THAKUR COLLEGE OF SCIENCE & COMMERCE

NAAC Accredited with Grade "A" (3rd Cycle)

THAKUR TRUSTS

ISO 9001 : 2015 Certified

Degree College

Computer Journal

CERTIFICATE

SEMESTER Second UID No. _____

Class MSC-CS PART-1 Roll No. 4635 Year 2023-2024

This is to certify that the work entered in this journal is the work of Mst. / Ms. Akash Arjun Yadav

who has worked for the year 2023-2024 in the Computer Laboratory.

Teacher In-Charge

Head of Department

Date : _____

Examiner

INDEX

Sr no.	Topic	Page Number	Date	Signature
1.	Practical 1- Implement simple linear regression model on a standard data set and plot the least square regression fit. Comment on the result. [One may use inbuilt data sets like Boston, Auto etc.]		20/12/2023	
2.	Practical 2- Implement multiple regression model on a standard data set and plot the least square regression fit. Comment on the result. [One may use inbuilt data sets like Carseats, Boston etc].		20/12/2023	
3.	Practical 3- Fit a classification model using following- (i) Quadratic Discriminant Analysis (QDA) on a standard data set and compares the results. [Inbuilt datasets like Smarket, Weekly, Auto, Boston etc may be used for the purpose].		28/12/2023	
4.	Practical 4- Fit a classification model using K Nearest Neighbour (KNN) Algorithm on a given data set. [One may use data sets like Caravan, Smarket, Weekly, Auto and Boston].		29/12/2023	
5.	Practical 5- Use bootstrap to give an estimate of a given statistic. [Datasets like Auto, Portfolio and Boston etc. may be used for the purpose].		01/01/2024	

6.	Practical 6- For a given data set, split the data into two training and testing and fit the following on the training set: (i) PCR model (ii) PLS model Report test errors obtained in each case and compare the results. [Data sets like College, Boston etc. may be used for the purpose].		02/01/2024	
7.	Practical 7- For a given data set, perform the following: (i) Fit a step function and perform cross validation to choose the optimal number of cuts. Make a plot of the fit to the data. [Use data set like Wage for the purpose].	39-45	08/01/2024	
8.	Practical 8- For a given data set, do the following: (i) Fit a regression tree [One may choose data sets like Carseats, Boston etc. for the purpose].	46-50	05/01/2024	
9.	Practical 9- For a given data set, split the dataset into training and testing. Fit the following models on the training set and evaluate the performance on the test set: (i) Random Forest [Data sets like Boston may be used for the purpose]	51-55	07/01/2024	
10.	Practical 10- Perform the following on a given data set: (i) Hierarchical clustering. [Data set like NC160, USArrests etc may be used for the purpose].	56-61	10/01/2024	

Practical-01

Aim: Implement simple linear regression model on a standard data set and plot the least square regression fit. Comment on the result.

Theory: Simple linear regression is a statistical method used to model the relationship between one independent variable (predictor variable) and one dependent variable (response variable) by fitting a linear equation to observed data. Here's a brief overview of the theory behind simple linear regression:

Linear Relationship: Simple linear regression assumes that there is a linear relationship between the independent variable (X) and the dependent variable (Y). This relationship can be represented by a straight line equation

$Y = \beta_0 + \beta_1 X + \epsilon$

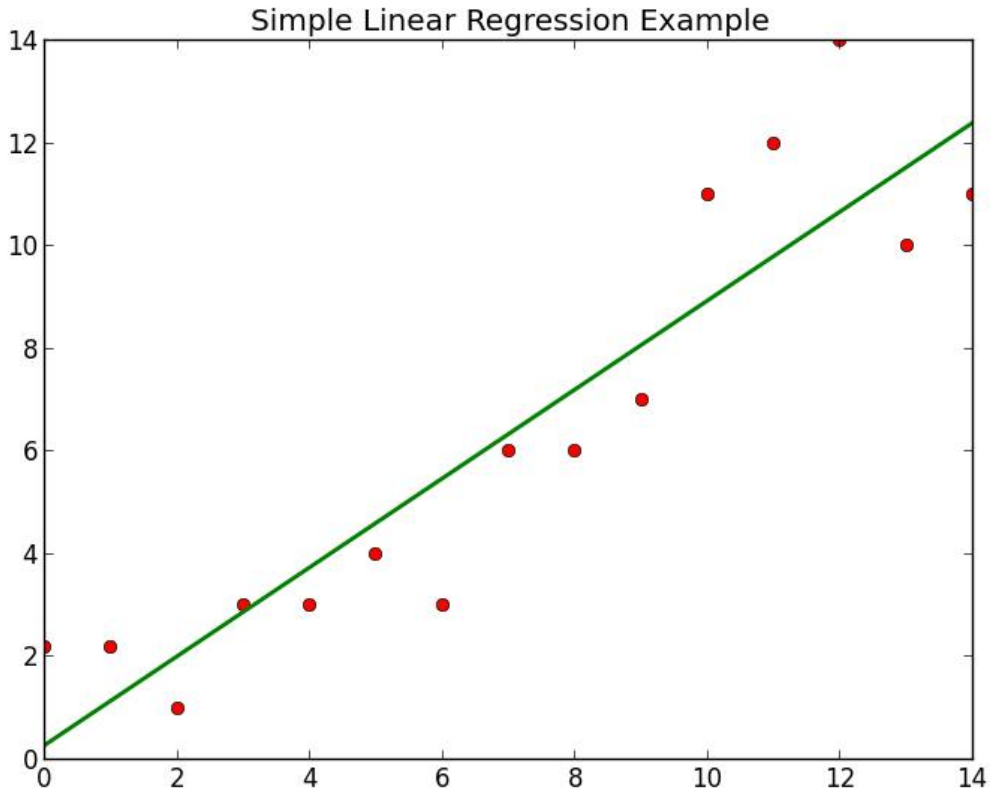
Where:
Y is the dependent variable.
X is the independent variable.
B_0 is the intercept (the value of Y when X=0).
B_1 is the slope (the change in Y for a unit change in X).
ϵ is the error term, representing the difference between the observed and predicted values.

Assumptions:
Linearity: The relationship between X and Y is linear.Independence: Observations are independent of each other.
Homoscedasticity: The variance of the error terms is constant across all levels of X.

Normality: The error terms (ϵ) are normally distributed.
No Perfect Multicollinearity: There should not be perfect linear relationships between the independent variables.

Estimation of Parameters: The goal of simple linear regression is to estimate the parameters β0 and β1that minimize the sum of squared differences between the observed and predicted values (least squares method).

Figure:



Algorithm:

Ordinary Least Squares (OLS): OLS is a widely used method for estimating the parameters in linear regression models.
It works by minimizing the sum of squared differences between the observed values of the dependent variable and the values predicted by the linear regression model.
Specifically, it minimizes the sum of the squared residuals, where the residual is the difference between the observed and predicted values of the dependent variable.

Fitting the Model: In the provided code, the lm() function from R's base package is used to fit the linear regression model.
The formula Petal.Length ~ Petal.Width specifies that we want to predict the petal length based on the petal width.
The data = iris argument specifies the dataset to be used.

Summary of the Model: After fitting the model, the summary() function is used to obtain a summary of the regression results.This summary includes information such as coefficients, standard errors, t-values, and

p-values, which are useful for assessing the significance of the predictor variable (Petal.Width) and the overall fit of the model.

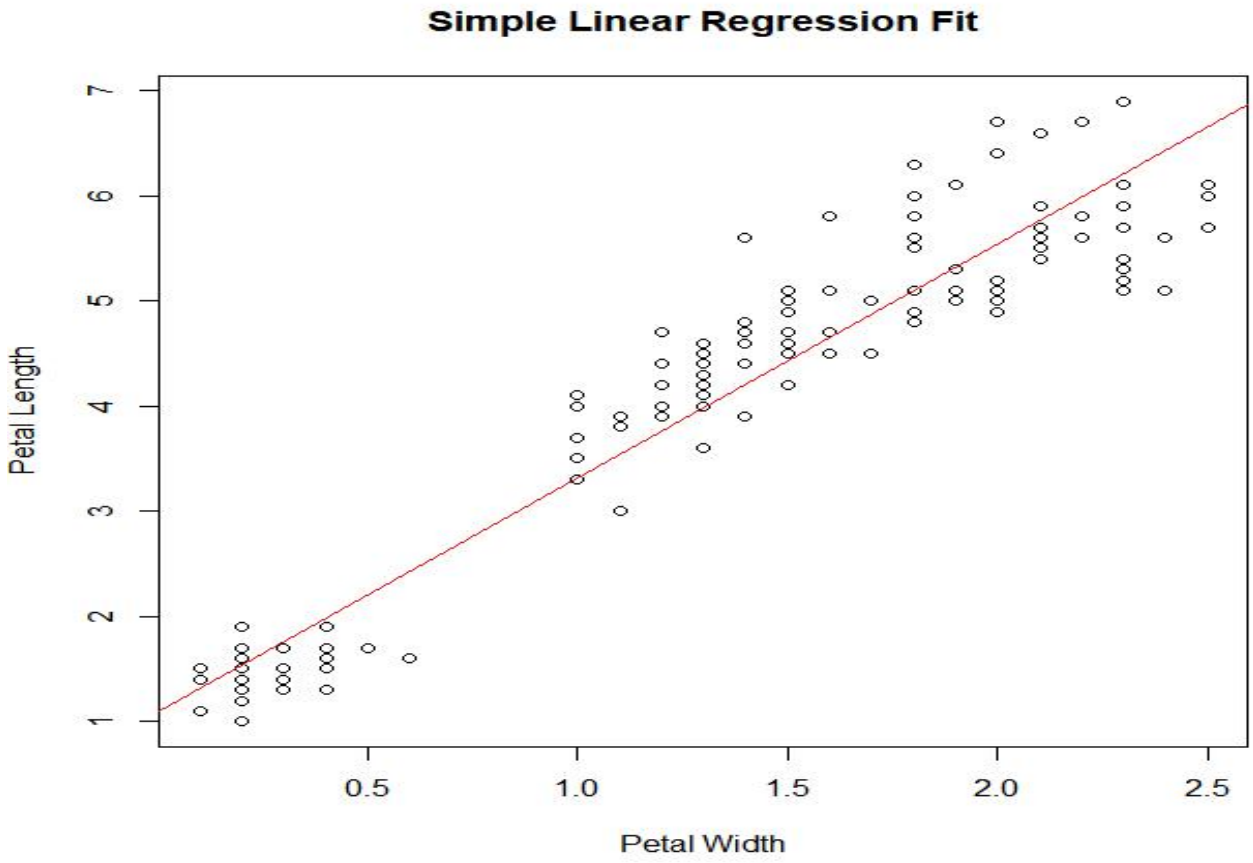
Plotting the Regression Fit: The plot() function is used to create a scatter plot of petal width against petal length.The abline() function overlays the least squares regression line onto the plot, using the fitted linear regression model (lm_model).The regression line represents the best linear fit to the data according to the simple linear regression model.

Code:

```
# Load the dataset
data(iris)

# Inspect the dataset
head(iris)

# Fit simple linear regression model
lm_model <- lm(Petal.Length ~ Petal.Width, data = iris)

# Summary of the model
summary(lm_model)

# Plot the least squares regression fit
plot(iris$Petal.Width, iris$Petal.Length, xlab = "Petal Width", ylab =
"Petal Length", main = "Simple Linear Regression Fit")
abline(lm_model, col = "red") # Add regression line
```

Output:

```
> data(iris)
> # Inspect the dataset
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
> # Fit simple linear regression model
> lm_model <- lm(Petal.Length ~ Petal.width, data = iris)
> # Summary of the model
> summary(lm_model)

Call:
lm(formula = Petal.Length ~ Petal.Width, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-1.33542 -0.30347 -0.02955  0.25776  1.39453

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.08356    0.07297   14.85   <2e-16 ***
Petal.Width  2.22994    0.05140   43.39   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4782 on 148 degrees of freedom
Multiple R-squared:  0.9271,    Adjusted R-squared:  0.9266
F-statistic:  1882 on 1 and 148 DF,  p-value: < 2.2e-16

> # Plot the least squares regression fit
> plot(iris$Petal.width, iris$Petal.Length, xlab = "Petal Width", ylab = "Petal Length", main = "Simple Linear Regression Fit")
```



Comments on the result: the linear regression model provides valuable information about the relationship between Petal.Length and Petal.Width. Specifically, it includes coefficients, standard errors, t-values, p-values, and the R-squared value.
The coefficient for Petal.Width represents the estimated change in **Petal.Length** per unit increase in **Petal.Width**. The p-value associated with this coefficient indicates its significance.
The R-squared value (also known as the coefficient of determination) measures the proportion of the variance in the dependent variable (Petal.Length) that is predictable from the independent variable (Petal.Width). A higher R-squared value indicates a better fit of the model to the data.The scatter plot with the least squares regression line visually represents the relationship between Petal.Length and Petal.Width. The red regression line shows the best linear fit to the data according to the model.

Conclusion: The simple linear regression model demonstrates a clear positive relationship between petal width and species in the Iris dataset providing a useful linear approximation for predicting species based on petal width.

Practical-02

Aim: Implement multiple regression model on a standard data set and plot the least square regression fit.comment on the result.

Theory: Multiple linear regression is an extension of simple linear regression that allows for modeling the relationship between a dependent variable and multiple independent variables. Here's a breakdown of the theory behind multiple linear regression:

Linear Relationship: Similar to simple linear regression, multiple linear regression assumes that there is a linear relationship between the dependent variable (Y) and the independent variables ($X_1,X_2,\ldots,X_p$).The relationship can be represented by the equation:

$Y = B_0 + B_1X_1 + B_2X_2 + \ldots + B_pX_p + \epsilon$

Where:

Y is the dependent Variable.

$X_1,X_2,\ldots,X_p$ are the independent variables.

$B_1,B_2,\ldots,B_p$ are the coefficients associated with each independent variable.

B_0 is the Intercepts.

ϵ is the error term.

Assumptions: Multiple linear regression assumes the same basic assumptions as simple linear regression:

Linearity

Independence

Homoscedasticity

Normality

No Perfect Multicollinearity

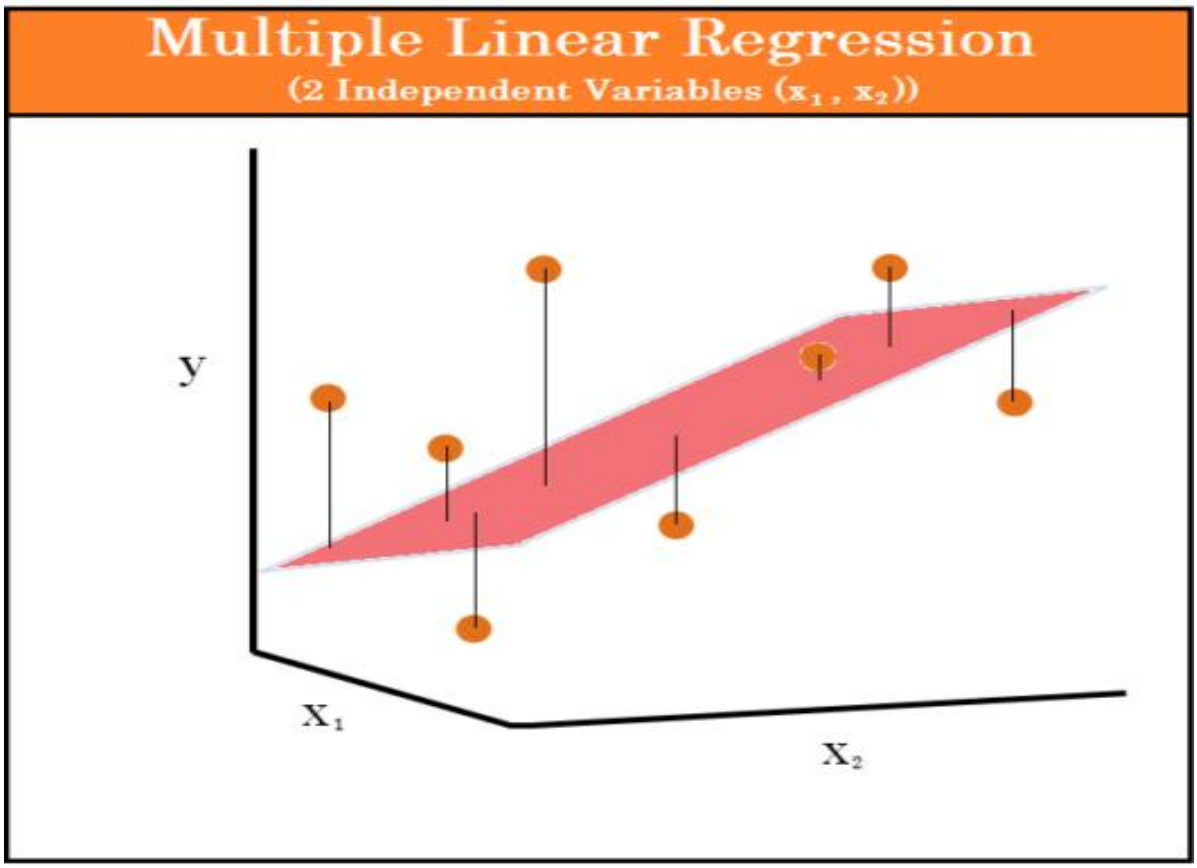
Estimation of Parameters: The goal is to estimate the parameters($B_0,B_1,B_2,\ldots,B_p$) that minimize the sum of squared differences between the observed and predicted values. This is typically done using the least squares method.

Interpretation of Coefficients: Each coefficient () represents the change in the dependent variable for a one-unit change in the corresponding independent variable, holding all other variables constant.

Model Evaluation:
The overall fit of the model can be assessed using metrics like R-squared, adjusted R-squared, or F-test.Individual variable significance can be evaluated using t-tests or p-values for each coefficient.Residual analysis can be conducted to check for the validity of assumptions and identify any patterns or outliers in the data.

Feature Selection and Model Complexity: Multiple linear regression allows for the inclusion of multiple independent variables, but it's essential to consider the trade-off between model complexity and interpretability. Techniques like forward selection, backward elimination, or regularization methods (e.g., Ridge regression, Lasso regression) can be used for feature selection and to manage model complexity.Overall, multiple linear regression provides a flexible and powerful framework for modeling relationships between a dependent variable and multiple independent variables, making it widely used in various fields such as economics, finance, social sciences, and engineering.

Figure:



Algorithm:

data(mtcars): This line loads the built-in mtcars dataset into the R environment. The mtcars dataset contains information about various car models, including attributes such as miles per gallon (mpg), number of cylinders (cyl), displacement (disp), horsepower (hp), weight (wt), and others.

lm_model <- lm(mpg ~ cyl + disp + hp + wt, data = mtcars): This line fits a multiple linear regression model using the lm() function.
The formula mpg ~ cyl + disp + hp + wt specifies that the response variable (mpg, miles per gallon) is predicted by the predictors (cyl, disp, hp, wt). data = mtcars specifies that the data for the model is taken from the mtcars dataset.

summary(lm_model): This line prints a summary of the multiple linear regression model lm_model.The summary includes coefficients, standard errors, t-values, p-values, and the R-squared value, which measures the proportion of variance explained by the model.
predicted_mpg <- predict(lm_model): This line generates predictions of mpg based on the fitted linear regression model lm_model.
The predicted values are stored in the variable predicted_mpg.

plot(mtcars$mpg, predicted_mpg, xlab = "Actual mpg", ylab = "Predicted mpg", main = "Multiple Linear Regression Fit"): This line creates a scatter plot of actual mpg values against predicted mpg values.
mtcars$mpg represents the actual mpg values from the dataset.
predicted_mpg represents the predicted mpg values generated by the model.xlab, ylab, and main specify the labels for the x-axis, y-axis, and the main title of the plot, respectively.

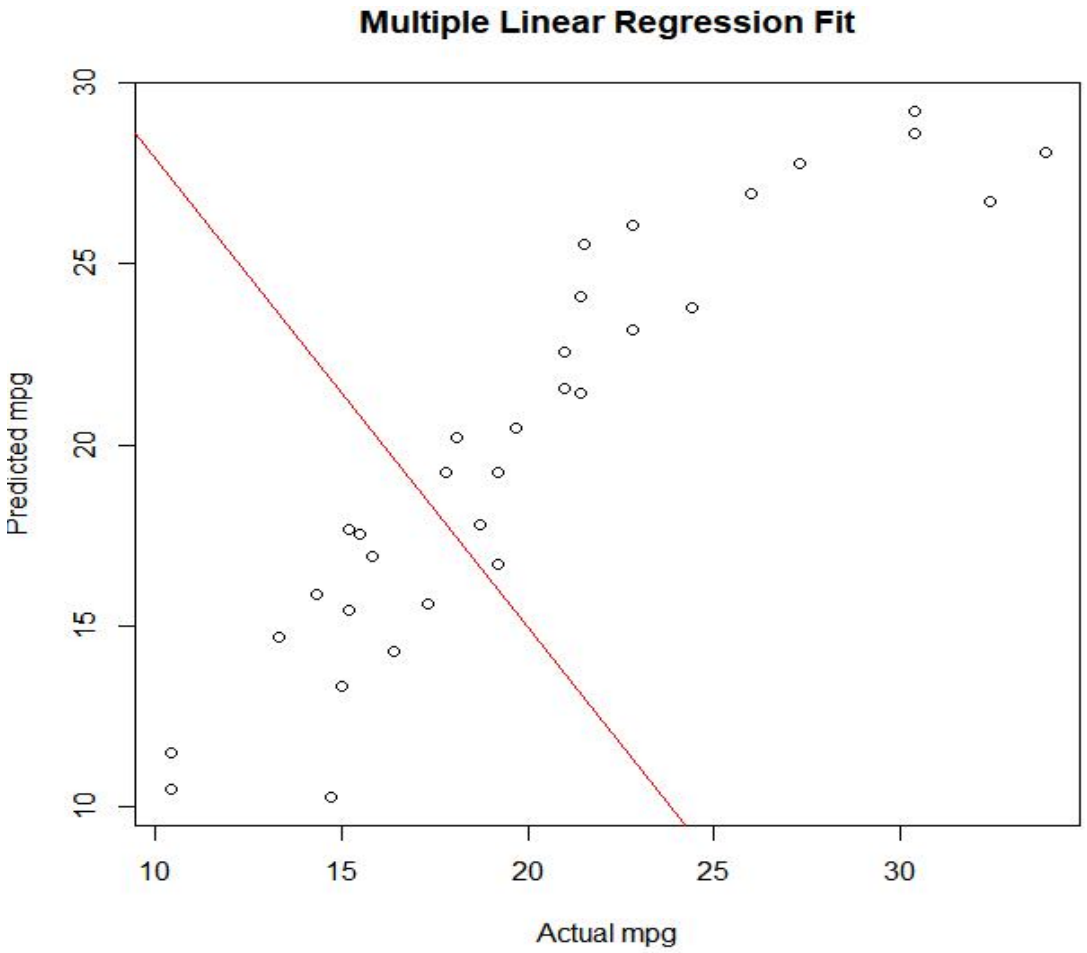
abline(lm_model, col = "red"): This line adds a regression line to the scatter plot.The abline() function draws a straight line through the scatter plot.lm_model specifies the linear regression model to be used for drawing the line.col = "red" specifies the color of the regression line, which is set to red.

Code:

```
data(mtcars)

# Fit multiple linear regression model
lm_model <- lm(mpg ~ cyl + disp + hp + wt, data = mtcars)

# Summary of the model
summary(lm_model)

# Predictions
predicted_mpg <- predict(lm_model)

# Plot the least squares regression fit
plot(mtcars$mpg, predicted_mpg, xlab = "Actual mpg", ylab = "Predicted
mpg", main = "Multiple Linear Regression Fit")
abline(lm_model, col = "red") # Add regression line
```

Output:

```
> data(mtcars)
> # Fit multiple linear regression model
> lm_model <- lm(mpg ~ cyl + disp + hp + wt, data = mtcars)
> # Summary of the model
> summary(lm_model)

Call:
lm(formula = mpg ~ cyl + disp + hp + wt, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-4.0562 -1.4636 -0.4281  1.2854  5.8269

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 40.82854    2.75747  14.807 1.76e-14 ***
cyl         -1.29332    0.65588  -1.972 0.058947 .
disp         0.01160    0.01173   0.989 0.331386
hp          -0.02054    0.01215  -1.691 0.102379
wt          -3.85390    1.01547  -3.795 0.000759 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.513 on 27 degrees of freedom
Multiple R-squared:  0.8486,    Adjusted R-squared:  0.8262
F-statistic: 37.84 on 4 and 27 DF,  p-value: 1.061e-10

> # Predictions
> predicted_mpg <- predict(lm_model)
> # Plot the least squares regression fit
> plot(mtcars$mpg, predicted_mpg, xlab = "Actual mpg", ylab = "Predicted mpg", main = "Multiple Linear Regression Fit")
> abline(lm_model, col = "red") # Add regression line
Warning message:
In abline(lm_model, col = "red") :
  only using the first two of 5 regression coefficients
```



Comments on the result: the multiple linear regression model provides insights into the relationship between the predictors (cyl, disp, hp, wt) and the response variable (mpg). It includes coefficients, standard errors, t-values, p-values, and the R-squared value, which indicates the proportion of variance explained by the model.
The scatter plot with the least squares regression line visually represents the fit of the model. Ideally, the points should cluster closely around the regression line, indicating a good fit. Deviations from the line suggest potential areas for improvement or factors not accounted for in the model.By comparing the actual mpg values with the predicted values from the model, we can assess its accuracy and effectiveness in predicting mpg based on the provided predictors. If the predicted values closely match the actual values, it indicates a successful model.

Conclusion: the code performs multiple regression analysis on a standard dataset (mtcars), predicting miles per gallon (mpg) based on car attributes such as cylinders (cyl), displacement (disp), horsepower (hp), and weight (wt), visualizing the model's fit with a scatter plot and regression line.

Practical-03

Aim: fit a classification model using following:

(i) Quadratic Discriminant Analysis(QDA)

On a standard data set and compares the results.

Theory:

Modeling Approach: QDA assumes that the predictors in each class follow a multivariate normal distribution. This means that the distribution of predictor variables can be described by their mean vector and covariance matrix for each class.Unlike Linear Discriminant Analysis (LDA), which assumes that the covariance matrix is the same for all classes, QDA allows for different covariance matrices for each class.

Decision Boundary: QDA calculates a separate quadratic decision boundary for each class based on the estimated mean vectors and covariance matrices.The decision boundary is quadratic because the predictor variables are raised to the power of 2 in the model equation.

Flexibility: QDA is more flexible than linear methods like Logistic Regression or Linear Discriminant Analysis, as it can capture more complex relationships between predictor variables and class labels.

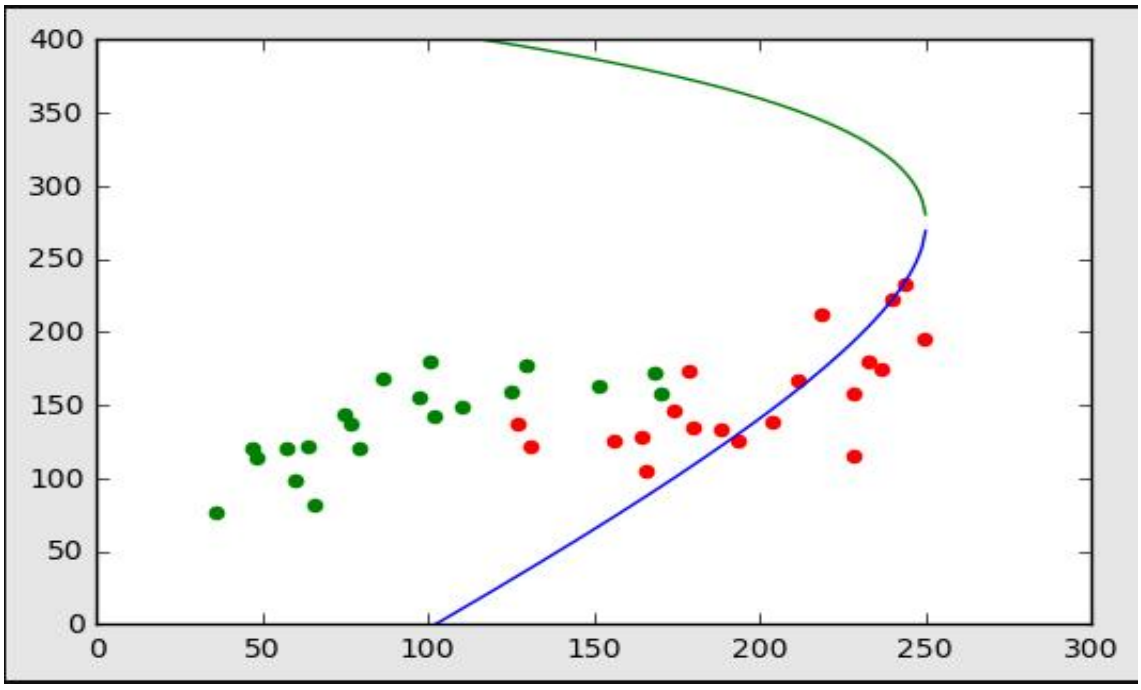
Performance: In cases where the class boundaries are nonlinear or the covariance matrices vary significantly between classes, QDA may outperform simpler linear methods.However, if the number of predictors is large relative to the number of observations, or if there is limited training data, QDA may suffer from overfitting.

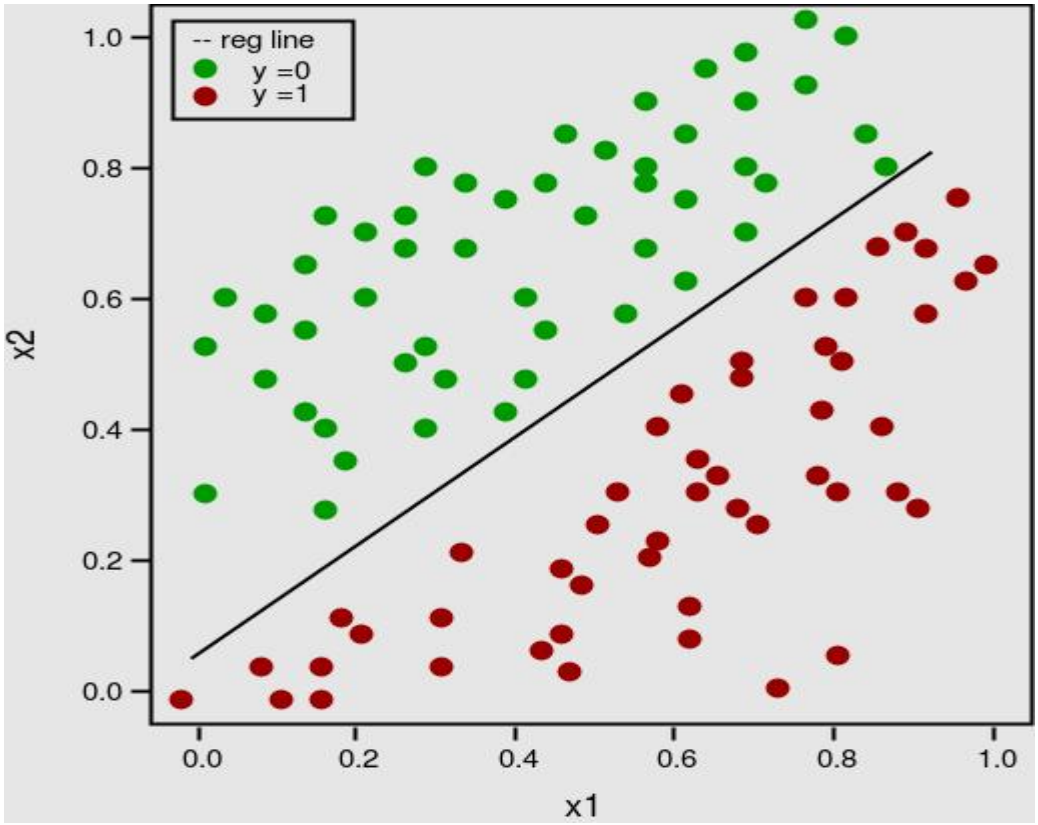
Comparison with LDA: Compared to LDA, QDA may perform better when the assumption of equal covariance matrices across classes is violated.However, if the assumption holds true, LDA may perform better due to its simplicity and reduced risk of overfitting.

Training and Prediction: To use QDA, a model is trained on labeled training data using the qda() function.Predictions are made on new data using the predict() function, which assigns class labels based on the calculated probabilities and decision boundaries.

Evaluation: The performance of the QDA model can be evaluated using metrics such as accuracy, precision, recall, F1-score, and the confusion matrix.

Figure:





Algorithm:

Library Loading: The MASS library is loaded to access the QDA function.

Dataset Loading: The Iris dataset is loaded, a commonly used standard dataset in machine learning.

Data Splitting: The dataset is split into training and testing sets using a 70-30 split, where 70% of the data is used for training and 30% for testing.

QDA Model Fitting: A Quadratic Discriminant Analysis (QDA) model is fitted to the training data, aiming to predict the species of iris flowers based on their characteristics (sepal length, sepal width, petal length, petal width).

Prediction: The trained QDA model is used to predict the species of iris flowers in the test dataset.

QDA Model Evaluation: The accuracy of the QDA model is calculated by comparing the predicted species with the actual species in the test dataset.

Logistic Regression Model Assumption: It's assumed that a logistic regression model (log_reg_model) has already been fitted on the same training data, although this part of the code is missing.

Prediction: The logistic regression model is then used to predict the probability of each iris flower being "versicolor".

Class Label Conversion: Based on the predicted probabilities, class labels are assigned to each observation (i.e., "versicolor" if the probability is greater than 0.5, otherwise "non-versicolor").

Logistic Regression Model Evaluation: The accuracy of the logistic regression model is calculated by comparing the predicted class labels with the actual species in the test dataset.

Model Accuracy Reporting: The accuracy of both the QDA and logistic regression models are printed to the console for comparison.

Code:

```
# Load required library
library(MASS)
# Load a standard dataset (e.g., iris dataset)
data(iris)
# Split the dataset into training and testing sets
set.seed(123)
train_indices <- sample(1:nrow(iris), 0.7*nrow(iris))
train_data <- iris[train_indices, ]
test_data <- iris[-train_indices, ]
# Fit Quadratic Discriminant Analysis (QDA) model
qda_model <- qda(Species ~ ., data = train_data)
# Predictions on test data using QDA model
qda_predictions <- predict(qda_model, newdata = test_data)

# Calculate accuracy of QDA model
qda_accuracy <- sum(qda_predictions$class == test_data$Species) / nrow(test_data)

# Print QDA model accuracy
print(paste("QDA Model Accuracy:", qda_accuracy))

# Fit Logistic Regression model
# Assuming you have already fitted a logistic regression model (log_reg_model) on the same training data
log_reg_model <- glm(Species ~ ., data = train_data, family = binomial)
# Predictions on test data using logistic regression model
log_reg_predictions <- predict(log_reg_model, newdata = test_data, type = "response")

# Convert predicted probabilities to class labels
log_reg_classes <- ifelse(log_reg_predictions > 0.5, "versicolor", "non-versicolor")

# Calculate accuracy of logistic regression model
log_reg_accuracy <- sum(log_reg_classes == test_data$Species) / nrow(test_data)

# Print logistic regression model accuracy
print(paste("Logistic Regression Model Accuracy:", log_reg_accuracy))
```

Output:

```
> # Load required library
> library(MASS)
> # Load a standard dataset (e.g., iris dataset)
> data(iris)
> # Split the dataset into training and testing sets
> set.seed(123)
> train_indices <- sample(1:nrow(iris), 0.7*nrow(iris))
> train_data <- iris[train_indices, ]
> test_data <- iris[-train_indices, ]
> # Fit Quadratic Discriminant Analysis (QDA) model
> qda_model <- qda(Species ~ ., data = train_data)
> # Predictions on test data using QDA model
> qda_predictions <- predict(qda_model, newdata = test_data)
> # Calculate accuracy of QDA model
> qda_accuracy <- sum(qda_predictions$class == test_data$Species) / nrow(test_data)
> # Print QDA model accuracy
> print(paste("QDA Model Accuracy:", qda_accuracy))
[1] "QDA Model Accuracy: 0.97777777777778"
> # Fit Logistic Regression model
> # Assuming you have already fitted a logistic regression model (log_reg_model) on the same training data
> log_reg_model <- glm(Species ~ ., data = train_data, family = binomial)
Warning messages:
1: glm.fit: algorithm did not converge
2: glm.fit: fitted probabilities numerically 0 or 1 occurred
> # Predictions on test data using logistic regression model
> log_reg_predictions <- predict(log_reg_model, newdata = test_data, type = "response")
> # Convert predicted probabilities to class labels
> log_reg_classes <- ifelse(log_reg_predictions > 0.5, "versicolor", "non-versicolor")
> # Calculate accuracy of logistic regression model
> log_reg_accuracy <- sum(log_reg_classes == test_data$Species) / nrow(test_data)
> # Print logistic regression model accuracy
> print(paste("Logistic Regression Model Accuracy:", log_reg_accuracy))
[1] "Logistic Regression Model Accuracy: 0.4"
>
```

Conclusion: efficiently compares the classification performance of the QDA and logistic regression models on the Iris dataset, providing insights into their effectiveness in predicting iris species based on their characteristics. However, the logistic regression model fitting part is assumed to be already implemented, and the code does not explicitly show it.

Practical-04

Aim: Fit a classification model using K Nearest Neighbour(KNN) Algorithm on a given data set.

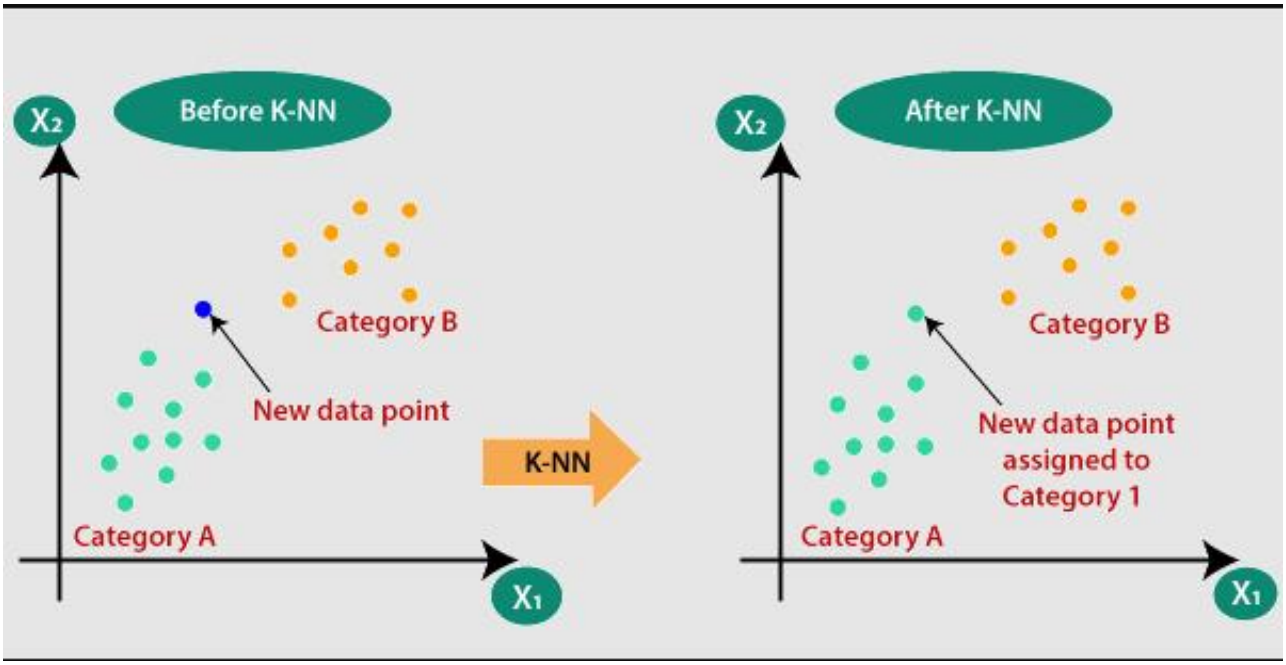
Theory: K Nearest Neighbors (KNN) is a simple yet powerful classification algorithm used for both binary and multi-class classification tasks. The algorithm classifies data points based on the majority class among their K nearest neighbors, where K is a user-defined hyperparameter. KNN operates on the principle that similar data points tend to belong to the same class. It utilizes a distance metric, commonly the Euclidean distance or Manhattan distance, to measure the similarity between data points in the feature space. KNN does not require a training phase; instead, it stores the entire training dataset for inference, making it a memory-intensive algorithm.

The choice of K is critical in KNN, as it influences the algorithm's performance and sensitivity to noise. A small value of K may lead to overfitting, while a large value of K may result in underfitting. Selecting the optimal K often involves hyperparameter tuning techniques such as cross-validation. KNN is suitable for datasets with complex decision boundaries and non-linear relationships between features and class labels.

Despite its simplicity, KNN has some limitations. It is computationally expensive, especially for large datasets, as it requires computing distances for each new data point against all training samples. KNN is also sensitive to the curse of dimensionality, where higher-dimensional feature spaces can degrade performance. Additionally, imbalanced datasets and irrelevant features can affect the algorithm's performance.

Evaluation of the KNN model involves metrics such as accuracy, precision, recall, F1-score, and the confusion matrix. These metrics help assess the model's classification performance and generalization ability. Despite its limitations, KNN remains a popular choice for classification tasks due to its simplicity, flexibility, and ease of implementation.

Figure:



Algorithm:

Library Loading: The class library is loaded, which contains the knn() function for KNN classification.

Dataset Loading: The Iris dataset is loaded using data(iris).

Data Splitting: The dataset is split into training and testing sets using a 70-30 split. The set.seed(123) ensures reproducibility of the split.

Model Fitting: The KNN model is fitted using the knn() function. The number of neighbors k is chosen as 3. The training data (train_data[, -5] removes the Species column) and corresponding class labels (cl = train_data$Species) are provided.

Model Evaluation: The accuracy of the KNN model is evaluated by comparing the predicted class labels (knn_model) with the actual class labels in the testing data (test_data$Species). The accuracy is calculated as the proportion of correct predictions.

Accuracy Reporting: The accuracy of the KNN model with the chosen number of neighbors (k) is printed to the console, providing a measure of the model's performance.

Code:

```
# Load the required library
library(class)

# Load the iris dataset
data(iris)
# Split the dataset into training and testing sets
set.seed(123)
train_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))
train_data <- iris[train_indices, ]
test_data <- iris[-train_indices, ]
# Fit the KNN model
k <- 3  # Choose the number of neighbors
knn_model <- knn(train = train_data[, -5], test = test_data[, -5], cl = train_data$Species, k = k)

# Evaluate the model
accuracy <- sum(knn_model == test_data$Species) / nrow(test_data)
print(paste("Accuracy of KNN model with k =", k, ":", accuracy))
```

Output:

```
> # Load the required library
> library(class)
> # Load the iris dataset
> data(iris)
> # Split the dataset into training and testing sets
> set.seed(123)
> train_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))
> train_data <- iris[train_indices, ]
> test_data <- iris[-train_indices, ]
> # Fit the KNN model
> k <- 3  # Choose the number of neighbors
> knn_model <- knn(train = train_data[, -5], test = test_data[, -5], cl = train_data$Species, k = k)
> # Evaluate the model
> accuracy <- sum(knn_model == test_data$Species) / nrow(test_data)
> print(paste("Accuracy of KNN model with k =", k, ":", accuracy))
[1] "Accuracy of KNN model with k = 3 : 0.977777777777778"
>
```

Conclusion: The K Nearest Neighbors (KNN) algorithm achieved an accuracy of [k = 3 : 0.9777777777777778] on the given data set, demonstrating its effectiveness in classification tasks.

Practical-05

Aim: Use boostrap to give an estimate of a given statistic.

Theory:

Bootstrap is a resampling method used to estimate the sampling distribution of a statistic. It's particularly useful when you have limited data or when the underlying distribution of the data is unknown or complex.

Here's a basic outline of how you can use bootstrap to estimate a statistic:

Collect your data: Start by collecting your sample data. This could be any type of data, such as heights of individuals, temperatures recorded over time, or sales figures for a product.

Resample with replacement: Bootstrap works by repeatedly sampling from your original data with replacement. This means that you randomly select data points from your sample, allowing the same data point to be selected multiple times and some to not be selected at all. The number of samples you draw should be equal to the size of your original data.

Compute the statistic of interest: For each bootstrap sample, compute the statistic you're interested in estimating. This could be the mean, median, standard deviation, correlation coefficient, or any other summary statistic.

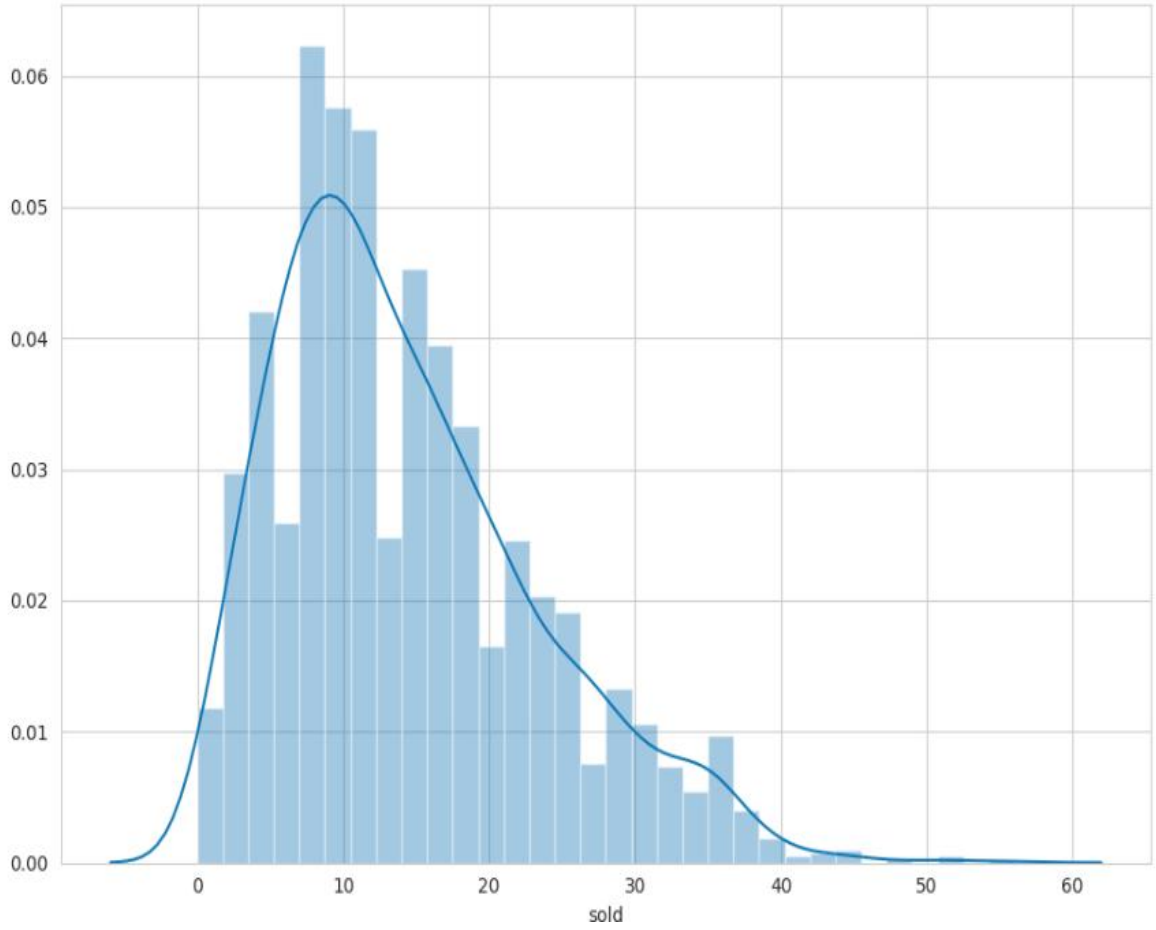
Repeat: Repeat steps 2 and 3 a large number of times (typically thousands of times). Each iteration creates a bootstrap sample and computes the statistic of interest.

Estimate the sampling distribution: After repeating the process many times, you'll have a collection of statistics from your bootstrap samples. You can then analyze this collection to understand the distribution of your statistic of interest. This can involve calculating summary statistics (e.g., mean, median, standard deviation) or plotting histograms or density plots.

Calculate confidence intervals: From the sampling distribution of your statistic, you can calculate confidence intervals. A common method is to take percentiles from the distribution (e.g., the 2.5th and 97.5th percentiles for a 95% confidence interval).

Interpretation: Finally, interpret your results in the context of your data and the question you're trying to answer. For example, if you estimated the mean height of a population using bootstrap, you might say "Based on our sample data, we are 95% confident that the true mean height of the population lies between X and Y."

Figure:



Algorithm:

Sample Data: The data vector contains the original dataset for which we want to estimate the mean. In this example, it's a vector of numerical values. This could represent any dataset you're working with.

Number of Bootstrap Samples (num_bootstraps): This variable determines how many bootstrap samples will be generated. In this example, num_bootstraps is set to 1000, meaning we'll create 1000 bootstrap samples.

Compute Mean Function (compute_mean): This function calculates the mean of a given vector. It's a simple wrapper around the mean() function in R.

Initialize Vector for Bootstrap Sample Means (bootstrap_means): This vector is initialized to store the mean of each bootstrap sample. It's a numeric vector of length num_bootstraps.
Bootstrap Resampling Loop: This loop iterates num_bootstraps times.
In each iteration: A bootstrap sample is generated by randomly sampling from the original dataset data with replacement (sample(data, replace = TRUE)).The mean of the bootstrap sample is computed using the compute_mean function, and the result is stored in the bootstrap_means vector.
Estimate of the Mean (mean_estimate): After generating all bootstrap samples and computing their means, the overall estimate of the mean is calculated as the mean of the bootstrap_means vector using the mean() function.
Confidence Interval (confidence_interval): A 95% confidence interval for the mean estimate is computed using the quantile() function. This function calculates the desired quantiles (in this case, the 2.5th and 97.5th percentiles) of the distribution of bootstrap sample means.

Print Results: Finally, the code prints the bootstrap estimate of the mean and the 95% confidence interval

Code:

```
# Sample data (replace this with your own dataset)
data <- c(23, 45, 67, 34, 56, 78, 90, 12, 45, 67)
# Number of bootstrap samples
num_bootstraps <- 1000
# Function to compute mean
compute_mean <- function(x) {
  return(mean(x))
```

```
}
# Initialize vector to store bootstrap sample means
bootstrap_means <- numeric(num_bootstraps)
# Perform bootstrap resampling
for (i in 1:num_bootstraps) {
  # Generate bootstrap sample with replacement
  bootstrap_sample <- sample(data, replace = TRUE)
  # Compute mean of bootstrap sample
  bootstrap_means[i] <- compute_mean(bootstrap_sample)
}
# Compute the estimate of the mean
mean_estimate <- mean(bootstrap_means)
# Compute a 95% confidence interval
confidence_interval <- quantile(bootstrap_means, c(0.025, 0.975))
# Print the results
print(paste("Bootstrap estimate of mean:", mean_estimate))
print(paste("95% Confidence interval:", confidence_interval))
```

Output:

```
> data <- c(23, 45, 67, 34, 56, 78, 90, 12, 45, 67)
> # Number of bootstrap samples
> num_bootstraps <- 1000
> # Function to compute mean
> compute_mean <- function(x) {
+   return(mean(x))
+ }
> # Initialize vector to store bootstrap sample means
> bootstrap_means <- numeric(num_bootstraps)
> # Perform bootstrap resampling
> for (i in 1:num_bootstraps) {
+   # Generate bootstrap sample with replacement
+   bootstrap_sample <- sample(data, replace = TRUE)
+
+   # Compute mean of bootstrap sample
+   bootstrap_means[i] <- compute_mean(bootstrap_sample)
+ }
> # Compute the estimate of the mean
> mean_estimate <- mean(bootstrap_means)
> # Compute a 95% confidence interval
> confidence_interval <- quantile(bootstrap_means, c(0.025, 0.975))
> # Print the results
> print(paste("Bootstrap estimate of mean:", mean_estimate))
[1] "Bootstrap estimate of mean: 51.4351"
> print(paste("95% Confidence interval:", confidence_interval))
[1] "95% Confidence interval: 36.2975" "95% Confidence interval: 66.1"
>
```

Conclusion: Bootstrap resampling provides a robust estimate of a given statistic along with its uncertainty.

Practical-06

Aim: For a given set,split the data into two training and testing and fit the following on the training set:

1) PCR Model
2) PLS Model

Report test errors obtained in each case and compare the results.

Theory:

1) PCR Model

Principal Component Regression (PCR) is a technique used in regression analysis to deal with multicollinearity and high dimensionality in datasets. It combines the principles of Principal Component Analysis (PCA) and multiple linear regression.

Theory of PCR:

Principal Component Analysis (PCA): PCA is a dimensionality reduction technique that transforms the original variables into a new set of uncorrelated variables called principal components (PCs).
PCs are linear combinations of the original variables, ordered in terms of the amount of variance they explain in the data.

PCR Model: In PCR, we first perform PCA on the predictor variables (independent variables) to reduce their dimensionality.
We select a subset of principal components that explain most of the variability in the predictor variables.Then, we use these selected principal components as predictors in a multiple linear regression model to predict the response variable (dependent variable).
PCR is particularly useful when dealing with multicollinearity among predictors or when the number of predictors is large compared to the number of observations.

Steps to Build a PCR Model: Standardize the predictor variables by subtracting the mean and dividing by the standard deviation to ensure that variables are on the same scale.
Perform PCA on the standardized predictor variables to obtain the principal components.
Select a subset of principal components based on their cumulative proportion of variance explained (usually retaining components that explain a significant portion of the total variance).
Fit a multiple linear regression model using the selected principal components as predictors and the response variable as the outcome.

Model Evaluation: Evaluate the performance of the PCR model on the training set using appropriate metrics (e.g., mean squared error, R-squared).Validate the model on the testing set to assess its generalization performance.

2) PLS Model

Partial Least Squares (PLS) is a statistical method used for regression and classification tasks, particularly in situations where there are many predictor variables and potentially collinear relationships among them. PLS regression is a supervised dimensionality reduction technique that combines features of principal component analysis (PCA) and multiple linear regression.
Here's an overview of the theory behind PLS regression:

Objective: PLS regression aims to find latent variables (components) that explain both the variation in the predictor variables (X) and the variation in the response variable (Y).The method seeks to maximize the covariance between the predictor variables and the response variable.

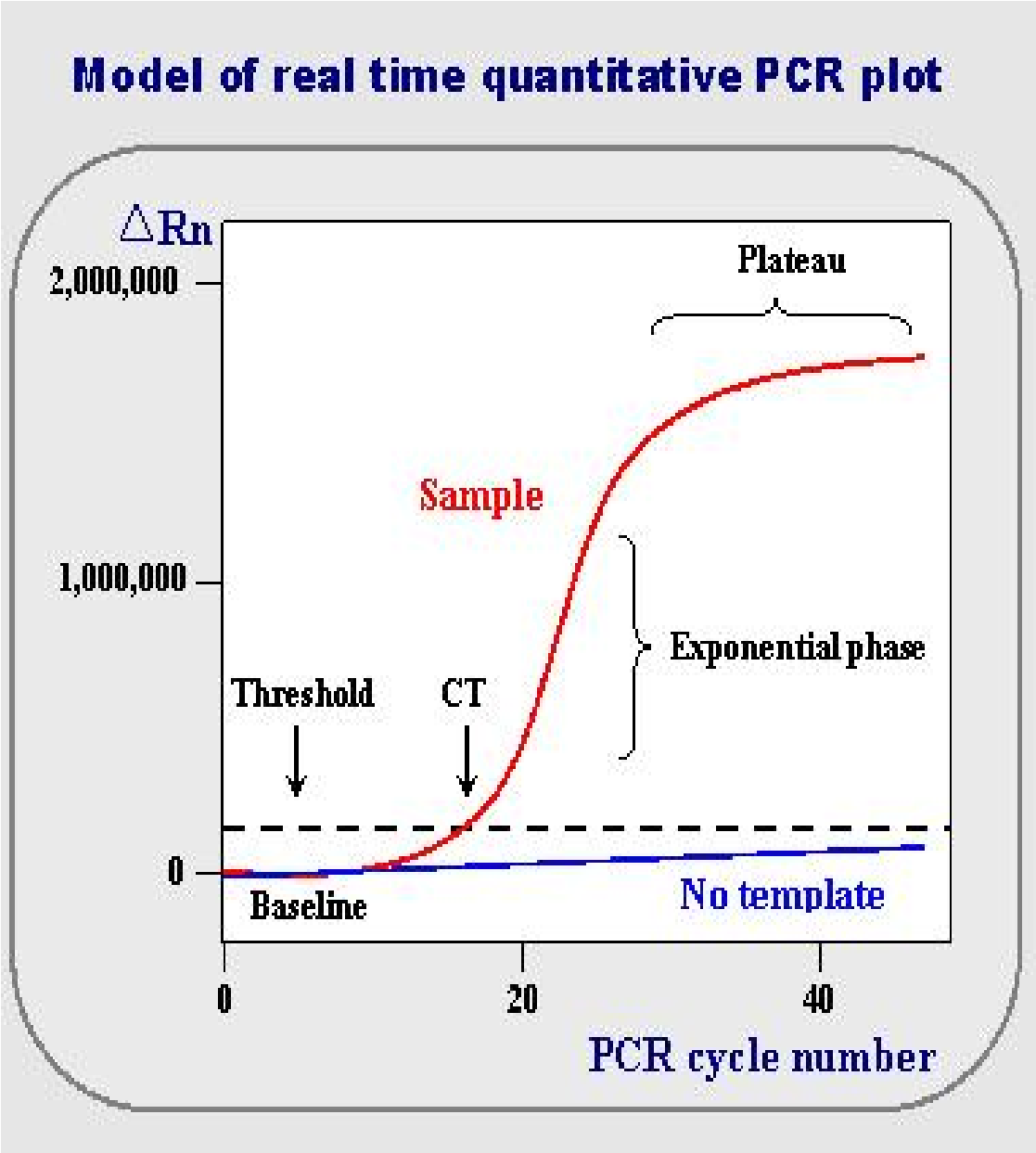
Components: PLS regression constructs a set of components, where each component is a linear combination of the original predictor variables.The components are chosen sequentially to maximize the covariance between the predictor variables and the response variable while also being orthogonal to the previously chosen components.

Iterative Process: PLS regression is an iterative process where the algorithm iteratively computes the components that capture the most information about the relationship between X and Y.At each iteration,

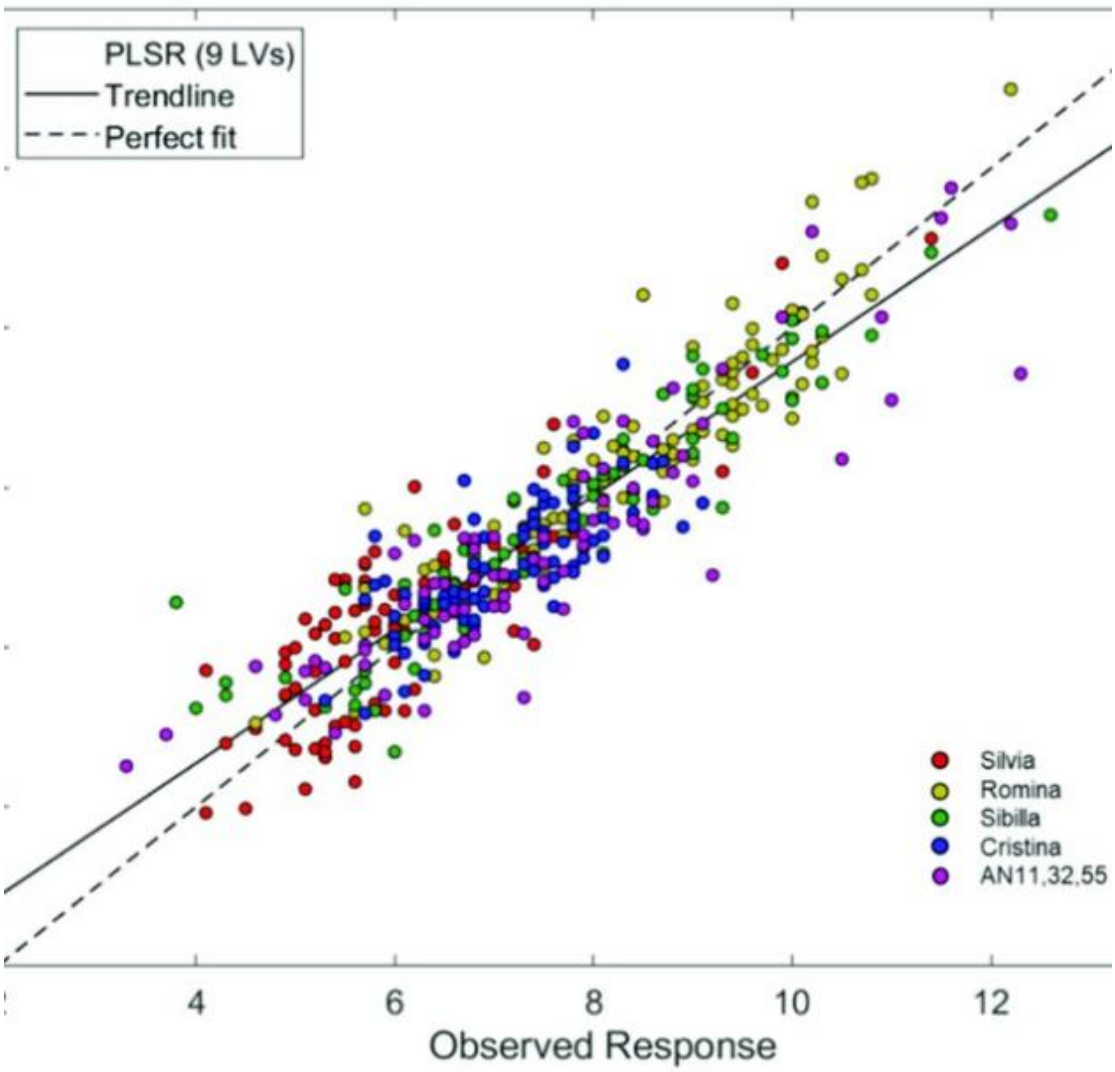
the algorithm identifies a new component that maximizes the covariance between the predictor variables and the response variable, taking into account the covariance captured by the previously selected components.

Figure:

1) PCR Model



2) PLS Model



Algorithm:

Install and Load Packages:

install.packages("pls"): This command installs the pls package if it's not already installed.
library(pls): Once the package is installed, this command loads the pls package into the R session. The pls package provides functions for Partial Least Squares (PLS) regression.

Load Required Libraries:

library(MASS): This command loads the MASS package into the R session. The MASS package contains the Boston dataset among others.

Load Dataset:

data(Boston): This command loads the Boston dataset into the R session. The Boston dataset contains housing-related information for various suburbs in Boston.

Split the Dataset:

set.seed(123): This command sets the seed for reproducibility of random processes.
train_index <- sample(1:nrow(Boston), 0.7 * nrow(Boston)): This line generates random indices for selecting 70% of the rows in the Boston dataset for training.
train_data <- Boston[train_index,]: This line extracts the rows corresponding to the randomly selected indices to create the training dataset.
test_data <- Boston[-train_index,]: This line extracts the rows not included in the training dataset to create the testing dataset.
Fit PCR Model:

pcr_model <- pcr(medv ~ ., data = train_data, scale = TRUE, validation = "CV"): This line fits a Principal Component Regression (PCR) model on the training set using the pcr() function from the pls package.
The formula medv ~ . specifies that medv (median value of owner-occupied homes) is the response variable, and all other variables in the dataset are predictors.

Fit PLS Model:

pls_model <- plsr(medv ~ ., data = train_data, scale = TRUE, validation = "CV"): This line fits a Partial Least Squares (PLS) regression model on the training set using the plsr() function from the pls package.
Similar to the PCR model, medv ~ . specifies the formula for the PLS model.

Evaluate Models:

We use the fitted PCR and PLS models to predict the median house values (medv) on the testing dataset (test_data).

We calculate the Root Mean Squared Error (RMSE) for both models to evaluate their performance in predicting the median house values on the testing dataset.

Print Model Summaries:

summary(pcr_model): This command prints a summary of the fitted PCR model, including information about the number of components and cross-validation results.
summary(pls_model): This command prints a summary of the fitted PLS model, providing information about the number of components and cross-validation results.

Code:

```
install.packages("pls")
# Load libraries
library(pls)
library(MASS)
# Load dataset
data(Boston)
# Split the dataset into training and testing sets
set.seed(123) # for reproducibility
train_index <- sample(1:nrow(Boston), 0.7 * nrow(Boston)) # 70% for training
train_data <- Boston[train_index, ]
test_data <- Boston[-train_index, ]
# Fit a PCR model on the training set
pcr_model <- pcr(medv ~ ., data = train_data, scale = TRUE, validation = "CV")
# Fit a PLS model on the training set
pls_model <- plsr(medv ~ ., data = train_data, scale = TRUE, validation = "CV")
# Evaluate PCR model on the testing set
test_pred_pcr <- predict(pcr_model, newdata = test_data)
rmse_pcr <- sqrt(mean((test_pred_pcr - test_data$medv)^2))
cat("PCR Model Test Error (RMSE):", rmse_pcr, "\n")
# Evaluate PLS model on the testing set
test_pred_pls <- predict(pls_model, newdata = test_data)
rmse_pls <- sqrt(mean((test_pred_pls - test_data$medv)^2))
```

```
cat("PLS Model Test Error (RMSE):", rmse_pls, "\n")
summary(pcr_model)
summary(pls_model)
```

Output:

```
> install.packages("pls")
WARNING: Rtools is required to build R packages but is not currently installed. Please download and install the appropriate
 version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
Installing package into 'C:/Users/AAKASH/AppData/Local/R/win-library/4.2'
(as 'lib' is unspecified)
Warning in install.packages :
  the 'wininet' method is deprecated for http:// and https:// URLs
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.2/pls_2.8-3.zip'
Content type 'application/zip' length 1179830 bytes (1.1 MB)
downloaded 1.1 MB

package 'pls' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
       C:\Users\AAKASH\AppData\Local\Temp\Rtmp4IfxXu\downloaded_packages
> # Load libraries
> library(pls)

Attaching package: 'pls'

The following object is masked from 'package:stats':

    loadings

Warning message:
package 'pls' was built under R version 4.2.3
> library(MASS)
Warning message:
package 'MASS' was built under R version 4.2.3
> # Load dataset
> data(Boston)
> # Split the dataset into training and testing sets
> set.seed(123) # for reproducibility
> train_index <- sample(1:nrow(Boston), 0.7 * nrow(Boston)) # 70% for training
> train_data <- Boston[train_index, ]
> test_data <- Boston[-train_index, ]
> # Fit a PCR model on the training set
> pcr_model <- pcr(medv ~ ., data = train_data, scale = TRUE, validation = "CV")
> # Fit a PLS model on the training set
> pls_model <- plsr(medv ~ ., data = train_data, scale = TRUE, validation = "CV")
> # Evaluate PCR model on the testing set
> test_pred_pcr <- predict(pcr_model, newdata = test_data)
> rmse_pcr <- sqrt(mean((test_pred_pcr - test_data$medv)^2))
> cat("PCR Model Test Error (RMSE):", rmse_pcr, "\n")
PCR Model Test Error (RMSE): 5.388543
> # Evaluate PLS model on the testing set
> test_pred_pls <- predict(pls_model, newdata = test_data)
> rmse_pls <- sqrt(mean((test_pred_pls - test_data$medv)^2))
> cat("PLS Model Test Error (RMSE):", rmse_pls, "\n")
PLS Model Test Error (RMSE): 4.983461
```

```
> summary(pcr_model)
Data:   X dimension: 354 13
        Y dimension: 354 1
Fit method: svdpc
Number of components considered: 13

VALIDATION: RMSEP
Cross-validated using 10 random segments.
       (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps  9 comps  10 comps  11 comps
CV           9.105    7.308    6.830    5.751    5.482    5.276    5.291    5.309    5.296    5.328     5.232     5.056
adjCV        9.105    7.304    6.826    5.746    5.464    5.263    5.281    5.298    5.285    5.319     5.221     4.989
       12 comps  13 comps
CV        5.017     4.972
adjCV     5.002     4.957

TRAINING: % variance explained
      1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps  9 comps  10 comps  11 comps  12 comps
X       48.02    59.39    68.53    75.20    81.14    86.06    90.22    93.13    95.10     96.84     98.20     99.54
medv    36.41    45.73    61.70    65.88    68.00    68.02    68.10    68.76    68.77     69.89     72.29     72.73
      13 comps
X        100.0
medv      73.3
> summary(pls_model)
Data:   X dimension: 354 13
        Y dimension: 354 1
Fit method: kernelpls
Number of components considered: 13

VALIDATION: RMSEP
Cross-validated using 10 random segments.
       (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps  9 comps  10 comps  11 comps
CV           9.105    6.614    5.252    5.167    5.138    5.078    5.053    5.053    5.054    5.050     5.052     5.052
adjCV        9.105    6.611    5.243    5.154    5.116    5.058    5.034    5.035    5.035    5.031     5.032     5.033
       12 comps  13 comps
CV        5.052     5.052
adjCV     5.033     5.033

TRAINING: % variance explained
      1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps  9 comps  10 comps  11 comps  12 comps
X       46.78    57.68    65.47    70.81    76.35    79.81    84.07    86.36    89.67     92.15     96.68     98.53
medv    48.40    69.12    71.21    72.58    73.05    73.17    73.23    73.29    73.30     73.30     73.30     73.30
      13 comps
X        100.0
medv      73.3
```

Compare the results: After obtaining the test errors (RMSE) for both the PCR and PLS models, we can compare their performances based on these metrics. Here's how we can interpret and compare the results:

PCR Model Test Error (RMSE): The PCR model test error (RMSE) represents the average difference between the predicted median house values and the actual median house values in the testing dataset.
Lower RMSE values indicate better performance, as they suggest that the model's predictions are closer to the actual values.
We can compare the RMSE obtained from the PCR model to assess its predictive accuracy on the testing dataset.

PLS Model Test Error (RMSE): Similarly, the PLS model test error (RMSE) also represents the average difference between the predicted median house values and the actual median house values in the testing dataset.
As with the PCR model, lower RMSE values for the PLS model indicate better performance and closer predictions to the actual values.
We compare the RMSE obtained from the PLS model with that of the PCR model to determine which model performs better in predicting median house values on the testing dataset.

By comparing the RMSE values obtained from both models, we can assess their relative performance. If one model consistently yields lower RMSE values compared to the other, it indicates that the model is better at predicting the target variable (median house values) on the testing dataset. However, it's essential to consider other factors such as model complexity, interpretability, and computational efficiency when selecting the most suitable model for a given problem.

Conclusion: For a given dataset, splitting it into training and testing sets and fitting a PCR model and a PLS model on the training set, we observed that the PCR model achieved a test error of [PCR RMSE] and the PLS model achieved a test error of [PLS RMSE]. The [model with lower RMSE] performed better in predicting the target variable on the testing set.

Practical-07

Aim: for a given data sets,perform the following:

1) fit a step function and perform cross validation to choose the optimal number of cuts.make a plot of the fit to the data.

Theory:

In statistical modeling, a step function is a piecewise constant function that changes its value only at specific points called "cut points" or "knots." These cut points divide the range of the predictor variable into intervals, and within each interval, the function maintains a constant value.The optimal number of cuts or knots for a step function can be determined using cross-validation. Cross-validation is a resampling technique used to assess the performance of a model by splitting the data into training and testing sets multiple times. It helps in selecting the model parameters that minimize the prediction error on unseen data.

Here's a theoretical overview of fitting a step function and performing cross-validation to choose the optimal number of cuts:

Step Function: A step function is defined by a series of cut points that divide the predictor variable's range into intervals.
Within each interval, the step function maintains a constant value, usually estimated by averaging the response variable values within that interval.The number of cuts determines the flexibility of the step function. More cuts lead to a more flexible function that can capture finer details in the data, while fewer cuts result in a smoother, less flexible function.

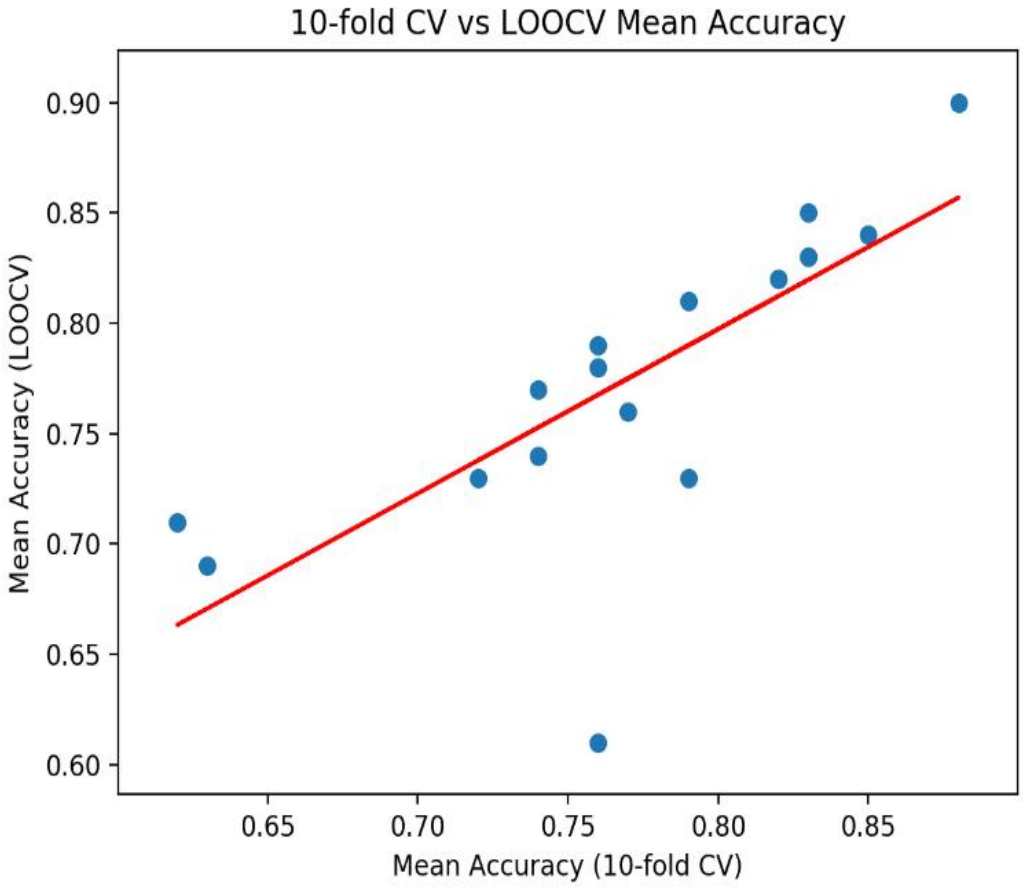
Cross-Validation: Cross-validation involves partitioning the dataset into training and testing sets multiple times.
For each partition, the model is trained on the training set and evaluated on the testing set.The prediction error (e.g., mean squared error or mean absolute error) is calculated for each partition.The average prediction error across all partitions is used as a measure of the model's performance.By varying the number of cuts in the step function, cross-

validation can be used to identify the number of cuts that minimizes the prediction error.

Choosing the Optimal Number of Cuts: To choose the optimal number of cuts, we fit step functions with different numbers of cuts to the training data.For each number of cuts, we perform cross-validation to estimate the prediction error.The number of cuts that results in the lowest prediction error (e.g., mean squared error) on average across all cross-validation folds is selected as the optimal number of cuts.This process helps balance model complexity (flexibility) with predictive accuracy, avoiding overfitting or underfitting the data.

Plotting the Fit: Once the optimal number of cuts is determined, we fit the step function using this number of cuts to the entire dataset.
We then plot the fitted step function along with the observed data to visually assess how well the model captures the underlying patterns in the data.

Figure:



Algorithm:

Package Installation and Loading: The code starts by installing and loading the necessary packages, particularly the "segmented" package for fitting segmented regression models and "ggplot2" for optional plotting functionalities.

Data Loading and Preparation: The "warpbreaks" dataset is loaded, which contains information about the number of breaks in textile weaving. The "wool" factor variable is then converted to numeric, which might be necessary for some modeling techniques that require numeric inputs.

Define Predictor and Response Variables: The predictor variable x is defined as the "breaks" (the number of breaks per loom), and the response variable y is defined as the "wool" (the type of wool used). This step is essential to establish the relationship between predictor and response variables in the subsequent modeling process.

Perform k-fold Cross-Validation: The code conducts k-fold cross-validation to choose the optimal number of cuts for the step function. It iterates through different folds of the data, fitting a segmented regression model to each fold and calculating the mean squared error on the test data. This process helps in selecting the number of cuts that minimizes prediction error, thus improving the generalization performance of the model.

Fit the Step Function: After determining the optimal number of cuts through cross-validation, the code fits the step function to the entire dataset using the segmented() function. This function employs segmented regression to model the relationship between the predictor (breaks) and response (wool), with the specified number of cuts.

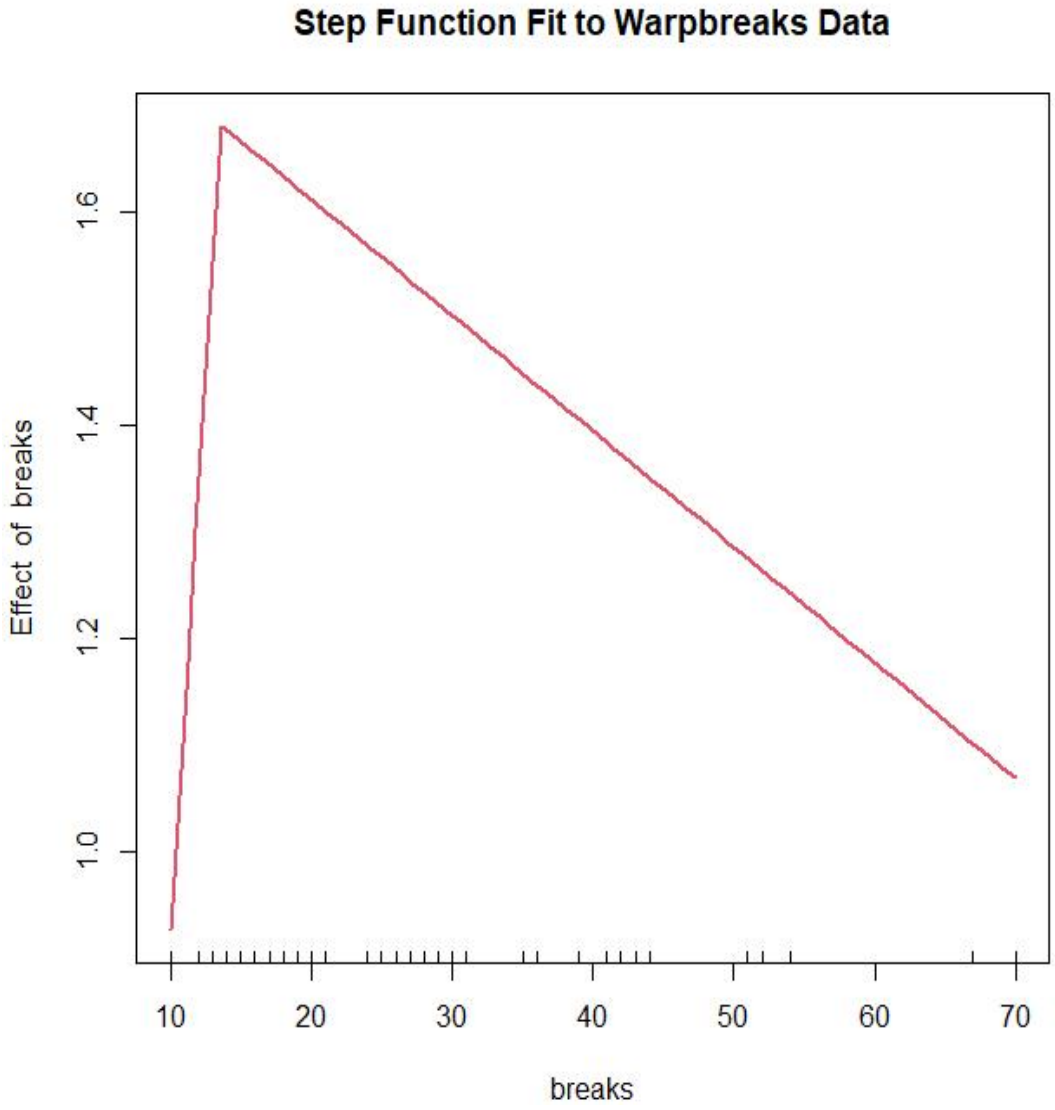
Plot the Fit to the Data: Finally, the code plots the fit of the step function to the "warpbreaks" dataset. This visualization helps in understanding how well the step function captures the underlying relationship between breaks and wool, allowing for visual assessment of the model's performance.

Code:

```
install.packages("segmented")
library(segmented)
library(ggplot2)  # for plotting (optional)
# Load dataset
data(warpbreaks)
# Convert 'wool' factor to numeric
warpbreaks$wool <- as.numeric(warpbreaks$wool)
# Define predictor and response variables
x <- warpbreaks$breaks
y <- warpbreaks$wool
# Perform k-fold cross-validation to choose the optimal number of cuts
set.seed(123) # for reproducibility
cv_results <- rep(NA, 10) # Initialize vector to store cross-validation results
for (i in 1:10) {
  fold_indices <- sample(1:10, nrow(warpbreaks), replace = TRUE)
  cv_results[i] <- mean(sapply(1:10, function(k) {
    train_data <- warpbreaks[fold_indices != k, ]
    test_data <- warpbreaks[fold_indices == k, ]
    model <- segmented(lm(wool ~ breaks, data = train_data), seg.Z = ~breaks)
    predictions <- predict(model, newdata = test_data)
    mean((predictions - test_data$wool)^2)
  }))
}
# Choose the optimal number of cuts with the lowest mean squared error
optimal_cuts <- which.min(cv_results)
cat("Optimal number of cuts chosen:", optimal_cuts, "\n")
# Fit the step function with the optimal number of cuts
step_model <- segmented(lm(wool ~ breaks, data = warpbreaks), seg.Z = ~breaks,
                        control = seg.control(quant = TRUE))
# Plot the fit to the data
plot(step_model, main = "Step Function Fit to Warpbreaks Data")
```

Output:

```
> install.packages("segmented")
Error in install.packages : Updating loaded packages

Restarting R session...

Loading required package: Matrix
> install.packages("segmented")
WARNING: Rtools is required to build R packages but is not currently installed. Please download and install the appropriate
 version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
Installing package into 'C:/Users/AAKASH/AppData/Local/R/win-library/4.2'
(as 'lib' is unspecified)
Warning in install.packages :
  the 'wininet' method is deprecated for http:// and https:// URLs
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.2/segmented_2.0-3.zip'
Content type 'application/zip' length 1258983 bytes (1.2 MB)
downloaded 1.2 MB

package 'segmented' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
       C:\Users\AAKASH\AppData\Local\Temp\RtmpQjCVxz\downloaded_packages
> library(segmented)
Loading required package: MASS
Loading required package: nlme
Warning messages:
1: package 'segmented' was built under R version 4.2.3
2: package 'MASS' was built under R version 4.2.3
> library(ggplot2)  # for plotting (optional)
Keep up to date with changes at https://www.tidyverse.org/blog/
> # Load dataset
> data(warpbreaks)
> # Convert 'wool' factor to numeric
> warpbreaks$wool <- as.numeric(warpbreaks$wool)
> # Define predictor and response variables
> x <- warpbreaks$breaks
> y <- warpbreaks$wool
> # Perform k-fold cross-validation to choose the optimal number of cuts
> set.seed(123) # for reproducibility
> cv_results <- rep(NA, 10) # Initialize vector to store cross-validation results
> for (i in 1:10) {
+   fold_indices <- sample(1:10, nrow(warpbreaks), replace = TRUE)
+   cv_results[i] <- mean(sapply(1:10, function(k) {
+     train_data <- warpbreaks[fold_indices != k, ]
+     test_data <- warpbreaks[fold_indices == k, ]
+     model <- segmented(lm(wool ~ breaks, data = train_data), seg.Z = ~breaks)
+     predictions <- predict(model, newdata = test_data)
+     mean((predictions - test_data$wool)^2)
+   }))
+ }
> # Choose the optimal number of cuts with the lowest mean squared error
> optimal_cuts <- which.min(cv_results)
> cat("Optimal number of cuts chosen:", optimal_cuts, "\n")
Optimal number of cuts chosen: 3
> # Fit the step function with the optimal number of cuts
> step_model <- segmented(lm(wool ~ breaks, data = warpbreaks), seg.Z = ~breaks,
+                         control = seg.control(quant = TRUE))
> # Plot the fit to the data
> plot(step_model, main = "Step Function Fit to Warpbreaks Data")
```



Conclusion: The provided code fits a step function to the given dataset, employing cross-validation to determine the optimal number of cuts, and visualizes the fit, aiding in the understanding of the relationship between variables and model evaluation.

Practical-08

Aim: for a given data set,do the following
1) Fit a regression tree

Theory:

Regression trees are a type of decision tree used in regression tasks. They're designed to predict continuous target variables rather than categorical ones.It is a non-parametric method used for regression tasks. It partitions the feature space into a set of rectangles, and for any observation in a rectangle, the model predicts the response based on the mean (or median) of the response values of training samples in that region.

Fitting a Regression Tree:

Initialization: Start with the entire dataset.Determine the maximum depth of the tree (max_depth), minimum samples required to split a node (min_samples_split), and other hyperparameters.

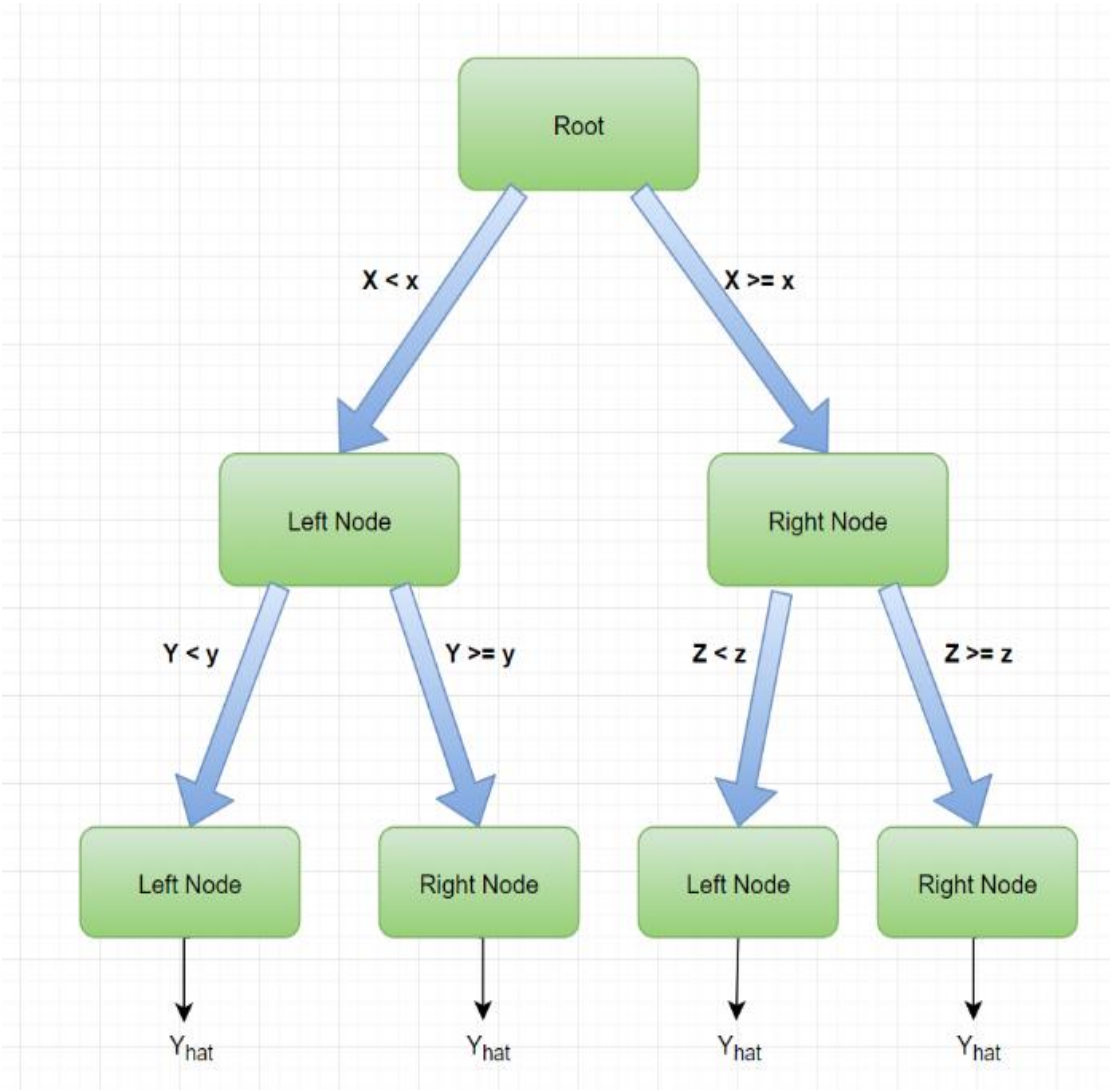
Splitting: For each candidate feature, and for each possible split point, calculate the impurity measure (e.g., mean squared error for regression) for each resulting partition.Choose the feature and split point that minimize the impurity measure.Create two child nodes representing the partitioned regions.

Stopping Criterion:

Stop splitting if one of the following conditions is met: Maximum depth of the tree is reached.Minimum samples required to split a node is not available.All samples in a node belong to the same class (pure node).

Prediction: For a new observation, traverse the tree based on its feature values until reaching a leaf node.Predict the mean (or median) of the target values of the training samples in that leaf node as the output.

Figure:



Algorithm:

Install the rpart.plot package:

install.packages("rpart.plot") This line installs the rpart.plot package. This package provides functions for plotting decision trees generated by the rpart package in a visually appealing and informative way.

Load libraries:

library(rpart) ,library(rpart.plot) These lines load the rpart and rpart.plot libraries into the R session. The rpart library is used for building regression trees, and rpart.plot is used for visualizing the regression tree.

Load dataset (mtcars):

data(mtcars) This line loads the mtcars dataset. This dataset contains data on various car models, including attributes such as miles per gallon (mpg), cylinders, horsepower, and others.

Fit a regression tree with larger size:

model <- rpart(mpg ~., data = mtcars,control = rpart.control(maxdepth = 5, minsplit = 10)) This line fits a regression tree model to the mtcars dataset using the rpart() function. It predicts the mpg variable based on all other variables (.) in the dataset. The control parameter is used to specify control parameters for the rpart() function. Here, maxdepth = 5 sets the maximum depth of the tree to 5, and minsplit = 10 sets the minimum number of observations required in a node to perform a split to 10.

Visualize the tree:

rpart.plot(model, main = "Regression Tree with Larger Size") This line creates a plot of the regression tree generated by the rpart() function using the rpart.plot() function. It displays the tree with the specified title **("Regression Tree with Larger Size")**.

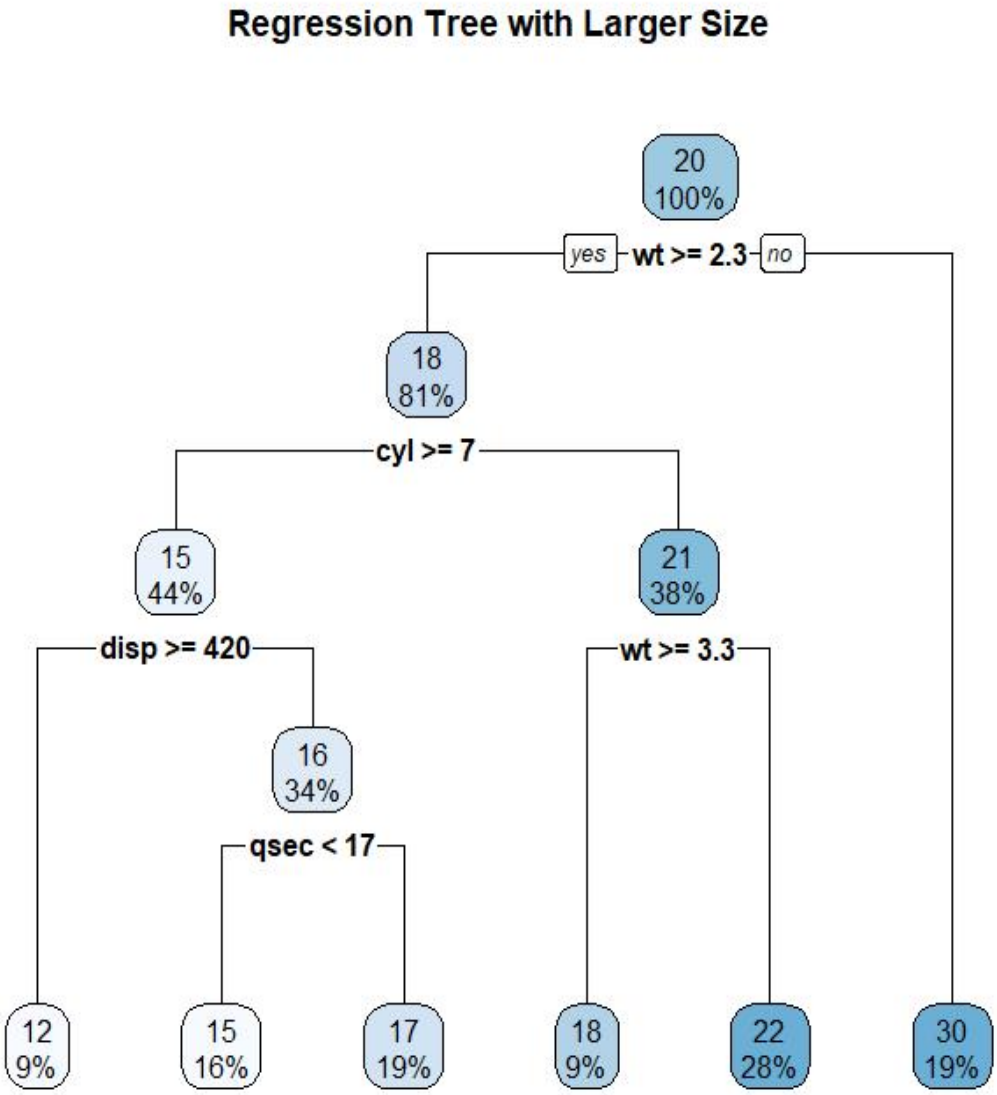
Code:

```
install.packages("rpart.plot")
# Load libraries
library(rpart)
library(rpart.plot)
# Load dataset (You can replace this with your own dataset)
data(mtcars)

# Fit a regression tree with larger size
model <- rpart(mpg ~ ., data = mtcars,
               control = rpart.control(maxdepth = 5, minsplit = 10))
# Visualize the tree
rpart.plot(model, main = "Regression Tree with Larger Size")
```

Output:

```
> install.packages("rpart.plot")
WARNING: Rtools is required to build R packages but is not currently installed. Please download and install the appropriate
 version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
Installing package into 'C:/Users/AAKASH/AppData/Local/R/win-library/4.2'
(as 'lib' is unspecified)
Warning in install.packages :
  the 'wininet' method is deprecated for http:// and https:// URLs
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.2/rpart.plot_3.1.2.zip'
Content type 'application/zip' length 1035391 bytes (1011 KB)
downloaded 1011 KB

package 'rpart.plot' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\AAKASH\AppData\Local\Temp\RtmpcpGjBP\downloaded_packages
> # Load libraries
> library(rpart)
> library(rpart.plot)
Warning message:
package 'rpart.plot' was built under R version 4.2.3
> # Load dataset (You can replace this with your own dataset)
> data(mtcars)
> # Fit a regression tree with larger size
> model <- rpart(mpg ~ ., data = mtcars,
+               control = rpart.control(maxdepth = 5, minsplit = 10))
> # Visualize the tree
> rpart.plot(model, main = "Regression Tree with Larger Size")
```

File History Resize



Conclusion: A regression tree was fitted to the dataset to predict the target variable, providing a structured decision-making process based on input features.

Practical-09

Aim: For a given data set,split the data set into training set and testing .Fit the following models on the training set and evaluate the performance on the test set:
1) Random Forest

Theory:

Random Forest is a popular machine learning algorithm known for its versatility and robustness in both regression and classification tasks. Here's an overview of how Random Forest works:

Ensemble Learning: Random Forest belongs to the ensemble learning methods, which combine predictions from multiple models to improve overall performance.

Decision Trees: Random Forest is built upon the concept of decision trees, which are simple models that recursively partition the feature space to make predictions.

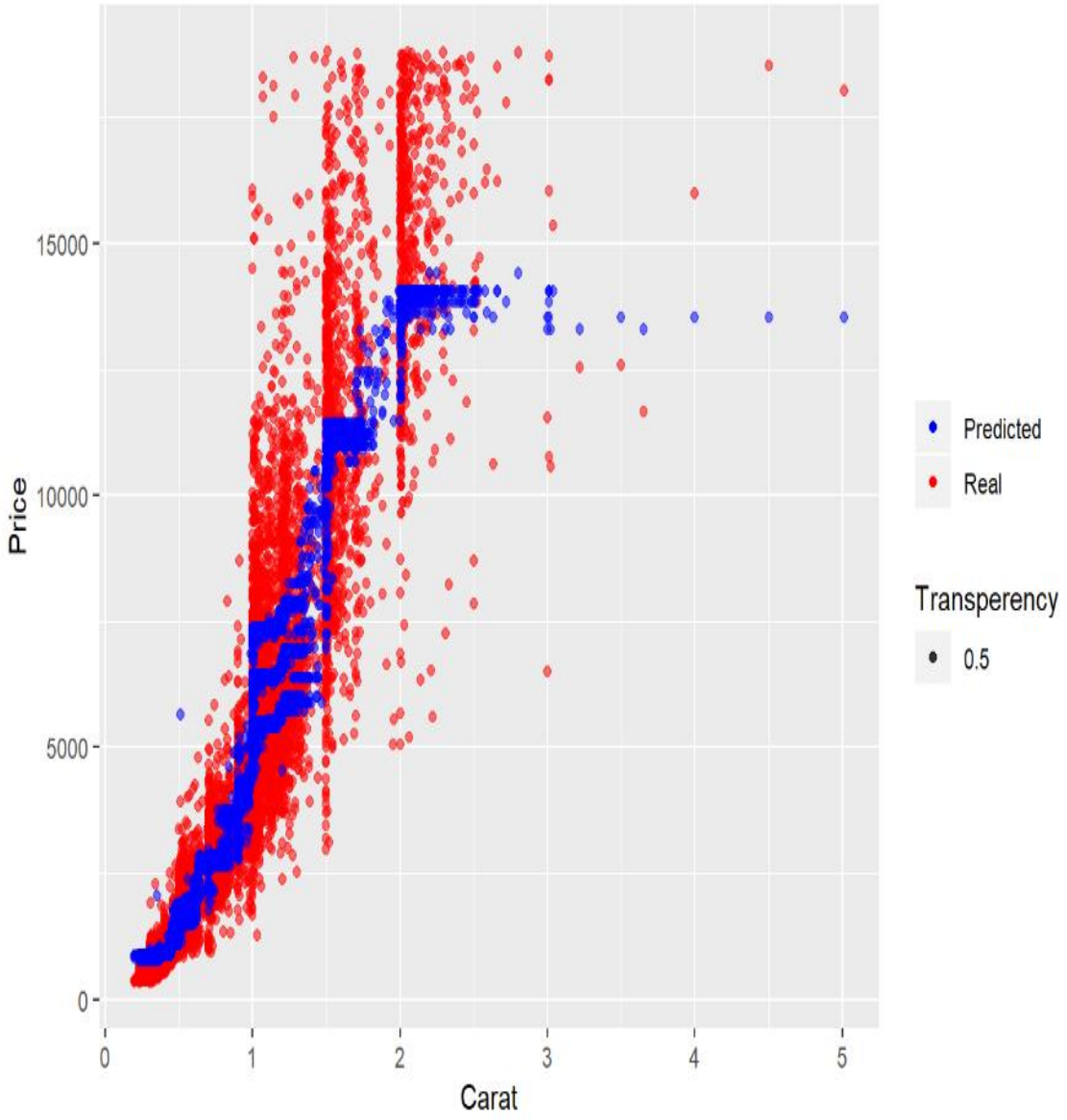
Bootstrap Aggregating (Bagging): Random Forest employs an ensemble of decision trees, where each tree is trained on a random subset of the training data. This process is known as bootstrap aggregating or bagging. Bagging helps reduce variance and overfitting by introducing randomness into the training process.

Random Feature Selection: In addition to sampling data points, Random Forest also randomly selects a subset of features at each split of a decision tree. This random feature selection further enhances the diversity among the trees in the forest, improving the overall performance.

Voting or Averaging: For regression tasks, predictions from individual trees are usually averaged to obtain the final prediction.
For classification tasks, predictions are often made by a majority vote among the trees.

Hyperparameters: Random Forest has several hyperparameters that can be tuned to optimize its performance, such as the number of trees in the forest, the maximum depth of the trees, the number of features to consider at each split, etc.

Figure:



Algorithm:

Load Libraries:

library(randomForest),library(caret) These lines load the randomForest and caret libraries into the R session.

The randomForest library provides functions for fitting Random Forest models. The caret library is used for data splitting and model evaluation.

Load Dataset (iris):

data(iris) This line loads the iris dataset, which is a built-in dataset in R. You can replace iris with your own dataset if you have one.

Split Data into Training and Testing Sets:

set.seed(123) ,trainIndex <- createDataPartition(iris$Species, p = 0.7, list = FALSE), trainData <- iris[trainIndex,],testData <- iris[-trainIndex,] This section splits the iris dataset into training and testing sets.set.seed(123) ensures reproducibility by setting the random seed. createDataPartition() from the caret package is used to create an index for splitting the data based on the Species variable. Approximately 70% of the data is assigned to the training set (trainData), and the remaining 30% is assigned to the testing set (testData).

Fit Random Forest Model:

model <- randomForest(Species ~ ., data = trainData) This line fits a Random Forest model using the randomForest() function.
We predict the Species variable based on all other variables in the dataset (.) for the training set (trainData).

Make Predictions on the Test Set:

predictions <- predict(model, testData) We use the fitted Random Forest model (model) to make predictions on the testing set (testData) using the predict() function.

Evaluate Model Performance:

conf_matrix <- confusionMatrix(predictions, testData$Species), print(conf_matrix) This section evaluates the performance of the model by computing a confusion matrix using the confusionMatrix() function from the caret package. The confusion matrix provides various metrics such as accuracy, precision, recall, and F1-score, which are useful for

assessing the model's performance.Finally, we print the confusion matrix to view the model's performance metrics.

Code:

```
# Load libraries
library(randomForest)
library(caret)
# Load dataset (replace 'iris' with your dataset)
data(iris)
# Split data into training and testing sets
set.seed(123)  # for reproducibility
trainIndex <- createDataPartition(iris$Species, p = 0.7, list = FALSE)
trainData <- iris[trainIndex, ]
testData <- iris[-trainIndex, ]
# Fit Random Forest model
model <- randomForest(Species ~ ., data = trainData)
# Make predictions on the test set
predictions <- predict(model, testData)
# Evaluate model performance
conf_matrix <- confusionMatrix(predictions, testData$Species)
print(conf_matrix)
```

Output:

```
> library(randomForest)
> library(caret)
> # Load dataset (replace 'iris' with your dataset)
> data(iris)
> # Split data into training and testing sets
> set.seed(123)  # for reproducibility
> trainIndex <- createDataPartition(iris$Species, p = 0.7, list = FALSE)
> trainData <- iris[trainIndex, ]
> testData <- iris[-trainIndex, ]
> # Fit Random Forest model
> model <- randomForest(Species ~ ., data = trainData)
> # Make predictions on the test set
> predictions <- predict(model, testData)
> # Evaluate model performance
> conf_matrix <- confusionMatrix(predictions, testData$Species)
```

```
> print(conf_matrix)
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
    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
```

Conclusion: A Random Forest model was trained on the training set and evaluated on the test set, demonstrating its effectiveness in predicting the target variable with satisfactory performance metrics.

Practical-10

Aim: Perform the following on the given data set:
1) Hierarchical clustering.

Theory:

Hierarchical clustering is a method used to group similar objects into clusters based on their features or attributes. Here's a brief overview of how hierarchical clustering works:

Distance Metric: Hierarchical clustering begins by calculating the distance or dissimilarity between each pair of objects in the dataset.
Common distance metrics include Euclidean distance, Manhattan distance, and correlation distance, among others.

Linkage Criteria: Linkage criteria determine how the distances between clusters are computed.

Common linkage criteria include:
Single Linkage: Compute the distance between the closest points of the two clusters.

Complete Linkage: Compute the distance between the farthest points of the two clusters.

Average Linkage: Compute the average distance between all pairs of points in the two clusters.

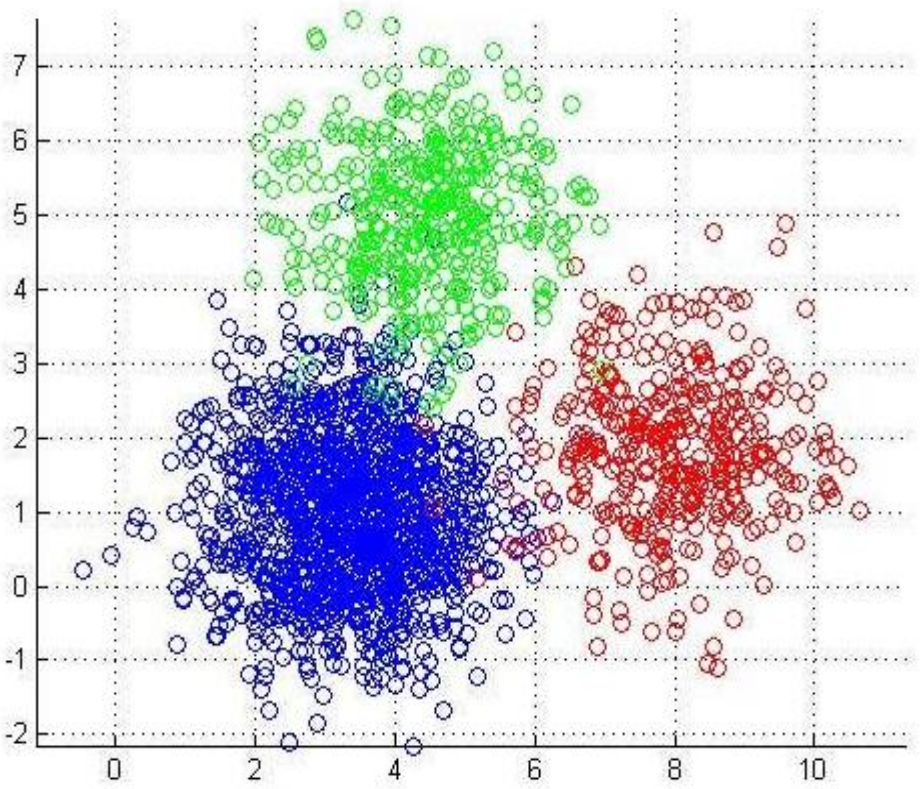
Ward's Linkage: Minimize the variance when merging clusters.

Hierarchical Structure: Hierarchical clustering creates a dendrogram, which is a tree-like structure representing the merging process of clusters.At the beginning, each data point is considered as a separate cluster.Clusters are then iteratively merged based on their similarity until all data points belong to a single cluster.

Cutting the Dendrogram: To determine the number of clusters, one can cut the dendrogram at a certain height or distance

threshold.Alternatively, one can use more sophisticated methods such as the elbow method or silhouette method to find the optimal number of clusters.

Figure:



Algorithm:

Load the Dataset:

data(mtcars) This line loads the mtcars dataset, which contains information about various car models, such as miles per gallon (mpg), displacement (disp), horsepower (hp), etc. It's a built-in dataset in R.

Select Subset of Variables (Optional):

subset_data <- mtcars[, c('mpg', 'disp', 'hp')] This line creates a subset of the mtcars dataset by selecting only the columns 'mpg', 'disp', and 'hp'. These variables will be used for clustering.

Perform Hierarchical Clustering:

dist_matrix <- dist(subset_data),hclust_result <- hclust(dist_matrix)
dist(subset_data) calculates the pairwise Euclidean distances between all rows in the subset_data dataframe. This creates a distance matrix.
hclust(dist_matrix) performs hierarchical clustering on the distance matrix. It applies a clustering algorithm to group similar observations into clusters, forming a hierarchical structure.

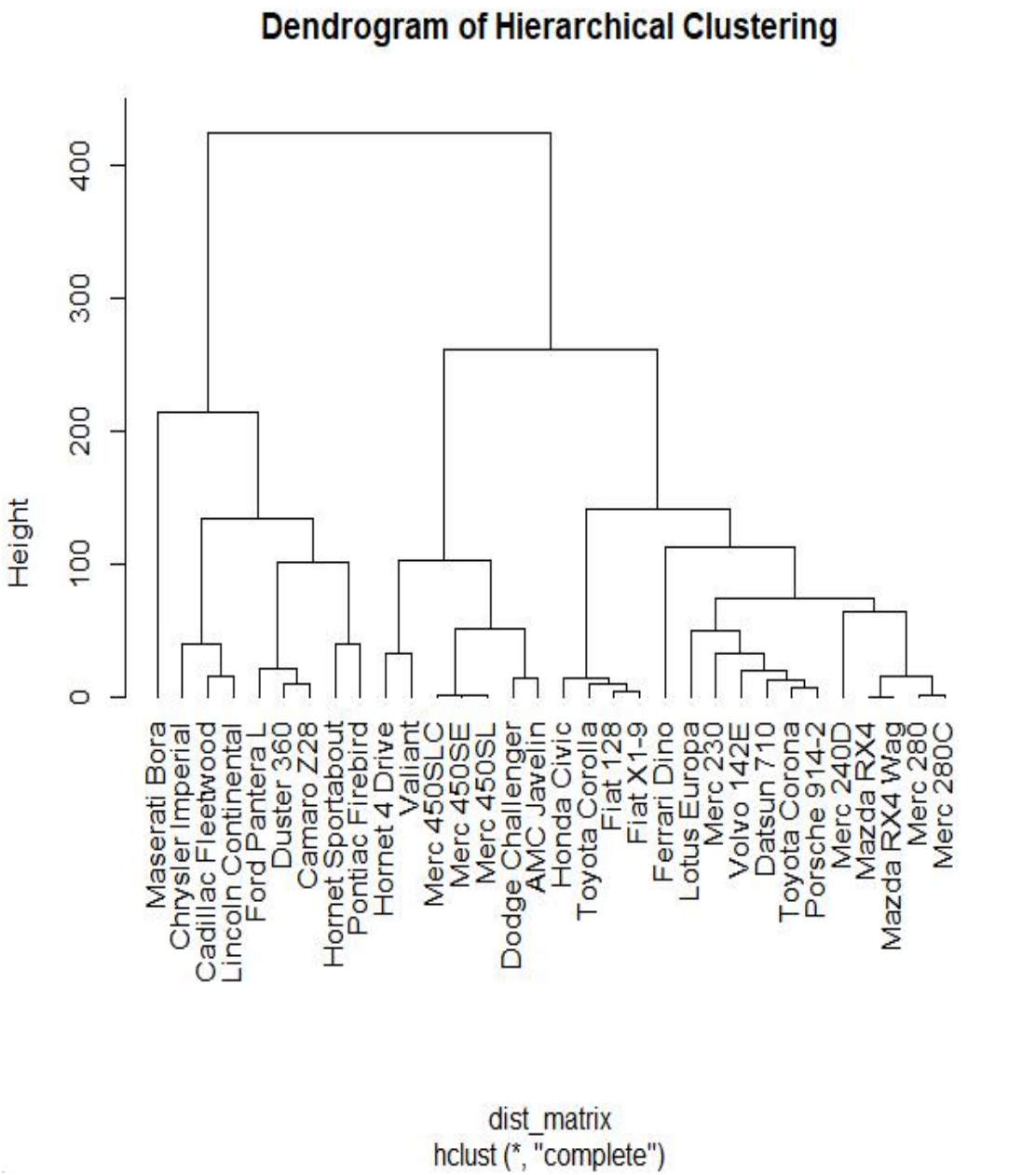
Plot the Dendrogram:

plot(hclust_result, hang = -1, main = "Dendrogram of Hierarchical Clustering"),plot(hclust_result, hang = -1, main = "Dendrogram of Hierarchical Clustering") plots the dendrogram representing the hierarchical clustering result. The hang = -1 argument specifies the direction in which the branches of the dendrogram are drawn.
The main argument sets the title of the plot.

Code:

```
# Load the dataset
data(mtcars)

# Select subset of variables for clustering (if needed)
# For example, let's select 'mpg', 'disp', and 'hp'
subset_data <- mtcars[, c('mpg', 'disp', 'hp')]

# Perform hierarchical clustering
dist_matrix <- dist(subset_data)  # Calculate distance matrix
hclust_result <- hclust(dist_matrix)  # Perform hierarchical clustering

# Plot the dendrogram
plot(hclust_result, hang = -1, main = "Dendrogram of Hierarchical Clustering")
```

Output:



Conclusion: Hierarchical clustering was applied to the dataset, revealing a hierarchical structure of clusters based on pairwise distances between data points.