**Practical 6 A**

**Aim :** Perform one-way ANOVA to compare means across multiple groups.

**Program :** import numpy as np

from scipy.stats import f\_oneway

# Create sample data for three groups

np.random.seed(42)

group1 = np.random.normal(loc=30, scale=10, size=50)

group2 = np.random.normal(loc=40, scale=15, size=50)

group3 = np.random.normal(loc=35, scale=12, size=50)

# Perform one-way ANOVA

statistic, p\_value = f\_oneway(group1, group2, group3)

# Print the results

print("One-way ANOVA:")

print(f"Statistic: {statistic}")

print(f"P-value: {p\_value}")

# Check for significance level (e.g., alpha=0.05)

alpha = 0.05

if p\_value < alpha:

print("Reject the null hypothesis. There are significant differences between group means.")

else:

print("Fail to reject the null hypothesis. There are no significant differences between group means.")

**Practical 6 B**

**Aim :** Conduct post-hoc tests to identify significant differences between group means.

**Program :** import pandas as pd

import pingouin as pg

import seaborn as sns

import matplotlib.pyplot as plt

# Replace this with your actual data

data = {

'group': ['A', 'A', 'B', 'B', 'C', 'C', 'D', 'D'],

'value': [10, 12, 15, 18, 9, 11, 16, 20]

}

df = pd.DataFrame(data)

# Performing ANOVA

from statsmodels.formula.api import ols

from statsmodels.stats.anova import anova\_lm

model = ols('value ~ group', data=df).fit()

anova\_table = anova\_lm(model)

# Performing post-hoc Tukey-Kramer test

mc = pg.pairwise\_tukey(data=df, dv='value', between='group')

# Displaying the summary of the Tukey-Kramer test

print(mc)

# Create a bar plot to visualize group means

sns.barplot(x='group', y='value', data=df, palette="viridis")

plt.show()

**Practical 7 A**

**Aim :** Create meaningful visualizations using data visualization tools.

**Program :** import matplotlib.pyplot as plt

import numpy as np

# Generate example data

np.random.seed(42)

x = np.random.rand(50)

y = 2 \* x + 1 + 0.1 \* np.random.randn(50)

# Create a scatter plot

plt.scatter(x, y, label='Data Points', color='blue', alpha=0.7)

# Add title and labels

plt.title('Scatter Plot Example')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

# Add a regression line

coefficients = np.polyfit(x, y, 1)

regression\_line = np.poly1d(coefficients)

plt.plot(x, regression\_line(x), color='red', label='Regression Line')

# Show legend

plt.legend()

# Show the plot

plt.show()

**Practical 7 B**

**Aim :** Combine multiple visualizations to tell a compelling data story.

**Program :** import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Generate example data

np.random.seed(42)

data = pd.DataFrame({

'X': np.random.rand(50),

'Y': 2 \* np.random.rand(50),

'Category': np.random.choice(['A', 'B'], size=50)

})

# Start the storytelling with Markdown cells

print("# Exploratory Data Analysis (EDA)")

# Visualization 1: Scatter plot with color-coded categories

plt.figure(figsize=(12, 4))

sns.scatterplot(x='X', y='Y', hue='Category', data=data)

plt.title('Scatter Plot with Color-Coded Categories')

plt.show()

# Visualization 2: Pair plot for overall exploration

print("## Pair Plot for Overall Exploration")

sns.pairplot(data, hue='Category')

plt.show()

# Explain insights from the visualizations

print("Insights:")

print("- The scatter plot shows a clear separation between categories A and B.")

print("- Pair plot reveals relationships between variables and category distribution.")

# Further analysis and visualization

print("# Further Analysis")

# Visualization 3: Box plot to analyze Y distribution by category

plt.figure(figsize=(8, 6))

sns.boxplot(x='Category', y='Y', data=data)

plt.title('Box Plot for Y Distribution by Category')

plt.show()

# Visualization 4: Histogram of X values

plt.figure(figsize=(8, 6))

sns.histplot(data['X'], bins=20, kde=True)

plt.title('Histogram of X Values')

plt.xlabel('X-axis')

plt.show()

# Conclude with final insights

print("Final Insights:")

print("- Category B has higher variability in Y values.")

print("- X values are approximately normally distributed.")

# End with recommendations or next steps

print("# Recommendations")

print("- Consider further analysis on the relationship between X, Y, and Category.")

**Practical 7 C**

**Aim :** Present the findings and insights in a clear and concise manner.

**Program :** import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Generate example data

np.random.seed(42)

data = pd.DataFrame({

'X': np.random.rand(50),

'Y': 2 \* np.random.rand(50),

'Category': np.random.choice(['A', 'B'], size=50)

})

# Start the storytelling with Markdown cells

print("# Data Storytelling and Analysis")

print("## Exploratory Data Analysis (EDA)")

# Visualization 1: Scatter plot with color-coded categories

plt.figure(figsize=(12, 4))

sns.scatterplot(x='X', y='Y', hue='Category', data=data)

plt.title('Scatter Plot with Color-Coded Categories')

plt.show()

# Visualization 2: Pair plot for overall exploration

print("### Pair Plot for Overall Exploration")

sns.pairplot(data, hue='Category')

plt.show()

# Explain insights from the visualizations

print("#### Insights:")

print("- The scatter plot shows a clear separation between categories A and B.")

print("- Pair plot reveals relationships between variables and category distribution.")

# Further analysis and visualization

print("## Further Analysis")

# Visualization 3: Box plot to analyze Y distribution by category

plt.figure(figsize=(8, 6))

sns.boxplot(x='Category', y='Y', data=data)

plt.title('Box Plot for Y Distribution by Category')

plt.show()

# Visualization 4: Histogram of X values

plt.figure(figsize=(8, 6))

sns.histplot(data['X'], bins=20, kde=True)

plt.title('Histogram of X Values')

plt.xlabel('X-axis')

plt.show()

# Conclude with final insights

print("### Final Insights:")

print("- Category B has higher variability in Y values.")

print("- X values are approximately normally distributed.")

# End with recommendations or next steps

print("## Recommendations")

print("- Consider further analysis on the relationship between X, Y, and Category.")

**Practical 8 A**

**Aim :** Implement simple linear regression using a dataset.

**Program :** import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

# Generating a sample dataset

np.random.seed(42)

X = 2 \* np.random.rand(100, 1)

y = 4 + 3 \* X + np.random.randn(100, 1)

# Splitting the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Creating and training the linear regression model

model = LinearRegression()

model.fit(X\_train, y\_train)

# Making predictions on the test set

y\_pred = model.predict(X\_test)

# Calculating Mean Squared Error (MSE)

mse = mean\_squared\_error(y\_test, y\_pred)

print(f'Mean Squared Error: {mse}')

# Plotting the results

plt.scatter(X\_test, y\_test, color='black', label='Actual Data')

plt.plot(X\_test, y\_pred, color='blue', linewidth=3, label='Regression Line')

plt.xlabel('X')

plt.ylabel('y')

plt.legend()

plt.show()

**Practical 8 B**

**Aim :** Explore and interpret the regression model coefficients and goodness-of-fit measures.

**Program :** import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

# Generating a sample dataset

np.random.seed(42)

X = 2 \* np.random.rand(100, 1)

y = 4 + 3 \* X + np.random.randn(100, 1)

# Splitting the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Creating and training the linear regression model

model = LinearRegression()

model.fit(X\_train, y\_train)

# Getting the regression coefficients

intercept = model.intercept\_[0]

slope = model.coef\_[0][0]

print(f'Intercept (b0): {intercept}')

print(f'Slope (b1): {slope}')

# Making predictions on the test set

y\_pred = model.predict(X\_test)

# Calculating Mean Squared Error (MSE) and R-squared

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f'Mean Squared Error: {mse}')

print(f'R-squared: {r2}')

# Plotting the results

plt.scatter(X\_test, y\_test, color='black', label='Actual Data')

plt.plot(X\_test, y\_pred, color='blue', linewidth=3, label='Regression Line')

plt.xlabel('X')

plt.ylabel('y')

plt.legend()

plt.title('Linear Regression Model')

plt.show()

**Practical 8 C**

**Aim :** Extend the analysis to multiple linear regression and assess the impact of additional predictors.

**Program :** import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

# Generating a sample dataset with two features

np.random.seed(42)

X = 2 \* np.random.rand(100, 2)

# y = 4 + 3\*X1 + 2\*X2 + noise

y = 4 + 3 \* X[:, 0] + 2 \* X[:, 1] + np.random.randn(100)

# Splitting the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Creating and training the multiple linear regression model

model = LinearRegression()

model.fit(X\_train, y\_train)

# Getting the regression coefficients and intercept

intercept = model.intercept\_

coefficients = model.coef\_

print(f'Intercept: {intercept}')

print(f'Coefficients: {coefficients}')

# Making predictions on the test set

y\_pred = model.predict(X\_test)

# Calculating Mean Squared Error (MSE) and R-squared

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f'Mean Squared Error: {mse}')

print(f'R-squared: {r2}')

# Plotting the results (for one feature, as visualization becomes challenging with multiple features)

plt.scatter(X\_test[:, 0], y\_test, color='black', label='Actual Data')

plt.scatter(X\_test[:, 0], y\_pred, color='blue', label='Predicted Data', marker='x')

plt.xlabel('X1')

plt.ylabel('y')

plt.legend()

plt.title('Multiple Linear Regression Model')

plt.show()

**Practical 9 A**

**Aim :** Build a logistic regression model to predict a binary outcome.

**Program :** import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

# Create a synthetic dataset for demonstration

np.random.seed(42)

data = pd.DataFrame({

'feature1': np.random.randn(100),

'feature2': np.random.randn(100),

'target': np.random.randint(0, 2, size=100)

})

# Split the dataset into features (X) and target variable (y)

X = data[['feature1', 'feature2']]

y = data['target']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a logistic regression model

model = LogisticRegression()

# Train the model on the training set

model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = model.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

classification\_rep = classification\_report(y\_test, y\_pred)

print(f"Accuracy: {accuracy}")

print(f"Confusion Matrix:\n{conf\_matrix}")

print(f"Classification Report:\n{classification\_rep}")

**Practical 9 B**

**Aim :** Evaluate the model's performance using classification metrics (e.g., accuracy, precision, recall).

**Program :** import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report

# Create a synthetic dataset for demonstration

np.random.seed(42)

data = pd.DataFrame({

'feature1': np.random.randn(100),

'feature2': np.random.randn(100),

'target': np.random.randint(0, 2, size=100)

})

# Split the dataset into features (X) and target variable (y)

X = data[['feature1', 'feature2']]

y = data['target']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a logistic regression model

model = LogisticRegression()

# Train the model on the training set

model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = model.predict(X\_test)

# Evaluate the model using classification metrics

report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", report)

**Practical 9 C**

**Aim :** Construct a decision tree model and interpret the decision rules for classification.

**Program :** import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier, export\_text

# Create a synthetic dataset for demonstration

np.random.seed(42)

data = pd.DataFrame({

'feature1': np.random.randn(100),

'feature2': np.random.randn(100),

'target': np.random.randint(0, 2, size=100)

})

# Split the dataset into features (X) and target variable (y)

X = data[['feature1', 'feature2']]

y = data['target']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a decision tree model

model = DecisionTreeClassifier(random\_state=42)

# Train the model on the training set

model.fit(X\_train, y\_train)

# Interpret the decision rules

tree\_rules = export\_text(model, feature\_names=list(X.columns))

print("Decision Rules:\n", tree\_rules)

**Practical 10 A**

**Aim :** Apply the K-Means algorithm to group similar data points into clusters.

**Program :** import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.datasets import make\_blobs

# Generate sample data

np.random.seed(42)

X, \_ = make\_blobs(n\_samples=300, centers=4, random\_state=42)

# Apply K-Means algorithm

kmeans = KMeans(n\_clusters=4, random\_state=42)

kmeans.fit(X)

# Get cluster centers and labels

centroids = kmeans.cluster\_centers\_

labels = kmeans.labels\_

# Plot the original data and cluster centers

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', alpha=0.7, edgecolors='k')

plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', s=200, linewidths=3, color='r', label='Centroids')

plt.title('K-Means Clustering')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend()

plt.show()

**Practical 10 B**

**Aim :** Determine the optimal number of clusters using elbow method or silhouette analysis.

**Program (Elbow Method) :** import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.datasets import make\_blobs

# Generate sample data

np.random.seed(42)

X, \_ = make\_blobs(n\_samples=300, centers=4, random\_state=42)

# Elbow method to find optimal number of clusters

inertia = []

for k in range(1, 11):

kmeans = KMeans(n\_clusters=k, random\_state=42)

kmeans.fit(X)

inertia.append(kmeans.inertia\_)

# Plot the elbow curve

plt.plot(range(1, 11), inertia, marker='o')

plt.title('Elbow Method for Optimal k')

plt.xlabel('Number of Clusters (k)')

plt.ylabel('Sum of Squared Distances (Inertia)')

plt.show()

**Program (Silhouette Analysis) :** import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

from sklearn.datasets import make\_blobs

# Generate sample data

np.random.seed(42)

X, \_ = make\_blobs(n\_samples=300, centers=4, random\_state=42)

# Silhouette analysis to find the optimal number of clusters

silhouette\_scores = []

for k in range(2, 11):

kmeans = KMeans(n\_clusters=k, random\_state=42)

kmeans.fit(X)

silhouette\_scores.append(silhouette\_score(X, kmeans.labels\_))

# Plot silhouette scores

plt.plot(range(2, 11), silhouette\_scores, marker='o')

plt.title('Silhouette Analysis for Optimal k')

plt.xlabel('Number of Clusters (k)')

plt.ylabel('Silhouette Score')

plt.show()

**Practical 10 C**

**Aim :** Visualize the clustering results and analyze the cluster characteristics.

**Program :** import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

from sklearn.datasets import make\_blobs

# Generate sample data

np.random.seed(42)

X, \_ = make\_blobs(n\_samples=300, centers=4, random\_state=42)

# Determine the optimal number of clusters using silhouette analysis

optimal\_k = 4 # Replace this with the optimal number from silhouette analysis

# Apply K-Means with the optimal number of clusters

kmeans = KMeans(n\_clusters=optimal\_k, random\_state=42)

kmeans.fit(X)

# Visualize the clustering results

plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels\_, cmap='viridis', alpha=0.7, edgecolors='k')

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], marker='X', s=200, linewidths=3, color='r', label='Centroids')

plt.title('K-Means Clustering Results')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend()

plt.show()

# Analyze cluster characteristics

for cluster\_id in range(optimal\_k):

cluster\_points = X[kmeans.labels\_ == cluster\_id]

centroid = kmeans.cluster\_centers\_[cluster\_id]

print(f"\nCluster {cluster\_id + 1} Characteristics:")

print(f"Number of points: {len(cluster\_points)}")

print(f"Centroid: {centroid}")

print(f"Cluster points: {cluster\_points}")

**Practical 11**

**Aim :** Perform PCA on a dataset to reduce dimensionality.

**Program :**

import numpy as np

from sklearn.decomposition import PCA

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

# Load an example dataset (you can replace this with your own dataset)

iris = load\_iris()

X = iris.data

y = iris.target

# Standardize the data (optional but recommended for PCA)

mean = np.mean(X, axis=0)

std\_dev = np.std(X, axis=0)

X\_standardized = (X - mean) / std\_dev

# Initialize PCA with the number of components you want to retain

n\_components = 2 # You can choose the number of components based on your requirements

pca = PCA(n\_components=n\_components)

# Fit the PCA model and transform the data

X\_pca = pca.fit\_transform(X\_standardized)

# Visualize the results (scatter plot for the first two principal components)

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=y, edgecolors='k', cmap=plt.cm.Paired)

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.title('PCA of Iris dataset')

plt.show()

# Explained variance ratio

explained\_variance\_ratio = pca.explained\_variance\_ratio\_

print(f"Explained Variance Ratio: {explained\_variance\_ratio}")