifying the columns from the DataFrame.
* Customize the plot by adding titles, labels, color coding, and other plot-specific attributes.
* Add interactive elements like hover data, tooltips, or facet plots for deeper insights.
* Render the plot using fig.show() to display it in a web browser or inline in a notebook.
* Save the plot to an HTML file or as a static image using fig.write\_html() or fig.write\_image().

**CODE:**

**Scatter Plot.R:**

# Install ggplot2 (if not already installed)

install.packages("ggplot2")

# Load the ggplot2 package

library(ggplot2)

# Scatter plot of Sepal.Length vs Sepal.Width, colored by Species

ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +

geom\_point(size = 3) + # Adds points

labs(title = "Scatter Plot of Sepal Dimensions",

x = "Sepal Length (cm)",

y = "Sepal Width (cm)") + # Adds axis labels and title

theme\_minimal() # Applies a minimal theme

**Bar Chart.R:**

# Install ggplot2 (if not already installed)

install.packages("ggplot2")

# Load the ggplot2 package

library(ggplot2)

# Bar plot of Species counts

ggplot(data = iris, aes(x = Species)) +

geom\_bar(fill = "steelblue") + # Adds bars filled with steel blue color

labs(title = "Count of Different Species in Iris Dataset",

x = "Species",

y = "Count") +

theme\_minimal()

**Histogram.R:**

# Install ggplot2 (if not already installed)

install.packages("ggplot2")

# Load the ggplot2 package

library(ggplot2)

# Histogram of Sepal Length

ggplot(data = iris, aes(x = Sepal.Length)) +

geom\_histogram(binwidth = 0.3, fill = "orange", color = "black") + # Adds histogram bars

labs(title = "Histogram of Sepal Length",

x = "Sepal Length (cm)",

y = "Frequency") +

theme\_minimal()

**Box Plot.R:**

# Install ggplot2 (if not already installed)

install.packages("ggplot2")

library(ggplot2)

# Box plot of Sepal Length for each Species

ggplot(data = iris, aes(x = Species, y = Sepal.Length, fill = Species)) +

geom\_boxplot() + # Adds box plot

labs(title = "Box Plot of Sepal Length by Species",

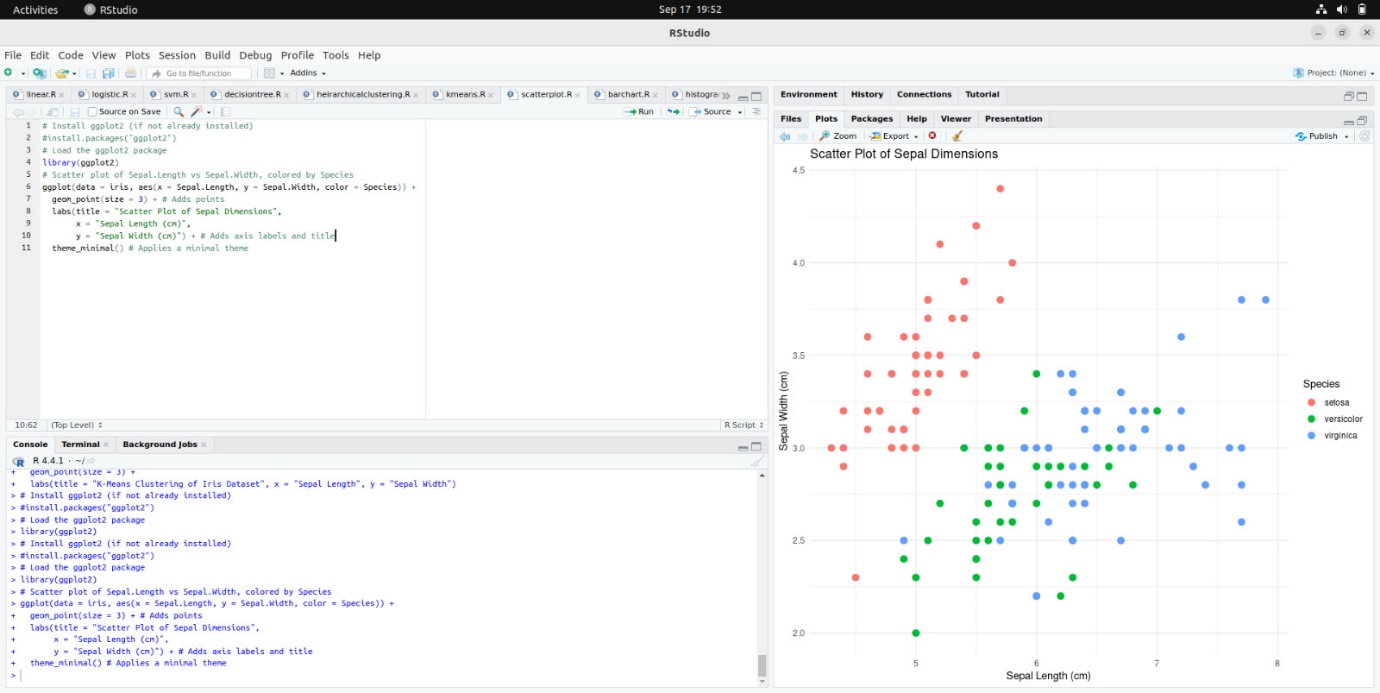
x = "Species",

y = "Sepal Length (cm)") +

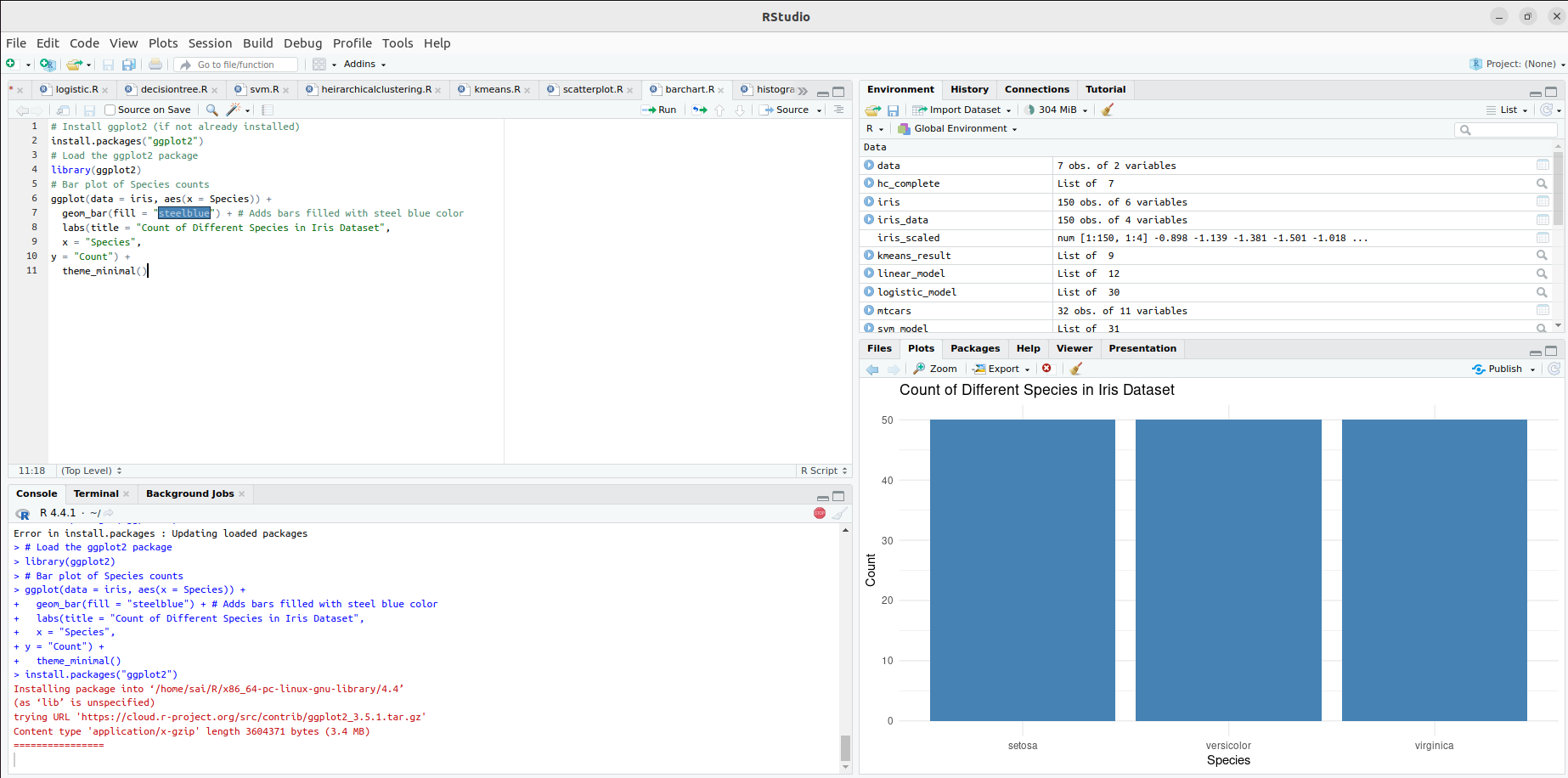
theme\_minimal()

**OUTPUT:**

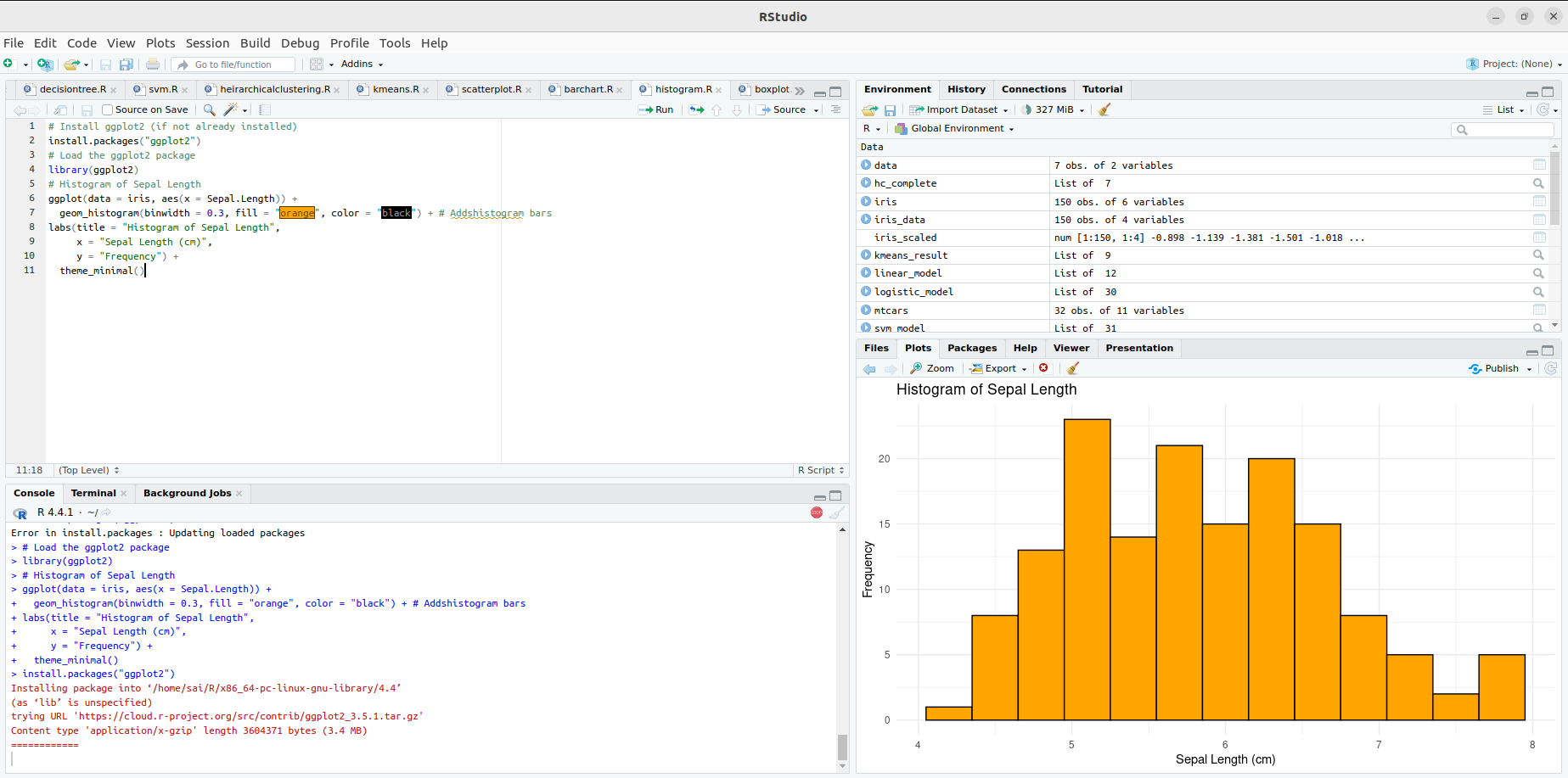
**Scatter Plot:**



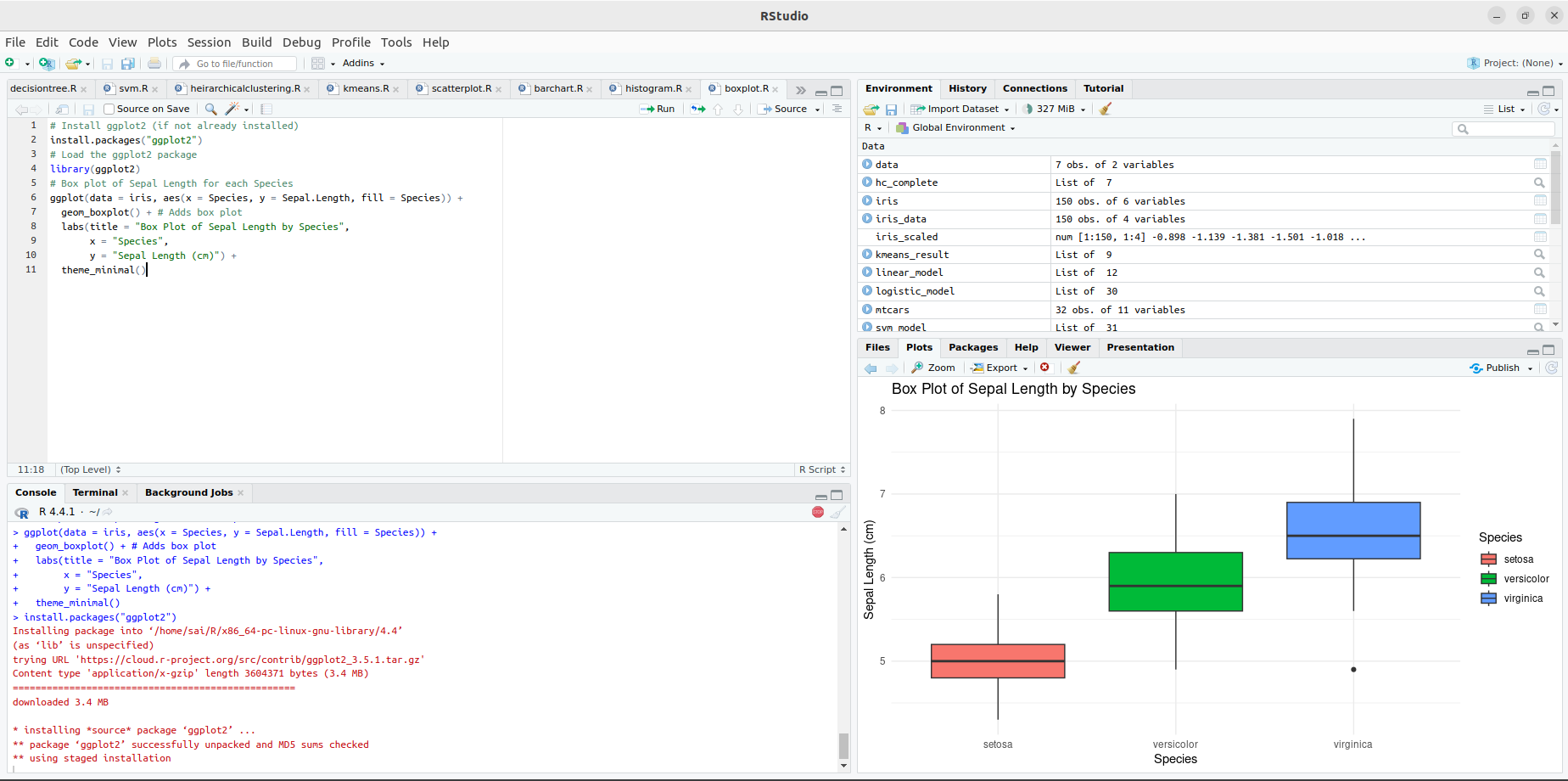
**Bar Chart:**

****

**Histogram:**

****

**Box Plot:**

****

**RESULT:**

Thus, Visualizing Data using any plotting framework using R programming has been successfully executed.Bottom of Form