CHAPTER 7

Statistics and Probability

7.1 Introduction

R is a popular programming language and environment that is widely used for statistical computing and graphics. It provides a comprehensive set of tools for data manipulation, calculation, and graphical display, making it a powerful tool for statistical analysis. Probability is a fundamental concept in statistics that deals with the likelihood of specific events occurring. R provides a robust framework for working with various probability distributions and performing probabilistic calculations.

7.2 Data Visualization

Data visualization is a critical component of data analysis, allows to represent complex data in a visually appealing and understandable manner. In R, there are various powerful packages that can help you create a wide range of visualizations with the helps of visualization techniques, a human can easily obtain information about hidden patterns in data that might be neglected. Therefore, Data Visualization helps us interpret data quickly, examine different variables to see their effects on the patterns, and derive insights from our data.

7.2.1 R Visualization Packages

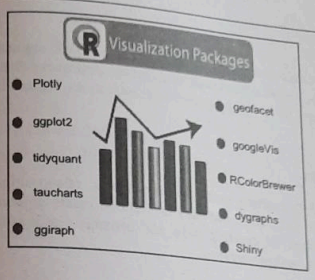
There are several powerful data visualization packages that can help you create a wide variety of static and interactive visualizations.

1. Plotly

Plotly is an interactive visualization library that support for the creation of interactive web-based graphs and to create interactive 2D and 3D plots.

2. ggplot2

ggplot2 used to create complex and customizable plots with a straightforward syntax. It is particularly well-suited for creating static, publication-quality graphics.



3. tidyquant

The tidyquant is a financial package that is used for quantitative financial analysis,importing, analyzing, and visualizing the data.

4. Tauchart

The library provides a declarative interface for rapid mapping of data fields to visual properties.

5. ggiraph

It is a tool that allows us to create dynamic ggplot graphs. This package allows to add tooltips, JavaScript actions, and animations to the graphics.

6. geofacets

This package provides geofaceting functionality for 'ggplot2'. Geofaceting arranges a sequence of plots for different geographical entities into a grid that preserves some of the geographical orientation.

7. googleVis

googleVis provides an interface between R and Google's charts tools. Helps to create web pages with interactive charts based on R data frames.

8. RColorBrewer

This package provides color schemes for maps and other graphics, which are designed by Cynthia Brewer.

9. Dygraphs

Dygraphs is an R interface to the dygraphs JavaScript library, which enables the creation of interactive time series plots. It supports the visualization of time series data with interactive zooming, panning, and interactive point selection.

10. Shiny

Shiny is an R package that allows to create interactive web applications. With Shiny,design interactive dashboards, visualizations, and data-driven applications that can be deployed on the web. It is a powerful tool for creating interactive visualizations that can be shared and accessed by others.

7.3 R Graphics

Graphics are used to examine marginal distributions, relationships between variables, and summary of very large data. there are various ways to create graphics and visualize data. R provides a rich ecosystem for creating static and dynamic visualizations for exploratory data analysis, communication, and presentation.

Base R Graphics

Base graphics system that provides functionalities for creating a variety of statistical plots,

- Pie Chart
- Bar Chart
- Line Chart
- Histograms
- Boxplots
- Scatter Plots

There are some key elements of a statistical graphic. These elements are the basics of the grammar of graphics. R provides some built-in functions which are included in the graphics package for data visualization in R.

```
#To load graphics package
    library("graphics")
#To load datasets package
    library("datasets")
#To load mtcars dataset
    data(mtcars)
#To analyze the structure of the dataset
    str(mtcars)
```


Output

```
'data.frame': 32 obs. of 11 variables:
$ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
$ cyl : num 6 6 4 6 8 6 8 4 4 6 ...
$ 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 ...
$ qsec: num 16.5 17 18.6 19.4 17 ...
$ vs  : num 0 0 1 1 0 1 0 1 1 1 ...
$ am  : num 1 1 1 0 0 0 0 0 0 0 ...
$ gear: num 4 4 4 3 3 3 3 4 4 4 ...
$ carb: num 4 4 1 1 2 1 4 2 2 4 ...
```

7.3.1 Pie Chart

A pie-chart is a representation of values in the form of slices of a circle with different colors. Slices are labeled with a description, and the numbers corresponding to each slice are also shown in the chart. pie () function, which takes positive numbers as vector input. Additional parameters are used to control labels, colors, titles, etc.

Syntax

pie(x, labels = NULL, clockwise = FALSE, main = NULL, col = NULL, ...)

- **x:** A vector of non-negative numerical values representing the areas of the sectors or slices.
- **labels:** A vector of character strings that will be used for labeling the slices. If NULL, the names of x will be used.
- **clockwise:** A logical value indicating whether the slices should be drawn in a clockwise or counterclockwise direction.
- **main:** A main title for the plot.
- **col:** A vector of colors for filling the slices. If not specified, R will use its default color palette.
- **...:** Additional graphical parameters can be passed.

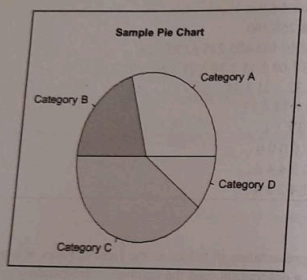
Example

```
# Example data
categories <- c("Category A", "Category B", "Category C", "Category D")
values <- c(30, 20, 40, 10)
```

 # Create a pie chart

pie(values, labels = categories, main = "Sample Pie Chart")

Output



In this example:

Categories is a vector containing the category labels.

- Values is a vector containing the values or proportions for each category.
- The pie() function is used to create the pie chart, with values as the first argument and categories as the labels argument.
- The main argument specifies the title of the pie chart, which is "Sample Pie Chart" in this case.

7.3.2 Bar Charts

Bar charts in R is a common way to visualize categorical data or compare different categories. The barplot() function in R is used to create basic bar charts.

Syntax

Barplot(height, names.arg, main, xlab, ylab, col, beside, legend.text, args.legend)

- **height:** A vector or matrix of values representing the heights of the bars. If it's a matrix, each row is a separate bar.
- **names.arg:** A vector of names or labels for the bars.
- **main:** The title of the bar chart.
- **xlab:** The label for the x-axis.
- **ylab:** The label for the y-axis.
- **col:** The color or colors to be used for the bars. If a vector of colors is provided, each bar will be filled with a different color.

- **beside:** A logical value that determines whether the bars should be plotted beside each other (if TRUE) or stacked (if FALSE).
- **legend.text:** A vector of text used in the legend.
- **args.legend:** A list of arguments to be passed to the legend function.

7.3.3 Vertical Bar Chart

Example

 # Example data

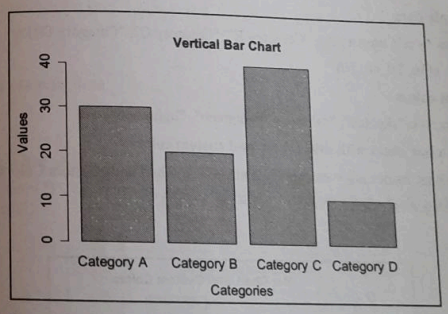
categories <- c("Category A", "Category B", "Category C", "Category D")

values <- c(30, 20, 40, 10)

 # Create a vertical bar chart

barplot(values, names.arg = categories, main = "Vertical Bar Chart", xlab = "Categories", ylab = "Values")

Example



Example: To create a horizontal bar chart, set the horiz argument to TRUE

 # Example data

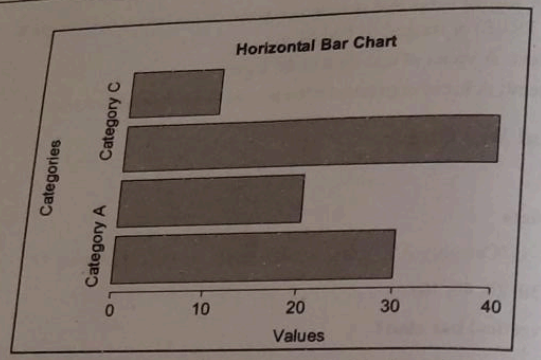
categories <- c("Category A", "Category B", "Category C", "Category D")

values <- c(30, 20, 40, 10)

 # Create a horizontal bar chart

barplot(values, names.arg = categories, main = "Horizontal Bar Chart", xlab = "Values", ylab = "Categories", horiz = TRUE)

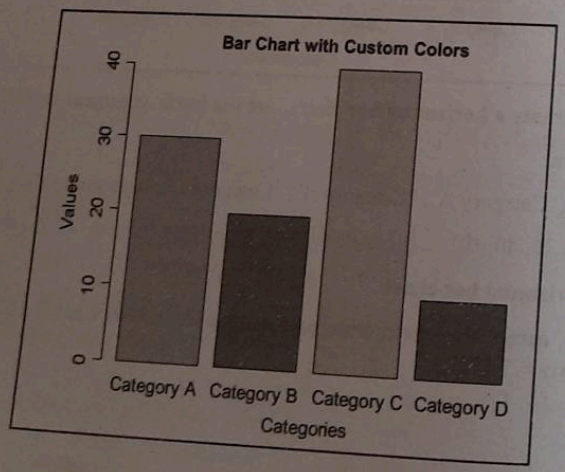
Output



Program for creating a bar chart in R with a title, labels, and custom colors using the barplot() function:

```
# Example data
categories <- c("Category A", "Category B", "Category C", "Category D")
values <- c(30, 20, 40, 10)
# Custom colors
bar_colors <- c("skyblue", "salmon", "lightgreen", "lightcoral")
# Create a bar chart with title, labels, and custom colors
barplot(values, names.arg = categories, main = "Bar Chart with Custom Colors",
xlab = "Categories", ylab = "Values", col = bar_colors)
```

Output



7.3.4 Line Chart

A line chart is a pictorial representation of information that changes continuously over time. A line graph can also be referred to as a line chart. Within a line graph, there are points connecting the data to show the continuous change. A line chart is used to connect a series of points by drawing line segments between them. Line charts are used to identify the trends in data.

Syntax

plot(x, y, type = "l", col = "blue", lwd = 2, xlab = "X-axis label", ylab = "Y-axis label", main = "Title of the Plot")

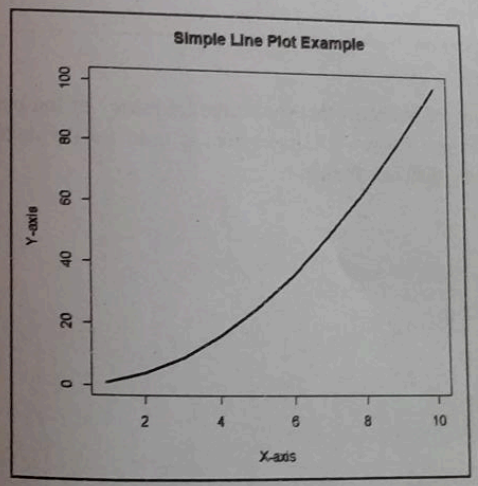
- x: a vector of x-axis values.
- y: a vector of corresponding y-axis values.
- type: specifies the type of plot, use "l" for a line graph.
- col: specifies the color of the line.
- lwd: specifies the width of the line.
- xlab: specifies the label for the x-axis.
- ylab: specifies the label for the y-axis.
- main: specifies the title of the plot.

Example

```
# Creating example data
x <- 1:10
y <- x^2
# Creating a line plot
```

plot(x, y, type = "l", col = "blue", lwd = 2, xlab = "X-axis", ylab = "Y-axis", main = "Simple Line Plot Example")

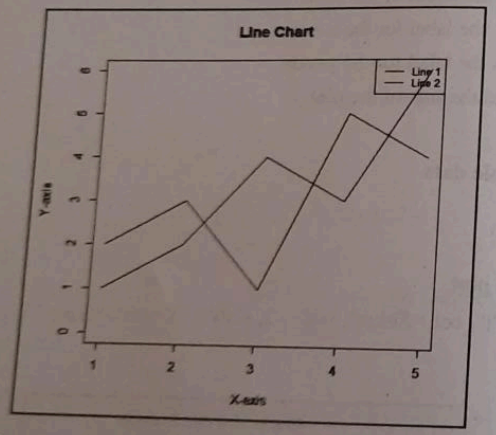
Output



Example

```
# Creating data
x <- c(1, 2, 3, 4, 5)
y1 <- c(2, 3, 1, 5, 4)
y2 <- c(1, 2, 4, 3, 6)
# Creating plot
plot(x, y1, type = "l", col = "blue", xlab = "X-axis", ylab = "Y-axis", ylim = c(0, max(y1, y2)), main = "Line Chart")
lines(x, y2, type = "l", col = "red")
legend("topright", legend = c("Line 1", "Line 2"), col = c("blue", "red"), lty = 1, cex = 0.8)
```

Output



7.3.5 Histogram

A histogram is a type of bar chart that shows the frequency of the number of values that are compared with a set of value ranges. The histogram is used for the distribution, whereas a bar chart is used for comparing different entities.

Syntax

```
hist(x,
main = "Histogram Title",
xlab = "X-axis Label",
ylab = "Y-axis Label",
col = "color",
```



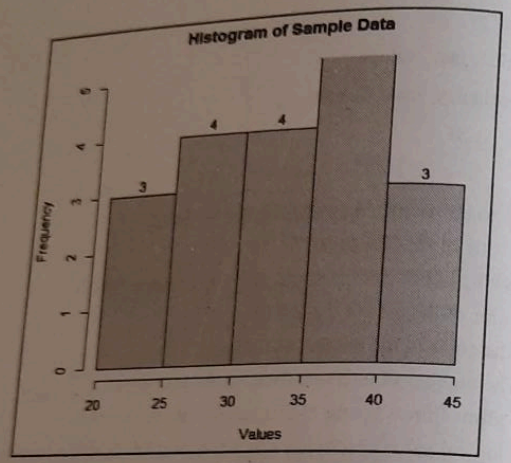
```
border = "color",
breaks = "number",
xlim = c(min_value, max_value),
ylim = c(min_frequency, max_frequency),
labels = TRUE/FALSE
)
```

- **x:** The data to be visualized as a histogram.
- **main:** The title of the histogram.
- **xlab and ylab:** Labels for the x-axis and y-axis, respectively.
- **col:** The color of the bars in the histogram.
- **border:** The color of the borders of the bars.
- **breaks:** The number of intervals (bins) in the histogram.
- **xlim and ylim:** Limits for the x-axis and y-axis, respectively.
- **labels:** A logical value specifying whether to add labels to the bars in the histogram.

Example

```
data <- c(22, 24, 25, 26, 28, 29, 30, 32, 34, 35, 35, 36, 37, 37, 38, 39, 40, 41, 42, 43)
hist(data,
  main = "Histogram of Sample Data",
  xlab = "Values",
  ylab = "Frequency",
  col = "lightblue",
  border = "black",
  breaks = 5,
  xlim = c(20, 45),
  ylim = c(0, 5),
  labels = TRUE
)
```

Output



7.3.6 Boxplot

A boxplot is a standardized way of displaying the distribution of data based on a five-number summary: minimum, first quartile (Q1), median, third quartile (Q3), and maximum. It is a useful tool for visualizing the spread and skewness of the data, identifying outliers, and comparing multiple distributions. The boxplot() function in R is used to create box-and-whisker plots. These plots display the distribution of numerical data through quartiles.

Syntax

```
boxplot(x,
    main = "Boxplot Title",
    xlab = "X-axis Label",
    ylab = "Y-axis Label",
    col = "color",
    border = "color",
    notch = FALSE/TRUE,
    horizontal = FALSE/TRUE
)
```

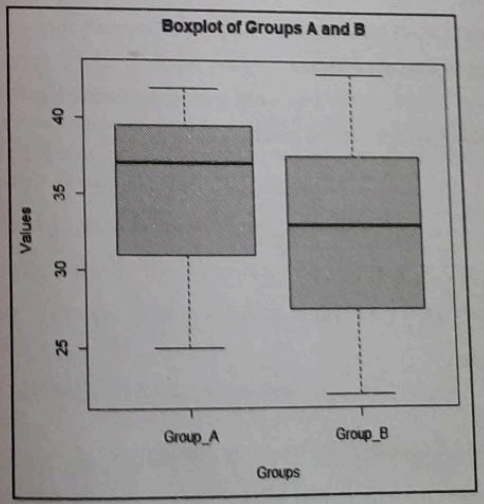
- **x:** The input data for which the boxplot is to be created.
- **main:** The title of the boxplot.
- **xlab and ylab:** Labels for the x-axis and y-axis, respectively.

- **col:** The color of the boxes in the boxplot.
- **border:** The color of the borders of the boxes.
- **notch:** A logical value specifying whether to create a notched box plot.
- **horizontal:** A logical value specifying whether to create a horizontal box plot.

Example

```
# Sample data
data <- data.frame(
  Group_A = c(25, 28, 30, 32, 35, 37, 38, 39, 40, 41, 42),
  Group_B = c(22, 24, 26, 29, 31, 33, 36, 37, 38, 40, 43)
)
# Create a boxplot
boxplot(data,
        main = "Boxplot of Groups A and B",   # Title of the boxplot
        xlab = "Groups",                      # X-axis label
        ylab = "Values",                      # Y-axis label
        col = c("lightblue", "lightgreen"),   # Colors of the boxes
        border = "black"                      # Color of the box borders
)
```

Output



7.3.7 Scatter Plot

A scatter plot is a type of plot or mathematical diagram using Cartesian coordinates to display values for typically two variables for a set of data. Each point in the plot represents an observation, and the position of the point depends on its values in the two variables being plotted. The plot() function with two numeric vectors as arguments.

A scatter plot helps to determine whether there is any relationship or pattern, such as correlation, between the two variables. It is a fundamental tool for understanding the nature of the relationship between variables and is essential in various fields, including statistics, data analysis, and machine learning.

Syntax

```
plot(x, y,
  main = "Scatterplot Title",
  xlab = "X-axis Label",
  ylab = "Y-axis Label",
  col = "color",
  xlim = c(min_value, max_value),
  ylim = c(min_value, max_value),
  pch = 1
)
```

- **x and y:** The data points for the x-axis and y-axis, respectively.
- **main:** The title of the scatter plot.
- **xlab and ylab:** Labels for the x-axis and y-axis, respectively.
- **col:** The color of the data points.
- **xlim and ylim:** Limits for the x-axis and y-axis, respectively.
- **pch:** The plotting symbol for the points.

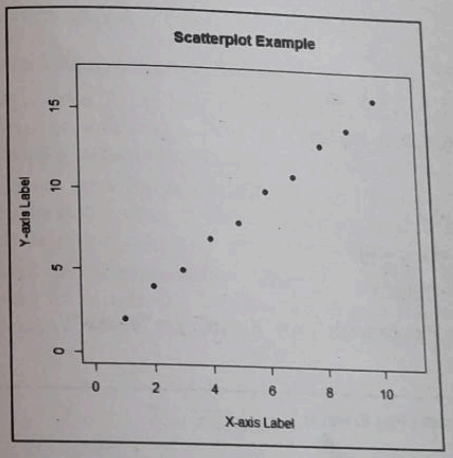
Example

```
# Sample data
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
y <- c(2, 4, 5, 7, 8, 10, 11, 13, 14, 16)
# Create a scatterplot
plot(x, y,
  main = "Scatterplot Example",   # Title of the scatterplot
  xlab = "X-axis Label",          # X-axis label
```



```
  ylab = "Y-axis Label",          # Y-axis label
  col = "blue",                   # Color of the points
  pch = 16,                       # Type of point (16 = solid circle)
  xlim = c(0, 11),                # X-axis limits
  ylim = c(0, 17)                 # Y-axis limits
)
```

Output



7.4 ggplot2

ggplot2 is widely used data visualization package that is built on the grammar of graphics, providing a structured approach to create plots. Developed by Hadley Wickham, ggplot2 allows users to construct customized and publication-quality graphs with relatively simple code. It follows the grammar of graphics, which breaks down a graphic into components like data, aesthetic mappings, geometric objects, and facets, among others.

The key components of ggplot2 include:

Data: This represents the dataset being visualized.

Aesthetic Mappings: These define how variables in the data are mapped to visual properties like color, shape, and size.

Geometric Objects (geoms): These are the actual visual elements such as points, lines, and bars that represent the data.

Facets: These allow for the creation of multiple plots that share the same structure but display different subsets of the data.

Statistical Transformations (stats): These are functions that summarize the data, providing a visual representation of statistical summaries.

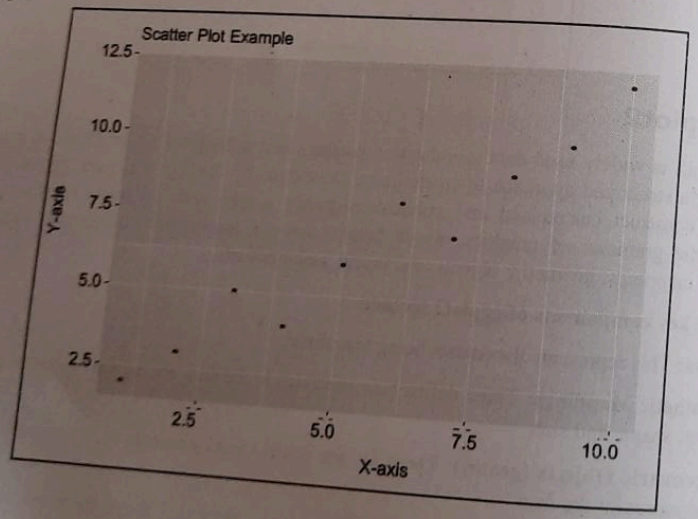
Scatter plot using the ggplot2 package in R, first need to install the and then load it using library(ggplot2).

Syntax

Example

```
library(ggplot2)
# Sample data
data <- data.frame(
  x = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
  y = c(2, 3, 5, 4, 6, 8, 7, 9, 10, 12)
)
# Create a scatter plot
ggplot(data, aes(x = x, y = y)) +
geom_point(color = "blue") +
labs(title = "Scatter Plot Example", x = "X-axis", y = "Y-axis")
```

Output



Advantages of Data Visualization in R

- **Rich Graphical Capabilities:** R provides a wide range of packages (e.g., ggplot2, plotly, lattice) that offer rich graphical capabilities for creating complex and customizable visualizations.
- **Customization and Flexibility:** R allows for extensive customization, enabling users to fine-tune every aspect of a plot, including colors, labels, axes, and annotations, to best represent the data.
- **Integration with Data Analysis:** R seamlessly integrates data visualization with data analysis, statistical modeling, and machine learning, making it easier for users to create visualizations directly from their data analysis workflow.

Disadvantages of Data Visualization in R

- **Steep Learning Curve:** R can have a steep learning curve, especially for beginners who are not familiar with programming or statistical concepts, which can make it challenging to grasp advanced data visualization techniques.
- **Performance Issues with Large Datasets:** R might face performance issues while handling large datasets, leading to slower execution times and memory limitations, which can affect the interactivity and responsiveness of visualizations.
- **Complex Syntax for Customization:** While R provides extensive customization options, the syntax for customizing plots can become complex, requiring users to have a strong understanding of the underlying grammar of graphics to create intricate visualizations.

EXERCISE

Short Questions

1. Define R programming language.
2. What is data visualization?
3. Mention data visualization packages in R.
4. Explain following Functions
   (a) ploty
   (b) ggplot2
   (c) tidyquant
   (d) tauchart
   (e) ggiraph
   (f) geofacets
   (g) googleVis

5. What is RColorBrewer?
6. Define dygraphs.
7. What is Shiny?
8. What is R Graphics?
9. Mention the different types of R Graphics.
10. Define pie chart with example.
11. Define bar charts with example.
12. What is vertical bar chart? Give an example.
13. What is line chart? Give an example.
14. Write the syntax for line graph.
15. Write the syntax for histogram.
16. Define boxplot.
17. What is scatter plot? Write its syntax.

Long Questions

1. Write a short note on visualization packages in R.
2. Explain Base R Graphics in detail.
3. Explain pie chart with syntax and example.
4. Explain bar charts with syntax and example.
5. Write a note on:
   (a) Vertical Bar Chart with example and syntax
   (b) Line Chart with syntax and example.
6. Explain histogram in detail.
7. What is Boxplot. explain with an example and write its syntax?
8. Explain scatter plot in detail.
9. Create a scatter plot.
10. Describe the key components included in ggplot2.
11. Mention the advantages and disadvantages of Data Visualization in R.

Ω♦Ω♦Ω♦Ω

CHAPTER 8

Common Probability Distribution

8.1 Introduction

R comes with built-in implementations of many probability distributions. probability distribution in R is associated with four functions that follow a naming convention: the probability density function always begins with 'd', the cumulative distribution function always begins with 'p', the inverse cumulative distribution (or quantile function) always begins with 'q', and a function that produces random variables always begins with 'r'. Each function takes a single argument at which to evaluate the function followed by specific parameters that define the particular distribution function to evaluate.

Some of the Common Probability Distribution Functions in R

Name	Probability Density	Cumulative Distribution	Quantile
Normal	dnorm(Z,mean,sd)	pnorm(Z,mean,sd)	qnorm(Q,mean,sd)
Poisson	dnorm(N,lambda)	pnorm(N,lambda)	qnorm(Q,lambda)
Binomial	dbinom(N,size,prob)	pbinom(N,size,prob)	qbinom(Q,size,prob)

8.2 Normal Distribution

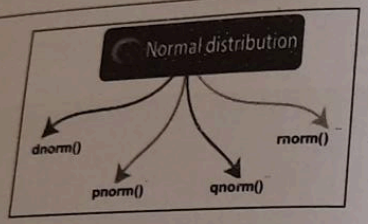
The normal distribution (Gaussian distribution) $N(\mu,\sigma)$ is represented R by dnorm, pnorm, and qnorm, where μ is the mean and σ is the standard deviation. The probability density dnorm and cumulative distribution pnorm are defined on the entire real axis.

Formula

The mean of the distribution is $\mu = (a + b)/2$

The standard deviation of the distribution is $\sigma = \sqrt{\sigma}$

R allows to generate normal distribution by providing the following functions:



There are four normal distribution available in R:

- Dnorm
- Pnorm
- Qnorm
- Rnorm

8.2.1 dnorm()

Dnorm(x,mean,sd) function computes the probability density function (PDF) at the specified values of 'x', where 'mean' and 'sd' are the mean and standard deviation, respectively.

$$f(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

Syntax

dnorm(x, mean = 0, sd = 1, log = FALSE)

- **x:** A numeric vector at which you want to evaluate the PDF.
- **mean:** The mean (average) of the normal distribution. The default is 0.
- **sd:** The standard deviation of the normal distribution. The default is 1.

Example

```
# Calculate the density of the normal distribution at a specific value
x <- 2
mean_value <- 0
sd_value <- 1
# Calculate the density of the normal distribution at the specified value
density_value <- dnorm(x, mean = mean_value, sd = sd_value)
# Print the calculated density value
print(density_value)
```

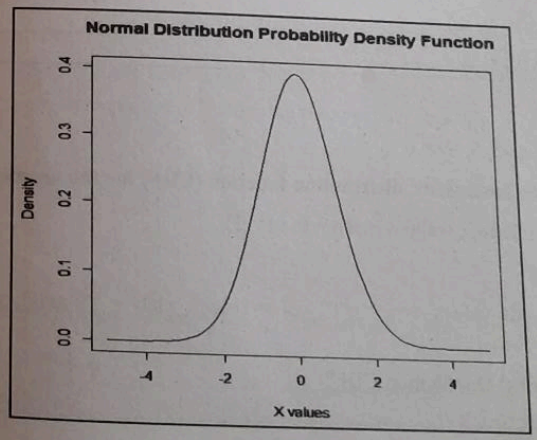

Output

```
[1] 0.05399097
```

Example: To visualize the probability density function (PDF) of the normal distribution using the dnorm function in R,create a line plot

```
Set the parameters
mean_value <- 0
sd_value <- 1
x_values <- seq(-5, 5, by = 0.1)
# Calculate the density of the normal distribution at the specified values
density_values <- dnorm(x_values, mean = mean_value, sd = sd_value)
# Create a line plot to visualize the probability density function (PDF) of the normal distribution
plot(x_values, density_values, type = "l",
     xlab = "X values", ylab = "Density",
     main = "Normal Distribution Probability Density Function",
     col = "blue")
```

Output



8.2.2 pnorm()

pnorm(q, mean, sd) function calculates the cumulative distribution function (CDF) at the specified quantile 'q', given the mean and standard deviation.

Syntax

pnorm(q, mean = 0, sd = 1, lower.tail = TRUE)

- **q:** The values at which the CDF of the normal distribution is to be calculated.
- **mean:** The mean of the normal distribution (default is 0).
- **sd:** The standard deviation of the normal distribution (default is 1).
- **lower.tail:** A logical value indicating whether to compute the lower tail probability (default is TRUE).

Example

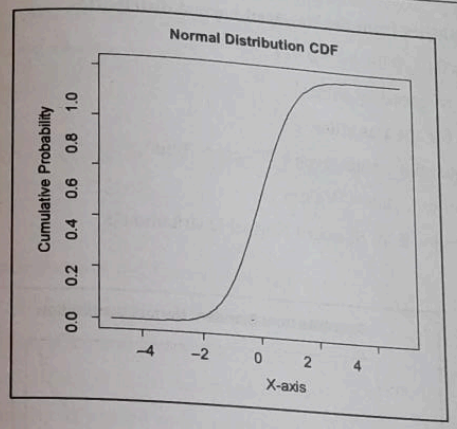
```
# Calculate the cumulative probability for a specific value
q <- 1.5
mean_value <- 0
sd_value <- 1
# Calculate the cumulative probability
cumulative_prob <- pnorm(q, mean = mean_value, sd = sd_value)
print(cumulative_prob)
```

Ouput: [1] 0.9331928

Example

```
x_values <- seq(-5, 5, by = 0.1)
# Calculate the cumulative distribution function (CDF) for the normal distribution
cdf_values <- pnorm(x_values, mean = 0, sd = 1)
# Plot the CDF
plot(x_values, cdf_values, type = "l", col = "blue", xlab = "X-axis", ylab = "Cumulative Probability",
main = "Normal Distribution CDF")
```


Ouput



8.2.3 qnorm()

qnorm(p, mean, sd) function provides the quantile function, returning the value corresponding to the quantile 'p' for the given mean and standard deviation.

Syntax

qnorm(p, mean = 0, sd = 1)

- **p:** The probability for which to compute the quantile.
- **mean:** The mean of the normal distribution (default is 0).
- **sd:** The standard deviation of the normal distribution (default is 1).

Example

```
p <- 0.95
mean_value <- 0
sd_value <- 1
quantile_value <- qnorm(p, mean = mean_value, sd = sd_value)
print(quantile_value)
```

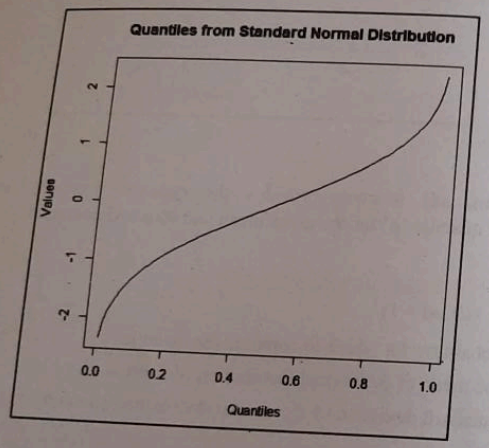
Output: [1] 1.644854

Example

Program using the qnorm function in R to generate quantiles from a normal distribution and then creating a corresponding chart.

```
# Generate quantiles from the standard normal distribution
quantiles <- seq(0.01, 0.99, by = 0.01)
quantile_values <- qnorm(quantiles)
# Create a plot for the quantiles
plot(quantiles, quantile_values, type = "l", col = "blue",
  xlab = "Quantiles", ylab = "Values",
  main = "Quantiles from Standard Normal Distribution")
```

Output



8.2.4 rnorm()

rnorm(n, mean, sd) function generates 'n' random numbers from a normal distribution with the specified mean and standard deviation.

Syntax

rnorm(n, mean = 0, sd = 1)

- **n:** The number of random values to generate.
- **mean:** The mean of the normal distribution (default is 0).
- **sd:** The standard deviation of the normal distribution (default is 1).

Example: Generate 10 random values from a normal distribution with mean 3 and standard deviation 2


```
random_values <- rnorm(10, mean = 3, sd = 2)
# Print the generated random values
print(random_values)
```

Ouput

```
[1] 1.8679108 -0.4020758  1.8689160  3.4796485  3.2377582  5.8934657
[7] 3.4419474  3.7484376  4.1152207 -1.8598992
```

Example

Program to find the Density and Cumulative distribution

```
# Parameters for the normal distribution
mean <- 0  # Mean of the normal distribution
sd <- 1    # Standard deviation of the normal distribution
# Compute the density at a given point using dnorm
density_at_zero <- dnorm(0, mean, sd)
cat("Density at 0:", density_at_zero, "\n")
# Compute the cumulative distribution at a given point using pnorm
cdf_at_zero <- pnorm(0, mean, sd)
cat("CDF at 0:", cdf_at_zero, "\n")
```

Output

```
Density at 0: 0.3989423
CDF at 0: 0.5
```

Example

```
p <- 0.95
quantile_value <- qnorm(p)
cat("Quantile value for", p, "probability:", quantile_value, "\n")
```

Output

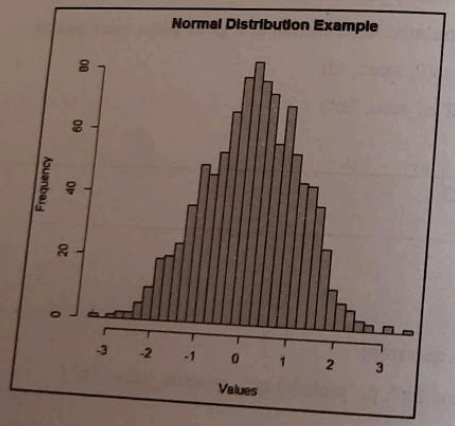
```
Quantile value for 0.95 probability: 1.644854
```

Example

Program to create a histogram to visualize the distribution

```
# Generate random data following a normal distribution with mean 0 and standard deviation
1
data <- rnorm(1000, mean = 0, sd = 1)
# Create a histogram to visualize the distribution
hist(data,
  main = "Normal Distribution Example",
  xlab = "Values",
  col = "lightblue",
  border = "black",
  breaks = 30  # Adjust the number of bins as needed
)
```

Output



8.3 Poisson Distribution

The Poisson distribution f(λ) is often used to represent the number of events occurring in a fixed interval of time or space.

$$f(x)=\frac{\lambda^x e^{-\lambda}}{x!}$$ where x = 0, 1, 2, 3, ...

There are four Poisson functions available in R:

- **dpois**
- **ppois**
- **qpois**
- **rpois**

The dpois (density), ppois (distribution function), rpois (random generation), and qpois (quantile function). The probability density dpois and cumulative distribution ppois are defined on non-negative integers.

The probability mass function (PMF) of the Poisson distribution is given by the formula:

$P(X=k)=(e-\lambda.\lambda k)/k!$

- $P(X=k)=$ is the probability of observing k events
- λ is the average rate of events in the given time interval,
- e is the base of the natural logarithm,
- k! represents the factorial of k.

8.3.1 dpois()

dpois() function is used for illustration of Poisson density in an R plot. The function dpois() calculates the probability of a random variable that is available within a certain range.

Syntax

dpois(x, lambda, log = FALSE)

- **x:** the number of events for which the probability is to be calculated.
- **lambda:** the average rate of occurrence (a non-negative numeric value).
- **log:** a logical value. If TRUE, probabilities are given as log values.

Example

```
# Set the average rate of occurrences (lambda)
lambda <- 2

# Compute the probability mass function (PMF) for a specific value of k
k <- 3

pmf_value <- dpois(k, lambda)

cat("Probability Mass Function (PMF) for k =", k, ":", pmf_value, "\n")
```

Output

```
Probability Mass Function (PMF) for k = 3 : 0.180447
```

8.3.2 ppois()

ppois() function is used for the illustration of the cumulative probability function in an R plot. The function ppois() calculates the probability of a random variable that will be equal to or less than a number.

Syntax

ppois(q, lambda, lower.tail, log)

- **K:** number of successful events happened in an interval
- **lambda:** mean per interval
- **lower.tail:** If TRUE then left tail is considered otherwise if FALSE right tail is considered
- **log:** If TRUE then the function returns probability in form of log

Example

```
lambda <- 4
k <- 3
# Calculate the CDF for k
cdf_value <- ppois(k, lambda)
# Print the result
cat("Cumulative Distribution Function (CDF) for k =", k, "and lambda =", lambda, ":", cdf_value, "\n")
```

Output

```
Cumulative Distribution Function (CDF) for k = 3 and lambda = 4 : 0.4334701
```

8.3.3 rpois()

The function rpois() is used for generating random numbers from a given Poisson's distribution.

Syntax

rpois(q, lambda)

q: number of random numbers needed

lambda: mean per interval

Example:

```
lambda <- 2.5 # Average rate of occurrences
random_deviates <- rpois(10, lambda)
```



```
# Print the random deviates
print(random_deviates)
```

Output: [1] 0 5 2 0 3 3 0 0 2 5

8.3.4 qpois()

The function qpois() is used for generating quantile of a given Poisson's distribution.

In probability, quantiles are marked points that divide the graph of a probability distribution into intervals (continuous) which have equal probabilities.

Syntax

qpois(q, lambda , lower.tail, log)

- K: number of successful events happened in an interval
- lambda: mean per interval
- lower.tail: If TRUE then left tail is considered otherwise if FALSE right tail is considered
- log: If TRUE then the function returns probability in form of log

Example

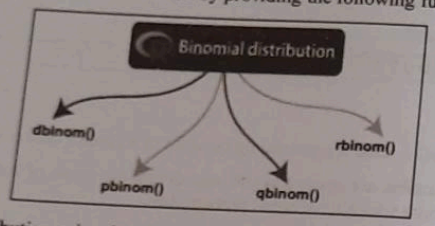
```
lambda <- 2
probability <- 0.7
quantile_value <- qpois(probability, lambda)
# Print the computed quantile
cat("Quantile value for probability", probability, "and lambda", lambda, ":", quantile_value, "\n")
```

Ouput: Quantile value for probability 0.7 and lambda 2 : 3

8.4 Binomial Distribution

The binomial distribution is also known as discrete probability distribution, which is used to find the probability of success of an event. The event has only two possible outcomes in a series of experiments. The tossing of the coin is the best example of the binomial distribution. When a coin is tossed, it gives either a head or a tail. The probability of finding exactly three heads in repeatedly tossing the coin ten times is approximate during the binomial distribution.

R allows to create binomial distribution by providing the following function:



The binomial distribution using functions such as

- **dbinom**
- **pbinom**
- **rbinom**
- **qbinom**

the probability mass function, the cumulative distribution function, generate random deviates, and compute quantiles, respectively. The probability mass function (PMF) of the binomial distribution is given by the formula

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

where:

- n is the number of trials,
- k is the number of successes,
- p is the probability of success on a single trial, and
- $(1-p)$ is the probability of failure on a single trial.

The term $\binom{n}{k}$ represents the number of ways to choose k successes from n trials and is calculated using the binomial coefficient formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Here,

- $n!$ denotes the factorial of n, which is the product of all positive integers up to n.

8.4.1 dbinom()

The dbinom() function of R calculates the probability density distribution at each point. In simple words, it calculates the density function of the particular binomial distribution.

Syntax

dbinom(x, size, prob)

Syntax:

- **x:** The number of successes.
- **size:** The number of trials.
- **prob:** The probability of success on each trial.

Example

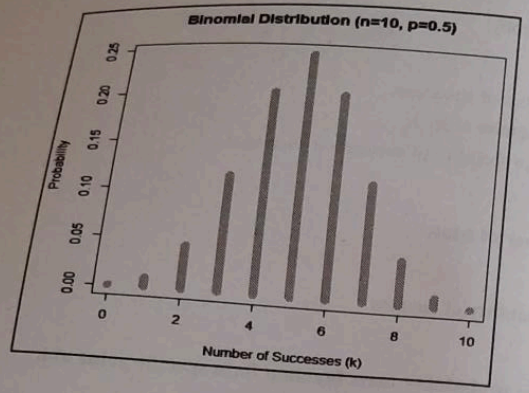
```
# Set the number of trials
n <- 5
# Set the probability of success
p <- 0.3
# Compute the probability mass function for a specific value of k
k <- 2
pmf_value <- dbinom(k, n, p)
# Print the computed probability mass function value
cat("Probability Mass Function (PMF) for k =", k, ":", pmf_value, "\n")
```

Output: Probability Mass Function (PMF) for k = 2 : 0.3087

Example: Program using the dbinom function in R to create a chart for the binomial distribution:

```
n <- 10  # number of trials
p <- 0.5  # probability of success
k_values <- 0:n
probabilities <- dbinom(k_values, size = n, prob = p)
plot(k_values, probabilities, type = "h", lwd = 10, col = "skyblue",
     main = "Binomial Distribution (n=10, p=0.5)",
     xlab = "Number of Successes (k)", ylab = "Probability")
```

Output



8.4.2 pbinom()

The pbinom function in R is used to compute the cumulative distribution function (CDF) for the binomial distribution.

Syntax

pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)

- **q:** the quantile (the number of successes).
- **size:** the number of trials.
- **prob:** the probability of success for each trial.
- **lower.tail:** a logical value. If TRUE, the cumulative probability is calculated as $P(X \leq q)$; if FALSE, it is calculated as $P(X > q)$.
- **log.p:** a logical value. If TRUE, probabilities are given as log values.

Example

```
# Set the number of trials
n <- 10

# Set the probability of success for each trial
p <- 0.3

# Compute the cumulative distribution function (CDF) for a specific value of k
k <- 4

cdf_value <- pbinom(k, n, p)
```



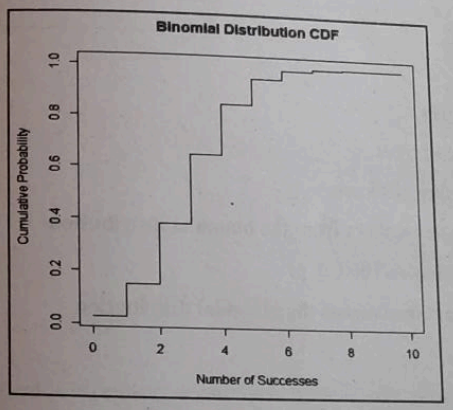
```
cat("Cumulative Distribution Function (CDF) for k =", k, ":", cdf_value, "\n")
```

Output: Cumulative Distribution Function (CDF) for k = 4 : 0.8497310037

Example: Program that creates a chart and computes the cumulative distribution function (CDF) of the binomial distribution:

```
n <- 10
p <- 0.3
x_values <- 0:n
cdf_values <- pbinom(x_values, size = n, prob = p)
plot(x_values, cdf_values, type = "s",
     xlab = "Number of Successes",
     ylab = "Cumulative Probability",
     main = "Binomial Distribution CDF")
```

Output



8.4.3 rbinom()

In R, the rbinom function is used to generate random deviates from the binomial distribution. Since the Bernoulli distribution is a special case of the binomial distribution with a single trial, you can use rbinom to simulate a Bernoulli distribution.

$$P(X = k) = (kn).pk.(1-p)^{n-k}$$

Syntax

rbinom(n, size, prob)

- *n* is the number of random values to generate.
- size is the number of trials.
- prob is the probability of success for each trial.

Example

Program using rbinom along with a bar chart to visualize the binomial distribution:

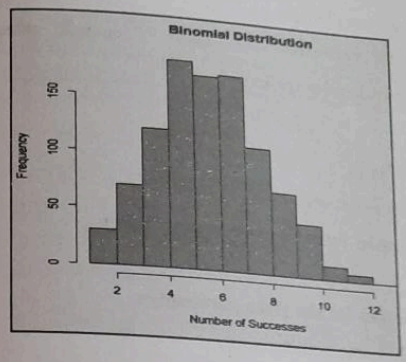
```
# Set the number of trials (n) and probability of success (p)
n <- 10
p <- 0.3
# Generate random deviates from the binomial distribution
random_deviates <- rbinom(5, n, p)
print(random_deviates)
```

Output: [1] 3 4 2 3 3

Example

```
#Set the parameters
n <- 20  # Number of trials
p <- 0.3  # Probability of success
# Generate random numbers from the binomial distribution
random_values <- rbinom(1000, n, p)
# Create a bar chart to visualize the binomial distribution
hist(random_values,
  breaks = seq(min(random_values), max(random_values) + 1, by = 1),
  main = "Binomial Distribution",
  xlab = "Number of Successes",
  ylab = "Frequency",
  col = "skyblue",
  border = "black"
)
```


Output



8.4.4 qbinom()

The qbinom function is used to compute the quantiles of the binomial distribution. It provides the inverse of the cumulative distribution function for the binomial distribution.

$$P(X \leq x) = \sum k = 0x(kn).pk.(1-p)^{n-k}$$

- n is the number of trials.
- p is the probability of success.
- x is the quantile value (the number of successes).

Syntax

qbinom(p, size, prob, lower.tail = TRUE)

- **p:** the probability value for which the quantile is to be calculated.
- **size:** the number of trials.
- **prob:** the probability of success for each trial.
- **lower.tail:** a logical value. If TRUE, the quantile is computed as $P(X \leq x)$; if FALSE, it is computed as $P(X > x)$.

Example

```
# Set the parameters
n <- 10  # Number of trials
p <- 0.3  # Probability of success
alpha <- 0.05  # Significance level
# Compute the quantile for the given significance level
```

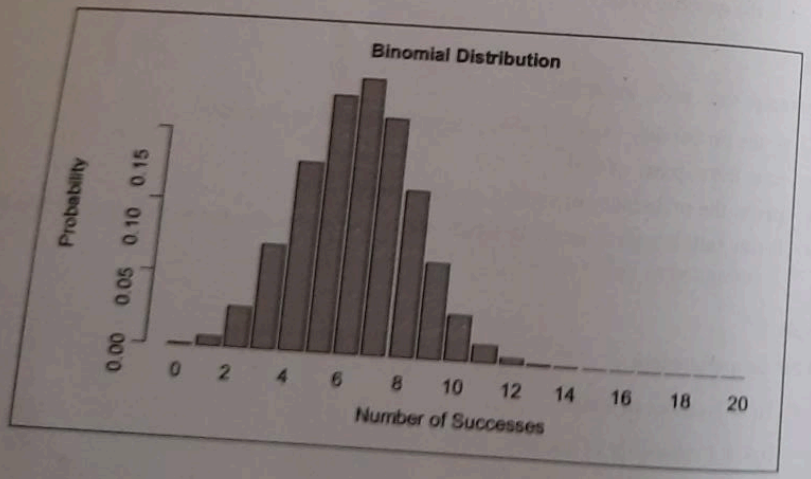
```
quantile_value <- qbinom(alpha, n, p)
cat("Quantile value for significance level", alpha, ":", quantile_value, "\n")
```

Output: Quantile value for significance level 0.05 : 1

Example

```
n <- 20  # Number of trials
p <- 0.3  # Probability of success
# Calculate the quantiles for the binomial distribution
quantiles <- 0:n
# Compute the quantiles for the given parameters
quantile_values <- qbinom(quantiles, n, p)
# Plot the binomial distribution using a bar chart
barplot(dbinom(quantiles, n, p), names.arg = quantiles,
  main = "Binomial Distribution",
  xlab = "Number of Successes",
  ylab = "Probability",
  col = "skyblue")
```

Output



8.5 The Uniform Distribution in R

A uniform distribution is a probability distribution in which every value between an interval from a to b is equally likely to be chosen. The probability that we will obtain a value between x_1 and x_2 on an interval from a to b can be found using the formula:

$P(\text{obtain value between } x_1 \text{ and } x_2) = (x_2 - x_1) / (b - a)$

The uniform distribution has the following properties:

- The mean of the distribution is $\mu = (a + b) / 2$
- The variance of the distribution is $\sigma^2 = (b - a)2 / 12$
- The standard deviation of the distribution is $\sigma = \sqrt{\sigma^2}$

8.5.1 dunif()

The dunif() function is used to compute the density of the uniform distribution at specified points. However, the uniform distribution is not as commonly used as other distributions. For continuous probability distribution, density is the value of the probability density function at x(i.e., f(x)).

Syntax

dunif(x,min=0,max=1)

- **x:** represents vector
- **min:** lower limit of the distribution (default value is 0 in R),
- **max:** upper limit of the distribution (default value is 1 in R).

Example

Program to create a custom dunif() .

```
dunif <- function(x, min, max) {
density <- ifelse(x >= min & x <= max, 1/(max - min), 0)
return(density)
}
x_values <- seq(0, 1, by = 0.1)
density_values <- dunif(x_values, 0, 1)
print(density_values)
```

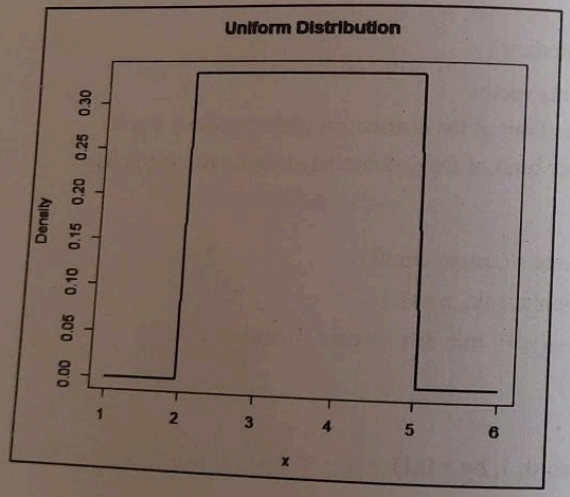
Output: [1] 1 1 1 1 1 1 1 1 1 1 1

Example

Program to create a custom dunif() function and generate a chart to visualize the uniform distribution:

```
dunif <- function(x, min, max) {
ifelse(x >= min & x <= max, 1/(max - min), 0)
}
# Set the parameters
min_value <- 2
max_value <- 5
x_values <- seq(min_value - 1, max_value + 1, length.out = 100)
# Calculate the density for the specified range
density_values <- dunif(x_values, min_value, max_value)
# Plot the uniform distribution
plot(x_values, density_values, type = "l", lwd = 2,
  col = "blue", xlab = "x", ylab = "Density",
  main = "Uniform Distribution")
```

Output



8.5.2 punif()

The punif function in R is used to compute the cumulative distribution function (CDF) for the uniform distribution. It calculates the probability that a random observation from a uniform distribution will be less than or equal to a specific value.

Syntax

punif(q, min = 0, max = 1, lower.tail = TRUE)

- **q:** the quantile (a numeric vector of values).
- **min:** the lower limit of the distribution (default is 0).
- **max:** the upper limit of the distribution (default is 1).
- **lower.tail:** a logical value indicating whether to compute the lower tail (default is TRUE).

Example

Program to calculate the cumulative distribution for a uniform distribution

```
q <- 0.4
min_value <- 0.2
max_value <- 0.8
# Compute the cumulative probability
cumulative_probability <- punif(q, min = min_value, max = max_value)
print(cumulative_probability)
```

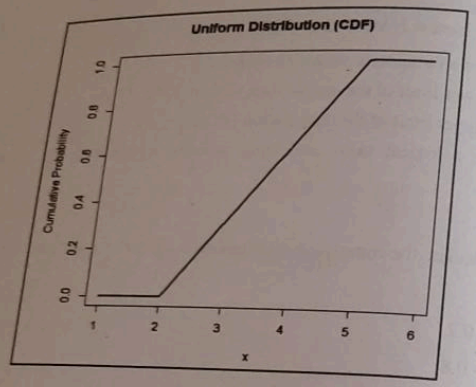
Output: [1] 0.3333333

Example

Program to create a chart for the uniform distribution using the punif function.

```
min_value <- 2
max_value <- 5
x_values <- seq(min_value - 1, max_value + 1, length.out = 100)
# Calculate the cumulative distribution for the specified range
cdf_values <- punif(x_values, min_value, max_value)
# Plot the uniform distribution
plot(x_values, cdf_values, type = "l", lwd = 2,
  col = "blue", xlab = "x", ylab = "Cumulative Probability",
  main = "Uniform Distribution (CDF)")
```

Output



8.5.3 qunif()

The qunif function in R is used to compute quantiles from the uniform distribution.

Syntax

qunif(p, min = 0, max = 1)

- **p** is the probability at which to compute the quantile.
- **min** is the minimum value of the distribution (lower limit of the interval).
- **max** is the maximum value of the distribution (upper limit of the interval).

The qunif function returns the quantiles corresponding to the probabilities provided in the p argument.

Example

Program to compute the quantiles for a uniform distribution

```
probabilities <- c(0.2, 0.5, 0.8)  # Probabilities
min_value <- 2  # Lower limit of the interval
max_value <- 5  # Upper limit of the interval
# Calculate the quantiles for the given probabilities
quantiles <- qunif(probabilities, min = min_value, max = max_value)
print(quantiles)
```

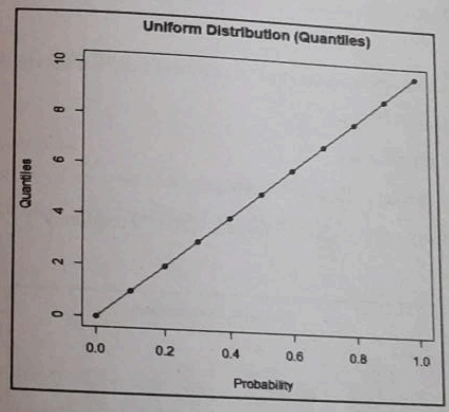
Output: [1] 2.6 3.5 4.4

Example

Program to create a chart to find the quantiles for the specified range

```
min_value <- 0
max_value <- 10
p_values <- seq(0, 1, by = 0.1)
# Calculate the quantiles for the specified range
quantiles <- qunif(p_values, min_value, max_value)
# Plot the uniform distribution
plot(p_values, quantiles, type = "o", pch = 19, col = "blue",
   xlab = "Probability", ylab = "Quantiles",
   main = "Uniform Distribution (Quantiles)")
```

Output



8.5.4 runif()

The runif() function in R is used to generate random numbers from a uniform distribution. It produces a specified number of random variates within a defined interval.

Syntax

runif(n, min = 0, max = 1)

- **n** is the number of random values to generate.
- **min** is the lower limit of the interval (default is 0).

- **max** is the upper limit of the interval (default is 1).

runif() function is used to create random variates within a specified range.

Example

```
# Generate a single random number between 0 and 1
runif(1)

# Generate 5 random numbers between 0 and 1
runif(5)

# Generate 10 random numbers between 5 and 10
runif(10, min = 5, max = 10)
```

Output

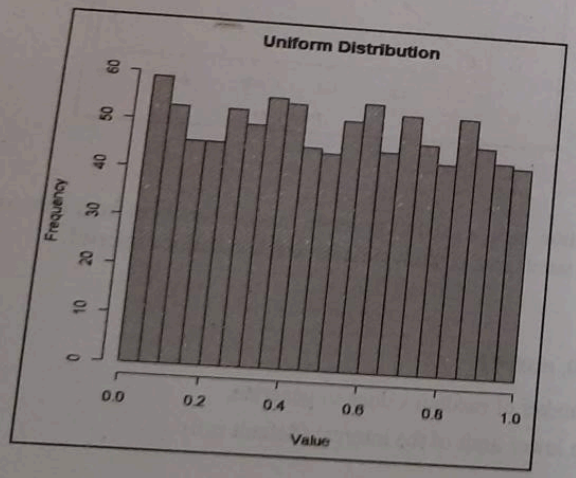
```
[1] 8.552517 9.490746 8.728362 6.250996 6.765012 5.633094 7.700866 8.516678
[9] 5.650098 8.942687
```

Example

Program to create a chart to generate random numbers from a uniform distribution

```
random_values <- runif(1000)

# Plot the uniform distribution

hist(random_values, breaks = 20, col = "skyblue",
main = "Uniform Distribution", xlab = "Value", ylab = "Frequency")
```

Output



8.6 Bernoulli Distribution

The Bernoulli distribution is a discrete probability distribution that represents the outcomes of a random experiment with two possible outcomes: success and failure. It is named after Jacob Bernoulli, a Swiss mathematician, and is a special case of the binomial distribution. The Bernoulli distribution is a takes value 1 with probability p and value 0 with probability 1−p, where $0 \leq p \leq 1$.

In this distribution,

- k takes on the value 1 with probability
- p and the value 0 with probability
- 1−p. The mean (expected value) of a Bernoulli random variable is

E[X]=p, and the variance is Var[X]=p(1−p).

In R Programming Language, there are 4 built-in functions to for Bernoulli distribution

8.6.1 dbern()

dbern( ) function in R programming measures the density function of the Bernoulli distribution.

Syntax: dbern(x, prob, log = FALSE)

Parameter

- **x:** vector of quantiles
- **prob:** probability of success on each trial
- **log:** logical; if TRUE, probabilities p are given as log(p)

Example

```
# Load the library
library(extraDistr)

# Define the probability of success
prob <- 0.3

# Define the values at which to evaluate the PDF
x_values <- c(0, 1)

# Compute the probability density function for the given values
densities <- dbern(x_values, prob)
print(densities)
```

Output: [1] 0.7 0.3

8.6.2 pbern()

pbern( ) function in R programming gives the distribution function for the Bernoulli distribution.

The distribution function or cumulative distribution function (CDF) or cumulative frequency function, describes the probability that a variate X takes on a value less than or equal to a number x.

Syntax

pbern(q, prob, lower.tail = TRUE, log.p = FALSE)

Parameter

- **q:** vector of quantiles
- **prob:** probability of success on each trial
- **lowe.tail:** logical
- **log.p:** logical; if TRUE, probabilities p are given as log(p).

Example

```
n <- 1
prob <- 0.3
# Compute the cumulative distribution function for the values 0 and 1
cumulative_prob <- pbinom(0:1, n, prob, lower.tail = TRUE)
print(cumulative_prob)
```

Output: [1] 0.7 1.0

8.6.3 qbern()

qbern( ) gives the quantile function for the Bernoulli distribution.

A quantile function in statistical terms specifies the value of the random variable such that the probability of the variable being less than or equal to that value equals the given probability.

Syntax

qbern(p, prob, lower.tail = TRUE, log.p = FALSE)

Parameter

- **p:** vector of probabilities.
- **prob:** probability of success on each trial.
- **lower.tail:** logical
- **log.p:** logical; if TRUE, probabilities p are given as log(p).

Example

```
# Installing and loading the extraDistr package
if (!require(extraDistr)) {
  install.packages("extraDistr")
}
library(extraDistr)
# Define the probability of success
prob <- 0.3
# Find the quantile for the given probability
quantile_value <- qbern(prob)
print(quantile_value)
```

Output: [1] 0

8.6.4 rbern()

rbern( ) function in R programming is used to generate a vector of random numbers which are Bernoulli distributed.

Syntax

rbern(n, prob)

Parameter

- **n:** number of observations.
- **prob:** number of observations.

Example

```
# Installing the 'extraDistr' package if not already installed
# install.packages("extraDistr")
# Loading the 'extraDistr' package
library(extraDistr)
prob <- 0.4
# Generating a sample of size 10 from a Bernoulli distribution
sample <- rbern(prob, 10)
print(sample)
```

Output: [1] 0 0 1 0 0 1 1 0 0 0

8.7 Student t-distribution

The Student's t-distribution (or simply the t-distribution) is a probability distribution that arises in the problem of estimating the mean of a normally distributed population when the sample size is small and the population standard deviation is unknown. It is also used for constructing confidence intervals and hypothesis tests on the population mean.

Key properties of the t-distribution include:

- **Shape:** The t-distribution is bell-shaped and symmetrical, like the standard normal distribution, but with heavier tails. As the degrees of freedom increase, the t-distribution approaches the standard normal distribution.
- **Centrality:** The mean, median, and mode of the t-distribution are all 0.
- **Degrees of Freedom (df):** The t-distribution is characterized by its degrees of freedom, denoted as v. The degrees of freedom determine the shape of the distribution. As the degrees of freedom increase, the t-distribution approaches the standard normal distribution. For small sample sizes, the t-distribution has more probability in the tails compared to the normal distribution.
- **Probability Density Function (PDF):** The PDF of the t-distribution involves the gamma function and is defined for any real number. The formula for the PDF is given in the response to the previous question.

The Student's t-distribution using various functions such as dt, pt, qt, and rt, which allow you to compute the probability density function (PDF), the cumulative distribution function (CDF), the quantile function, and generate random deviates, respectively. Here's a summary of these functions:

8.7.1 dt()

dt(x, df) computes the probability density function (PDF) of the Student's t-distribution at the specified values.

x: Vector of values at which to evaluate the PDF.

df: Degrees of freedom.

Example

```
density <- dt(1.5, 10)
print(density)
```

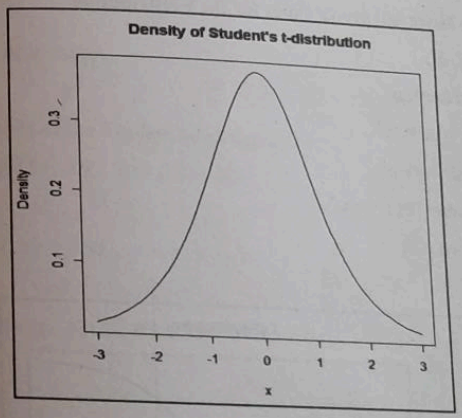
Output: [1] 0.1274448

Example

Program to demonstrate the use of dt() along with a chart program to visualize the t-distribution:


```
#Define the range of values
x <- seq(-3, 3, length.out = 100)
# Set the degrees of freedom
df <- 5
# Compute the probability density function for the given values
densities <- dt(x, df)
# Create a plot of the t-distribution
plot(x, densities, type = "l",
main = "Density of Student's t-distribution",
xlab = "x", ylab = "Density")
```

Output



8.7.2 pt()

pt(q, df) computes the cumulative distribution function (CDF) of the Student's t-distribution for the given values.

q: Vector of quantiles.

df: Degrees of freedom.

Example

```
cumulative_prob <- pt(1.5, 10)
print(cumulative_prob)
```

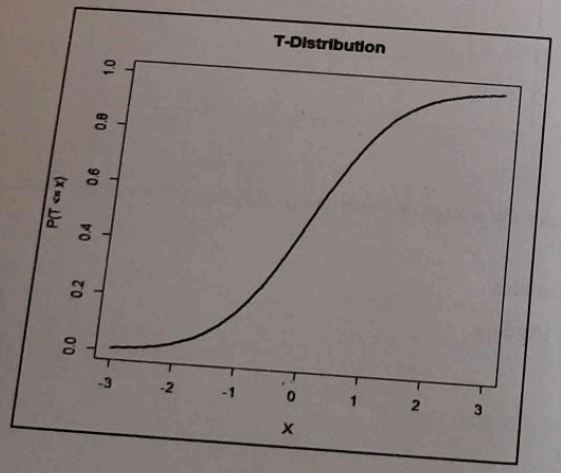
Ouput: [1] 0.9177463

Example

Program to demonstrate the use of pt() along with a chart program to visualize the t-distribution:

```
# Set the degrees of freedom
df <- 10
# Generate a sequence of values for the x-axis
x <- seq(-3, 3, length.out = 100)
# Calculate the lower tail probabilities for the t-distribution
p_values <- pt(x, df)
# Plot the t-distribution
plot(x, p_values, type = "l",
  main = "T-Distribution",
xlab = "X", ylab = "P(T <= x)",
col = "blue", lwd = 2)
```

Output



8.7.3 qt()

qt(p, df): Computes the quantiles of the Student's t-distribution for the given probabilities.

p: Vector of probabilities.

df: Degrees of freedom

Example

```
quantiles <- qt(0.95, 10)
print(quantiles)
```

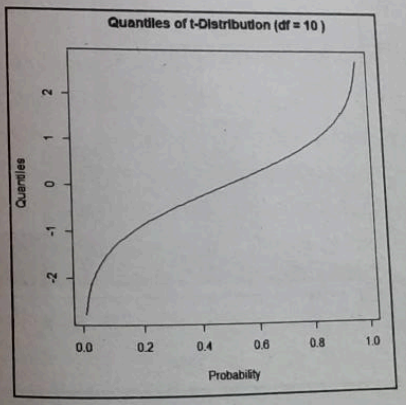
Output: [1] 1.812461

Example

Program to demonstrate the use of qt() along with a chart program to visualize the t-distribution:

```
# Set the degrees of freedom
df <- 10
# Compute quantiles of the Student's t-distribution
quantiles <- qt(seq(0.01, 0.99, by = 0.01), df)
# Create a chart
plot(seq(0.01, 0.99, by = 0.01), quantiles, type = "l",
xlab = "Probability", ylab = "Quantiles",
main = paste("Quantiles of t-Distribution (df =", df, ")"))
```

Ouput



8.7.4 rt()

rt(n, df): Generates random deviates from a Student's t-distribution.

n: Number of observations to generate.

df: Degrees of freedom.

Example

```
sample <- rt(10, 10)
print(sample)
```

Output:

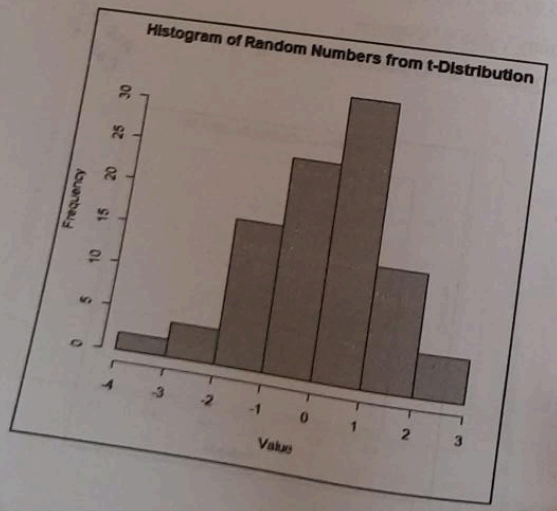
```
[1] -1.22383726  0.94088638 -0.97026774 -1.85703872  0.95586137 -1.11909061
[7] -0.02851959  0.72807861  0.46058078 -0.16748534
```

Example

Program to demonstrate the use of rt() along with a chart program to visualize the t-distribution:

```
#Generating random numbers from a t-distribution
random_t <- rt(n = 100, df = 5)
# Creating a histogram
hist(random_t, col = "skyblue", main = "Histogram of Random Numbers from t-Distribution", xlab = "Value")
```

Output



EXERCISE

Short Questions

1. List out the common probability distribution functions in R.
2. Define normal distribution. Write its formula.
3. Mention the four normal distribution available in R.
4. Define dnorm(). Write its syntax.
5. Define pnorm(). Write its syntax.
6. Define qnorm(). Write its syntax.
7. What is rnorm()?
8. Write the syntax of rnorm().
9. Define Poisson distribution.
10. Mention the functions of Poisson functions.
11. Write the formula for the probability mass function (PMF) of Poisson distribution.
12. Define following with its syntax
    (a) dpois()
    (b) ppois()
    (c) rpois ()
    (d) gpois()
13. What is Binomial Distribution?
14. List out the functions of binomial distribution.
15. Define following with its syntax.
    (a) dbniom()
    (b) pbniom()
    (c) qbniom()
    (d) rbniom()
16. Define the Uniform Distribution in R.
17. List out the functions of Uniform Distribution in R.
18. What is dunif()? Give an example. Write the syntax for dunif().
19. What is punif()? Give an example. Write the syntax for punif().
20. What is qunif()? Give an example. Write the syntax for qunif().
21. What is runif()? Give an example. Write the syntax for runif().
22. Define Bernoulli Distribution.
23. Define for the following terms with its syntax.
    (a) dbern()
    (b) pbern()

(c) qbern()
(d) rbern()

24. What is student t-distribution?
25. What are the key properties of t-distribution?
26. Define for the following terms with its syntax.
    (a) dt()
    (b) pt()
    (c) qt()
    (d) rt()

Long Questions

1. What is Normal Distribution? Explain its types in detail with example.
2. Write a program to create a histogram to visualize the distribution.
3. What is Poisson Distribution? Explain its types in detail with example.
4. What is Binomial Distribution? Explain its types in detail with example
5. Define the term Uniform Distribution in R briefly.
6. Explain the types of Uniform Distribution.
7. Explain Bernoulli Distribution in detail with example.
8. Explain the built-in functions in Bernoulli Distribution.
9. What is student t-distribution? Explain the key properties in detail.
10. Write a short note on the following functions.
    (a) dt()
    (b) pt()
    (c) qt()
    (d) rt()

CHAPTER 9

Statistical Testing and Modelling

9.1 Introduction

Statistical testing and modelling are fundamental aspects of data analysis and research, used to make inferences and draw conclusions from data. They help researchers and analysts make decisions, assess the significance of relationships, and understand the reliability of their findings. Here's an introduction to statistical testing and modelling:

9.1.1 Statistical Testing

Statistical testing involves the use of data-driven methods to make inferences about populations or data samples. The process typically follows these steps:

Formulate Hypotheses: Establish a null hypothesis (no effect) and an alternative hypothesis (there is an effect or difference).

Select a Test: Choose an appropriate statistical test based on the nature of the data and the research question.

Collect Data: Gather data from the sample or population of interest.

Compute the Test Statistic: Calculate a test statistic based on the collected data and the chosen test.

Determine Significance: Compare the test statistic to a critical value or compute a p-value to determine whether the results are statistically significant.

Common types of statistical tests include t-tests, ANOVA (analysis of variance), chi-square tests, and regression analysis.

9.1.2 Statistical Modelling

Statistical modeling involves the use of mathematical models to describe and understand relationships within data. This process helps in predicting outcomes, understanding complex systems, and identifying underlying patterns in data. Key steps in statistical modeling include:

Data Collection and Preparation: Gather and preprocess data for analysis, ensuring data quality and relevance.

Model Selection: Choose an appropriate statistical model based on the data characteristics and the research objectives.

Model Fitting: Estimate the parameters of the selected model using techniques such as least squares estimation, maximum likelihood estimation, or Bayesian inference.

Model Evaluation: Assess the model's performance and validity using various metrics, such as goodness-of-fit measures, prediction accuracy, and diagnostic tests.

Model Interpretation: Interpret the results and make conclusions about the relationships between variables and the overall model's predictive capability.

Statistical modeling techniques include linear regression, logistic regression, time series analysis, machine learning models, and more advanced methods like neural networks and decision trees.

Both statistical testing and modeling are essential tools in various fields such as economics, social sciences, natural sciences, and engineering.

9.2 Sampling Distributions in R

9.2.1 Samples

Samples is a common task, when working with statistical analyses, simulations, and data modeling. R provides several functions and methods to create samples from different types of distributions or datasets. Here are some common ways to generate samples in R:

Random Sampling from a Vector: the sample() function to randomly sample elements from a vector or a set of values.

 # Example of random sampling from a vector

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
random_sample <- sample(x, size = 5, replace = FALSE)
random_sample
```

Output: *[1] 5 3 9 1 4*

9.2.2 Sampling Distribution

A sampling distribution is a theoretical probability distribution that describes the behavior of a statistic based on repeated random sampling from a population. It provides information about the variability of a statistic, such as the sample mean or sample proportion, across multiple samples of the same size drawn from the same population. Understanding sampling distribution is crucial in statistical inference, as they help assess.

In statistics, a population is an entire pool from which a statistical sample is drawn. A population may refer to an entire group of people, objects, events, hospital visits, or measurements. A population can thus be said to be an aggregate observation of subjects grouped together by a common feature.

- A sampling distribution is a statistic that is arrived out through repeated sampling from a larger population.
- It describes a range of possible outcomes that of a statistic, such as the mean or mode of some variable, as it truly exists a population.
- The majority of data analyzed by researchers are actually drawn from samples, and not populations.

Syntax

hist(v,main,xlab,ylab,col)

where.

v is a vector containing values used in histogram.

main indicates title of the chart.

col is used to set color of the bars.

xlab is used to give description of x-axis.

ylab is used to give description of y-axis.

Steps to Calculate Sampling Distributions in R:

Step 1: Here, first we have to define a number of samples(n=1000).

n<-1000

Step 2: Next we create a vector(sample_means) of length 'n' with Null(NA) values [rep() function is used to replicate the values in the vector

Syntax: rep(value_to_be_replicated,number_of_times)

Step 3: Later we filled the created sample_means null vector with sample means from the considered population using the mean() function which are having a sample mean of 10(mean) and standard deviation of 10(sd) of 20 samples(n) using rnorm() which is used to generate normal distributions.

Syntax: mean(x, trim = 0)

: rnorm(n, mean, sd)

Step 4: To check the created samples we used head() which returns the first six samples of the dataframe (vector,list etc,.).

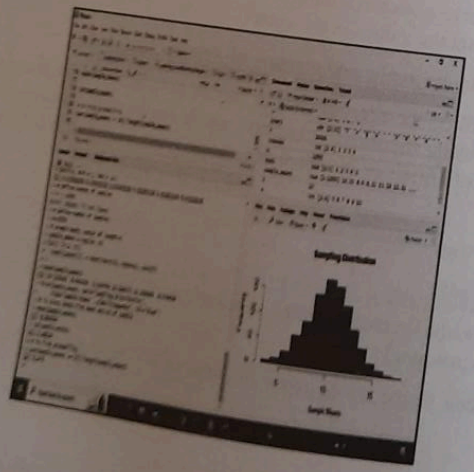
Syntax: head(data_frame,no_of_rows_be_returned) #By default second argument is set to 6 in R.

Step 5: Finally to visualize the sample_mean dataset we plotted a histogram (for better visualization) using hist() function in R.

Step 6: Finally we found the probability of generated sample means which are having mean greater than or equal to 10.

```
# define number of samples
n <-1000
# create empty vector of length n
sample_means = rep(NA, n)
# fill empty_vector with means
for(i in 1: n){
sample_means[i] = mean(rnorm(20, mean=10, sd=10))
}
head(sample_means)
# create histogram to visualize
hist(sample_means, main="Sampling Distribution",
  xlab="Sample Means", ylab="Frequency", col="blue")
# To cross check find mean and sd of sample
mean(sample_means)
sd(sample_means)
# To find probability
sum(sample_means >= 10)/length(sample_means)
```

Output

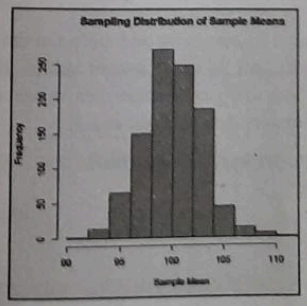


Example

Program to create a sampling distribution of the sample mean in R

```
# Setting the parameters
population_mean <- 100
population_sd <- 15
sample_size <- 30
num_samples <- 1000
# Generating the population data
population <- rnorm(10000, mean = population_mean, sd = population_sd)
# Creating an empty vector to store sample means
sample_means <- numeric(num_samples)
# Simulating multiple samples and calculating means
for (i in 1:num_samples) {
sample <- sample(population, size = sample_size)
sample_means[i] <- mean(sample)
}
# Plotting the sample distribution of means
hist(sample_means,
   main = "Sampling Distribution of Sample Means",
   xlab = "Sample Mean",
   ylab = "Frequency",
   col = "skyblue")
```

Output



This example demonstrates how to simulate a sampling distribution of the sample mean from a population and visualize it using a histogram.

9.3 Testing

"Testing" refers to the process of evaluating or examining something to assess its characteristics, performance, or quality. In the context of software development or quality assurance, testing refers to the systematic process of verifying that a software application or system meets specified requirements and functions as expected. It involves the identification of errors, bugs, or other issues that may affect the functionality, reliability, or security of the software.

Software Testing Includes Various Types and Levels of Testing

Hypothesis Testing: R provides functions and packages for conducting various hypothesis tests, such as t-tests, ANOVA, chi-square tests, and others, to assess the significance of relationships or differences in data.

Unit Testing: Unit testing in R involves the process of evaluating individual components or units of code to ensure that each component functions correctly in isolation. Testthat and RUnit are popular R packages used for unit testing.

Integration Testing: Integration testing in R involves testing how different components or functions work together to ensure that the integrated parts of the code function correctly as a whole.

Regression Testing: This type of testing in R involves checking whether recent code changes have affected the existing functionality and if any new bugs or errors have been introduced.

Each type of testing serves a specific purpose and helps ensure the overall quality and effectiveness of the software product. The goal of testing is to identify and resolve any defects or issues before the software is released to end-users, thereby enhancing the user experience and minimizing potential risks and vulnerabilities.

9.4 Factor in R

Factor in R is a variable used to categorize and store the data, having a limited number of different values. It stores the data as a vector of integer values. Factor in R is also known as a categorical variable that stores both string and integer data values as levels. Factor is mostly used in Statistical Modeling and exploratory data analysis with R.

In a dataset, we can distinguish two types of variables:

- categorical
- continuous

In descriptive statistics for categorical variables in R, the value is limited and usually based on a particular finite group. For example, a categorical variable in R can be countries, year, gender, occupation.

A continuous variable, however, can take any values, from integer to decimal. For example, we can have the revenue, price of a share, etc..

9.5 Categorical Variables

Categorical variables in R are stored into a factor. Let's check the code below to convert a character variable into a factor variable in R. Characters are not supported in machine learning algorithm, and the only way is to convert a string to an integer.

Syntax

```
factor(x = character(), levels, labels = levels, ordered = is.ordered(x))
```

Arguments

- **x:** A vector of categorical data in R. Need to be a string or integer, not decimal.
- **Levels:** A vector of possible values taken by x. This argument is optional. The default value is the unique list of items of the vector x.
- **Labels:** Add a label to the x categorical data in R. For example, 1 can take the label 'male' while 0, the label 'female'.
- **ordered:** Determine if the levels should be ordered in categorical data in R.

Example

```
# Create gender vector

gender_vector <- c("Male", "Female", "Female", "Male", "Male")

class(gender_vector)

# Convert gender_vector to a factor

factor_gender_vector <-factor(gender_vector)

class(factor_gender_vector)
```

Output

```
## [1] "character"
## [1] "factor"
```

A categorical variable in R can be divided into **nominal categorical variable** and **ordinal categorical variable.**

9.5.1 Nominal Categorical Variable

A categorical variable has several values but the order does not matter. For instance, male or female. Categorical variables in R does not have ordering.

Example

```
# Create a color vector
color_vector <- c('blue', 'red', 'green', 'white', 'black', 'yellow')
# Convert the vector to factor
factor_color <- factor(color_vector)
factor_color
```

Ouput

```
## [1] blue  red   green white black yellow
## Levels: black blue green red white yellow
```

9.5.2 Ordinal Categorical Variable

Ordinal categorical variables do have a natural ordering. We can specify the order, from the lowest to the highest with order = TRUE and highest to lowest with order = FALSE.

Example

```
# Create Ordinal categorical vector
day_vector <- c('evening', 'morning', 'afternoon', 'midday', 'midnight', 'evening')
# Convert `day_vector` to a factor with ordered level
factor_day <- factor(day_vector, order = TRUE, levels =c('morning', 'midday', 'afternoon', 'evening', 'midnight'))
factor_day
```

Output

```
[1] evening  morning  afternoon midday   midnight evening
Levels: morning < midday < afternoon < evening < midnight
# Count the number of occurence of each level
summary(factor_day)
```

Ouput

```
morning   midday  afternoon  evening  midnight
      1        1          1        2         1
```

R ordered the level from 'morning' to 'midnight' as specified in the levels parenthesis.

EXERCISE

Short Questions

1. What is Statistical Testing?
2. Mention the process of Statistical Testing.
3. What is Statistical Modelling?
4. Define Sampling Distributions in R.
5. What is Testing?
6. Mention the various types of Testing.
7. What is meant by Factor in R?
8. Mention the two types of variables in a dataset.
9. Define the0 -Categorical Variables.
10. Write the syntax for categorized variables and describe the arguments.
11. What is Nominal Categorical Variable?
12. Mention the types of Categorical Variables.

Long Questions

1. Explain Statistical Testing in detail.
2. Explain Statistical Modelling and its keys steps in detail.
3. Write a note on Sampling Distributions in R.
4. Write a program to create a sampling distribution of the sample mean in R.
5. Explain Testing and its levels in detail.
6. Define factor in R. Explain the various types of variables included in dataset.
7. What is Categorical Variables? Write its syntax and give an example.
8. Write a note on the types of Categorical Variables.

Ω♦Ω♦Ω♦Ω

CHAPTER 10

Hypothesis Testing

10.1 Introduction

Hypothesis testing is a fundamental statistical method used to make inferences about population parameters based on sample data. It involves evaluating the validity of a claim or hypothesis about a population parameter using sample evidence. The process of hypothesis testing follows a structured framework and aims to assess whether the evidence from the data supports or contradicts the proposed hypothesis.

The key steps involved in hypothesis testing include:

- ***Formulate Hypotheses:*** Define a null hypothesis (H0) that represents the default or status quo assumption and an alternative hypothesis (Ha or H1) that contradicts the null hypothesis and represents the claim or effect you want to test.
- ***Collect Data:*** Gather relevant data through experiments, surveys, or observations. The data collected should be representative of the population under investigation and should align with the research question and hypotheses.
- ***Choose a Statistical Test:*** Select an appropriate statistical test based on the type of data and the nature of the hypotheses being tested. Common tests include t-tests, z-tests, chi-square tests, ANOVA, regression analysis, and others, depending on the specific research context.
- ***Compute the Test Statistic and P-value:*** Calculate the test statistic from the sample data and determine the probability of observing the test statistic, or a more extreme value, under the assumption that the null hypothesis is true. This probability is known as the p-value.
- ***Compare the P-value with the Significance Level:*** Compare the calculated p-value with a predetermined significance level (α) to decide whether to reject the null hypothesis. If the p-value is less than or equal to the significance level, the null hypothesis is rejected in favor of the alternative hypothesis.

- ***Draw Conclusions:*** Based on the statistical results, draw conclusions about the validity of the null hypothesis and make inferences about the population parameter of interest. Consider the practical significance of the results in addition to the statistical significance.

The process of testing the hypothesis made by the researcher or to validate the hypothesis. To perform hypothesis testing, a random sample of data from the population is taken and testing is performed. Based on the results of the testing, the hypothesis is either selected or rejected. This concept is known as **Statistical Inference.**

- **One sample T-Testing**
- **Two-sample T-Testing**
- **Paired T-Test**

R provides a powerful environment for conducting various statistical tests and analyses, allowing researchers and data analysts to test specific hypotheses and draw meaningful conclusions.

10.1.1 The Objectives of Conducting Hypothesis Testing

- **Testing Research Hypotheses:** Hypothesis testing allows researchers to formally evaluate research questions or hypotheses based on empirical data.
- **Making Data-Driven Decisions:** By using hypothesis testing in R, data analysts can make informed decisions about whether to accept or reject a specific claim or hypothesis based on the available data.
- **Quantifying Uncertainty:** Hypothesis testing provides a framework for quantifying the uncertainty associated with sample estimates and determining the likelihood of observed effects or differences occurring by chance.
- **Comparing Groups or Conditions:** Hypothesis testing facilitates the comparison of group means, proportions, variances, or other parameters, enabling researchers to identify significant differences or similarities between different groups or conditions.
- **Evaluating Relationships:** Hypothesis testing allows analysts to assess the strength and significance of relationships between variables, helping to determine whether observed associations are statistically meaningful.

10.1.2 Four-Step Process of Hypothesis Testing

The four-step process of hypothesis testing in R involves a systematic approach to evaluate a claim about a population parameter based on sample data. Here's an overview of the four steps and how they can be implemented in R:

1. State the Hypotheses

- **Null Hypothesis (H0):** A statement of no effect or no difference.
- **Alternative Hypothesis (H1 or Ha):** A statement contradicting the null hypothesis.

 # Example of stating hypotheses

Syntax

t.test(x, mu)

Parameters

- x: represents numeric vector of data
- mu: represents true value of the mean

Example

```
# Defining sample vector
x <- rnorm(100)
# One Sample T-Test
t.test(x, mu = 5)
```

Output

```
One Sample t-test
data: x
t = -49.504, df = 99,
p-value < 2.2e-16
Alternative hypothesis: true mean is not equal to 5
```

95 percent confidence interval: -0.1910645 0.2090349 sample estimates: mean of x 0.008985172

- Data: The dataset 'x' was used for the test.
- The determined t-value is -49.504.
- Degrees of Freedom (df): The t-test has 99 degrees of freedom.
- The p-value is 2.2e-16, which indicates that there is substantial evidence refuting the null hypothesis.
- Alternative hypothesis: The true mean is not equal to five, according to the alternative hypothesis.
- 95 percent confidence interval: (-0.1910645, 0.2090349) is the confidence interval's value. This range denotes the values that, with 95% confidence, correspond to the genuine population mean.

Example

```
# Example data
data <- c(5.1, 4.8, 5.3, 5.2, 4.9, 5.0, 5.1, 4.9, 5.2, 5.0)
```



```
# H0: Population mean is equal to a specific value
# H1: Population mean is not equal to a specific value
```

$$\# H0: \mu = 100$$

$$\# H1: \mu \neq 100$$

2. Set the Significance Level

- Choose the significance level (α), typically 0.05 or 0.01, which represents the threshold for deciding whether to reject the null hypothesis.

```
# Example of setting the significance level
alpha <- 0.05
```

3. Calculate the Test Statistic and P-value/Analyze Sample Data

- Conduct the appropriate statistical test based on the data and hypotheses to calculate the test statistic and the corresponding p-value.

```
# Example of calculating the test statistic and p-value
# Assuming data is in the vector 'data'
result <- t.test(data)
# Extract the test statistic and p-value
test_statistic <- result$statistic
p_value <- result$p.value
```

4. Make a Decision/Interpret Decision

- Compare the p-value to the significance level (α) to decide whether to reject the null hypothesis or fail to reject it.

```
# Example of making a decision
if (p_value < alpha) {
print("Reject the null hypothesis")
} else {
print("Fail to reject the null hypothesis")
}
```

10.2 One Sample T-Testing

One-sample t-test is used to compare the mean of a single sample to a known or hypothesized population mean. In R, it is possible to conduct a one-sample t-test using the t.test() function.

```
# Assuming the null hypothesis that the true mean is 5
result <- t.test(data, mu = 5)
print(result)
```

Output

```
One Sample t-test
data: data
t = 1, df = 9, p-value = 0.3434
alternative hypothesis: true mean is not equal to 5
95 percent confidence interval:
4.936892 5.163108
sample estimates:
mean of x
5.05
```

In this example, data represents your sample data. The t.test() function conducts the one-sample t-test, with mu representing the hypothesized population mean. The function will provide the test statistic, the p-value, and the confidence interval for the mean.

10.3 Two-Sample T-Test

10.3.1 t.test()

A two-sample t-test is used to compare the means of two independent groups to determine whether they are significantly different from each other. In R, you can perform a two-sample t-test using the t.test() function.

Syntax

t.test(x, y)

Parameters

x and y: Numeric vectors

Example

```
# Sample data for two groups
group1 <- c(25, 27, 26, 28, 30, 29, 31, 32, 30, 28)
group2 <- c(22, 24, 26, 27, 28, 25, 29, 31, 30, 28)
```



```
# Perform a two-sample t-test
result <- t.test(group1, group2)
print(result)
```

Output

```
Welch Two Sample t-test
data: group1 and group2
t = 1.4191, df = 17.142, p-value = 0.1738
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.7771851  3.9771851
sample estimates:
mean of x mean of y
28.6      27.0
```

10.3.2 wilcox.test()

wilcox.test()function is to Conduct a Wilcoxon rank-sum test (Mann-Whitney U test) to compare the medians of two independent samples. This non-parametric test is used when the assumptions of the t-test are not met, such as when the data are not normally distributed or when the sample sizes are small.

Syntax

wilcox.test(x, y, alternative = c("two.sided", "less", "greater"), mu = 0, paired = FALSE, conf.int = FALSE, conf.level = 0.95)

- **x and y:** Numeric vectors containing the two samples that you want to compare.
- **alternative:** A character string specifying the alternative hypothesis, which can be "two.sided" (default), "less", or "greater".
- **mu:** A numeric value giving the hypothesized median under the null hypothesis.
- **paired:** A logical value indicating whether the samples are paired.
- **conf.int:** A logical value indicating whether to compute the confidence interval of the true location shift.
- **conf.level:** The confidence level of the interval, which is set to 0.95 by default.

Example

```
# Create two independent samples
group1 <- c(17, 12, 20, 21, 19)
group2 <- c(15, 14, 18, 16, 13)
```

```
# Perform Wilcoxon rank sum test
result <- wilcox.test(group1, group2, alternative = "two.sided")
print(result)
```

Output

```
Wilcoxon rank sum exact test
data: group1 and group2
W = 19, p-value = 0.2222
alternative hypothesis: true location shift is not equal to 0
```

10.4 Paired T-Test

Paired t-test is performed using the t.test() function. The paired t-test is used when you have two related groups or samples. The data should be paired because the measurements in each group are not independent, and each observation in one group has a unique corresponding measurement in the other group.

Example

```
# Example data for paired t-test
before <- c(23, 21, 29, 18, 25)
after <- c(19, 20, 28, 17, 22)
# Perform paired t-test
result <- t.test(before, after, paired = TRUE)
print(result)
```

Output

```
Paired t-test
data: before and after
t = 3.1623, df = 4, p-value = 0.03411
alternative hypothesis: true mean difference is not equal to 0
95 percent confidence interval:
0.2440219 3.7559781
sample estimates:
mean difference
2
```

- before and after are the two sets of paired data. Each element in before corresponds to the same element at the same index in after.

- t.test() is used to perform the paired t-test. Setting paired = TRUE indicates that the two samples are paired.
- The result is stored in the result variable.
- Finally, the result is printed to the console.

10.5 Chi-Square Test

The **Chi-Square Test** is used to analyze the frequency table (i.e., contingency table), which is formed by two categorical variables. The chi-square test evaluates whether there is a significant relationship between the categories of the two variables.

The Chi-Square Test is a statistical method which is used to determine whether two categorical variables have a significant correlation between them. These variables should be from the same population and should be categorical like- Yes/No, Red/Green, Male/Female, etc. group chi-square test is also useful while comparing the tallies or counts of categorical responses between two(or more) independent s.

Syntax

chisq.test(x, y = NULL, correct = TRUE)

- **x:** A contingency table (a matrix or a data frame) containing the observed counts or frequencies of the categories.
- **y:** If you have a contingency table, you can provide it as x, and y can be left as NULL. If you have a matrix or data frame with rows and columns representing two categorical variables, you can provide both x and y.
- **correct:** A logical value indicating whether to apply continuity correction. It's usually set to TRUE.

An example of how to use the chi-square test in R:

Example

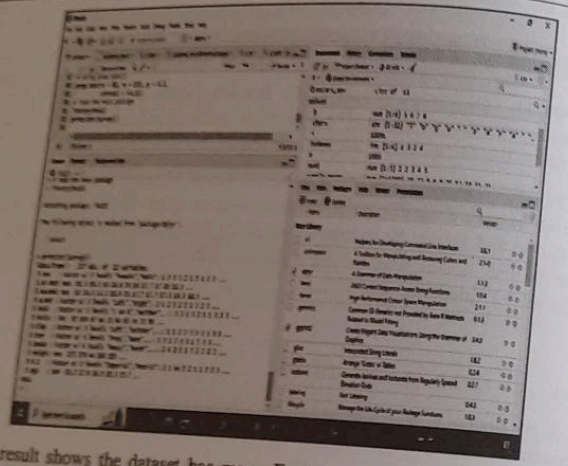
To take the survey data in the MASS library which represents the data from a survey conducted on students.

```
# load the MASS package
library(MASS)
print(str(survey))
```

Output

```
'data.frame': 237 obs. of 12 variables: $ Sex : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2
1 2 2 ... $ Wr.Hnd: num 18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ... $ NW.Hnd: num 18 20.5
13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ... $ W.Hnd : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2
2 2 2 2 ... $ Fold : Factor w/ 3 levels "L on R","Neither",..: 3 3 1 3 2 1 1 3 3 3 ... $ Pulse : int 92
104 87 NA 35 64 83 74 72 90 ... $ Clap : Factor w/ 3 levels "Left","Neither",..: 1 1 2 2 3 3 3 3 3
```

3 ... $ Exer : Factor w/ 3 levels "Freq","None",..: 3 2 2 2 3 3 1 1 3 3 ... $ Smoke : Factor w/ 4 levels "Heavy","Never",..: 2 4 3 2 2 2 2 2 2 2 ... $ Height: num 173 178 NA 160 165 ... $ M.I : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2 2 ... $ Age : num 18.2 17.6 16.9 20.3 23.7 ... NULL



The above result shows the dataset has many Factor variables which can be considered as categorical variables.

In this, model we consider the variables "Exer" and "Smoke".

The Smoke column records the students smoking habits. the Exer column records their exercise level.

To test the hypothesis whether the students smoking habit is independent of their exercise level at .05 significance level.

Example

```
# Create a data frame from the main data set.
stu_data = data.frame(survey$Smoke,survey$Exer)
# Create a contingency table with the needed variables.
stu_data = table(survey$Smoke,survey$Exer)
```

Output

	Freq	None	Some
Heavy	7	1	3
Never	87	18	84

Occas	12	3	4
Regul	9	1	7

finally we apply the chisq.test() function to the contingency table stu_data.

```
# applying chisq.test() function
print(chisq.test(stu_data))
```

Output

```
Pearson's Chi-squared test
data: stu_data X-squared = 5.4885, df = 6, p-value = 0.4828
```

As the p-value 0.4828 is greater than the .05, we conclude that the smoking habit is independent of the exercise level of the student and hence there is a weak or no correlation between the two variables.

Example

```
# Create a contingency table
observed_data <- matrix(c(45, 30, 15, 60), nrow = 2, byrow = TRUE)
colnames(observed_data) <- c("Category A", "Category B")
rownames(observed_data) <- c("Group 1", "Group 2")
# Perform the chi-square test
result <- chisq.test(observed_data)
# Print the results
print(result)
```

Output

```
Pearson's Chi-squared test with Yates' continuity correction
data:  observed_data
X-squared = 23.361, df = 1, p-value = 1.343e-06
```

In this example:

- two rows representing two groups and two columns representing two categories.
- We specify row and column names for the table.
- We use the chisq.test() function to perform the chi-square test on the contingency table.
- The results are stored in the result variable.
- We print the results, which include the chi-squared statistic, degrees of freedom, the p-value, and other information.

10.6 Advantages and Disadvantages of Hypothesis Testing

Advantages

Objective Decision Making: Hypothesis testing provides a structured framework for making objective decisions based on data analysis, reducing the influence of personal bias.

Scientific Validity: It allows researchers to draw conclusions about population parameters based on sample data, ensuring scientific validity in the research process.

Standardized Approach: Hypothesis testing follows a standardized approach, making it easier for other researchers to replicate the analysis and verify the results.

Quantitative Results: It provides quantitative results in the form of test statistics and p-values, enabling researchers to make statistically sound inferences about the research hypotheses.

Controlled Experiments: Hypothesis testing is well-suited for controlled experiments, where researchers can establish causality and test specific hypotheses.

Disadvantages

Assumptions: Hypothesis testing relies on various assumptions about the data, such as normality, independence, and homogeneity of variance. Violations of these assumptions can affect the validity of the results.

Simplification of Reality: Hypothesis testing often simplifies complex real-world scenarios, leading to a reduction in the richness of the data and potentially overlooking important nuances.

Risk of Errors: There is a risk of committing Type I errors (rejecting a true null hypothesis) and Type II errors (failing to reject a false null hypothesis), especially when the sample size is small or the effect size is small.

Limited Scope: Hypothesis testing is constrained by the specific hypotheses formulated by the researchers. It may not capture the full complexity of relationships between variables in the data.

10.7 Proportions Testing

Proportion testing is commonly used to analyze categorical data, especially when working with binary outcomes or proportions. It helps assess whether the proportions of successes or events differ significantly across groups or whether a sample proportion significantly deviates from an expected proportion.

In R you can perform various types of proportions testing using specific functions and methods. Some common types of proportions testing include:

Syntax

The **prop.test()** function is used to perform proportions testing

prop.test(x, n, p = NULL, alternative = "two.sided", conf.level = 0.95)

- **x:** Number of successes (a vector of counts of successes).
- **n:** Number of trials or total sample size (a vector of the same length as x).
- **p:** A single number giving the hypothetical probability of success. If not given, it is set to the overall proportion of successes.
- **alternative:** A character string specifying the alternative hypothesis, which can be "two.sided," "less," or "greater."
- **conf.level:** Confidence level of the interval.

Example

Program to perform a proportions test in R:

```
# Define the number of successes and the total number of trials
successes <- 20
total_trials <- 50
# Perform a proportions test
prop.test(successes, total_trials, alternative = "two.sided", conf.level = 0.95)
```

Output

```
1-sample proportions test with continuity correction
data: successes out of total_trials, null probability 0.5
X-squared = 1.62, df = 1, p-value = 0.2031
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
0.2673293 0.5479516
sample estimates:
p
0.4
```

Where successes represents the number of successes and total_trials represents the total number of trials. The alternative parameter is set to "two.sided" to test for a two-tailed alternative hypothesis, and the conf.level parameter is set to 0.95 to calculate a 95% confidence interval.

10.7.1 One Sample T-Testing (Testing Mean)

- One sample T-Testing approach collects a huge amount of data and tests it on random samples.
- To perform T-Test in R, normally distributed data is required. This test is used to test the mean of the sample with the population.

For example, the height of persons living in an area is different or identical to other persons ving in other areas.

Syntax

t.test(x, mu)

Parameters

x: represents numeric vector of data

mu: represents true value of the mean

Example

```
# Defining sample vector
x <- rnorm(100)
# One Sample T-Test
t.test(x, mu = 5)
```

Output

```
One Sample t-test
data: x
t = -49.504, df = 99,
p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 5
95 percent confidence interval: -0.1910645 0.2090349 sample estimates: mean of x
0.008985172
```

Where,

- **Data:** The dataset 'x' was used for the test.
- The determined t-value is -49.504.
- **Degrees of Freedom (df):** The t-test has 99 degrees of freedom.

  The p-value is 2.2e-16, which indicates that there is substantial evidence refuting the null hypothesis.
- **Alternative Hypothesis:** The true mean is not equal to five, according to the alternative hypothesis.
- **95 Percent Confidence Interval:** (-0.1910645, 0.2090349) is the confidence interval's value. This range denotes the values that, with 95% confidence, correspond to the genuine population mean.

10.7.2 One-Proportion Z-Test

A one-proportion z-test to assess whether a sample proportion significantly differs from a hypothesized population proportion. The test evaluates whether there is enough evidence to reject the null hypothesis that the sample proportion is equal to the specified value.

The One proportion Z-test is used to compare an observed proportion to a theoretical one when there are only two categories.

For example, we have a population of mice containing half male and half females (p = 0.5 = 50%).

Some of these mice (n = 160) have developed spontaneous cancer, including 95 males and 65 females.

We want to know, whether cancer affects more males than females? So in this problem:

- The number of successes (male with cancer) is 95
- The observed proportion (po) of the male is 95/160
- The observed proportion (q) of the female is 1 – po
- The expected proportion (pe) of the male is 0.5 (50%)
- The number of observations (n) is 160

The Formula for One-Proportion Z-Test

The test statistic (also known as z-test) can be calculated as follow:

$$r = \frac{p_0 - p_e}{\sqrt{p_0 q / n}}$$

where, p_o: the observed proportion **q:** $1 - p_o$

p_e: the expected proportion

n: the sample size

Implementation in R

In R Language, the function used for performing a z-test is **binom.test()** and **prop.test()**.

Syntax

binom.test(x, n, p = 0.5, alternative = "two.sided") prop.test(x, n, p = NULL, alternative = "two.sided", correct = TRUE)

Parameters

- **x** = number of successes and failures in data set.
- **n** = size of data set.
- **p** = probabilities of success. It must be in the range of 0 to1
- **alternative** = a character string specifying the alternative hypothesis.

correct = a logical indicating whether Yates' continuity correction should be applied where possible

Example 1

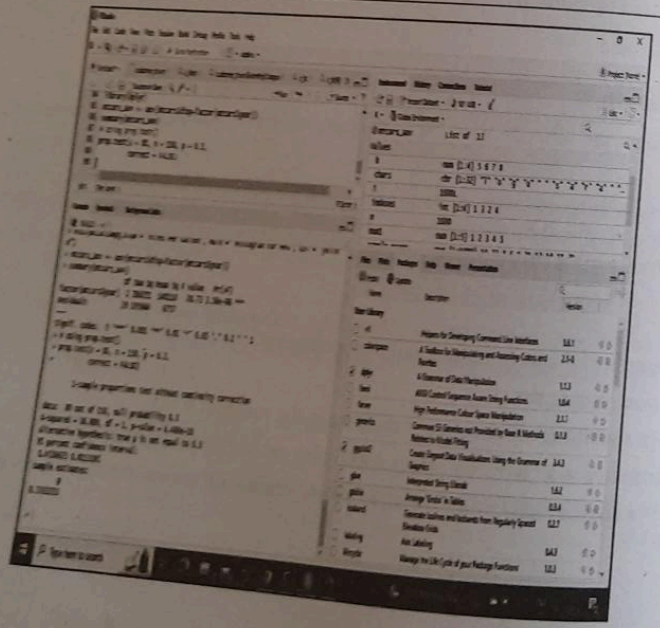
Let's assume that 30 out of 70 people recommend Street Food to their friends. To test this claim, a random sample of 150 people obtained. Of these 150 people, 80 indicate that they recommend Street Food to their friend. Is this claim accurate? Use alpha = 0.05.

Solution: Now x=.80, P=.30, n=150. We want to know, whether the People recommend street food to their friend than healthy food? Let's use the function **prop.test()** in R.

```
# Using prop.test()
prop.test(x = 80, n = 150, p = 0.3,
correct = FALSE)
```

Output

```
1-sample proportions test without continuity correction
data:  80 out of 150, null probability 0.3
X-squared = 38.889, df = 1, p-value = 4.486e-10
alternative hypothesis: true p is not equal to 0.3
95 percent confidence interval:
0.4536625 0.6113395
sample estimates:
   p
0.5333333
```



10.7.3 Two-Proportions Z-Test

The two-proportion z-test compares the proportions of two independent groups or samples. The z-test is used to determine if there is a significant difference between the proportions of success in the two groups.the prop.test() function to conduct a two-proportion z-test.

For example, we have two groups of individuals:

Group A with lung cancer: n = 500

Group B, healthy individuals: n = 500

The number of smokers in each group is as follow:

Group A with lung cancer: n = 500, 490 smokers, pA=490/500=98

Group B, healthy individuals: n = 500, 400 smokers, pB=400/500=80

In this setting:

The overall proportion of smokers is p=frac(490+400)500+500=89

The overall proportion of non-smokers is q=1−p=11

```
prop.test(x, n, p = NULL, alternative = "two.sided",
correct = TRUE)
```

- **x:** a vector of counts of successes
- **n:** a vector of count trials
- **alternative:** a character string specifying the alternative hypothesis
- **correct:** a logical indicating whether Yates' continuity correction should be applied where possible

Example

```
res <- prop.test(x = c(490, 400), n = c(500, 500))
# Printing the results
Res
```

Output

```
2-sample test for equality of proportions with continuity correction
data:  c(490, 400) out of c(500, 500)
X-squared = 80.909, df = 1, p-value < 2.2e-16
alternative hypothesis: two.sided
95 percent confidence interval:
 0.1408536 0.2191464
sample estimates:
prop 1  prop 2
0.98    0.80
```

10.8 Error in Hypothesis Testing

In Hypothesis testing, Error is the estimate of the approval or rejection of a particular hypothesis. There are mainly **two types of errors in hypothesis testing:**

1. **Type I Error (also known as alpha error):** Type I error occurs when we reject the Null hypothesis but the Null hypothesis is correct. This case is also known as a false positive.
2. **Type II Error (also known as beta error):** Type II error occurs when we fail to remove the Null Hypothesis when the Null hypothesis is incorrect/the alternative hypothesis is correct. This case is also known as a false negative.

Type I and Type II Error

Null hypothesis is ...	True	False
Rejected	Type I error False positive Probability = α	Correct decision True position Probability = 1 – β
Not rejected	Correct decision True negative Probability = 1 – α	Type II error False negative Probability = β

10.8.1 Type I Error

A Type I error, also known as an alpha error, occurs when the null hypothesis is rejected even though it is actually true. This means that the test results lead to the conclusion that there is a significant effect, when in fact there is no such effect. In the context of R, the probability of committing a Type I error is represented by the significance level, denoted by alpha (α).

general syntax for conducting a hypothesis test that may result in a Type I error in R:

```
# Sample data
data <- c(3, 5, 4, 6, 7, 8, 2, 3, 5, 4)
# One-sample t-test
result <- t.test(data, mu = 4.5)
# Extract p-value
p_value <- result$p.value
# Define the significance level (alpha)
alpha <- 0.05
# Check for Type I error
if (p_value < alpha) {
```



```
print("Reject the null hypothesis, Type I error might have occurred.")
} else {
print("Fail to reject the null hypothesis.")
}
```

In this example, a one-sample t-test using the t.test() function in R. We then extract the p-value from the test results and define the significance level (alpha). If the p-value is less than the significance level, we conclude that the null hypothesis should be rejected and print a message indicating that a Type I error might have occurred. Otherwise, we print a message stating that we fail to reject the null hypothesis.

Output

```
[1] "Fail to reject the null hypothesis."
```

10.8.2 Type II Error

A Type II error, often denoted as β (beta), is a statistical term used in hypothesis testing that occurs when the null hypothesis is not rejected when it is actually false. In simpler terms, it is the error of failing to detect a real effect or difference when it exists. This error is particularly relevant when evaluating the power of a statistical test, which is the probability of correctly rejecting the null hypothesis when it is false.

An example of a Type II error in the context of a hypothesis test in R:

```
# Generate two sets of data with different means
set.seed(123)
data1 <- rnorm(100, mean = 5, sd = 2)
data2 <- rnorm(100, mean = 7, sd = 2)
# Perform a two-sample t-test with unequal variances
result <- t.test(data1, data2, var.equal = FALSE)
# Extract the p-value and true means
p_value <- result$p.value
true_mean1 <- mean(data1)
true_mean2 <- mean(data2)
# Set a significance level
alpha <- 0.05
# Check for Type II error
if (p_value > alpha) {
```

```
    print("Fail to reject the null hypothesis, Type II error might have occurred.")
} else {
    print("Reject the null hypothesis.")
}
```

In this script, two sets of data are generated with different means. The t.test() function is then used to perform a two-sample t-test with unequal variances. The script extracts the p-value and the true means of the two sets of data, and a significance level (alpha) is defined. If the p-value is greater than the significance level, the script prints a message indicating that a Type II error might have occurred. Otherwise, it prints a message stating that the null hypothesis is rejected.

Output

```
[1] "Reject the null hypothesis."
```

10.8.3 Power of Hypothesis Testing

Hypothesis testing, the power of a statistical test is the probability that the test will correctly reject the null hypothesis when the alternative hypothesis is true. In simpler terms, it is the probability of detecting an effect or a difference if it truly exists. High statistical power is desirable as it indicates that the test can effectively identify true effects, thus reducing the likelihood of Type II errors (false negatives).

The power of a statistical test is influenced by several factors:

Effect Size (ES): The magnitude of the difference or effect you are trying to detect. A larger effect size increases the statistical power.

Sample Size (n): The number of observations or participants in your study. Increasing the sample size generally enhances statistical power.

Significance Level (α): The chosen level of significance, often set at 0.05. A lower alpha increases power but also increases the risk of Type I errors.

Variability (Standard Deviation or Variance): The amount of variability or spread in the data. A smaller variance increases power.

To calculate statistical power in hypothesis testing, you can use statistical software like R or specialized power analysis packages. Here's a general formula for power calculation:

$$\text{Power} = 1 - P(\text{Type II Error}) = 1 - \beta$$

Where:

- Power is the statistical power.
- P(Type II Error) is the probability of a Type II error.
- β (beta) is the probability of a Type II error.

for specific types of statistical tests (e.g., t-tests, ANOVA, chi-squared tests), different functions and packages can be used to perform power analysis and calculate power. For instance,

in R, you can use packages like pwr, pwr.t.test, pwr.anova.test, or pwr.chisq.test to conduct power analyses for various test scenarios.

Example

To calculate the power of a t-test in R using the pwr.t.test function:

```
# Load the required library
library(pwr)
# Set parameters
effect_size <- 0.8 # This is the standardized effect size
n <- 50 # Sample size per group
alpha <- 0.05 # Significance level
# Conduct a power analysis
power_analysis <- pwr.t.test(n = n, d = effect_size, sig.level = alpha, type = "two.sample")
# Extract the power value
power_value <- power_analysis$power
# Print the result
print(paste("Statistical power:", power_value))
```

In this example, we are performing a two-sample t-test and assuming equal variances between the groups. We set the effect size, sample size per group, and significance level. The pwr.t.test function calculates the power of the t-test, and we extract the power value from the analysis. The script then prints the statistical power to the console.

10.10 Analysis of Variance (ANOVA)

Analysis of Variance (ANOVA) is a statistical technique, commonly used to study differences between two or more group means. ANOVA test is centered on the different sources of variation in a typical variable. ANOVA in R primarily provides evidence of the existence of the mean equality between the groups. This statistical method is an extension of the t-test. It is used in a situation where the factor variable has more than one group.ANOVA is a statistical test for estimating how a quantitative dependent variable change according to the levels of one or more categorical independent variables. ANOVA tests whether there is a difference in the means of the groups at each level of the independent variable.

The null hypothesis (H0) of the ANOVA is no difference in means, and the alternative hypothesis (Ha) is that the means are different from one another. ANOVA using the aov() function, which stands for "analysis of variance." ANOVA is commonly used in various fields, including psychology, biology, and social sciences, to compare group means and understand the effects of different factors on the outcome variable.

Example

- A car company wishes to compare the average petrol consumption of THREE similar models of car and has available six vehicles of each model.
- A teacher is interested in a comparison of average percentage marks attained in the examinations of FIVE different subjects and has available the marks of eight students who all completed each examination.

10.10.1 The Basic Performing ANOVA in R

Data Preparation: Organize your data in a way that represents the groups or factors you want to compare. This typically involves setting up a data frame where each column represents a different group or factor.

ANOVA Function in R: Use the aov() function in R to perform ANOVA. This function takes a formula as an argument, typically in the form of response_variable ~ group_variable.

Interpreting ANOVA Results: After running the ANOVA, you can use the summary() function to view the ANOVA table, which provides key statistics such as the F-value, degrees of freedom, and p-value.

Some of The Key Properties of ANOVA

- **Decomposition of Variance:** ANOVA decomposes the total variance in the data into different components, such as the variance between groups and the variance within groups. This decomposition helps to understand the relative contributions of different sources of variability.
- **F-Test:** ANOVA uses the F-test to assess whether the variation among group means is larger than the variation within the groups. The F-test compares the ratio of the mean square variation between groups to the mean square variation within groups.
- **Assumptions:** ANOVA relies on several assumptions, including the normality of the data within each group, homogeneity of variances, and independence of observations. Violations of these assumptions can affect the validity of ANOVA results.
- **One-Way and Two-Way ANOVA:** ANOVA can be categorized into one-way and two-way ANOVA based on the number of factors being studied. One-way ANOVA examines the effects of a single factor, whereas two-way ANOVA investigates the effects of two factors simultaneously.
- **Post hoc Tests:** In cases where ANOVA indicates significant differences between groups, post hoc tests can be used to identify which specific groups differ significantly from each other. Common post hoc tests include Tukey's HSD (Honestly Significant Difference), Bonferroni, and Scheffe tests.

10.10.2 One-Way ANOVA

The process of a one-way ANOVA (one independent variable) and a two-way ANOVA (two independent variables).

ANOVA is a statistical method used to compare means of three or more groups to determine whether there are any statistically significant differences among the means. It is called "one-way" because there is only one independent variable affecting the dependent variable. This independent variable typically represents the different groups or levels being compared.

Syntax

```
model <- aov(response_variable ~ group_variable, data = your_data_frame)
```

Here's

- **model:** This is the variable that will store the ANOVA model.
- **aov():** The function used to fit the ANOVA model.
- **response_variable:** This is the continuous response variable you are analyzing across different groups.
- **group_variable:** This represents the categorical variable that defines the different groups.
- **your_data_frame:** This refers to the data frame that contains your data.

 # Example data for one-way ANOVA

```
group1 <- c(18, 20, 22, 24, 19)
group2 <- c(15, 17, 16, 19, 18)
group3 <- c(21, 23, 25, 24, 22)
```

 # Combine the data into a data frame

```
data <- data.frame(
value = c(group1, group2, group3),
group = factor(rep(1:3, each = 5))
)
```

 # Perform one-way ANOVA

```
model <- aov(value ~ group, data = data)
summary(model)
```

Output

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Group	2	91.2	45.6	12.67	0.0011 **
Residuals	12	43.2	3.6		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

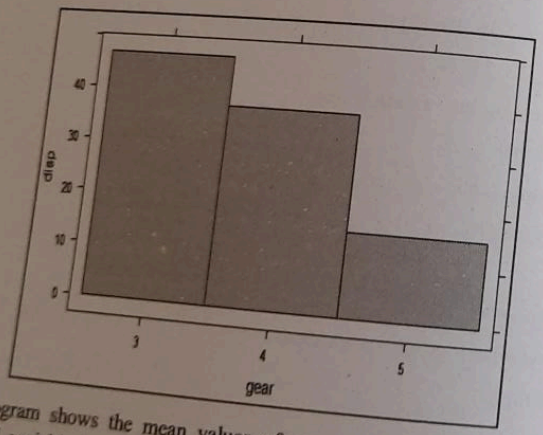
In this example:

- There are three groups of data: group1, group2, and group3.
- To combine the data into a data frame data, where the variable value contains the measurements, and the variable group represents the group labels.
- The aov() function to fit the one-way ANOVA model, with value as the response variable and group as the independent variable.

Example

```
# Installing the package
install.packages("dplyr")
# Loading the package
library(dplyr)
# Variance in mean within group and between group
histogram(mtcars$disp~factor(mtcars$gear),color='B',
xlab = "gear", ylab = "disp")
```

Output



- The Histogram shows the mean values of gear with respect of displacement. Hear categorical variable is the gear on which factor function is used and the continuous variable is disp.

10.10.3 Two-way ANOVA

The two-way ANOVA is used to analyze the effects of two categorical independent variables on a continuous dependent variable. Here's an example of how to perform a two-way ANOVA in R:


```
# Creating a sample dataset
set.seed(123)
group1 <- gl(2, 4, labels = c("A", "B"))
group2 <- gl(2, 2, 8, labels = c("X", "Y"))
response <- rnorm(8, mean = 10, sd = 2)
# Combining the data into a data frame
data <- data.frame(group1, group2, response)
# Performing two-way ANOVA
model <- aov(response ~ group1 + group2 + group1:group2, data = data)
# Summarizing the ANOVA results
summary(model)
```

Output

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Group 1	1	0.020	0.020	0.005	0.946
Group 1	1	0.026	0.026	0.007	0.939
Group 1: Group 2	1	12.844	12.844	3.286	0.144
Residuals	4	15.635	3.909		

- A sample dataset with two categorical independent variables (group1 and group2) and one continuous dependent variable (response).
- The gl() function is used to generate factor levels for the groups.
- The data is combined into a data frame data.
- The aov() function is used to perform the two-way ANOVA, including the main effects of group1 and group2 as well as their interaction (group1:group2).
- The summary() function is then used to print the ANOVA results, including the F-values, p-values, and other relevant statistics.

Calculate test statistics using aov function.

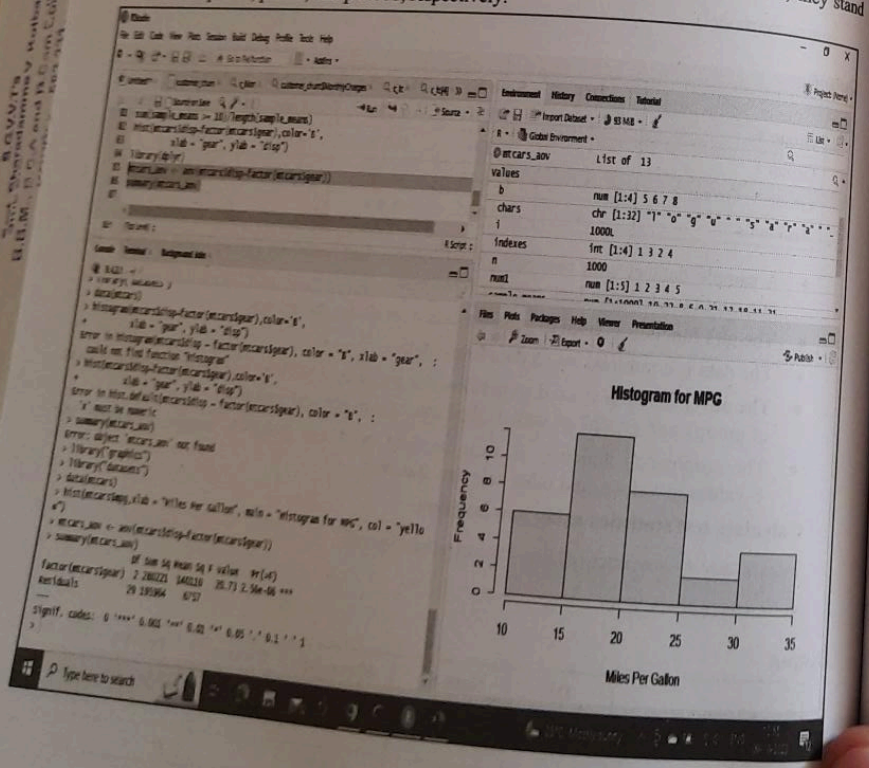
```
mtcars_aov <- aov(mtcars$disp~factor(mtcars$gear))
summary(mtcars_aov)
```

Output

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
factor(mtcars$gear)	2	280221	140110	20.73	2.56e-06 ***
Residuals	29	195964	6757		

— Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

- **Df:** The model's degrees of freedom.
- **Sum Sq:** The sums of squares, which represent the variability that the model is able to account for.
- **Mean Sq:** The variance explained by each component is represented by the mean squares.
- **F-value:** It is the measure used to compare the mean squares both within and between groups.
- **Pr(>F):** The F-statistics p-value, which denotes the factors' statistical significance.
- **Residuals:** Relative deviations from the group mean, are often known as residuals and their summary statistics.
- **Identifier Codes:** Asterisks (*) are used to show the degree of significance; they stand for p 0.05, p 0.01, and p 0.001, respectively.



EXERCISE

Short Questions

1. Define hypothesis testing.
2. Mention the key steps in hypothesis testing.
3. Write any 2 objective of conducting hypothesis testing.
4. What is Four-Step process in hypothesis testing?
5. Define One Sample T-Testing.
6. Define Two-Sample T-Testing.
7. Write the syntax of t.test().
8. What is wilcox.test()? Write its syntax.
9. Define paired T-test with syntax and example.
10. Define Chi –Square Test with syntax and example.
11. Mention any 2 advantages of hypothesis testing.
12. Mention any 2 disadvantages of hypothesis testing.
13. What is proportions testing?
14. Define One-Proportion Z-test.
15. Write the formula for One-Proportion Z-test.
16. Define Two-Proportion Z-test.
17. What is Error in Hypothesis Testing?
18. Mention the 2 main types of errors in hypothesis testing.
19. Mention the factors in power of hypothesis testing.
20. Define analysis of variance (ANOVA).
21. List out the key properties of ANOVA.
22. Define One-Way ANOVA with its syntax.
23. Define Two-Way ANOVA with an example
24. Write the syntax of Two-Way ANOVA.

Long Questions

1. Explain the key steps involved in hypothesis testing in detail.
2. What are the objectives of conducting hypothesis testing?
3. Explain the Four-Step Process of Hypothesis Testing.
4. Explain One Sample T-Testing with an example and write its syntax.

5. Write a short note on:
   (a) t.test()
   (b) wilcox.test()
6. Explain Paired T-Test in detail.
7. Explain the Chi-Square Test briefly.
8. Write down the advantages of Hypothesis Testing in detail.
9. What are the disadvantages of Hypothesis Testing explain briefly?
10. Explain Proportions testing in detail.
11. Explain One Sample T-Testing (Testing Macro)
12. Explain One-Proportion Z-Test and implementation in R.
13. Explain Two-Proportion Z-Test in detail.
14. Define error in hypothesis testing. Explain its types.
15. What is Power of hypothesis testing and describe the factors involved in the power of statistical test?
16. Write a program to calculate the power of a t-test in R using the prop.test function.
17. Explain the basic performing ANOVA in R and properties of ANOVA.
18. Explain the One-Way ANOVA in detail.
19. Explain the Two-Way ANOVA in detail.

CHAPTER 11

Regression

11.1 Introduction

Regression is a statistical tool to estimate the relationship between two or more variables. There is always one response variable and one or more predictor variables. Regression analysis is widely used to fit the data accordingly and further, predicting the data for forecasting. It helps businesses and organizations to learn about the behavior of their product in the market using the dependent/response variable and independent/predictor variable.

11.1.1 Types of Regression in R

There are mainly three types of Regression in R programming that is widely used. They are:

- Linear Regression
- Multiple Regression
- Logistic Regression

11.2 Linear Regression

The Linear Regression model is one of the most widely used three of the regression types. Linear regression is a statistical technique used to model the relationship between a dependent variable (often denoted as Y) and one or more independent variables (often denoted as X). It assumes a linear relationship between the dependent and independent variables.

The general mathematical equation for a linear regression is

$y = ax + b$

Parameters

- y is the response variable.
- x is the predictor variable.
- a and b are constants which are called the coefficients.

Implementation in R

In R programming, lm() function is used to create linear regression model.

Syntax: lm(formula, data)

Parameter

formula: This is a symbolic description of the model to be fitted. It is usually written in the form response ~ predictor1 + predictor2 + ... Here, response is the dependent variable, and predictor1, predictor2, etc., are the independent variables.

data: This argument specifies the data frame containing the variables in the formula.

Example

A Program where we have a dataset of heights and weights, and we want to perform a simple linear regression to predict weights based on heights.

```
# Generate some sample data
heights <- c(65, 71, 69, 68, 72, 66, 77, 73, 74, 60)
weights <- c(120, 150, 140, 130, 160, 125, 180, 170, 175, 110)
# Create a data frame from the data
data <- data.frame(heights, weights)
# Perform linear regression
model <- lm(weights ~ heights, data = data)
# Print the summary of the linear regression model
summary(model)
```

Output

```
Call:
lm(formula = weights ~ heights, data = data)
Residuals:
```

Min	1Q	Median	3Q	Max
-8.787	-4.025	-2.640	5.871	9.685

```
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) -188.2247   30.6403  -6.143 0.000276 ***
heights        4.8090    0.4399  10.933 4.34e-06 ***
---
```



```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 6.561 on 8 degrees of freedom
Multiple R-squared:  0.9373,    Adjusted R-squared:  0.9294
F-statistic: 119.5 on 1 and 8 DF,  p-value: 4.344e-06
```

11.2.1 Predict() Function

The predict() function in R is used to make predictions based on a fitted statistical model. It allows you to apply a model to new data or to the existing data to estimate the values of the dependent variable. The specific usage of the predict() function depends on the type of model you have fitted.

Syntax

predict(object, newdata, ...)

- **Object:** is the formula which is already created using the lm() function.
- **Newdata:** is the vector containing the new value for predictor variable. This is an optional argument that specifies the data frame or matrix containing the new data for which you want to make predictions.
- ...: Additional optional arguments that can be specific to the type of model you are using. For example, in the case of a linear regression model, you might specify interval="prediction" to compute prediction intervals.

Example

```
# Create a simple linear regression model
heights <- c(65, 71, 69, 68, 72, 66, 77, 73, 74, 60)
weights <- c(120, 150, 140, 130, 160, 125, 180, 170, 175, 110)
data <- data.frame(heights, weights)
model <- lm(weights ~ heights, data = data)
# Predict new weights for given heights
new_heights <- c(63, 70, 75)
new_data <- data.frame(heights = new_heights)
predictions <- predict(model, newdata = new_data)
# Display the predictions
print(predictions)
```

Output:

1	2	3
114.7416	148.4045	172.4494

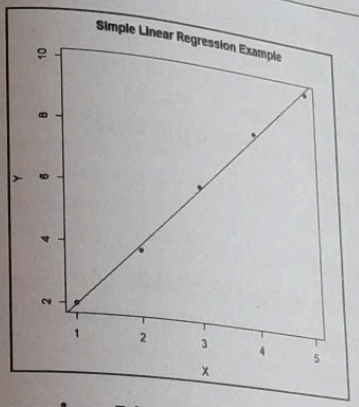
In this example, we first create a linear regression model using the lm() function. Then, we use the predict() function to make predictions for new heights provided in the new_data data frame. The function returns a vector of predicted values, which are the estimated weights corresponding to the new heights.

Example

Program to perform linear regression and plot the graph.

```
# Simple Linear Regression Example in R
# Create some sample data
x <- c(1, 2, 3, 4, 5)
y <- c(2, 3.8, 6.1, 8.2, 9.9)
# Perform linear regression
model <- lm(y ~ x)
# Print the summary of the linear regression model
summary(model)
# Plot the data points and the linear regression line
plot(x, y, main="Simple Linear Regression Example", xlab="X", ylab="Y", pch=16, col="blue")
abline(model, col="red")
# Adding more points to the graph
new_x <- c(6, 7)
new_y <- predict(model, data.frame(x=new_x))
points(new_x, new_y, pch=16, col="green")
```


Output



11.3 Linear Regression Line

A linear line showing the relationship between the dependent and independent variables is called a regression line. A regression line can show two types of relationship.

Positive Linear Relationship: If the dependent variable increases on the Y-axis and independent variable increases on X-axis, then such a relationship is termed as a Positive linear relationship.

Negative Linear Relationship: If the dependent variable decreases on the Y-axis and independent variable increases on the X-axis, then such a relationship is called a negative linear relationship.

Salary Dataset

Years Experienced	Salary
1.1	39343.00
1.3	46205.00
1.5	37731.00
2.0	43525.00
2.2	39891.00
2.9	56642.00
3.0	60150.00
3.2	54445.00
3.2	64445.00
3.7	57189.00

For general purposes, we define:

- x as a feature vector, i.e x = [x_1, x_2, …., x_n],
- y as a response vector, i.e y = [y_1, y_2, …., y_n]

for n observations (in the above example, n=10).

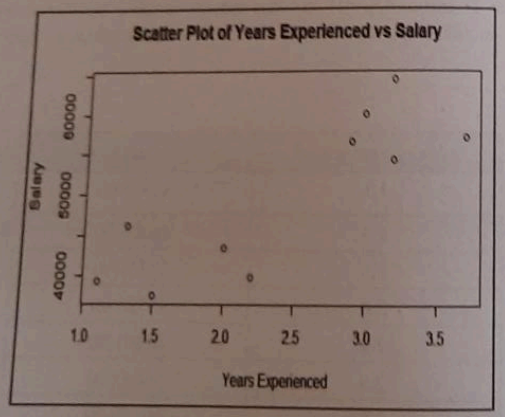
First we convert these data values into R Data Frame.

```
# Create the data frame
data <- data.frame(
Years_Exp = c(1.1, 1.3, 1.5, 2.0, 2.2, 2.9, 3.0, 3.2, 3.2, 3.7),
Salary = c(39343.00, 46205.00, 37731.00, 43525.00,
           39891.00, 56642.00, 60150.00, 54445.00, 64445.00, 57189.00)
```

Scatter plot of the given dataset:

```
# Create the scatter plot
plot(data$Years_Exp, data$Salary,
  xlab = "Years Experienced",
  ylab = "Salary",
  main = "Scatter Plot of Years Experienced vs Salary")
```

Output



- Now, we have to find a line that fits the above scatter plot through which we can predict any value of y or response for any value of x. The line which best fits is called the **Regression line.**
- The equation of **the regression line** is given by: y = a + bx

11.3.1 Advantages of Linear Regression

1. **Easy to implement:** R provides built-in functions, such as lm(), to perform Simple Linear Regression quickly and efficiently.
2. **Easy to interpret:** Simple Linear Regression models are easy to interpret, as they model a linear relationship between two variables.
3. **Useful for prediction:** Simple Linear Regression can be used to make predictions about the dependent variable based on the independent variable.
4. **Provides a measure of goodness of fit:** Simple Linear Regression provides a measure of how well the model fits the data, such as the R-squared value.

11.3.2 Disadvantages of Linear Regression

1. Assumes linear relationship: Simple Linear Regression assumes a linear relationship between the variables, which may not be true in all cases.
2. Sensitive to outliers: Simple Linear Regression is sensitive to outliers, which can significantly affect the model coefficients and predictions.
3. Assumes independence of observations: Simple Linear Regression assumes that the observations are independent, which may not be true in some cases, such as time series data.
4. Cannot handle non-numeric data: Simple Linear Regression can only handle numeric data and cannot be used for categorical or non-numeric data.
5. Overall, Simple Linear Regression is a useful tool for modeling the relationship between two variables, but it has some limitations and assumptions that need to be carefully considered.

11.4 Multiple Regression

Multiple regression is a statistical technique used to model the relationship between a dependent variable and multiple independent variables. It's an extension of simple linear regression, where you have more than one predictor variable. In multiple regression, you can analyze how each independent variable contributes to the variation in the dependent variable while controlling for the others..

Multiple Linear Regression basically describes how a single response variable Y depends linearly on a number of predictor variables.

The basic examples where Multiple Regression can be used are as follows:

1. The selling price of a house can depend on the desirability of the location, the number of bedrooms, the number of bathrooms, the year the house was built, the square footage of the lot, and a number of other factors.
2. The height of a child can depend on the height of the mother, the height of the father, nutrition, and environmental factors.

The general mathematical equation for multiple regression is –

$y = a + b_1x_1 + b_2x_2 + ...b_nx_n$

Parameters

- y is the response variable.
- a, b_1, b_2...bn are the coefficients.
- x_1, x_2, ...xn are the predictor variables.

The regression model is created using the **lm()** function in R. The model determines the value of the coefficients using the input data. Next we can predict the value of the response variable for a given set of predictor variables using these coefficients.

Implementation in R

Multiple regression in R programming uses the same **lm()** function to create the model.

Syntax: lm(formula, data)

Parameters

- **Formula:** It represents the formula on which data has to be fitted.
- **Data:** It represents dataframe on which formula has to be applied.

11.4.1 lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in multiple regression is –

lm(y ~ x1+x2+x3...,data)

Following is the description of the parameters used –

- **Formula** is a symbol presenting the relation between the response variable and predictor variables.
- **Data** is the vector on which the formula will be applied.

Example

Multiple Linear Regression

```
# Create some sample data
x1 <- c(1, 2, 3, 4, 5)
x2 <- c(3, 4, 5, 6, 7)
y <- c(3, 4.8, 6.9, 9.2, 10.9)
```



```
# Create a data frame
data <- data.frame(x1, x2, y)
# Perform multiple linear regression
model <- lm(y ~ x1 + x2, data = data)
# Print the summary of the multiple linear regression model
summary(model)
# Make predictions using the model
new_data <- data.frame(x1 = c(6, 7), x2 = c(8, 9))
new_predictions <- predict(model, newdata = new_data)
print(new_predictions)
```

Output

Call:

lm(formula = y ~ x1 + x2, data = data)

Residuals:

1	2	3	4	5
0.08	–0.14	–0.06	0.22	–0.10

Coefficients: (1 not defined because of singularities)

	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	0.90000	0.17963	5.01	0.0153	*
x1	2.02000	0.05416	37.30	4.24e-05	***
x2	NA	NA	NA	NA	

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1713 on 3 degrees of freedom

Multiple R-squared: 0.9978, Adjusted R-squared: 0.9971

F-statistic: 1391 on 1 and 3 DF, p-value: 4.24e-05

 # New Predictions

1	2
13.02	15.04

Example

```
# Sample data for multiple regression
set.seed(123)
x1 <- rnorm(100)
x2 <- rnorm(100)
y <- 2 * x1 - 3 * x2 + rnorm(100)
# Perform multiple regression
model <- lm(y ~ x1 + x2)
# Summary of the multiple regression model
summary(model)
# Visualization of the regression
par(mfrow=c(2, 2)) # Divide the plotting area into a 2x2 grid
# Scatterplot of x1 against y
plot(x1, y, main="Scatterplot of x1 against y", xlab="x1", ylab="y")
abline(model$coefficients[1], model$coefficients[2], col="red")
# Scatterplot of x2 against y
plot(x2, y, main="Scatterplot of x2 against y", xlab="x2", ylab="y")
abline(model$coefficients[1], model$coefficients[3], col="blue")
# Scatterplot of the fitted values against y
plot(fitted(model), y, main="Scatterplot of fitted values against y", xlab="Fitted values", ylab="y")
abline(0, 1, col="green")
# Residuals plot
plot(model, which=1)
```

Output

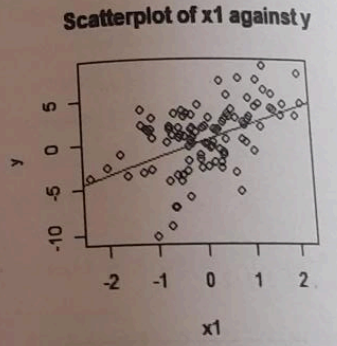
Call:

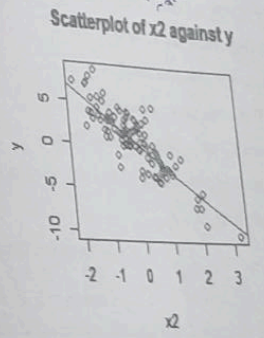
lm(formula = y ~ x1 + x2)

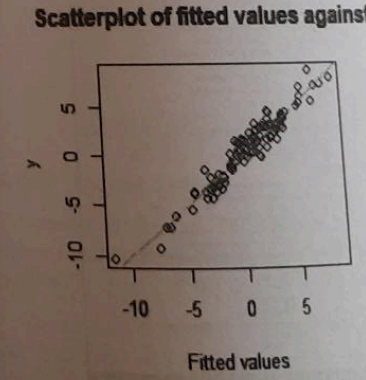
Residuals:

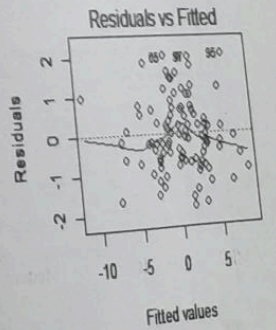
Min	1Q	Median	3Q	Max
–1.8730	–0.6607	–0.1245	0.6214	2.0798


```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.13507    0.09614   1.405    0.163
x1           1.86683    0.10487  17.801   <2e-16 ***
x2          -2.97619    0.09899 -30.064   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9513 on 97 degrees of freedom
Multiple R-squared:  0.9294,    Adjusted R-squared:  0.9279
F-statistic: 638.4 on 2 and 97 DF,  p-value: < 2.2e-16
```







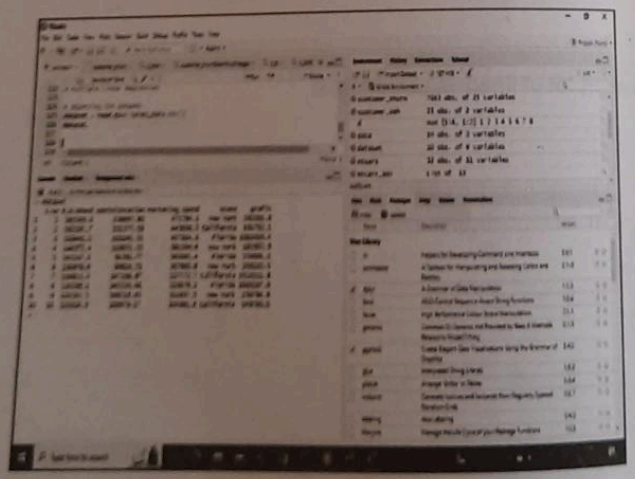
Example

Program to create a dataset and performing the multiple linear regression.

```
> dataset
   R.D.Spend Administration Marketing.Spend       State   Profit
1  165349.2       136897.80        471784.1    New York 192261.8
2  162597.7       151377.59        443898.5  California 191792.1
3  153441.5       101145.55        407934.5     Florida 191050.4
4  144372.4       118671.85        383199.6    New York 182902.0
5  142107.3        91391.77        366168.4     Florida 166187.9
6  131876.9        99814.71        362861.4    New York 156991.1
7  134615.5       147198.87        127716.8  California 156122.5
8  130298.1       145530.06        323876.7     Florida 155752.6
9  120542.5       148718.95        311613.3    New York 152211.8
10 123334.9       108679.17        304981.6  California 149760.0
```

```
# Multiple Linear Regression

# Importing the dataset

dataset = read.csv('data2.csv')

# Encoding categorical data

dataset$State = factor(dataset$State,

levels = c('New York', 'California', 'Florida'),

labels = c(1, 2, 3))
```

```
> dataset$State
 [1] 1 2 3 1 3 1 2 3 1 2
Levels: 1 2 3
```




```
# Splitting the dataset into the Training set and Test set

# install.packages('caTools')

library(caTools)

set.seed(123)

split = sample.split(dataset$Profit, SplitRatio = 0.8)

training_set = subset(dataset, split == TRUE)

test_set = subset(dataset, split == FALSE)

# Feature Scaling

# training_set = scale(training_set)

# test_set = scale(test_set)

# Fitting Multiple Linear Regression to the Training set

regressor = lm(formula = Profit ~ .,

data = training_set)

# Predicting the Test set results

y_pred = predict(regressor, newdata = test_set)
```

```
> regressor

Call:
lm(formula = Profit ~ ., data = training_set)

Coefficients:
    (Intercept)        R.D.Spend   Administration   Marketing.Spend
      2.816e+04        8.884e-01        5.670e-02        2.859e-02
         State2           State3
     -2.861e+03        9.172e+03


> y_pred
       5        8
179233.6 170602.2
```

11.4.2 Input Data

Consider the data set "mtcars" available in the R environment. It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.

The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables. We create a subset of these variables from the mtcars data set for this purpose.

Example

```
input <- mtcars[,c("mpg","disp","hp","wt")]
print(head(input))
```

Output

	mpg	disp	hp	wt
Mazda RX4	21.0	160	110	2.620
Mazda RX4 Wag	21.0	160	110	2.875
Datsun 710	22.8	108	93	2.320
Hornet 4 Drive	21.4	258	110	3.215
Hornet Sportabout	18.7	360	175	3.440
Valiant	18.1	225	105	3.460

11.5 Logistic Regression

Logistic Regression is another widely used regression analysis technique and predicts the value with a range. Moreover, it is used for predicting the values for categorical data. For example, Email is either spam or non-spam, winner or loser, male or female, etc. Mathematically,

$y = 1/(1+e^{-(a+ b_1x_1+b_2 x_2+b_3x_3+...)})$

where,

- y represents response variable
- **x** is the predictor variable
- **a** and **b** are the coefficients which are numeric constants.

Implementation in R

glm() function is used to create a logistic regression model.

Syntax

glm(formula, data, family)

Parameters

- **Formula:** It represents a formula on the basis of which model has to be fitted.
- **Data:** It represents dataframe on which formula has to be applied
- **Family:** It represents the type of function to be used. "binomial" for logistic regression

Example

The in-built data set "mtcars" describes different models of a car with their various engine specifications. In "mtcars" data set, the transmission mode (automatic or manual) is described by

the column am which is a binary value (0 or 1). We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.

```
# Select some columns form mtcars.
input <- mtcars[,c("am","cyl","hp","wt")]
print(head(input))
```

Output

```
                  am cyl  hp    wt
Mazda RX4          1   6 110 2.620
Mazda RX4 Wag      1   6 110 2.875
Datsun 710         1   4  93 2.320
Hornet 4 Drive     0   6 110 3.215
Hornet Sportabout  0   8 175 3.440
Valiant            0   6 105 3.460
```

11.6 Linear Model Selection

Linear Model Selection refers to the process of choosing the most appropriate set of predictor variables (or features) to include in a linear regression model. This selection process is crucial for building a reliable and interpretable model that captures the essential relationships between the predictors and the target variable.The primary goal of linear model selection in R is to strike a balance between model complexity and predictive performance. A more complex model may lead to overfitting, where the model fits the noise in the data rather than the underlying patterns, resulting in poor generalization to new data. On the other hand, a model that is too simple may fail to capture the true relationships between the predictors and the target variable.

R provides several methods for linear model selection, including forward selection, backward elimination, and stepwise selection. These methods help in systematically adding or removing predictors from the model based on certain criteria, such as p-values, Akaike Information Criterion (AIC), or Bayesian Information Criterion (BIC).

These criteria aim to balance the goodness of fit of the model with the complexity of the model, penalizing excessive complexity. By using these model selection techniques in R, you can identify the most relevant predictors and build a linear regression model that best explains the variability in the data while avoiding overfitting.

There are several methods and packages available for performing linear model selection. These methods include:

11.6.1 Leaps Package

The leaps package in R provides functions for performing subset selection, including best subset selection, forward selection, and backward elimination, in linear regression models. This

package is particularly useful when dealing with datasets containing a large number of predictors, as it efficiently explores various combinations of predictors to identify the best subset that optimizes model performance. The leaps package is commonly used for model selection and feature selection tasks in statistical analysis and data science.

The functions available in the leaps package are:

- **Regsubsets:** This function fits all possible models with a specified number of predictors and returns a detailed summary that includes information about the best-fitting models based on different selection criteria such as AIC, BIC, or adjusted R-squared.
- **Leaps:** This function computes the best subsets of variables for linear regression models using the exhaustive search method. It returns a list of results containing information about the best subset models.
- **Summary.regsubsets:** This function provides a summary of the results from the regsubsets function, displaying information about the best models and their respective criteria values.

11.7 Best Subset Selection

Best subset selection is a method for selecting the best-fitting model by considering all possible combinations of predictor variables. It involves fitting all possible regression models using a specific number of predictors and then selecting the model that best balances goodness of fit with model complexity. In R, you can perform best subset selection using the leaps package.

11.7.1 Steps to Perform Best Subset Selection

Data Preparation

Load your dataset and prepare it for regression analysis. Ensure that all the predictor variables are numeric and that the response variable is also numeric.

Install and Load Necessary Packages

If you haven't already, you might need to install and load the leaps package, which provides functions for best subset selection.

```
# Install and load the 'leaps' package
install.packages("leaps")
library(leaps)
```

Perform Best Subset Selection

Use the regsubsets function from the leaps package to perform best subset selection. This function fits all possible models and provides information about the best-fitting models for different numbers of predictors.

```
# Example data (replace with your own dataset)
```



```
data <- mtcars
# Best subset selection
fit <- regsubsets(mpg ~ ., data = data, nvmax = 3) # Select up to 3 variables
# Summary of the best subset selection results
summary(fit)
```

Output

Subset selection object

Call: regsubsets.formula(mpg ~ ., data = data, nvmax = 3)

10 Variables (and intercept)

	Forced in	Forced out
cyl	FALSE	FALSE
disp	FALSE	FALSE
hp	FALSE	FALSE
drat	FALSE	FALSE
wt	FALSE	FALSE
qsec	FALSE	FALSE
vs	FALSE	FALSE
am	FALSE	FALSE
gear	FALSE	FALSE
carb	FALSE	FALSE

1 subsets of each size up to 3

Selection Algorithm: exhaustive

cyl disp hp drat wt qsec vs am gear carb

1 (1) "

2 (1) "

3 (1) "

In this example, nvmax specifies the maximum number of predictor variables to consider in each model.

11.8 Stepwise Selection

Stepwise selection is a method used for selecting variables in a regression model. It involves systematically adding or removing variables from the model based on their statistical significance. In R, the 'stepAIC' function from the 'MASS' package is commonly used for stepwise regression.

There are different types of stepwise selection methods commonly used in the context of linear regression. These methods include forward selection, backward elimination, and stepwise regression.

11.8.1 Forward Selection

Forward selection is a stepwise regression approach that involves building a regression model by sequentially adding variables that improve the model fit the most at each step. It starts with a model that includes only the intercept and then systematically incorporates the most significant predictor variables one at a time until no more variables can significantly enhance the model's performance. We use the step function from the stats package to perform forward selection. The step function can be used for both forward and backward stepwise selection.

Syntax

```
# Fit a linear regression model and perform forward selection using the step function
model <- lm(response_variable ~ ., data = your_data)
forward_model <- step(model, direction = "forward")
summary(forward_model)
```

- **Response_variable:** The dependent variable you want to predict.
- **Your_data:** The dataset containing both the response variable and potential predictor variables.
- **lm(response_variable ~ ., data = your_data):** This fits an initial model using all available predictor variables (denoted by .).
- **Step(model, direction = "forward"):** This is where the forward selection process is initiated, and it uses the step function from the stats package. The direction argument specifies "forward," indicating that predictors should be added step by step.
- **Summary(forward_model):** This summarizes the results of the final model obtained through forward selection.

Example

```
# Example data (replace with your own dataset)
data <- mtcars
# Fit a simple linear regression model with intercept only
initial_model <- lm(mpg ~ 1, data = data)
# Perform forward selection
final_model <- step(initial_model, scope = list(upper = ~., lower = ~1), direction = "forward")
# Summary of the final model
summary(final_model)
```

Output

```
Call:
lm(formula = mpg ~ 1, data = data)

Residuals:
```

Min	1Q	Median	3Q	Max
–9.6906	–4.6656	–0.8906	2.7094	13.3094

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   20.091      1.065   18.86  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.027 on 31 degrees of freedom
```


11.8.2 Backward Elimination

Backward elimination is a stepwise regression technique used in linear regression to iteratively remove the least statistically significant predictors from the model. Here's the syntax for performing backward elimination in R using the step function from the stats package:

Syntax

```
model <- lm(response_variable ~ ., data = your_data)
backward_model <- step(model, direction = "backward")
summary(backward_model)
```

- **Response_variable** is the dependent variable you want to predict.
- **Your_data** is the dataset containing both the response variable and potential predictor variables.
- **lm(response_variable ~ ., data = your_data)** fits an initial model using all available predictor variables (denoted by .).
- **Step(model, direction = "backward")** initiates the backward elimination process using the step function from the stats package, with the direction argument set to "backward".

- **Summary(backward_model)** summarizes the results of the final model obtained through backward elimination.

Example

Program to perform backward elimination in R using the step function from the stats package:

```
library(stats)

# Assuming 'response_variable' is the dependent variable and 'predictor1', 'predictor2', etc. are the independent variables in your dataset
model <- lm(response_variable ~ predictor1 + predictor2 + predictor3, data = your_data)

# Perform backward elimination using the step function
backward_model <- step(model, direction = "backward")

# Display the summary of the backward elimination model
summary(backward_model)
```

Output

```
Call:
lm(formula = mpg ~ wt + qsec + am, data = mtcars)

Residuals:
```

Min	1Q	Median	3Q	Max
–3.4811	–1.5555	0.7257	1.4110	4.6610

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   9.6178     6.9596   1.382 0.177915
wt           -3.9165     0.7112  -5.507 6.95e-06 ***
qsec          1.2259     0.2887   4.247 0.000216 ***
am            2.9358     1.4109   2.081 0.046716 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.459 on 28 degrees of freedom
Multiple R-squared:  0.8497,    Adjusted R-squared:  0.8336
F-statistic: 52.75 on 3 and 28 DF,  p-value: 1.21e-11
```


11.8.3 Advantages of Linear Model Selection

- **Data Snooping:** Overfitting can occur if the model selection process is driven by the data. It may result in a model that performs well on the training data but poorly on new data.
- **Loss of Information:** Removing variables can lead to loss of information that might be relevant in a broader context or for future research.
- **Instability:** Model selection can be sensitive to the specific dataset, and different subsets of data can lead to different model choices. This can make the selection process less stable.

11.8.4 Disadvantages of Linear Model Selection

- **Data Snooping:** Overfitting can occur if the model selection process is driven by the data. It may result in a model that performs well on the training data but poorly on new data.
- **Loss of Information:** Removing variables can lead to loss of information that might be relevant in a broader context or for future research.
- **Instability:** Model selection can be sensitive to the specific dataset, and different subsets of data can lead to different model choices. This can make the selection process less stable.

EXERCISE

Short Questions

1. Define Regression.
2. What are the types of Regression in R?
3. Define the following terms.
   (a) Linear Regression
   (b) Multiple Regression
   (c) Logistic Regression
4. Write the syntax of Linear Regression.
5. Define predict() function.
6. Mention the advantages of Linear Regression.
7. Mention the disadvantages of Linear Regression.
8. Write the general mathematical equation for multiple regression.
9. What is the lm function in R?
10. Define input data.
11. What are the functions available in the leaps package?
12. Define leaps package.

13. What is best subset selection?
14. Define stepwise selection.
15. Write the syntax for Forward selection.
16. Define Backward elimination.
17. Mention any 2 advantages of Linear Model Selection.
18. Mention any 2 disadvantages of Linear Model Selection.

Long Questions

1. What is Regression? Explain the types of Regression in detail.
2. Write a short note on the following terms with example and syntax:
   (a) Linear Regression
   (b) Multiple Regression
   (c) Logistic Regression
3. Explain predict() function in detail.
4. What is Linear Regression Line? Explain its types.
5. Explain the advantages and disadvantages of Linear Regression.
6. Explain Multiple Regression in detail with example.
7. What0 is input data? Explain with an example.
8. Write a note on Logistic Regression.
9. Explain Linear Model Selection and also its functions briefly.
10. Define Best Subset Selection. Explain the steps to perform best subset selection.
11. Explain the stepwise selection in detail.
12. Explain the advantages and disadvantages of Linear Model Selection.

♣♦♣♦♣♦♣

CHAPTER 12

Advanced Graphics

12.1 Introduction

Advanced graphics typically refer to the creation of more complex and sophisticated visualizations that go beyond basic plots and charts. In the context of data visualization, advanced graphics involve the use of techniques and tools that allow for the creation of intricate and highly customizable visual representations of data. The lattice package provides a comprehensive system for visualizing multivariate data, including the ability to create plots conditioned on one or more variables. The ggplot2 package offers an elegant system for generating univariate and multivariate graphs based on the grammar of graphics. The graph types include probability plots, mosaic plots, and correlograms

Advanced graphics include:

- **Interactivity:** Advanced graphics often involve interactive elements that allow users to manipulate and explore data directly within the visualization.
- **Customization:** They offer extensive customization options, enabling users to adjust various aspects of the visualization, such as color schemes, shapes, sizes, and layouts.
- **Complex Data Structures:** They are capable of handling complex data structures and large datasets, allowing for the representation of multidimensional and multivariate data.
- **3D Visualization:** Some advanced graphics techniques include the creation of 3D plots and visualizations, which provide a more immersive and comprehensive view of data.
- **Dynamic Visualization:** They enable the creation of dynamic and animated graphics that can effectively convey changes and patterns in the data over time or in response to user interactions.

12.2 Packages

There are several packages specifically designed for creating advanced and sophisticated graphics. These packages provide extensive capabilities for data visualization and allow for the creation of interactive, dynamic, and highly customizable plots.

Some of the packages for advanced graphics in R include:

- **ggplot2:** ggplot2 is a powerful and widely used package for creating static and publication-quality graphics. It follows the grammar of graphics framework and allows for the creation of a wide range of plots with extensive customization options.
- **Plotly:** Plotly is an interactive and web-based visualization package that enables the creation of dynamic, interactive, and high-quality plots. It supports a variety of graph types and can produce web-based visualizations that can be easily shared.
- **Lattice:** Lattice is a package for creating Trellis graphics, which are particularly useful for conditioning plots, including scatter plots, bar plots, and line plots. It allows for the creation of complex multi-paneled displays.
- **ggvis:** ggvis is an interactive graphics package that integrates seamlessly with ggplot2. It enables the creation of interactive web-based graphics using a grammar of graphics framework, allowing for dynamic and responsive visualizations.
- **dygraphs:** dygraphs is a package specifically designed for time series data. It provides interactive time series charting capabilities and is particularly useful for visualizing and exploring temporal data.
- **rBokeh:** rBokeh is an R interface to the Bokeh visualization library in Python. It allows for the creation of interactive visualizations in R, providing a wide range of interactive plots, including bar plots, line plots, and scatter plots.
- **Shiny:** Shiny is not solely a graphics package, but a web application framework that allows the creation of interactive web applications directly from R. It can be used in conjunction with various plotting libraries to create dynamic and interactive dashboards and applications.

These packages provide R users with a diverse set of tools for creating advanced and interactive visualizations that are suitable for various types of data analysis and communication.

12.3 Customizing Plots

Customizing plots in R allows us to create visually appealing and informative graphics tailored to the specific needs. We can adjust various aspects of the plot, such as the title, axes, labels, colors, and annotations.

Basic Plot Customization

- Adjusting the plot title using the 'main' parameter.
- Customizing axis labels using the xlab and ylab parameters.
- Modifying axis limits using the xlim and ylim parameters.
- Changing axis ticks and labels using functions like 'axis' and 'at'.

Color Customization

- Setting colors for points, lines, or bars using the 'col' parameter.

- Creating color palettes with functions like 'rainbow', 'heat.colors', and 'colorRampPalette'.

Text Customization

- Modifying text properties such as font size, font family, and font style using the 'cex' and family parameters.
- Adding annotations and text labels using functions like text and 'mtext'.

Legend Customization

- Modifying legend labels and position using the legend function and x and y parameters.
- Changing legend titles and text properties using the 'title' and 'text.font' parameters.

Layout Customization

- Adjusting the layout of multiple plots using functions like 'par' and 'layout'.
- Creating multi-panel plots using packages like 'ggplot2' and 'gridExtra'.

Line and Marker Customization

- Modifying line types and widths using the 'lty' and 'lwd' parameters.
- Changing point shapes, sizes, and colors using the 'pch' and 'cex' parameters.

Background Customization

- Adjusting the background color using the 'bg' parameter.
- Adding grid lines using functions like 'abline' and grid.

When customizing plots, consider the requirements of your specific visualization and the best practices for conveying information effectively

12.3.1 Plotting Function

Function and Arguments	Output Plot
plot (x, y)	Scatterplot of x and y numeric vectors
plot (factor)	Barpot of the factor
plot (factor, y)	Boxplot of the numeric vector and the levels of the factor
plot (time-series)	Time series plot
plot (data_frame)	Correlation plot of all dataframe columns (more than two columns)
plot (date, y)	Plot a data-based vector
plot (function, lower, upper)	Plot of the function between the lower and maximum value specified

Here are some common customization options and their corresponding R code:

Changing Plot Title and Axis Labels

```
plot(x, y, main = "Custom Plot Title", xlab = "X Axis Label", ylab = "Y Axis Label")
```

Modifying Plot Colors and Line Types

```
plot(x, y, type = "l", col = "blue", lwd = 2, lty = 2)  # blue color, thicker line, dashed line type
```

Adding Points and Legends

```
plot(x, y, type = "l", col = "blue", lwd = 2, lty = 2)
points(x, y, col = "red", pch = 16)  # Add points with red color and solid circle shape
legend("topright", legend = c("Line", "Points"), col = c("blue", "red"), lty = c(2, 0), pch = c(NA, 16))
```

Adjusting Axes Limits and Ticks

```
plot(x, y, xlim = c(0, 10), ylim = c(0, 20), xaxs = "i", yaxs = "i")  # Set custom axis limits and remove the outer tick marks
```

Adding Text and Annotations

```
text(5, 15, "Sample Text", col = "green", cex = 1.2)  # Add text at coordinates (5, 15) with green color and larger size
abline(h = 10, col = "red", lty = 2)  # Add a horizontal line at y = 10 with red color and dashed line type
```

Customizing Plot Layout and Appearance

```
par(mfrow = c(2, 2))  # Divide the plotting area into a 2x2 grid
par(mar = c(5, 4, 4, 2) + 0.1)  # Set the margins of the plot
```

These are just a few examples of how you can customize your plots in R. You can further customize plots by exploring additional parameters and functions based on your specific visualization requirements.

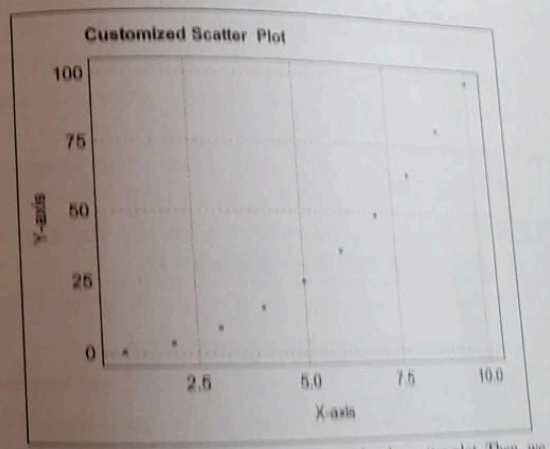
Example

```
library(ggplot2)

# Create sample data
x <- 1:10
y <- x^2

# Create a basic scatter plot
p <- ggplot(data = data.frame(x, y), aes(x = x, y = y)) +
```



```
  geom_point(color = "blue") +
  labs(title = "Customized Scatter Plot", x = "X Axis", y = "Y Axis")

# Customize the plot
p + theme_minimal() +
  theme(plot.title = element_text(color = "red", size = 16, face = "bold"),
        axis.title.x = element_text(color = "green", size = 12),
        axis.title.y = element_text(color = "purple", size = 12),
        axis.text = element_text(size = 10),
        panel.background = element_rect(fill = "lightyellow"),
        panel.grid.minor = element_blank(),
        panel.grid.major = element_line(color = "gray", linetype = "dashed"))

# Save the plot as a PNG file
ggsave("customized_plot.png", plot = p, width = 6, height = 4, dpi = 300)
```

Output



In this example, we first create sample data for a simple scatter plot. Then, we customize the plot using various parameters and options from the theme function. Finally, we save the plot as a PNG file. You can run this code in R after installing and loading the ggplot2 package.

12.4 Colors

Colors can be defined using various formats, including predefined color names, hexadecimal color codes, RGB values, and other color models. These color definitions can be used in plots, graphs, and other visualizations to add aesthetic appeal and convey additional information.

Here are some common ways to define colors in R:

Predefined Color Names: R provides a set of standard color names, such as "red," "blue," "green," and "purple," which can be used directly in plotting functions.

Hexadecimal Color Codes: Hexadecimal color codes represent colors using a combination of red, green, and blue (RGB) values in a hexadecimal format. For example, "#FF0000" represents the color red.

RGB Values: Colors can be defined using RGB values, which specify the intensity of red, green, and blue components on a scale of 0 to 255. For instance, the color red can be defined as RGB (255, 0, 0).

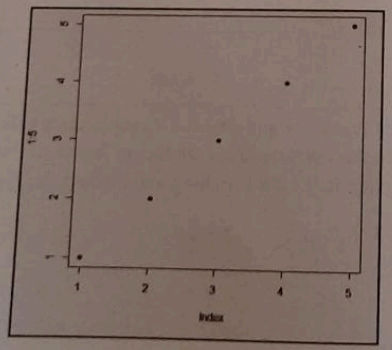
RGBA Values: Similar to RGB, the RGBA color model includes an additional alpha channel that represents the opacity or transparency of the color.

Example

Program to defining colors using different methods in R

```
# Using predefined color names
plot(1:5, col = "blue", pch = 19)
# Using hexadecimal color codes
plot(1:5, col = "#FF0000", pch = 19)
# Using RGB values
plot(1:5, col = rgb(255, 0, 0, maxColorValue = 255), pch = 19)
```

Output



By utilizing these color definitions, you can customize and enhance the visual appearance of your plots and graphics in R, making them more engaging and informative.

12.5 Customizing Traditional R plots

Customizing traditional R plots involves modifying various graphical parameters to make plots more visually appealing, informative, and suitable for the specific data and research objectives. Traditional R plots are typically created using base R graphics functions, and customization can enhance their appearance, layout, and interpretability. Here's the key aspects you can customize in traditional R plots:

Title and Labels

- **Main Title (main):** Add a title to describe the content or purpose of the plot.
- **X-axis Label (xlab) and Y-axis Label (ylab):** Label the axes to provide context for the data being displayed.

Colors and Line Types

- **Color (col):** Change the line or point color to differentiate multiple data series.
- **Line Type (lty):** Adjust the line type (solid, dashed, etc.) to distinguish lines.
- **Point Character (pch):** Modify the point character to change the shape of data points.

Line Width (lwd): Increase or decrease the line width to make lines more visible or subtle.

Axis Limits (xlim, ylim): Adjust the range of the x-axis and y-axis to focus on specific data intervals.

Legend (legend): Add a legend to identify data series or categories in the plot. Customize the legend's position and labels.

Grid Lines (grid): Add grid lines to improve readability and make it easier to estimate values.

Text Labels (text): Place text labels on the plot to provide additional information or annotations.

Background and Margins

- **Plot Margin (par(mar)):** Adjust the margin size around the plot area to make room for titles and labels.
- **Background Color (bg):** Change the background color of the plot area.

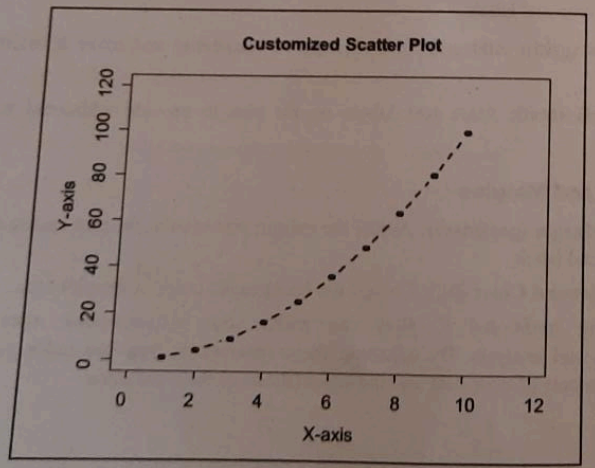
Customizing traditional R plots can make your visualizations more effective for communication and analysis. By adjusting these parameters, you can tailor your plots to the specific requirements of your data and the expectations of your audience.

Example

Program to demonstrate how to customize a traditional R plot:

```
# Create example data
x <- 1:10
y <- x^2
# Create a scatter plot with customized parameters
plot(x, y,
     type = "b",          # 'b' for both points and lines
     col = "blue",        # Set point color to blue
     pch = 19,            # Set point shape (solid circle)
     lty = 2,             # Set line type (dashed)
     lwd = 2,             # Set line width
     xlab = "X-axis",     # X-axis label
     ylab = "Y-axis",     # Y-axis label
     main = "Customized Scatter Plot",  # Main title
     xlim = c(0, 12),     # Set X-axis limits
     ylim = c(0, 120),    # Set Y-axis limits
     col.axis = "green",  # Set axis label color
     col.lab = "purple"   # Set axis label text color
)
```

Output



13.6 Specialized Text Notation

Specialized text notation in R refers to using specific symbols, formatting, or characters to represent special characters, mathematical equations, or other notations within text strings. This is often used when creating data labels, titles, axis labels, or annotations in R plots or when writing text for mathematical or scientific documents. R supports specialized text notation using various mechanisms. Here are some common examples:

1. Greek Letters

Greek letters can be included in text strings using the expression function. For example, expression(alpha) represents the Greek letter alpha.

 # Example of using Greek letter alpha in a plot title

```
plot(1:10, main = expression("Scatterplot with " ~ alpha ~ " Symbol"))
```

2. Subscripts and Superscripts

Subscripts and superscripts can be added to text using the substitute and paste functions.

 # Example of using subscripts and superscripts in a plot title

```
plot(1:10, main = substitute(paste("H" ^ 2, "O"), list()))
```

3. Special Characters

Special characters, such as degree symbol (°) or copyright symbol (©), can be included using their Unicode code points.

 # Example of using special characters in a plot title

```
plot(1:10, main = "Temperature (C) vs. Time")
```

4. Mathematical Equations

Mathematical equations can be included within text using LaTeX notation. For example, $\alpha + \beta = \gamma$ represents a mathematical equation.

 # Example of including a mathematical equation in a plot title

```
plot(1:10, main = "Linear Regression Model: $y = \\alpha + \\beta x + \\epsilon$")
```

5. Scientific Notation

Scientific notation can be used to format numbers with exponents.

 # Example of scientific notation in axis labels

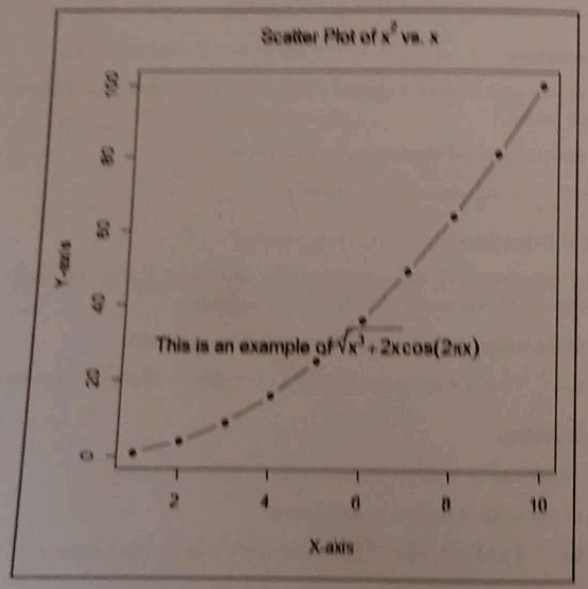
```
plot(1:10, xlab = "Time (s)", ylab = "Distance (m)", main = "Experimental Data: $2.5 \\times 10^{-3}$ kg")
```

By using specialized text notations, you can make your R plots and documents more informative and mathematically accurate, which is especially important in scientific and technical fields.

Example

```
# Create example data
x <- 1:10
y <- x^2
# Create a scatter plot
plot(x, y, type = "b", pch = 19, col = "blue", xlab = "X-axis", ylab = "Y-axis",
main = expression(paste("Scatter Plot of ", x^2, " vs. ", x)))
# Add a mathematical expression to the plot
text(5, 30, expression(paste("This is an example of ", sqrt(x^3 + 2*x) * cos(2*pi*x))), cex =
1.2, col = "red")
```

Output



13.7 Specialized Label Notation

Specialized label notation in format and customize the appearance of text labels in your plots. Specialized label notation allows you to add mathematical symbols, Greek letters, subscripts, superscripts, and other formatting to your text annotations. The expression and bquote functions are commonly used for creating specialized labels in R plots.

The **expression function** is used to create a label with mathematical notation. The label "f(x) = α + β * x^2" includes Greek letters (α, β) and superscripts (^).

```
# Create a plot with a specialized label
plot(1:5, 1:5, type = "n", xlab = "", ylab = "")
text(3, 3, expression(f(x) == alpha + beta * x^2))
```

The **bquote function** allows you to create dynamic labels that include the values of variables, in this case, x_val and y_val.

```
#Create a plot with a dynamic label using bquote
x_val <- 2
y_val <- 8
plot(1:10, 1:10, type = "n", xlab = "", ylab = "")
text(5, 5, bquote("A point at (" * .(x_val) * ", " * .(y_val) * ")"))
```

The **substitute function** enables partial label substitution, making it easy to include variable names in the labels.

```
# Create a plot with a partially substituted label using substitute
variable <- "temperature"
plot(1:5, 1:5, type = "n", xlab = "", ylab = "")
text(3, 3, substitute("The " * variable * " is increasing", list(variable = variable)))
```

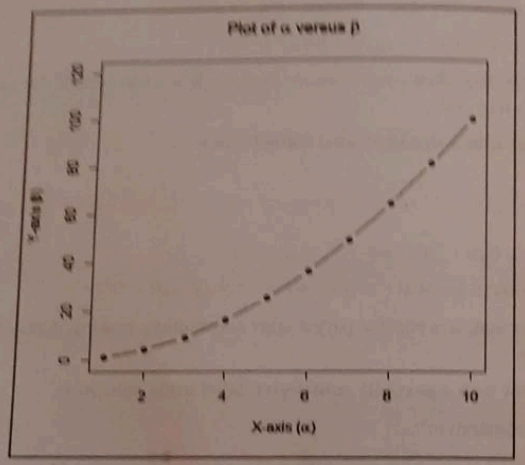
Example

Program to demonstrates how to use specialized label notation to add mathematical expressions to an R plot:

```
# Create example data
x <- 1:10
y <- x^2
# Create a scatter plot with specialized label notation
plot(x, y,
     type = "b",
     col = "blue",
```

```
  pch = 19,
  xlab = expression(paste("X-axis (", alpha, ")")),  # Math notation for the X-axis label
  ylab = expression(paste("Y-axis (", beta, ")")),  # Math notation for the Y-axis label
  main = expression(paste("Plot of ", alpha, " versus ", beta)),  # Math notation for the main
title
  ylim = c(0, 120)
)
```

Output



12.8 Plotting Region

The "plotting region" specifies the size and location of the plot within a graphical device such as a window or file. This is typically done using functions like par, which sets various graphical parameters, and 'plot', which creates the actual plot within the specified region. Here's how you can define and use a plotting region in R:

Defining a Plotting Region with Par

The **'par'** function to set graphical parameters, including the size and location of the plotting region.

Common parameters for defining the plotting region include:

- **mfrow or mfcol:** Specifies the number of rows and columns for multiple plots in a grid.
- **mar:** Sets the margins around the plotting region.

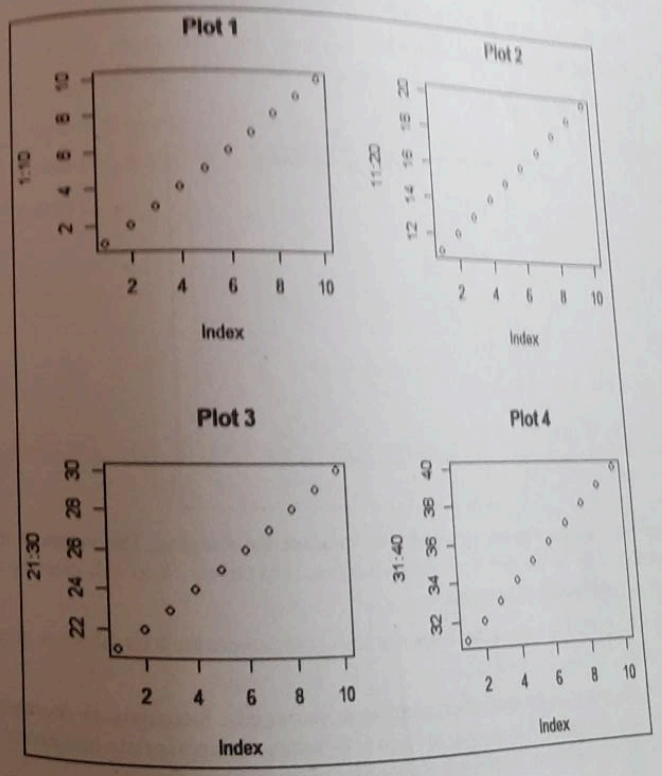
- **oma:** Specifies the outer margins of the entire plot.
- **plt:** Defines the location of the plotting region within the graphical device.

Example

Define a 2 × 2 grid for multiple plots

```
par(mfrow = c(2, 2))
# Create individual plots within the plotting region
plot(1:10, main = "Plot 1")
plot(11:20, main = "Plot 2")
plot(21:30, main = "Plot 3")
plot(31:40, main = "Plot 4")
```

Output



13.9 Plotting Margins

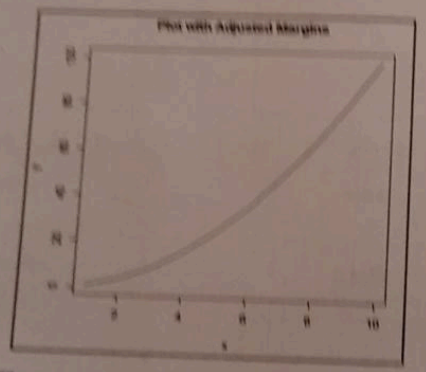
Plotting margins using the 'par' function: The par function is used to set various graphical parameters for plotting. To change the margin sizes, you can modify the 'mar' parameter, which represents the number of lines of margin to be specified on the four sides of the plot (bottom, left, top, right).

Example

Program to adjust the plotting margins in R

```
# Create example data
x <- 1:10
y <- x^2

# Set the plotting margins using the par function
par(mar = c(5, 4, 4, 2) + 0.1)  # Adjust the margins for the plot

# Create the plot with adjusted margins
plot(x, y, type = "l", col = "blue", main = "Plot with Adjusted Margins")
```

Output



The 'par' function: The mar parameter is set to adjust the margins. The values in the mar vector represent the bottom, left, top, and right margins, respectively. You can customize these values to suit your specific requirements.

The + 0.1 is used to increase the margin size slightly to prevent the axis labels or titles from being cut off.

Adjust the values in the mar vector according to your needs. Increasing or decreasing the values will alter the width of the margins in the plot. By setting the appropriate margin sizes, you can control the space between the plot area and the edge of the graphics device.

13.10 Point-and-click

A point-and-click coordinate interaction in an R plot to use the locator function. The locator function in R to interactively click on a plot, and it records the coordinates of the points where you click. The locator function interactively selects points or coordinates by clicking on a plot. This can be particularly useful for obtaining specific data points or coordinates from a graph. It is useful for identifying specific data points or regions of interest.

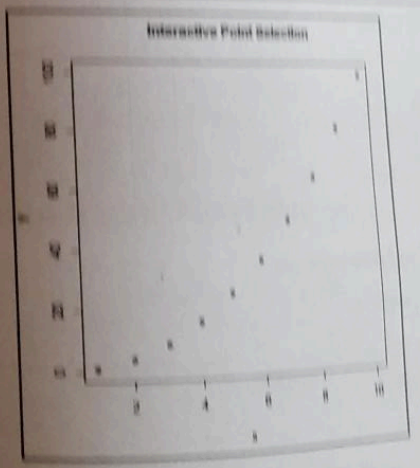
Example

Program to use the locator function to interactively click on a plot and retrieve the coordinates:

```
# Create an example scatterplot
x <- 1:10
y <- x^2
plot(x, y, type = "p", col = "blue", pch = 16, main = "Interactive Point Selection")

# Use locator to interactively click on the plot
points <- locator(1)

# Print the coordinates of the clicked points
cat("Clicked Coordinates:\n")
print(points)
```

Output

```
$x [1] 5.237939
$y [1] 69.9835
```



In this example:

- The scatterplot of x and y values.
- The locator function is used to interactively click on the plot. When you click on the plot, it records the coordinates of the point you clicked on.
- The clicked coordinates are stored in the points variable.
- We print the coordinates to the console.
- By clicking on multiple points on the plot, the locator function will record each set of coordinates. This interactive feature is useful for exploring data and identifying specific data points on a graph.

12.11 3D Scatter Plots

3D scatter plots in R to visualize data points in a three-dimensional space, typically with points representing data points in a 3D space. the scatterplot3d function from the scatterplot3d package to create 3D scatter plots. a 3D scatter plot, the position of each data point is determined by its values along the three dimensions, and the points are displayed in the three-dimensional coordinate system.

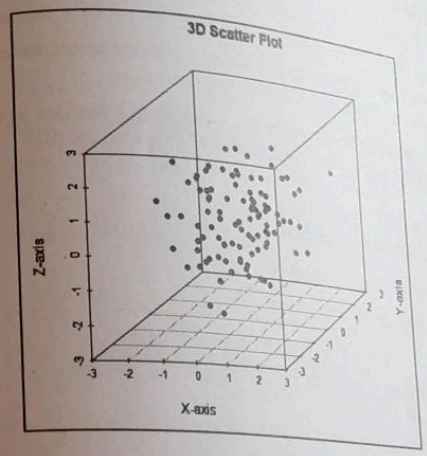
3D scatter plots using various libraries, such as scatterplot3d or rgl, to visualize data in three dimensions. Here's an example of how to create a 3D scatter plot using the scatterplot3d library:

Example

Program to create 3D Scatterplot3d

```
# Install and load the necessary library
install.packages("scatterplot3d")
library(scatterplot3d)
# Create example data
x <- rnorm(100)
y <- rnorm(100)
z <- rnorm(100)
# Create a 3D scatter plot
scatterplot3d(x, y, z, color = "blue",
              main = "3D Scatter Plot",
              xlab = "X-axis",
              ylab = "Y-axis",
              zlab = "Z-axis",
              pch = 19
)
```


Output



In this example:

- The scatterplot3d function from the scatterplot3d library is used to create a 3D scatter plot.
- The variables x, y, and z represent the coordinates in the three dimensions.
- The color parameter is used to set the color of the points in the scatter plot.
- Other parameters, such as main for the title, and xlab, ylab, and zlab for the labels of the axes, are customized to provide additional context to the plot.
- The pch argument is used to specify the type of points in the plot.

You can install the scatterplot3d package if you haven't already and use it to create 3D scatter plots to visualize data in three dimensions. Adjust the example according to your specific data and visualization requirements.

12.12 Plotting in Higher Dimensions

Plotting in higher dimensions is a challenging task, as visualizing data beyond three dimensions directly on a 2D plot is not feasible. However, there are several techniques in R that can help visualize and analyze high-dimensional data:

Scatterplot Matrices: Use the pairs function to create a matrix of scatterplots, where each variable is plotted against every other variable. This provides an overview of pairwise relationships in the data.

Parallel Coordinate Plots: Represent each observation as a line that traverses across a set of parallel axes, with each axis representing a different variable. This allows for the visualization of multivariate data points.

Heatmaps: Use the heatmap or geom_tile function in R to create heatmaps that display numerical data in a matrix format, where colors represent the values of the data points.

Interactive 3D Visualization: Packages like rgl and plotly enable the creation of interactive 3D plots, allowing for the exploration of data in three dimensions.

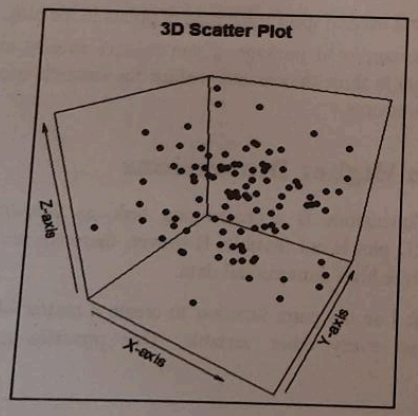
Hierarchical Clustering Dendrogram: Use the heatmap.2 function in the gplots package to create a dendrogram that displays hierarchical relationships within the data.

Example

Program to demonstrate a 3D scatter plot

```
# Install and load the required package
# install.packages("plot3D")
library(plot3D)
# Create example data
x <- rnorm(100)
y <- rnorm(100)
z <- rnorm(100)
# Create a 3D scatter plot
scatter3D(x, y, z, colvar = z, col = "blue", pch = 16, theta = 30, phi = 30,
xlab = "X-axis", ylab = "Y-axis", zlab = "Z-axis", main = "3D Scatter Plot")
```

Output

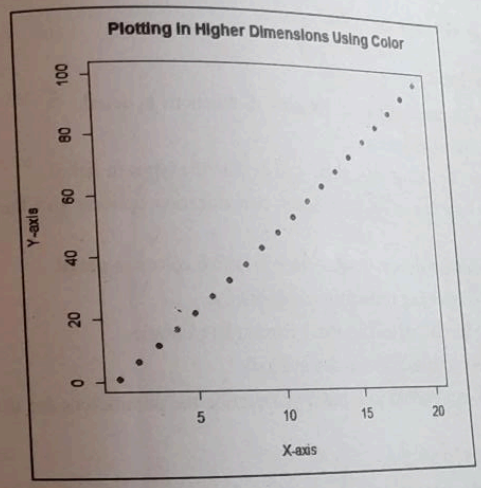


Example

Program to demonstrates plotting in higher dimensions using color in R

```
# Create example data
x <- 1:20
y <- seq(1, 100, length.out = 20)
z <- seq(10, 200, length.out = 20)
color <- z # Assign color based on the third dimension
# Create a scatter plot with color representing the third dimension
plot(x, y, col = color, pch = 19, main = "Plotting in Higher Dimensions Using Color",
  xlab = "X-axis", ylab = "Y-axis")
```

Output



EXERCISE

Short Questions

1. Define Advanced Graphics.
2. What are the functions included in Advanced Graphics?
3. Define Packages.
4. Mention some packages for Advanced Graphics in R.
5. Define Cus tomizing plots.

6. Mention the types of Customizing plots.
7. What are the plotting functions?
8. Mention some common Customization options and their corresponding R code.
9. Define colors.
10. Mention different ways to fine colors in R.
11. List out the key aspects to customize in traditional R plots.
12. Define Specialized text notation with examples.
13. Why Specialized label notation is used?
14. What is meant by Plotting Regions?
15. Mention the common parameters used for defining the plotting region.
16. Define plotting margins.
17. What is point-and-click?
18. What 3D scatter plots?
19. Define plotting in higher dimensions.

Long Questions

1. Explain Advanced Graphics. Describe its functions in detail.
2. Define Packages in detail.
3. What are the Customizing plots? And explain the types in detail.
4. Give a brief description on common customization options and their corresponding R code.
5. Define colors and some common ways to define colors in detail.
6. Explain customizing traditional R plots briefly.
7. What are Specialized text notation? Explain its examples.
8. Explain Specialized label notation in detail.
9. Define plotting regions with example. Describe the parameters for defining the plotting region.
10. Explain plotting margins.
11. Write a program to adjust the plotting margins in R
12. Explain Point-and-click with an example
13. Define 3D scatter plots with an example program.
14. Explain plotting in higher dimensions in detail.
15. Write a program to demonstrate plotting in higher dimensions using color in R.

♣♦♣♦♣♦♣

Lab Manual

Part A

1. Write a program for different types of data structures in R

```
# Creating a Numeric Vector
n <- c(1, 2, 3, 4, 5)
print("Numeric Vector:")
print(n)
# Creating a Character Vector
cv <- c("apple", "banana", "cherry")
print("Character Vector:")
print(cv)

# Creating a List
lst <- list(1, "apple", TRUE)
print("List:")
print(lst)
# Creating a Matrix
m <- matrix(1:6, nrow = 2, ncol = 3)
print("Matrix:")
print(m)
# Creating a Data Frame
df <- data.frame(Name = c("Alice", "Bob", "Charlie"),
                 Age = c(25, 30, 22))
print(" Data Frame:")
print(df)
```