Exercise 7:

Correlationand covariance	a. Find the correlation matrix. b. Plot the correlation plot on Dataset and visualize giving an overview of relationships among data on iris data.

Program:

```
# Install corrplot package

install.packages("corrplot")

# Load necessary libraries

library(corrplot)


# Load the Iris dataset

data(iris)


# a. Find the correlation matrix

cor_matrix <- cor(iris[, 1:4])


# Print the correlation matrix

print(cor_matrix)


# b. Plot the correlation plot

corrplot(cor_matrix, method = "color", type = "upper", addrect = 3)
```

Explanation:

 # Install corrplot package

install.packages("corrplot")

- This line installs the `corrplot` package from CRAN. This package is used for visualizing correlation matrices.

 # Load necessary libraries

library(corrplot)

- Here, we load the `corrplot` library into the R session. Loading the library makes its functions available for use in the current R session.

 # Load the Iris dataset

data(iris)

- This line loads the Iris dataset, assuming it's not already loaded. The Iris dataset is a built-in dataset in R, commonly used for practicing data analysis and machine learning.

 # a. Find the correlation matrix

cor_matrix <- cor(iris[, 1:4])

- In this step, we use the `cor` function to calculate the correlation matrix for the first four columns of the Iris dataset, which are assumed to be numerical variables (Sepal.Length, Sepal.Width, Petal.Length, and Petal.Width). The resulting `cor_matrix` is a 4x4 matrix containing correlation coefficients between these variables.

 # Print the correlation matrix

print(cor_matrix)

- This line prints the correlation matrix to the R console. The correlation matrix shows how each variable correlates with every other variable in the dataset.

 # b. Plot the correlation plot

corrplot(cor_matrix, method = "color", type = "upper", addrect = 3)

- Finally, this code creates a correlation plot using the `corrplot` function. It visualizes the correlation matrix in a graphical form. Here are the key arguments used in the `corrplot` function:

- `cor_matrix`: The correlation matrix we calculated earlier.

- `method = "color"`: Specifies that we want to use color to represent the strength and direction of the correlation.

- `type = "upper"`: Indicates that we only want to display the upper triangle of the correlation matrix to avoid redundancy.

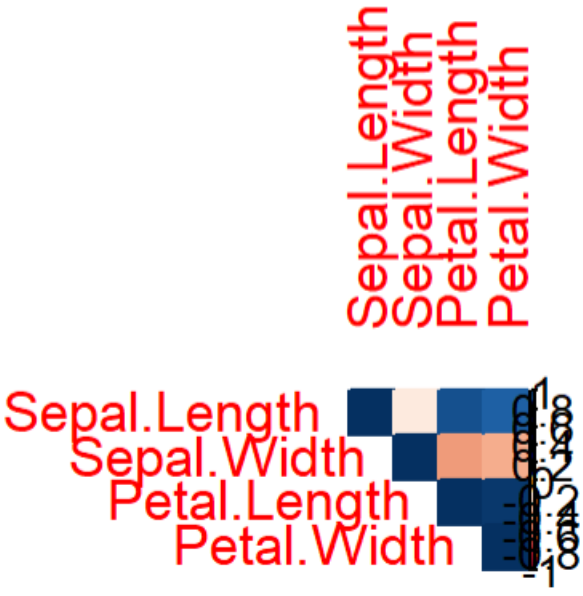
- `addrect = 3`: Adds rectangles around highly correlated variables for better visualization.

When we run this code, we should see a visual representation of the correlation matrix, where colors and intensity indicate the strength and direction of the correlations between variables in the Iris dataset. This plot provides a quick overview of the relationships among different features in the dataset.

Output:

```
package 'corrplot' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Rohini\AppData\Local\Temp\RtmpiEqPaZ\downloaded_packages
corrplot 0.92 loaded
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
Warning message:
package 'corrplot' was built under R version 4.3.2
>
```



Exercise 8:

Introduction to MachineLearningwith R	a. Installing and loadingnecessary packages b. Splitting data into training andtesting sets c. Building a simple machine learning d. Model Evaluation & prediction

Program:

```
# Install and load required packages
install.packages("caret")
install.packages("e1071")

library(caret)
library(e1071)
# Load the Iris dataset
data(iris)

# Set seed for reproducibility
set.seed(123)

# Split the data into training (70%) and testing (30%) sets
trainIndex <- createDataPartition(iris$Species, p = 0.7, list = FALSE, times = 1)
trainData <- iris[trainIndex, ]
testData <- iris[-trainIndex, ]
# Build an SVM model
svm_model <- svm(Species ~ ., data = trainData)
# Make predictions on the test set
predictions <- predict(svm_model, newdata = testData)
# Evaluate the model
print(confusionMatrix(predictions, testData$Species))
```

Program Explanation:

a. Installing and loading necessary packages

```
# Install and load required packages
install.packages("caret")
install.packages("e1071")

library(caret)
library(e1071)
```

- In this part, we are installing and loading two essential packages: `caret` and `e1071`.
  - `caret`: This package provides a set of functions for streamlining the model training and evaluation process.
  - `e1071`: This package contains the functions needed for Support Vector Machine (SVM) modeling.

b. Splitting data into training and testing sets

```
# Load the Iris dataset
data(iris)
# Set seed for reproducibility
set.seed(123)
# Split the data into training (70%) and testing (30%) sets
trainIndex <- createDataPartition(iris$Species, p = 0.7,
                                  list = FALSE,
                                  times = 1)
trainData <- iris[trainIndex, ]
testData <- iris[-trainIndex, ]
```

- Here, we are using the famous Iris dataset, which comes pre-loaded with R. We set a seed for reproducibility, ensuring that the random split of the data into training and testing sets is consistent across runs. The `createDataPartition` function from the `caret` package is used to perform the split.

c. Building a simple machine learning model

```
# Build an SVM model
svm_model <- svm(Species ~ ., data = trainData)
```

- Now, we're creating a Support Vector Machine (SVM) model using the `svm` function from the `e1071` package. The formula `Species ~ .` indicates that we want to predict the "Species" variable using all other variables in the dataset.

d. Model evaluation and prediction

```
# Make predictions on the test set
predictions <- predict(svm_model, newdata = testData)

# Evaluate the model
confusionMatrix(predictions, testData$Species)
```

- Here, we use the trained SVM model to make predictions on the test set. The `predict` function is used for this purpose. After making predictions, we evaluate the model's performance using the `confusionMatrix` function from the `caret` package. This function computes various metrics such as accuracy, sensitivity, and specificity and presents them in a confusion matrix.

Output:

```
Console   Terminal ×   Background Jobs ×
R 4.3.1 · ~/

package 'e1071' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Rohini\AppData\Local\Temp\RtmpOGH4f3\downloaded_packages
Loading required package: ggplot2
Loading required package: lattice
Confusion Matrix and Statistics

            Reference
Prediction   setosa versicolor virginica
  setosa         15          0         0
  versicolor      0         14         2
  virginica       0          1        13

Overall Statistics

               Accuracy : 0.9333
                 95% CI : (0.8173, 0.986)
    No Information Rate : 0.3333
```

```
Console   Terminal ×   Background Jobs ×
R 4.3.1 · ~/

    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: setosa Class: versicolor Class: virginica
Sensitivity                 1.0000            0.9333           0.8667
Specificity                 1.0000            0.9333           0.9667
Pos Pred Value              1.0000            0.8750           0.9286
Neg Pred Value              1.0000            0.9655           0.9355
Prevalence                  0.3333            0.3333           0.3333
Detection Rate              0.3333            0.3111           0.2889
Detection Prevalence        0.3333            0.3556           0.3111
Balanced Accuracy           1.0000            0.9333           0.9167
```

Explanation:

In this example, the confusion matrix is divided into three classes: setosa, versicolor, and virginica (the species of iris flowers in the dataset). The diagonal elements represent the number of correctly predicted instances for each class, while off-diagonal elements represent misclassifications.

The overall accuracy of the model is also provided, along with additional statistics such as sensitivity, specificity, and balanced accuracy for each class. These metrics give insights into how well the model performs for each class.

Exercise 10:

10	Regressionmodel	Create a regressionmodel for a given dataset

Creating a regression model involves several steps, such as loading the dataset, exploring and preprocessing the data, splitting it into training and testing sets, building the regression model, and evaluating its performance.

Below is a simple example using R. In this example, We have used the built-in dataset "mtcars," which contains information about various car models.

Program:

```
# Load the dataset

data(mtcars)


# Explore the dataset

head(mtcars)


# Check for missing values

summary(mtcars)


# Split the dataset into training and testing sets

set.seed(123)  # for reproducibility

train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))

train_data <- mtcars[train_indices, ]

test_data <- mtcars[-train_indices, ]


# Build a linear regression model

model <- lm(mpg ~ wt + hp, data = train_data)


# Summary of the model

summary(model)
```

```
# Make predictions on the test set
predictions <- predict(model, newdata = test_data)

# Evaluate the model
mse <- mean((predictions - test_data$mpg)^2)
rmse <- sqrt(mse)
r_squared <- 1 - (sum((test_data$mpg - predictions)^2) / sum((test_data$mpg - mean(test_data$mpg))^2))

# Print evaluation metrics
cat("Mean Squared Error:", mse, "\n")
cat("Root Mean Squared Error:", rmse, "\n")
cat("R-squared:", r_squared, "\n")
```

This code performs the following steps:

1. Load the dataset: Use the `data` function to load the "mtcars" dataset.

2. Explore the dataset: Use `head` to display the first few rows of the dataset and `summary` to check for missing values or get summary statistics.

3. Split the dataset: Randomly split the dataset into a training set (70% of the data) and a testing set (30% of the data).

4. Build a linear regression model: Use the `lm` function to create a linear regression model. In this example, we predict the miles per gallon (`mpg`) based on the weight (`wt`) and horsepower (`hp`) of the cars.

5. Summary of the model: Display a summary of the linear regression model.

6. Make predictions on the test set: Use the `predict` function to make predictions on the testing set.

7. Evaluate the model: Calculate Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared to evaluate the model's performance.

Explanation:

1. Load the dataset:

data(mtcars)

- In this step, we're loading the built-in "mtcars" dataset. This dataset contains information about various car models, including features like miles per gallon (`mpg`), weight (`wt`), and horsepower (`hp`).

2. Explore the dataset:

head(mtcars)

summary(mtcars)

- The `head` function shows the first few rows of the dataset, providing a quick look at its structure. The `summary` function provides summary statistics for each variable in the dataset, including measures like mean, median, and quartiles. This helps in understanding the distribution of data and checking for any missing values.

3. Check for missing values:

summary(mtcars)

- The `summary` function, as mentioned earlier, can help identify missing values in the dataset. If any variable has fewer observations than others, it may indicate missing values.

4. Split the dataset into training and testing sets:

```
set.seed(123)
train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))
train_data <- mtcars[train_indices, ]
test_data <- mtcars[-train_indices, ]
```

- We set a seed for reproducibility and then use the `sample` function to randomly select 70% of the indices for the training set. The remaining 30% form the testing set. This step is crucial to assess how well the model generalizes to new, unseen data.

5. Build a linear regression model:

```
model <- lm(mpg ~ wt + hp, data = train_data)
```

- The `lm` function is used to create a linear regression model. Here, we're predicting the miles per gallon (`mpg`) based on the weight (`wt`) and horsepower (`hp`) of the cars. The formula `mpg ~ wt + hp` indicates the relationship we want to model.

6. Summary of the model:

```
summary(model)
```

- The `summary` function provides detailed information about the linear regression model, including coefficients, standard errors, t-values, and p-values. This information helps in understanding the significance of each predictor in the model.

7. Make predictions on the test set:

```
predictions <- predict(model, newdata = test_data)
```

- The `predict` function is used to make predictions on the test set based on the trained model.

8. Evaluate the model:

mse <- mean((predictions - test_data$mpg)^2)

rmse <- sqrt(mse)

r_squared <- 1 - (sum((test_data$mpg - predictions)^2) / sum((test_data$mpg - mean(test_data$mpg))^2))

- Mean Squared Error (MSE): It measures the average of the squared differences between predicted and actual values. Smaller values indicate better model performance.

- Root Mean Squared Error (RMSE): It is the square root of MSE, providing a measure in the same unit as the target variable.

- R-squared (R^2): It represents the proportion of variance in the dependent variable that is predictable from the independent variables. A value close to 1 indicates a good fit.

Output:

```
> source("~/.active-rstudio-document")
Mean Squared Error: 4.242533
Root Mean Squared Error: 2.059741
R-squared: 0.6636754
>
```

Exercise 11:

11	Multiple regression model	Apply multiple regressions, if data have a continuous Independent variable

Let's extend the same example to a multiple regression model. In this case, we will use the "mtcars" dataset again, but we will include more independent variables. We will predict the miles per gallon (mpg) based on multiple features such as weight (wt), horsepower (hp) and the number of cylinders (cyl). The dependent variable (mpg) is continuous.

Program:

```
# Load the dataset
data(mtcars)

# Explore the dataset
head(mtcars)
summary(mtcars)

# Split the dataset into training and testing sets
set.seed(123)
train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))
train_data <- mtcars[train_indices, ]
test_data <- mtcars[-train_indices, ]

# Build a multiple regression model
model <- lm(mpg ~ wt + hp + cyl, data = train_data)

# Summary of the model
summary(model)

# Make predictions on the test set
predictions <- predict(model, newdata = test_data)
```

```
# Evaluate the model
mse <- mean((predictions - test_data$mpg)^2)
rmse <- sqrt(mse)
r_squared <- 1 - (sum((test_data$mpg - predictions)^2) / sum((test_data$mpg - mean(test_data$mpg))^2))

# Print evaluation metrics
cat("Mean Squared Error:", mse, "\n")
cat("Root Mean Squared Error:", rmse, "\n")
cat("R-squared:", r_squared, "\n")
```

In this updated code:

We have added the variable cyl to the predictors in the formula inside the lm function to include it in the multiple regression model.

The model is then trained using the training data and a summary of the model is printed to the console.

Predictions are made on the test set and the evaluation metrics (MSE, RMSE, R-squared) are calculated and printed.

Run this code in your R environment to see the results for the multiple regression model with the additional independent variable (cyl). As before, interpret the coefficients in the model summary and assess the model's performance using the evaluation metrics.

Explanation:

Certainly, let's go through the multiple regression implementation step by step:

1. Load the dataset:

```
data(mtcars)
```

- As before, we load the "mtcars" dataset, which contains information about various car models.

2. Explore the dataset:

head(mtcars)

summary(mtcars)

- The `head` function shows the first few rows of the dataset, and `summary` provides summary statistics for each variable in the dataset. This helps in understanding the structure and distribution of the data.

3. Split the dataset into training and testing sets:

set.seed(123)

train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))

train_data <- mtcars[train_indices,]

test_data <- mtcars[-train_indices,]

- We set a seed for reproducibility, and then we randomly select 70% of the indices for the training set. The remaining 30% forms the testing set.

4. Build a multiple regression model:

model <- lm(mpg ~ wt + hp + cyl, data = train_data)

- We use the `lm` function to create a multiple regression model. In this case, we're predicting the miles per gallon (`mpg`) based on the weight (`wt`), horsepower (`hp`), and the number of cylinders (`cyl`) of the cars.

5. Summary of the model:

summary(model)

- The `summary` function provides detailed information about the multiple regression model, including coefficients, standard errors, t-values, and p-values for each predictor. Look for the coefficients associated with `wt`, `hp`, and `cyl`.

6. Make predictions on the test set:

predictions <- predict(model, newdata = test_data)

- The `predict` function is used to make predictions on the test set based on the trained multiple regression model.

7. Evaluate the model:

mse <- mean((predictions - test_data$mpg)^2)

rmse <- sqrt(mse)

r_squared <- 1 - (sum((test_data$mpg - predictions)^2) / sum((test_data$mpg - mean(test_data$mpg))^2))

- Mean Squared Error (MSE): Measures the average of the squared differences between predicted and actual values. Lower values are better.

- Root Mean Squared Error (RMSE): Similar to MSE but in the same unit as the target variable. Lower values are better.

- R-squared (R^2): Represents the proportion of variance in the dependent variable that is predictable from the independent variables. A value close to 1 indicates a better fit.

8. Print evaluation metrics:

cat("Mean Squared Error:", mse, "\n")

cat("Root Mean Squared Error:", rmse, "\n")

cat("R-squared:", r_squared, "\n")

- These lines print the calculated evaluation metrics to the console for easy interpretation.

Now, when we run this code in your R environment, we will see the summary of the multiple regression model and the evaluation metrics based on the "mtcars" dataset with three predictors: `wt`, `hp`, and `cyl`. Adjust the model and analysis based on your specific dataset and research questions.

Output:

```
> source("~/.active-rstudio-document")
Mean Squared Error: 5.152415
Root Mean Squared Error: 2.269893
R-squared: 0.591545
>
```

Exercise 12:

12	Regression model for prediction	Apply regression Model techniques to predict the data on the given dataset.

To provide a regression model implementation, let's use a sample dataset and demonstrate a simple linear regression using the popular `lm()` function in R. For this example, let us use the built-in dataset "mtcars" and predict miles per gallon (`mpg`) based on one of its continuous predictors, such as weight (`wt`).

Program:

```
# Load the dataset
data(mtcars)

# Explore the dataset
head(mtcars)
summary(mtcars)

# Split the dataset into training and testing sets
set.seed(123)
train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))
train_data <- mtcars[train_indices, ]
test_data <- mtcars[-train_indices, ]

# Build a simple linear regression model
model <- lm(mpg ~ wt, data = train_data)

# Summary of the model
summary(model)

# Make predictions on the test set
```

predictions <- predict(model, newdata = test_data)

Evaluate the model

mse <- mean((predictions - test_data$mpg)^2)

rmse <- sqrt(mse)

r_squared <- 1 - (sum((test_data$mpg - predictions)^2) / sum((test_data$mpg - mean(test_data$mpg))^2))

Print evaluation metrics

cat("Mean Squared Error:", mse, "\n")

cat("Root Mean Squared Error:", rmse, "\n")

cat("R-squared:", r_squared, "\n")

In this code:

1. Load the dataset:

data(mtcars)

- Load the "mtcars" dataset.

2. Explore the dataset:

head(mtcars)

summary(mtcars)

- Display the first few rows and summary statistics of the dataset to understand its structure.

3. Split the dataset into training and testing sets:

set.seed(123)

train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))

train_data <- mtcars[train_indices,]

test_data <- mtcars[-train_indices,]

- Randomly split the dataset into training and testing sets. This ensures that the model's performance is evaluated on unseen data.

4. Build a simple linear regression model:

model <- lm(mpg ~ wt, data = train_data)

- Use the `lm` function to create a simple linear regression model predicting miles per gallon (`mpg`) based on weight (`wt`).

5. Summary of the model:

summary(model)

- Display a summary of the linear regression model, including coefficients and statistical significance.

6. Make predictions on the test set:

predictions <- predict(model, newdata = test_data)

- Use the trained model to make predictions on the test set.

7. Evaluate the model:

mse <- mean((predictions - test_data$mpg)^2)

rmse <- sqrt(mse)

r_squared <- 1 - (sum((test_data$mpg - predictions)^2) / sum((test_data$mpg - mean(test_data$mpg))^2))

- Calculate Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared to evaluate the model's performance.

8. Print evaluation metrics:

cat("Mean Squared Error:", mse, "\n")

cat("Root Mean Squared Error:", rmse, "\n")

cat("R-squared:", r_squared, "\n")

- Print the calculated evaluation metrics to the console for interpretation.

Output:

```
> source("~/.active-rstudio-document")
Mean Squared Error: 4.567618
Root Mean Squared Error: 2.137199
R-squared: 0.6379045
>
```

Exercise 13:

13	Principal Component Analysis	Perform PrincipalComponent Analysis (PCA) using R

Make sure you have the necessary R packages installed or install them using:

install.packages(c("ggplot2", "stats"))

Program:

```
# Load the USArrests dataset
data(USArrests)

# View the first few rows of the dataset
head(USArrests)

# Standardize the features (mean=0, sd=1)
scaled_data <- scale(USArrests)

# Perform PCA
pca_result <- prcomp(scaled_data)

# Summary of the PCA
print(summary(pca_result))


# Access the loadings (weights) of each variable on each principal component
loadings_matrix <- pca_result$rotation
print(loadings_matrix)

# Access the scores of each data point on each principal component
scores <- as.data.frame(pca_result$x)
```

```
print(head(scores))

# Scree plot to visualize the variance explained by each principal component
png(filename = "scree_plot.png", width = 800, height = 400)  # Adjust width and height as needed
par(mar = c(5, 5, 2, 2))  # Adjust the margins
plot(1:4, pca_result$sdev^2 / sum(pca_result$sdev^2), type = "b",
    main = "Scree Plot", xlab = "Principal Component", ylab = "Proportion of Variance Explained")
dev.off()

# Biplot to visualize the data points and loadings
png(filename = "biplot.png", width = 800, height = 800)  # Adjust width and height as needed
par(mar = c(5, 5, 2, 2))  # Adjust the margins
biplot(pca_result)
dev.off()
```

This R code is a comprehensive script that performs Principal Component Analysis (PCA) on the USArrests dataset, visualizes the results, and saves the plots as PNG images. Let's go through each part of the code step by step.

```
# Load the USArrests dataset
data(USArrests)

# View the first few rows of the dataset
head(USArrests)
```

- This section loads the USArrests dataset, which is a built-in dataset in R containing data on violent crime rates in the United States. The `head()` function is used to display the first few rows of the dataset.

 # Standardize the features (mean=0, sd=1)

scaled_data <- scale(USArrests)

- Here, the features in the dataset are standardized (scaled) so that they have a mean of 0 and a standard deviation of 1. This is a common preprocessing step in PCA.

 # Perform PCA

pca_result <- prcomp(scaled_data)

- The `prcomp()` function is used to perform Principal Component Analysis on the standardized dataset (`scaled_data`), and the result is stored in the variable `pca_result`.

 # Summary of the PCA

print(summary(pca_result))

- This part prints a summary of the PCA, including information about the standard deviations, proportion of variance explained, and cumulative proportion of variance explained by each principal component.

 # Access the loadings (weights) of each variable on each principal component

loadings_matrix <- pca_result$rotation

print(loadings_matrix)

- Here, the loadings matrix is extracted from the PCA result. The loadings represent the weights of each original variable on each principal component.

 # Access the scores of each data point on each principal component

scores <- as.data.frame(pca_result$x)

print(head(scores))

- The scores matrix is extracted, representing the coordinates of each data point in the new space defined by the principal components.

 # Scree plot to visualize the variance explained by each principal component

png(filename = "scree_plot.png", width = 800, height = 400) # Adjust width and height as needed

par(mar = c(5, 5, 2, 2)) # Adjust the margins

plot(1:4, pca_result$sdev^2 / sum(pca_result$sdev^2), type = "b",

main = "Scree Plot", xlab = "Principal Component", ylab = "Proportion of Variance Explained")

dev.off()

- This section creates a scree plot, which shows the proportion of variance explained by each principal component. The plot is saved as a PNG file, and adjustments are made to the width, height, and margins.

 # Biplot to visualize the data points and loadings

png(filename = "biplot.png", width = 800, height = 800) # Adjust width and height as needed

par(mar = c(5, 5, 2, 2)) # Adjust the margins

biplot(pca_result)

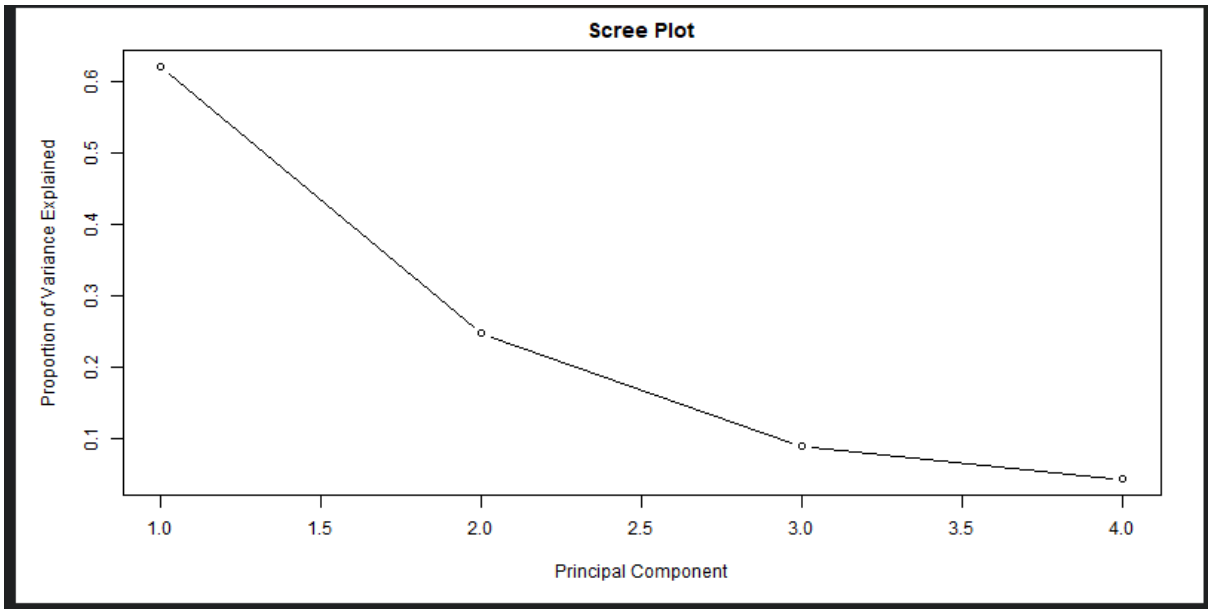
dev.off()

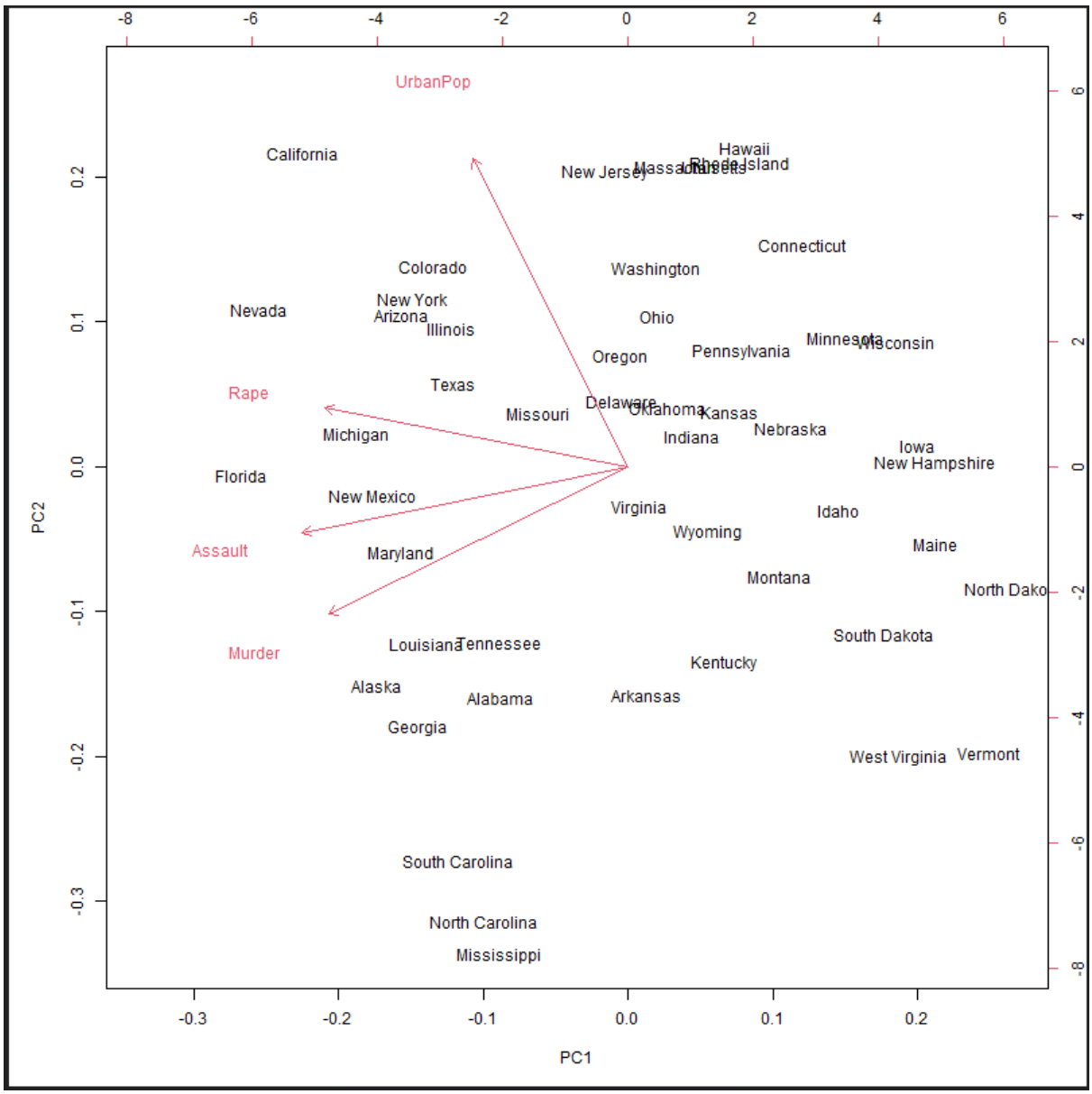
- This part creates a biplot, which visualizes both the data points and the loadings in the new space defined by the principal components. The biplot is saved as a PNG file, and adjustments are made to the width, height, and margins.

After running this script, we will find two PNG files ("scree_plot.png" and "biplot.png") in your working directory, each containing the respective plots.

Output:

```
> source("~/.active-rstudio-document")
Importance of components:
                          PC1    PC2     PC3     PC4
Standard deviation     1.5749 0.9949 0.59713 0.41645
Proportion of Variance 0.6201 0.2474 0.08914 0.04336
Cumulative Proportion  0.6201 0.8675 0.95664 1.00000
               PC1        PC2        PC3         PC4
Murder   -0.5358995 -0.4181809  0.3412327  0.64922780
Assault  -0.5831836 -0.1879856  0.2681484 -0.74340748
UrbanPop -0.2781909  0.8728062  0.3780158  0.13387773
Rape     -0.5434321  0.1673186 -0.8177779  0.08902432
                PC1        PC2         PC3          PC4
Alabama    -0.9756604 -1.1220012  0.43980366  0.154696581
Alaska     -1.9305379 -1.0624269 -2.01950027 -0.434175454
Arizona    -1.7454429  0.7384595 -0.05423025 -0.826264240
Arkansas    0.1399989 -1.1085423 -0.11342217 -0.180973554
California -2.4986128  1.5274267 -0.59254100 -0.338559240
Colorado   -1.4993407  0.9776297 -1.08400162  0.001450164
>
```




UrbanPop
Rape
Assault
Murder
California
Hawaii
New Jersey
Connecticut
Colorado
Washington
Nevada
New York
Arizona
Illinois
Ohio
Oregon
Pennsylvania
Minnesota
Wisconsin
Texas
Missouri
Delaware
Oklahoma
Kansas
Indiana
Nebraska
Michigan
Iowa
New Hampshire
Florida
New Mexico
Virginia
Idaho
Wyoming
Maine
Maryland
Montana
North Dako
South Dakota
Louisiana
Tennessee
Kentucky
Alaska
Alabama
Arkansas
Georgia
West Virginia
Vermont
South Carolina
North Carolina
Mississippi
PC1
PC2

Exercise 14:

14	Factor Analysis	Perform FactorAnalysis using R.

Program:

```
# Install and load necessary packages
#install.packages("psych")
library(psych)

# Load the dataset
data(mtcars)

# Select relevant variables for factor analysis
# For this example, I'll use a subset of variables
my_data <- mtcars[, c("mpg", "disp", "hp", "drat", "wt")]

# Check the structure of the data
str(my_data)

# Perform factor analysis
factor_result <- fa(my_data, nfactors = 2, rotate = "varimax")

# Print the factor analysis results
print(factor_result)

# Plot eigenvalues to determine the number of factors
eigenvalues <- factor_result$values
plot(1:length(eigenvalues), eigenvalues, type = "b",
    main = "Scree Plot", xlab = "Factor", ylab = "Eigenvalue")
```

Explanation:

 # Install and load necessary packages

 # install.packages("psych")

library(psych)

- This section installs and loads the `psych` package, which is essential for performing factor analysis. If you haven't installed the package yet, you can uncomment the `install.packages("psych")` line to install it.

 # Load the dataset

data(mtcars)

- This code loads the built-in `mtcars` dataset, which contains information about various car models.

 # Select relevant variables for factor analysis

 # For this example, I'll use a subset of variables

my_data <- mtcars[, c("mpg", "disp", "hp", "drat", "wt")]

- Here, a subset of variables (`mpg`, `disp`, `hp`, `drat`, `wt`) is selected from the `mtcars` dataset. These variables will be used in the factor analysis.

 # Check the structure of the data

str(my_data)

- This code checks and prints the structure of the selected data. It's a good practice to inspect the structure to ensure the data is loaded correctly and has the expected format.

 # Perform factor analysis

factor_result <- fa(my_data, nfactors = 2, rotate = "varimax")

- Here, factor analysis is performed using the `fa` function from the `psych` package. It specifies that we want to extract 2 factors (`nfactors = 2`) and use the varimax rotation method (`rotate = "varimax"`).

 # Print the factor analysis results

print(factor_result)

- This line prints the results of the factor analysis, including factor loadings, communalities, and other relevant information.

 # Plot eigenvalues to determine the number of factors

eigenvalues <- factor_result$values

plot(1:length(eigenvalues), eigenvalues, type = "b",

main = "Scree Plot", xlab = "Factor", ylab = "Eigenvalue")

- This code generates a scree plot to help determine the number of factors to retain. The scree plot displays the eigenvalues for each factor, and you can look for an "elbow" in the plot to decide how many factors to keep.

This script provides a basic example of how to perform factor analysis, inspect the results, and visualize the eigenvalues using R.

Output:

```
> source("~/.active-rstudio-document")
'data.frame':   32 obs. of  5 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
Factor Analysis using method =  minres
Call: fa(r = my_data, nfactors = 2, rotate = "varimax")
Standardized loadings (pattern matrix) based upon correlation matrix
       MR1   MR2   h2       u2 com
mpg  -0.74 -0.55 0.85  0.15003 1.8
disp  0.77  0.55 0.90  0.10117 1.8
hp    0.34  0.94 1.00 -0.00035 1.3
drat -0.75 -0.21 0.61  0.38870 1.2
wt    0.86  0.40 0.89  0.10610 1.4

                       MR1  MR2
SS loadings           2.57 1.69
Proportion Var        0.51 0.34
Cumulative Var        0.51 0.85
Proportion Explained  0.60 0.40
Cumulative Proportion 0.60 1.00
```

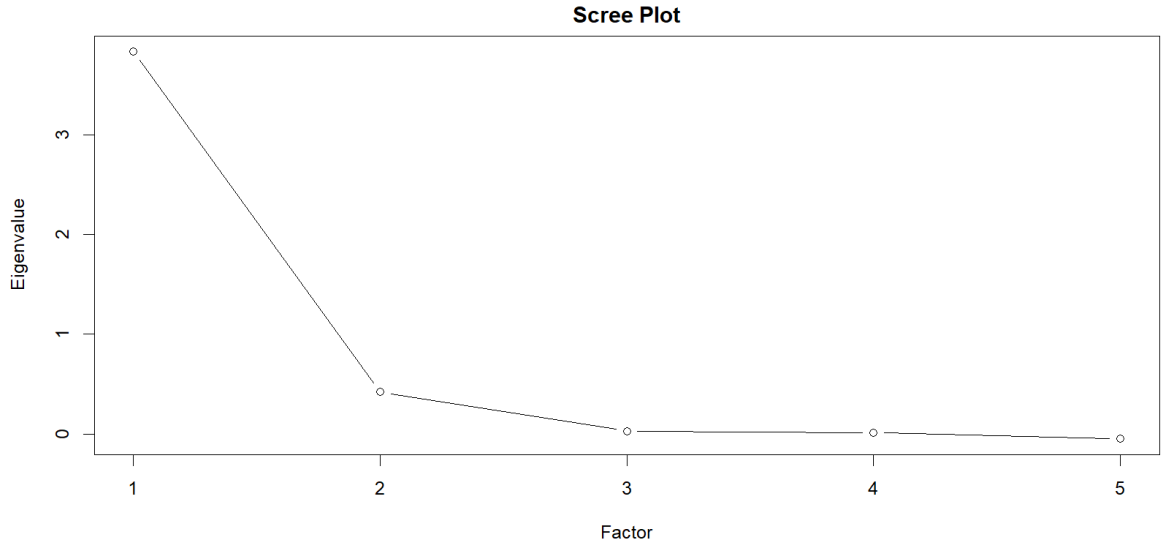
```
Mean item complexity =  1.5
Test of the hypothesis that 2 factors are sufficient.

df null model =  10  with the objective function =  5.2 with Chi Square =  148.26
df of  the model are 1  and the objective function was  0.13

The root mean square of the residuals (RMSR) is  0.01
The df corrected root mean square of the residuals is  0.04

The harmonic n.obs is  32 with the empirical chi square  0.11  with prob <  0.74
The total n.obs was  32  with Likelihood Chi Square =  3.43  with prob <  0.064

Tucker Lewis Index of factoring reliability =  0.815
RMSEA index =  0.274  and the 90 % confidence intervals are  0 0.628
BIC =  -0.03
Fit based upon off diagonal values = 1
Measures of factor score adequacy
                                                MR1 MR2
Correlation of (regression) scores with factors   0.96   1
Multiple R square of scores with factors          0.92   1
Minimum correlation of possible factor scores     0.85   1
```



Exercise 15:

15	Classification Algorithms	Implement k-Nearest Neighbors (kNN) classification using R

Program:

```
# Load necessary libraries
library(class)

# Load the Iris dataset
data(iris)

# Split the dataset into training and testing sets
set.seed(123)  # Set seed for reproducibility
sample_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))  # 70% for training
train_data <- iris[sample_indices, ]
test_data <- iris[-sample_indices, ]

# k-Nearest Neighbors (kNN) classification
k <- 3  # Set the number of neighbors

# Predict the class of test_data using kNN
predicted_classes <- knn(train = train_data[, 1:4], test = test_data[, 1:4], cl = train_data$Species, k = k)

# Confusion matrix
conf_matrix <- table(predicted_classes, test_data$Species)
print("Confusion Matrix:")
print(conf_matrix)
```

```
# Calculate accuracy
accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)
print(paste("Accuracy:", round(accuracy, 2)))
```

Explanation of the code:

Load necessary libraries:

- The class library provides the knn function for k-Nearest Neighbors classification.

Load the Iris dataset:

- The Iris dataset is a built-in dataset in R.

Split the dataset:

- We split the dataset into training and testing sets. In this example, 70% of the data is used for training, and the remaining 30% is used for testing.

k-Nearest Neighbors (kNN) classification:

- We set the number of neighbors (k) to 3.
- The knn function is used to predict the class of the test data based on the training data.

Confusion matrix:

- We create a confusion matrix to evaluate the performance of the classification.

Calculate accuracy:

- We calculate the accuracy of the kNN classification by comparing predicted classes to the actual classes.

We can run this script in your R environment to see the results of the kNN classification on the Iris dataset. Note that we can modify the k value and other parameters based on our specific requirements.

Detailed explanation:

Let's break down the provided R code step by step, explaining each section:

Load Necessary Libraries

library(class)

- This line loads the `class` library, which provides the `knn` function for k-Nearest Neighbors classification.

Load the Iris Dataset

data(iris)

- This line loads the famous Iris dataset, which is included in the base R packages. The dataset contains measurements of sepal length, sepal width, petal length, and petal width for three species of iris flowers.

Split the Dataset into Training and Testing Sets

set.seed(123)

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

train_data <- iris[sample_indices,]

test_data <- iris[-sample_indices,]

- Here, we set the seed for reproducibility, and then we randomly sample 70% of the indices to create a training set (`train_data`) and use the remaining 30% for the testing set (`test_data`).

k-Nearest Neighbors (kNN) Classification

k <- 3

predicted_classes <- knn(train = train_data[, 1:4], test = test_data[, 1:4], cl = train_data$Species, k = k)

- In this section:

- We set the value of `k` to 3, which means we will consider the three nearest neighbors for classification.

- The `knn` function is used for k-Nearest Neighbors classification. We provide the training data (`train_data[, 1:4]` - the first four columns representing the features), the test data (`test_data[, 1:4]`), the class labels of the training data (`cl = train_data$Species`), and the value of `k`.

Confusion Matrix

conf_matrix <- table(predicted_classes, test_data$Species)

print("Confusion Matrix:")

print(conf_matrix)

- This section creates a confusion matrix to evaluate the performance of the kNN classification. The confusion matrix shows the number of correct and incorrect predictions for each class.

Calculate Accuracy

accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)

print(paste("Accuracy:", round(accuracy, 2)))

- Here, the accuracy of the kNN classification is calculated by summing the diagonal elements (correct predictions) of the confusion matrix and dividing it by the total number of predictions. The result is printed as the accuracy.

Summary

In summary, this R script demonstrates how to perform k-Nearest Neighbors classification on the Iris dataset, split the data into training and testing sets, generate a confusion matrix, and calculate the accuracy of the classification. We can modify the `k` value or other parameters to experiment with different settings. This example serves as a starting point for understanding and implementing kNN classification in R.

Output:

```
> source("~/.active-rstudio-document")
[1] "Confusion Matrix:"

predicted_classes setosa versicolor virginica
       setosa         14          0         0
       versicolor      0         17         0
       virginica       0          1        13
[1] "Accuracy: 0.98"
>
```

Exercise 16:

16	Classification Algorithms	Evaluate the performance of Naive Bayes classifier usingR.

Program:

```
# Load necessary library
library(e1071)

# Load the Iris dataset
data(iris)

# Split the dataset into training and testing sets
set.seed(123)  # Set seed for reproducibility
sample_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))  # 70% for training
train_data <- iris[sample_indices, ]
test_data <- iris[-sample_indices, ]

# Fit a Naive Bayes classifier
naive_bayes_model <- naiveBayes(Species ~ ., data = train_data)

# Make predictions on the test set
predicted_classes <- predict(naive_bayes_model, newdata = test_data)

# Confusion matrix
conf_matrix <- table(predicted_classes, test_data$Species)
print("Confusion Matrix:")
print(conf_matrix)

# Calculate accuracy
accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)
```

```
print(paste("Accuracy:", round(accuracy, 2)))

# Calculate precision, recall, and F1 score
precision <- conf_matrix[2, 2] / sum(conf_matrix[, 2])
recall <- conf_matrix[2, 2] / sum(conf_matrix[2, ])
f1_score <- 2 * (precision * recall) / (precision + recall)

print(paste("Precision:", round(precision, 2)))
print(paste("Recall:", round(recall, 2)))
print(paste("F1 Score:", round(f1_score, 2)))
```

Explanation:

Load Necessary Library

library(e1071)

- This line loads the `e1071` library, which provides the `naiveBayes` function for training a Naive Bayes classifier.

Load the Iris Dataset

data(iris)

- This line loads the Iris dataset, a built-in dataset in R that contains measurements of sepal length, sepal width, petal length, and petal width for three species of iris flowers.

Split the Dataset into Training and Testing Sets

set.seed(123)

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

train_data <- iris[sample_indices,]

test_data <- iris[-sample_indices,]

- Here, we set the seed for reproducibility, and then we randomly sample 70% of the indices to create a training set (`train_data`) and use the remaining 30% for the testing set (`test_data`).

Fit a Naive Bayes Classifier

```
naive_bayes_model <- naiveBayes(Species ~ ., data = train_data)
```

- This line fits a Naive Bayes classifier to the training data. The formula `Species ~ .` indicates that we are predicting the 'Species' variable based on all other variables in the dataset.

Make Predictions on the Test Set

```
predicted_classes <- predict(naive_bayes_model, newdata = test_data)
```

- Using the trained Naive Bayes model, we make predictions on the test data.

Confusion Matrix

```
conf_matrix <- table(predicted_classes, test_data$Species)
print("Confusion Matrix:")
print(conf_matrix)
```

- This section creates a confusion matrix, which is a table that shows the number of true positive, true negative, false positive, and false negative predictions. It provides a detailed view of the classifier's performance.

Calculate Accuracy

```
accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)
print(paste("Accuracy:", round(accuracy, 2)))
```

- The accuracy is calculated by summing the diagonal elements (correct predictions) of the confusion matrix and dividing by the total number of predictions.

Calculate Precision, Recall, and F1 Score

```
precision <- conf_matrix[2, 2] / sum(conf_matrix[, 2])
recall <- conf_matrix[2, 2] / sum(conf_matrix[2, ])
f1_score <- 2 * (precision * recall) / (precision + recall)

print(paste("Precision:", round(precision, 2)))
print(paste("Recall:", round(recall, 2)))
print(paste("F1 Score:", round(f1_score, 2)))
```

- Precision, recall, and F1 score are metrics that provide more insights into the classifier's performance, especially in situations where there is an imbalance in the class distribution. Precision is the ratio of true positive predictions to the total predicted positive instances, recall is the ratio of true positive predictions to the total actual positive instances, and F1 score is the harmonic mean of precision and recall.

In summary, this R script demonstrates how to implement a Naive Bayes classifier, make predictions on a test set, generate a confusion matrix, and calculate accuracy, precision, recall, and F1 score for evaluation.

Output:

```
> source("~/.active-rstudio-document")
[1] "Confusion Matrix:"

predicted_classes setosa versicolor virginica
       setosa         14          0         0
       versicolor      0         18         0
       virginica       0          0        13
[1] "Accuracy: 1"
[1] "Precision: 1"
[1] "Recall: 1"
[1] "F1 Score: 1"
```

Exercise 17:

17	Classification Algorithms	Evaluate the performance of theDecisio Treeclassifier using R.

Program:

```
# Load necessary library
library(rpart)

# Load the Iris dataset
data(iris)

# Split the dataset into training and testing sets
set.seed(123)  # Set seed for reproducibility
sample_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))  # 70% for training
train_data <- iris[sample_indices, ]
test_data <- iris[-sample_indices, ]

# Fit a Decision Tree classifier
decision_tree_model <- rpart(Species ~ ., data = train_data, method = "class")

# Visualize the Decision Tree
plot(decision_tree_model)
text(decision_tree_model, cex = 0.8)

# Make predictions on the test set
predicted_classes <- predict(decision_tree_model, newdata = test_data, type = "class")

# Confusion matrix
conf_matrix <- table(predicted_classes, test_data$Species)
```

```
print("Confusion Matrix:")
print(conf_matrix)

# Calculate accuracy
accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)
print(paste("Accuracy:", round(accuracy, 2)))

# Calculate precision, recall, and F1 score
precision <- conf_matrix[2, 2] / sum(conf_matrix[, 2])
recall <- conf_matrix[2, 2] / sum(conf_matrix[2, ])
f1_score <- 2 * (precision * recall) / (precision + recall)

print(paste("Precision:", round(precision, 2)))
print(paste("Recall:", round(recall, 2)))
print(paste("F1 Score:", round(f1_score, 2)))
```

Explanation:

Load Necessary Library

library(rpart)

- This line loads the `rpart` library, which provides the `rpart` function for training a Decision Tree classifier.

Load the Iris Dataset

data(iris)

- This line loads the Iris dataset, which is a built-in dataset in R.

Split the Dataset into Training and Testing Sets

set.seed(123) # Set seed for reproducibility

sample_indices <- sample(1:nrow(iris), 0.7 * nrow(iris)) # 70% for training

train_data <- iris[sample_indices,]

test_data <- iris[-sample_indices,]

- Here, we set the seed for reproducibility, and then we randomly sample 70% of the indices to create a training set (`train_data`) and use the remaining 30% for the testing set (`test_data`).

Fit a Decision Tree Classifier

decision_tree_model <- rpart(Species ~ ., data = train_data, method = "class")

- This line fits a Decision Tree classifier to the training data. The formula `Species ~ .` indicates that we are predicting the 'Species' variable based on all other variables in the dataset. The `method = "class"` argument specifies that this is a classification problem.

Visualize the Decision Tree

plot(decision_tree_model)

text(decision_tree_model, cex = 0.8)

- These lines visualize the Decision Tree. The `plot` function creates the initial plot, and the `text` function overlays text on the plot, including node labels.

Make Predictions on the Test Set

predicted_classes <- predict(decision_tree_model, newdata = test_data, type = "class")

- Using the trained Decision Tree model, we make predictions on the test data. The `type = "class"` argument specifies that we want class predictions.

Confusion Matrix

conf_matrix <- table(predicted_classes, test_data$Species)

print("Confusion Matrix:")

print(conf_matrix)

- This section creates a confusion matrix, which is a table that shows the number of true positive, true negative, false positive, and false negative predictions. It provides a detailed view of the classifier's performance.

Calculate Accuracy, Precision, Recall, and F1 Score

```
accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)
print(paste("Accuracy:", round(accuracy, 2)))

precision <- conf_matrix[2, 2] / sum(conf_matrix[, 2])
recall <- conf_matrix[2, 2] / sum(conf_matrix[2, ])
f1_score <- 2 * (precision * recall) / (precision + recall)

print(paste("Precision:", round(precision, 2)))
print(paste("Recall:", round(recall, 2)))
print(paste("F1 Score:", round(f1_score, 2)))
```

- Accuracy is calculated by summing the diagonal elements (correct predictions) of the confusion matrix and dividing by the total number of predictions. Precision, recall, and F1 score are computed using the confusion matrix, providing additional metrics for evaluating the classifier's performance.

This script demonstrates how to implement a Decision Tree classifier, visualize the resulting tree, and evaluate its performance on a test set using R.

Output:

```
> source("~/.active-rstudio-document")
[1] "Confusion Matrix:"

predicted_classes setosa versicolor virginica
       setosa         14          0         0
       versicolor      0         18         1
       virginica       0          0        12
[1] "Accuracy: 0.98"
[1] "Precision: 1"
[1] "Recall: 0.95"
[1] "F1 Score: 0.97"
>
```

Exercise 18:

18	Classification Algorithms	Evaluate the performance of Random ForestClassifier using R.

Evaluating the performance of a Random Forest classifier involves assessing metrics such as accuracy, precision, recall, and F1 score. The below example shows the implementation of a Random Forest classifier and evaluate its performance using R. we will use the built-in Iris dataset as an example.

Program:

```
# Load necessary library
library(randomForest)

# Load the Iris dataset
data(iris)

# Split the dataset into training and testing sets
set.seed(123)  # Set seed for reproducibility
sample_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))  # 70% for training
train_data <- iris[sample_indices, ]
test_data <- iris[-sample_indices, ]

# Fit a Random Forest classifier
random_forest_model <- randomForest(Species ~ ., data = train_data)

# Make predictions on the test set
predicted_classes <- predict(random_forest_model, newdata = test_data)

# Confusion matrix
conf_matrix <- table(predicted_classes, test_data$Species)
print("Confusion Matrix:")
```

```
print(conf_matrix)

# Calculate accuracy
accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)
print(paste("Accuracy:", round(accuracy, 2)))

# Calculate precision, recall, and F1 score
precision <- conf_matrix[2, 2] / sum(conf_matrix[, 2])
recall <- conf_matrix[2, 2] / sum(conf_matrix[2, ])
f1_score <- 2 * (precision * recall) / (precision + recall)

print(paste("Precision:", round(precision, 2)))
print(paste("Recall:", round(recall, 2)))
print(paste("F1 Score:", round(f1_score, 2)))
```

Explanation of the code:

1. Load Necessary Library

library(randomForest)

- This line loads the `randomForest` library, which provides the `randomForest` function for training a Random Forest classifier.

2. Load the Iris Dataset

data(iris)

- This line loads the Iris dataset, which is a built-in dataset in R.

3. Split the Dataset into Training and Testing Sets

```
set.seed(123)  # Set seed for reproducibility
sample_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))  # 70% for training
train_data <- iris[sample_indices, ]
test_data <- iris[-sample_indices, ]
```

- Here, we set the seed for reproducibility, and then we randomly sample 70% of the indices to create a training set (`train_data`) and use the remaining 30% for the testing set (`test_data`).

4. Fit a Random Forest Classifier

random_forest_model <- randomForest(Species ~ ., data = train_data)

- This line fits a Random Forest classifier to the training data. The formula `Species ~ .` indicates that we are predicting the 'Species' variable based on all other variables in the dataset.

5. Make Predictions on the Test Set

predicted_classes <- predict(random_forest_model, newdata = test_data)

- Using the trained Random Forest model, we make predictions on the test data.

6. Confusion Matrix

conf_matrix <- table(predicted_classes, test_data$Species)

print("Confusion Matrix:")

print(conf_matrix)

- This section creates a confusion matrix, which is a table that shows the number of true positive, true negative, false positive, and false negative predictions. It provides a detailed view of the classifier's performance.

7. Calculate Accuracy, Precision, Recall, and F1 Score

accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)

print(paste("Accuracy:", round(accuracy, 2)))

precision <- conf_matrix[2, 2] / sum(conf_matrix[, 2])

recall <- conf_matrix[2, 2] / sum(conf_matrix[2,])

f1_score <- 2 * (precision * recall) / (precision + recall)

print(paste("Precision:", round(precision, 2)))

print(paste("Recall:", round(recall, 2)))

print(paste("F1 Score:", round(f1_score, 2)))

- Accuracy is calculated by summing the diagonal elements (correct predictions) of the confusion matrix and dividing by the total number of predictions. Precision, recall, and F1 score are computed using the confusion matrix, providing additional metrics for evaluating the classifier's performance.

This script demonstrates how to implement a Random Forest classifier, make predictions, generate a confusion matrix, and calculate accuracy, precision, recall, and F1 score for evaluation.

Output:

```
[1] "Confusion Matrix:"

predicted_classes setosa versicolor virginica
       setosa         14          0         0
       versicolor      0         17         0
       virginica       0          1        13
[1] "Accuracy: 0.98"
[1] "Precision: 0.94"
[1] "Recall: 1"
[1] "F1 Score: 0.97"
>
```

Exercise 19: Final Project

Project Titles

1. Customer Segmentation and Behavior Analysis:

- Use clustering algorithms (such as k-means) to segment customers based on their purchasing behavior. Analyze each segment's characteristics and tailor marketing strategies accordingly.

2. Predictive Maintenance for Equipment:

- Develop a predictive maintenance model using machine learning techniques to forecast when equipment is likely to fail. This can help reduce downtime and maintenance costs.

3. Credit Risk Assessment:

- Build a credit scoring model to assess the credit risk of individuals or businesses. Use historical data to train the model and predict the likelihood of default for new applicants.

4. Natural Language Processing (NLP) for Sentiment Analysis:

- Implement NLP techniques to analyze sentiment in text data. Apply sentiment analysis to social media, customer reviews, or news articles to understand public opinion or customer feedback.

5. Healthcare Data Analysis and Prediction:

- Analyze healthcare data to identify patterns and trends. Build predictive models for disease diagnosis or patient outcomes. Explore datasets such as electronic health records or public health datasets.

These project titles cover a range of data science topics, including clustering, predictive modeling, sentiment analysis, and healthcare analytics. Students can choose a project that aligns with their interests and goals in the field of data science.