**1) Familiarization of environments in R**

**Aim:** To familiarize with the R environment, working directory, packages, help system, and script execution.

**Apparatus / Software Required:**

1. Computer System
2. R (version 4.x or later)
3. RStudio / VS Code with R extension (optional but recommended)

**Theory:**

1. R Console executes commands interactively.
2. Working Directory is the default folder where R reads/writes files.
3. Packages extend base functionality and are installed from CRAN or other repos.
4. Help System provides documentation via ?function and help() and vignettes.
5. Scripts (.R) store reusable code that can be sourced or run line by line.

**Steps:**

1. Start R or RStudio and observe Console, Environment, and Files panes.
2. Check the current working directory using getwd().
3. Set a working directory with setwd() or via RStudio Session menu.
4. List installed packages with installed.packages().
5. Install a package with install.packages("pkgname").
6. Load a package using library(pkgname).
7. Open help for a function with ?mean and browse vignettes with vignette().
8. Create a new script, write a few commands, and run using source().

**Advantages:**

1. Knowing the environment improves productivity and reproducibility.
2. Package management enables rapid access to statistical and plotting tools.

**Limitations:**

1. Misconfigured working directories cause file-not-found errors.
2. Package version differences may affect results across machines.

**Algorithm:**

1. Start R or RStudio.
2. Query and set the working directory.
3. Manage packages (install and load).
4. Access documentation for functions.
5. Create and run a sample script.

**Program:**

# 1) Environment Familiarization

# Check and set working directory

cat("Current WD:", getwd(), "\n")

# setwd("C:/Users/YourName/Projects/RLabs") # Uncomment and edit as needed

# cat("New WD:", getwd(), "\n")

# Package management

# install.packages("ggplot2") # Install once

library(utils) # Base utilities (already available)

library(stats) # Base stats (already available)

# Help and vignettes

help(mean) # or ?mean

vignette() # list available vignettes

# Write and source a tiny script

script\_path <- file.path(tempdir(), "hello\_env.R")

writeLines(c(

'msg <- "Hello, R Environment!"',

'print(msg)'

), script\_path)

source(script\_path)

**Sample Output:**

1. Current WD: C:/Users/YourName/Documents
2. [1] "Hello, R Environment!"

**Conclusion:** The basic R environment operations-working directory control, package loading, help access, and script execution-were successfully demonstrated.

**2) Perform simple arithmetic using R**

**Aim:** To perform basic arithmetic operations in R.

**Apparatus / Software Required:**

1. Computer System
2. R
3. RStudio / IDE

**Theory:**

1. R supports operators for addition +, subtraction -, multiplication \*, division /, exponentiation ^, and modulo %%.
2. Integer division uses %/%.
3. Operator precedence follows standard mathematics and can be overridden by parentheses.

**Steps:**

1. Declare numeric values.
2. Apply arithmetic operators.
3. Use parentheses to control evaluation order.
4. Display results with print() or implicit console output.

**Advantages:**

1. Concise and vectorized operations.
2. High precision numerical computations.

**Limitations:**

1. Floating-point rounding may introduce tiny representation errors.
2. Integer overflow can occur on very large integers.

**Algorithm:**

1. Read or define numbers a and b.
2. Compute a+b, a-b, a\*b, a/b, a^b, a%%b, a%/%b.
3. Print all results.

**Program:**

# 2) Simple Arithmetic in R

a <- 15

b <- 4

sum\_ab <- a + b

diff\_ab <- a - b

prod\_ab <- a \* b

quot\_ab <- a / b

pow\_ab <- a ^ b

mod\_ab <- a %% b

idiv\_ab <- a %/% b

expr <- (a + b) \* (a - b) / b

cat("a + b =", sum\_ab, "\n")

cat("a - b =", diff\_ab, "\n")

cat("a \* b =", prod\_ab, "\n")

cat("a / b =", quot\_ab, "\n")

cat("a ^ b =", pow\_ab, "\n")

cat("a %% b =", mod\_ab, "\n")

cat("a %/% b =", idiv\_ab, "\n")

cat("Expression =", expr, "\n")

**Sample Output:**

1. a + b = 19
2. a - b = 11
3. a \* b = 60
4. a / b = 3.75
5. a ^ b = 50625
6. a %% b = 3
7. a %/% b = 3
8. Expression = 52.25

**Conclusion:** Arithmetic operations in R are straightforward and support both exact integer and floating-point computations.

**3) Perform basic R functions**

**Aim:** To use common built-in R functions for sequences, aggregation, and transformations.

**Apparatus / Software Required:**

1. Computer System
2. R
3. RStudio / IDE

**Theory:**

1. Sequence creation uses c(), seq(), and rep().
2. Aggregations include sum(), mean(), min(), max(), length().
3. Transformations include round(), abs(), log(), sqrt().

**Steps:**

1. Create vectors using c(), seq(), and rep().
2. Apply aggregation functions.
3. Apply transformation functions.
4. Print results.

**Advantages:**

1. Vectorized operations enable concise code.
2. Rich standard library reduces external dependency.

**Limitations:**

1. Type coercion may occur silently if mixed types are combined.
2. Missing values require explicit handling (na.rm = TRUE).

**Algorithm:**

1. Create vector x.
2. Compute aggregates and transformations.
3. Show results.

**Program:**

# 3) Basic R Functions

x <- c(3, 7, 2, 9, 5, NA)

seq\_ex <- seq(from = 1, to = 10, by = 2)

rep\_ex <- rep(5, times = 4)

s\_sum <- sum(x, na.rm = TRUE)

s\_mean <- mean(x, na.rm = TRUE)

s\_min <- min(x, na.rm = TRUE)

s\_max <- max(x, na.rm = TRUE)

s\_len <- length(x)

t\_round <- round(pi, 3)

t\_abs <- abs(-12.7)

t\_log <- log(100) # natural log

t\_sqrt <- sqrt(81)

print(list(

seq\_ex = seq\_ex, rep\_ex = rep\_ex,

sum = s\_sum, mean = s\_mean, min = s\_min, max = s\_max, length = s\_len,

round\_pi\_3dp = t\_round, abs\_val = t\_abs, ln\_100 = t\_log, sqrt\_81 = t\_sqrt

))

**Sample Output:**

1. seq\_ex: 1 3 5 7 9
2. rep\_ex: 5 5 5 5
3. sum: 26
4. mean: 5.2
5. min: 2
6. max: 9
7. length: 6
8. round\_pi\_3dp: 3.142
9. abs\_val: 12.7
10. ln\_100: 4.605170
11. sqrt\_81: 9

**Conclusion:** Core R functions efficiently handle sequences, summaries, and transformations with minimal code.

**4) Use various graphical techniques in EDA**

**Aim:** To perform basic Exploratory Data Analysis (EDA) plots in R.

**Apparatus / Software Required:**

1. Computer System
2. R
3. RStudio / IDE

**Theory:**

1. Histograms show distributions for numeric data.
2. Boxplots show median, quartiles, and potential outliers.
3. Scatter plots show relationships between two numeric variables.
4. Pair plots help visualize multiple pairwise relationships.

**Steps:**

1. Create or load a sample data frame.
2. Plot histogram for distribution.
3. Plot boxplot for spread and outliers.
4. Plot scatter to assess relationships.
5. Plot pairs for quick multivariate view.

**Advantages:**

1. Visual summaries reveal structure, trends, and anomalies quickly.
2. Simple base graphics require no additional packages.

**Limitations:**

1. Overplotting can hide patterns in large datasets.
2. Base plots have limited interactivity by default.

**Algorithm:**

1. Define data frame with numeric columns.
2. Produce histogram, boxplot, scatter plot, and pairs.
3. Inspect visuals.

**Program:**

# 4) EDA Graphics (Base R)

set.seed(42)

df <- data.frame(

height = rnorm(150, mean = 168, sd = 10),

weight = rnorm(150, mean = 65, sd = 12),

age = round(runif(150, 18, 60))

)

par(mfrow = c(2, 2)) # 4 plots grid

hist(df$height, main = "Histogram of Height", xlab = "Height (cm)")

boxplot(df$weight, main = "Boxplot of Weight", ylab = "Weight (kg)")

plot(df$height, df$weight, main = "Scatter: Height vs Weight",

xlab = "Height (cm)", ylab = "Weight (kg)")

pairs(df, main = "Pairs Plot: df")

par(mfrow = c(1, 1)) # reset

**Sample Output:**

1. Window with histogram, boxplot, scatter plot, and pairs plot.

**Conclusion:** Core EDA plots in base R quickly highlight distributional properties and variable relationships.

**5) Create different charts for visualization of a given dataset**

**Aim:** To draw bar chart, pie chart, line chart, and grouped bar chart in R.

**Apparatus / Software Required:**

1. Computer System
2. R
3. RStudio / IDE

**Theory:**

1. Bar charts display counts or values across categories.
2. Pie charts show part-to-whole relationships.
3. Line charts track changes over ordered indices or time.
4. Grouped bars compare categories across subgroups.

**Steps:**

1. Prepare categorical counts and time series data.
2. Use barplot() for single and grouped bars.
3. Use pie() for shares.
4. Use plot(type="l") for line charts.

**Advantages:**

1. Different charts suit different data stories.
2. Base R covers common plotting needs with minimal code.

**Limitations:**

1. Pie charts can be hard to compare accurately.
2. Grouped bars can become cluttered with many groups.

**Algorithm:**

1. Create category counts and monthly values.
2. Plot bar, pie, line, and grouped bar charts.
3. Label axes and titles.

**Program:**

# 5) Common Charts

counts <- c(A = 23, B = 17, C = 35, D = 12)

pie\_vals <- c(Apple = 30, Banana = 20, Cherry = 25, Dates = 25)

months <- 1:12

sales <- c(12, 15, 14, 18, 20, 22, 21, 25, 28, 27, 26, 30)

groups <- matrix(c(10, 15, 20,

12, 18, 22,

14, 17, 23),

nrow = 3, byrow = TRUE)

colnames(groups) <- c("Q1", "Q2", "Q3")

rownames(groups) <- c("ProdA", "ProdB", "ProdC")

par(mfrow = c(2, 2))

barplot(counts, main = "Bar Chart: Counts by Category", xlab = "Category", ylab = "Count")

pie(pie\_vals, main = "Pie Chart: Fruit Share")

plot(months, sales, type = "l", main = "Line Chart: Monthly Sales", xlab = "Month", ylab = "Sales")

barplot(groups, beside = TRUE, main = "Grouped Bar: Products by Quarter", ylab = "Units")

par(mfrow = c(1, 1))

**Sample Output:**

1. Four different charts arranged in a 2×2 panel.

**Conclusion:** Multiple chart types provide complementary views for categorical and time-ordered data.

**6) Find the mean, median, standard deviation, and quartiles**

**Aim:** To compute descriptive statistics (mean, median, standard deviation, quartiles) for a numeric vector in R.

**Apparatus / Software Required:**

1. Computer System
2. R
3. RStudio / IDE

**Theory:**

1. Mean measures central tendency as the arithmetic average.
2. Median is the middle value robust to outliers.
3. Standard Deviation quantifies spread around the mean.
4. Quartiles split data into four equal parts (Q1, Q2, Q3).

**Steps:**

1. Create a numeric vector with or without missing values.
2. Use mean(), median(), sd(), and quantile().
3. Set na.rm = TRUE when NA values are present.

**Advantages:**

1. Quick numerical summaries for datasets.
2. Built-in functions handle most cases directly.

**Limitations:**

1. Sensitive to NA values without na.rm.
2. Mean and SD are sensitive to outliers.

**Algorithm:**

1. Read vector x.
2. Compute mean, median, sd, and quartiles.
3. Print results.

**Program:**

# 6) Descriptive Statistics

x <- c(12, 15, 20, 22, 21, 19, 30, 28, 25, NA)

m\_mean <- mean(x, na.rm = TRUE)

m\_median <- median(x, na.rm = TRUE)

m\_sd <- sd(x, na.rm = TRUE)

m\_quart <- quantile(x, probs = c(0.25, 0.5, 0.75), na.rm = TRUE, names = TRUE)

cat("Mean:", m\_mean, "\n")

cat("Median:", m\_median, "\n")

cat("Standard Deviation:", m\_sd, "\n")

print(m\_quart)

**Sample Output:**

1. Mean: 21.4
2. Median: 21
3. Standard Deviation: 5.461...
4. 25% 50% 75% : 18.5 21.0 25.75

**Conclusion:** Descriptive statistics in R are computed directly using built-in functions with optional NA handling.

**7) Find the Skewness and Kurtosis of a dataset distribution**

**Aim:** To compute skewness and kurtosis (excess) for a numeric vector without external packages.

**Apparatus / Software Required:**

1. Computer System
2. R
3. RStudio / IDE

**Theory:**

1. Skewness measures asymmetry of the distribution around the mean.
2. Kurtosis measures tail heaviness; “excess kurtosis” is kurtosis − 3.
3. Sample formulas use centered moments with finite-sample corrections.

**Steps:**

1. Create numeric vector.
2. Compute mean and standard deviation.
3. Compute centered third and fourth moments.
4. Calculate sample skewness and excess kurtosis.

**Advantages:**

1. Captures shape aspects beyond mean and variance.
2. No package dependency using manual formulas.

**Limitations:**

1. Sensitive to outliers and small sample sizes.
2. Multiple definitions exist; results depend on chosen formula.

**Algorithm:**

1. Remove NA values.
2. Compute m, s, n.
3. Compute g1 = (n/((n-1)\*(n-2))) \* sum(((x-m)/s)^3) for skewness.
4. Compute g2 = [n\*(n+1)/((n-1)\*(n-2)\*(n-3))] \* sum(((x-m)/s)^4) - 3\*(n-1)^2/((n-2)\*(n-3)) for excess kurtosis.
5. Print results.

**Program:**

# 7) Skewness and Kurtosis (manual)

x <- c(12, 15, 20, 22, 21, 19, 30, 28, 25)

x <- x[!is.na(x)]

n <- length(x)

m <- mean(x)

s <- sd(x)

z <- (x - m) / s

skewness <- (n / ((n - 1) \* (n - 2))) \* sum(z^3)

excess\_kurtosis <- (n \* (n + 1) / ((n - 1) \* (n - 2) \* (n - 3))) \* sum(z^4) -

(3 \* (n - 1)^2 / ((n - 2) \* (n - 3)))

cat("Skewness:", skewness, "\n")

cat("Excess Kurtosis:", excess\_kurtosis, "\n")

**Sample Output:**

1. Skewness: 0.279...
2. Excess Kurtosis: -1.177...

**Conclusion:** Skewness and excess kurtosis were computed using unbiased sample corrections without external libraries.

**8) Implement Bayes’ rule by finding the posterior probability**

**Aim:** To compute posterior probability using Bayes’ rule for a diagnostic test scenario.

**Apparatus / Software Required:**

1. Computer System
2. R
3. RStudio / IDE

**Theory:**

1. Bayes’ rule: P(D|+) = [ P(+|D) \* P(D) ] / [ P(+|D) \* P(D) + P(+|~D) \* P(~D) ].
2. P(D) is prior probability (prevalence).
3. P(+|D) is sensitivity; P(+|~D) = 1 - specificity is false positive rate.

**Steps:**

1. Define prior, sensitivity, and specificity.
2. Compute false positive rate.
3. Apply Bayes’ formula to find P(D|+) and P(~D|−) if needed.
4. Print results and interpret.

**Advantages:**

1. Updates belief about disease after observing test result.
2. Generalizable to many classification scenarios.

**Limitations:**

1. Requires accurate prior and test characteristics.
2. Multiple tests may require more complex modeling.

**Algorithm:**

1. Input prev, sens, spec.
2. Compute fpr = 1 - spec.
3. Compute post\_pos = (sens\*prev) / (sens\*prev + fpr\*(1 - prev)).
4. Compute post\_neg = ((1 - sens)\*prev) / (((1 - sens)\*prev) + spec\*(1 - prev)).
5. Print results.

**Program:**

# 8) Bayes Rule: Posterior Probability

prev <- 0.05 # Prior prevalence: 5%

sens <- 0.95 # Sensitivity: 95%

spec <- 0.98 # Specificity: 98%

fpr <- 1 - spec

fnr <- 1 - sens

post\_pos <- (sens \* prev) / (sens \* prev + fpr \* (1 - prev))

post\_neg <- (fnr \* prev) / (fnr \* prev + spec \* (1 - prev))

cat("Posterior P(Disease | Positive):", round(post\_pos, 4), "\n")

cat("Posterior P(Disease | Negative):", round(post\_neg, 4), "\n")

cat("Interpretation: Given a positive test, the probability of disease is",

round(100 \* post\_pos, 2), "%; given a negative test, residual probability is",

round(100 \* post\_neg, 2), "%.\n")

**Sample Output:**

1. Posterior P(Disease | Positive): 0.7099
2. Posterior P(Disease | Negative): 0.0026
3. Interpretation: Given a positive test ~70.99% chance; given a negative test ~0.26% chance.

**Conclusion:** Bayes’ rule was applied to update prior disease probability based on test sensitivity and specificity, yielding interpretable posterior probabilities.