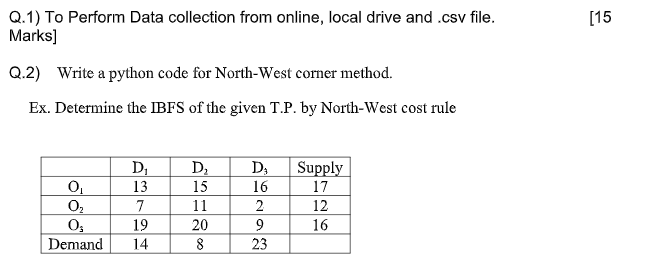
**Practical 1**



Q1.

**For python**

import requests

url = "https://api.example.com/data"

response = requests.get(url)

data = response.json() # or response.text for raw text

**for local file**

import pandas as pd

data = pd.read\_excel("C:/path/to/yourfile.xlsx")

**For csv**

import pandas as pd

data = pd.read\_csv("C:/path/to/yourfile.csv")

**Q2. North west method**

import numpy as np

import pandas as pd

# Define the cost matrix, supply, and demand

cost\_matrix = np.array([

[13, 15, 16],

[7, 11, 2],

[19, 20, 9]

])

supply = [17, 12, 16]

demand = [14, 8, 23]

# Initialize allocation matrix with zeros

allocation = np.zeros\_like(cost\_matrix, dtype=int)

# Apply North-West Corner Method

i, j = 0, 0

while i < len(supply) and j < len(demand):

# Find the minimum of supply and demand

allocation\_value = min(supply[i], demand[j])

allocation[i][j] = allocation\_value

# Update supply and demand

supply[i] -= allocation\_value

demand[j] -= allocation\_value

# Move to the next cell

if supply[i] == 0:

i += 1

elif demand[j] == 0:

j += 1

# Display the allocation matrix

allocation\_df = pd.DataFrame(allocation, columns=['D1', 'D2', 'D3'], index=['O1', 'O2', 'O3'])

print("Initial Basic Feasible Solution (IBFS) by North-West Corner Method:")

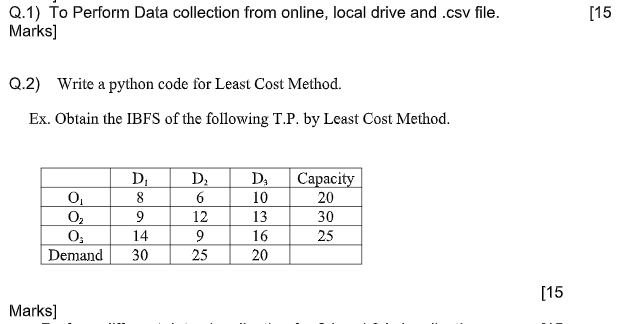
print(allocation\_df)

# Optional: Calculate the total cost

total\_cost = (allocation \* cost\_matrix).sum()

print("\nTotal Transportation Cost:", total\_cost)

**Practical 2**



**Q2. IBFS by least cost method**

import numpy as np

import pandas as pd

# Define the cost matrix, supply, and demand

cost\_matrix = np.array([

[8, 6, 10],

[9, 12, 13],

[14, 9, 16]

])

supply = [20, 30, 25]

demand = [30, 25, 20]

# Initialize allocation matrix with zeros

allocation = np.zeros\_like(cost\_matrix, dtype=int)

# Convert supply and demand to mutable lists

remaining\_supply = supply.copy()

remaining\_demand = demand.copy()

# Apply Least Cost Method

while any(remaining\_supply) and any(remaining\_demand):

# Find the minimum cost cell

min\_cost = np.inf

min\_pos = (-1, -1)

for i in range(len(remaining\_supply)):

for j in range(len(remaining\_demand)):

if remaining\_supply[i] > 0 and remaining\_demand[j] > 0 and cost\_matrix[i][j] < min\_cost:

min\_cost = cost\_matrix[i][j]

min\_pos = (i, j)

i, j = min\_pos

# Allocate as much as possible to the minimum cost cell

allocation\_value = min(remaining\_supply[i], remaining\_demand[j])

allocation[i][j] = allocation\_value

# Update supply and demand

remaining\_supply[i] -= allocation\_value

remaining\_demand[j] -= allocation\_value

# Display the allocation matrix

allocation\_df = pd.DataFrame(allocation, columns=['D1', 'D2', 'D3'], index=['O1', 'O2', 'O3'])

print("Initial Basic Feasible Solution (IBFS) by Least Cost Method:")

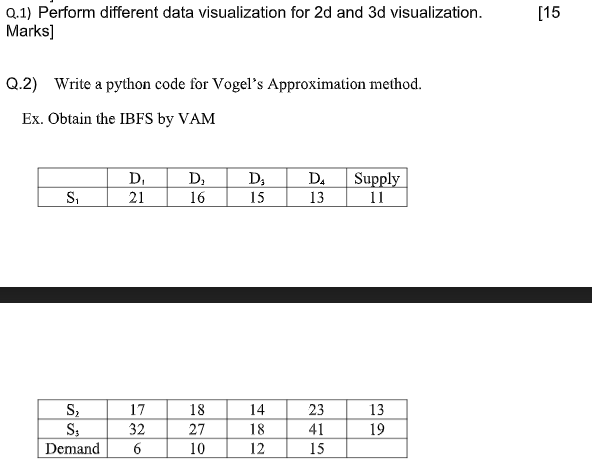
print(allocation\_df)

# Optional: Calculate the total cost

total\_cost = (allocation \* cost\_matrix).sum()

print("\nTotal Transportation Cost:", total\_cost)

**Practical 3**



**Q1. 2d and 3d visualization**

**Line plot**

import matplotlib.pyplot as plt

# Sample data

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']

sales = [250, 300, 350, 400, 450, 500]

# Create line plot

plt.figure(figsize=(8, 5))

plt.plot(months, sales, marker='o', linestyle='-', color='b')

plt.title('Monthly Sales Trend')

plt.xlabel('Month')

plt.ylabel('Sales')

plt.grid(True)

plt.show()

**Bar Plot**

import matplotlib.pyplot as plt

# Sample data

products = ['Product A', 'Product B', 'Product C', 'Product D']

sales = [500, 700, 300, 400]

# Create bar chart

plt.figure(figsize=(8, 5))

plt.bar(products, sales, color=['skyblue', 'salmon', 'lightgreen', 'violet'])

plt.title('Sales by Product')

plt.xlabel('Products')

plt.ylabel('Sales')

plt.show()

**Scatter plot**

import matplotlib.pyplot as plt

# Sample data

advertising = [50, 60, 70, 80, 90, 100, 110, 120]

sales = [200, 240, 290, 310, 400, 450, 500, 550]

# Create scatter plot

plt.figure(figsize=(8, 5))

plt.scatter(advertising, sales, color='green', marker='o')

plt.title('Advertising Spend vs Sales')

plt.xlabel('Advertising Spend ($)')

plt.ylabel('Sales ($)')

plt.grid(True)

plt.show()

**3D scatter plot**

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import numpy as np

# Sample data

np.random.seed(0)

x = np.random.randint(1, 100, 50)

y = np.random.randint(1, 100, 50)

z = np.random.randint(1, 100, 50)

colors = np.random.rand(50)

sizes = np.random.randint(20, 200, 50)

# Create 3D scatter plot

fig = plt.figure(figsize=(10, 7))

ax = fig.add\_subplot(111, projection='3d')

scatter = ax.scatter(x, y, z, c=colors, s=sizes, cmap='viridis', alpha=0.6)

ax.set\_title('3D Scatter Plot')

ax.set\_xlabel('X-axis')

ax.set\_ylabel('Y-axis')

ax.set\_zlabel('Z-axis')

plt.colorbar(scatter, ax=ax, label='Color Scale')

plt.show()

**3D Bar Plot**

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import numpy as np

# Sample data

regions = ['North', 'South', 'East', 'West']

products = ['A', 'B', 'C']

data = np.random.randint(10, 100, size=(len(regions), len(products)))

# Create 3D bar chart

fig = plt.figure(figsize=(10, 7))

ax = fig.add\_subplot(111, projection='3d')

\_x = np.arange(len(regions))

\_y = np.arange(len(products))

\_xx, \_yy = np.meshgrid(\_x, \_y)

x, y = \_xx.ravel(), \_yy.ravel()

top = data.ravel()

z = np.zeros\_like(x)

dx = dy = 0.5

dz = top

colors = plt.cm.viridis(dz / dz.max())

ax.bar3d(x, y, z, dx, dy, dz, color=colors, shade=True)

ax.set\_xticks(np.arange(len(regions)) + dx/2)

ax.set\_xticklabels(regions)

ax.set\_yticks(np.arange(len(products)) + dy/2)

ax.set\_yticklabels(products)

ax.set\_zlabel('Sales')

ax.set\_title('3D Bar Chart: Sales by Region and Product')

plt.show()

**Q2. IBFS by VAM**

import numpy as np

def vogels\_approximation\_method(cost\_matrix, supply, demand):

# Convert inputs to numpy arrays for easy manipulation

cost\_matrix = np.array(cost\_matrix)

supply = np.array(supply)

demand = np.array(demand)

# Initial allocation matrix (same shape as cost matrix)

allocation = np.zeros\_like(cost\_matrix, dtype=int)

while supply.sum() > 0 and demand.sum() > 0:

# Calculate penalties

row\_penalties = []

for row in cost\_matrix:

# Get the sorted non-blocked costs

sorted\_row = np.sort(row[row != -1])

if len(sorted\_row) > 1:

penalty = sorted\_row[1] - sorted\_row[0]

elif len(sorted\_row) == 1:

penalty = sorted\_row[0]

else:

penalty = 0 # No penalty if all elements are blocked

row\_penalties.append(penalty)

col\_penalties = []

for col in cost\_matrix.T:

# Get the sorted non-blocked costs

sorted\_col = np.sort(col[col != -1])

if len(sorted\_col) > 1:

penalty = sorted\_col[1] - sorted\_col[0]

elif len(sorted\_col) == 1:

penalty = sorted\_col[0]

else:

penalty = 0 # No penalty if all elements are blocked

col\_penalties.append(penalty)

# Choose the maximum penalty

max\_row\_penalty = max(row\_penalties)

max\_col\_penalty = max(col\_penalties)

if max\_row\_penalty >= max\_col\_penalty:

row = row\_penalties.index(max\_row\_penalty)

min\_cost\_index = np.where(cost\_matrix[row] == min(cost\_matrix[row][cost\_matrix[row] != -1]))[0][0]

allocate = min(supply[row], demand[min\_cost\_index])

allocation[row][min\_cost\_index] = allocate

supply[row] -= allocate

demand[min\_cost\_index] -= allocate

if supply[row] == 0:

cost\_matrix[row] = -1 # Mark row as filled

if demand[min\_cost\_index] == 0:

cost\_matrix[:, min\_cost\_index] = -1 # Mark column as filled

else:

col = col\_penalties.index(max\_col\_penalty)

min\_cost\_index = np.where(cost\_matrix[:, col] == min(cost\_matrix[:, col][cost\_matrix[:, col] != -1]))[0][0]

allocate = min(supply[min\_cost\_index], demand[col])

allocation[min\_cost\_index][col] = allocate

supply[min\_cost\_index] -= allocate

demand[col] -= allocate

if supply[min\_cost\_index] == 0:

cost\_matrix[min\_cost\_index] = -1 # Mark row as filled

if demand[col] == 0:

cost\_matrix[:, col] = -1 # Mark column as filled

return allocation

# Example input

cost\_matrix = [

[21, 16, 15, 13],

[17, 18, 14, 23],

[32, 27, 18, 41]

]

supply = [11, 13, 19]

demand = [6, 10, 12, 15]

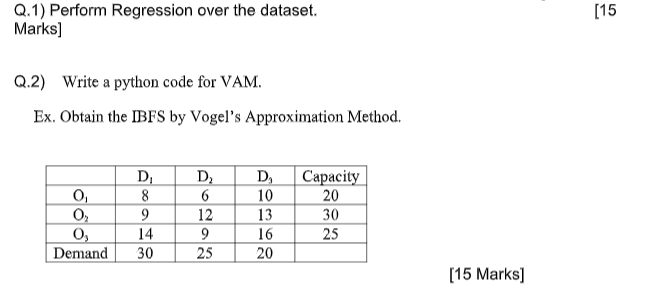
# Call the function and print the allocation

allocation = vogels\_approximation\_method(cost\_matrix, supply, demand)

print("Initial Basic Feasible Solution (Allocation):")

print(allocation)

**Practical 4**



**Q1. Regression**

import pandas as pd

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

# Load dataset (replace 'your\_dataset.csv' with your actual dataset file)

data = pd.read\_csv('your\_dataset.csv')

# Display the first few rows of the dataset

print("Dataset preview:")

print(data.head())

# Define features (X) and target variable (y)

# Replace 'feature\_column' and 'target\_column' with your actual column names

X = data[['feature\_column']] # Use double brackets for a DataFrame (for single feature)

y = data['target\_column']

# Split 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 Linear Regression model and fit it to the training data

model = LinearRegression()

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

# Predict target values for the testing set

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

# Print model coefficients

print("Coefficient:", model.coef\_)

print("Intercept:", model.intercept\_)

# Evaluate the model

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

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

print("Mean Squared Error:", mse)

print("R^2 Score:", r2)

# Plot the results

plt.figure(figsize=(10, 6))

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

plt.plot(X\_test, y\_pred, color='red', label='Predicted')

plt.xlabel('Feature')

plt.ylabel('Target')

plt.title('Linear Regression Model')

plt.legend()

plt.show()

**Q2. IBFS by VAM**

import numpy as np

def vogels\_approximation\_method(cost\_matrix, supply, demand):

# Convert inputs to numpy arrays for easy manipulation

cost\_matrix = np.array(cost\_matrix)

supply = np.array(supply)

demand = np.array(demand)

# Initial allocation matrix (same shape as cost matrix)

allocation = np.zeros\_like(cost\_matrix, dtype=int)

while supply.sum() > 0 and demand.sum() > 0:

# Step 1: Calculate penalties for each row

row\_penalties = []

for row in cost\_matrix:

sorted\_row = np.sort(row[row != -1]) # Ignore filled cells (-1)

penalty = sorted\_row[1] - sorted\_row[0] if len(sorted\_row) > 1 else 0

row\_penalties.append(penalty)

# Step 2: Calculate penalties for each column

col\_penalties = []

for col in cost\_matrix.T:

sorted\_col = np.sort(col[col != -1]) # Ignore filled cells (-1)

penalty = sorted\_col[1] - sorted\_col[0] if len(sorted\_col) > 1 else 0

col\_penalties.append(penalty)

# Step 3: Select the row or column with the highest penalty

max\_row\_penalty = max(row\_penalties)

max\_col\_penalty = max(col\_penalties)

if max\_row\_penalty >= max\_col\_penalty:

# Step 4: Allocate from the row with the maximum penalty

row = row\_penalties.index(max\_row\_penalty)

min\_cost\_index = np.where(cost\_matrix[row] == min(cost\_matrix[row][cost\_matrix[row] != -1]))[0][0]

allocate = min(supply[row], demand[min\_cost\_index])

allocation[row][min\_cost\_index] = allocate

supply[row] -= allocate

demand[min\_cost\_index] -= allocate

if supply[row] == 0:

cost\_matrix[row] = -1 # Mark row as filled

if demand[min\_cost\_index] == 0:

cost\_matrix[:, min\_cost\_index] = -1 # Mark column as filled

else:

# Step 5: Allocate from the column with the maximum penalty

col = col\_penalties.index(max\_col\_penalty)

min\_cost\_index = np.where(cost\_matrix[:, col] == min(cost\_matrix[:, col][cost\_matrix[:, col] != -1]))[0][0]

allocate = min(supply[min\_cost\_index], demand[col])

allocation[min\_cost\_index][col] = allocate

supply[min\_cost\_index] -= allocate

demand[col] -= allocate

if supply[min\_cost\_index] == 0:

cost\_matrix[min\_cost\_index] = -1 # Mark row as filled

if demand[col] == 0:

cost\_matrix[:, col] = -1 # Mark column as filled

return allocation

# Example input data for transportation problem

cost\_matrix = [

[8, 6, 10],

[9, 12, 13],

[14, 9, 16]

]

supply = [20, 30, 25]

demand = [30, 25, 20]

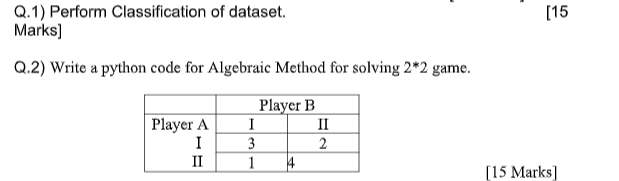
# Call the function and print the allocation

allocation = vogels\_approximation\_method(cost\_matrix, supply, demand)

print("Initial Basic Feasible Solution (Allocation):")

print(allocation)

**Practical 5**



**Q1. Classification**

import pandas as pd

import numpy as np

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

from sklearn.preprocessing import LabelEncoder

from sklearn.linear\_model import LogisticRegression

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

import matplotlib.pyplot as plt

import seaborn as sns

# Load the dataset

data = pd.read\_csv('your\_dataset.csv') # Replace with your dataset path

# Preview the dataset

print("Dataset preview:")

print(data.head())

# Assuming the target variable is 'target\_column' and all other columns are features

X = data.drop(columns=['target\_column']) # Features (independent variables)

y = data['target\_column'] # Target (dependent variable)

# Encode categorical features if necessary (for example, using LabelEncoder for a categorical target)

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(y) # Transform labels to numeric if categorical

# Split the dataset into training and testing sets (80% training, 20% testing)

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

# Initialize Logistic Regression model

model = LogisticRegression()

# Train the model

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)

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

# Print the evaluation metrics

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(class\_report)

# Plot the confusion matrix

plt.figure(figsize=(6, 4))

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=label\_encoder.classes\_, yticklabels=label\_encoder.classes\_)

plt.xlabel('Predicted')

plt.ylabel('True')

plt.title('Confusion Matrix')

plt.show()

**Q2. 2\*2 game**

from sympy import symbols, Eq, solve

# Define variables

p, q = symbols('p q')

# Set up the equations for expected payoffs

# Player A's expected payoff equations for I and II (Player B choosing strategies)

eq1 = Eq(q + 2, 4 - 3\*q) # Player A's equation (expected payoff for strategy I and II)

# Solve for q (Player B's optimal strategy)

q\_solution = solve(eq1, q)

q\_optimal = q\_solution[0]

print(f"Player B's optimal strategy q (probability of choosing I): {q\_optimal:.2f}")

# Calculate Player A's expected payoff using q's value

# Expected payoff for Player A choosing strategy I

expected\_payoff\_A = q\_optimal + 2

print(f"Player A's expected payoff when playing I: {expected\_payoff\_A:.2f}")

# Calculate Player A's expected payoff when playing II (to verify equality)

expected\_payoff\_A\_II = 4 - 3\*q\_optimal

print(f"Player A's expected payoff when playing II: {expected\_payoff\_A\_II:.2f}")

# Now for Player B's strategy, we use the equilibrium condition for Player B's payoff

# Player B's expected payoff equations for strategies I and II (Player A choosing strategies)

eq2 = Eq(3\*p + 1\*(1-p), 2\*p + 4\*(1-p)) # Player B's equation (expected payoff for strategy I and II)

# Solve for p (Player A's optimal strategy)

p\_solution = solve(eq2, p)

p\_optimal = p\_solution[0]

print(f"Player A's optimal strategy p (probability of choosing I): {p\_optimal:.2f}")

# Calculate Player B's expected payoff using p's value

expected\_payoff\_B = 3\*p\_optimal + 1\*(1-p\_optimal)

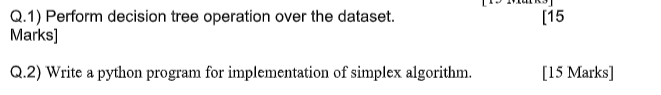
print(f"Player B's expected payoff when playing I: {expected\_payoff\_B:.2f}")

# Display optimal mixed strategies and expected payoffs

print(f"Player A's optimal strategy (p): {p\_optimal:.2f}")

print(f"Player B's optimal strategy (q): {q\_optimal:.2f}")

**Practical 6**



**Q1. Decision tree**

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

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

from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt

from sklearn.tree import plot\_tree

# Step 1: Load the dataset

# Replace 'your\_dataset.csv' with your actual dataset

data = pd.read\_csv('your\_dataset.csv')

# Preview the dataset

print("Dataset preview:")

print(data.head())

# Step 2: Preprocessing

# Assuming the target variable is 'target\_column', and all other columns are features

X = data.drop(columns=['target\_column']) # Features (independent variables)

y = data['target\_column'] # Target (dependent variable)

# If there are any categorical columns, encode them

# Example: Encode target labels if they are categorical

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(y) # Transform labels to numeric

# Step 3: Split the data into training and testing sets (80% training, 20% testing)

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

# Step 4: Initialize the Decision Tree Classifier

model = DecisionTreeClassifier(random\_state=42)

# Step 5: Train the model

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

# Step 6: Make predictions on the test set

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

# Step 7: Evaluate the model

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

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

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

# Print evaluation metrics

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(class\_report)

# Step 8: Visualize the decision tree

plt.figure(figsize=(12, 8))

plot\_tree(model, filled=True, feature\_names=X.columns, class\_names=label\_encoder.classes\_, rounded=True)

plt.title("Decision Tree Classifier Visualization")

plt.show()

**Q2. Write a python program for implementation of simplex algorithm.**

import numpy as np

def simplex(c, A, b):

"""

Simplex algorithm implementation for Linear Programming problems.

Parameters:

c: Coefficients of the objective function (1D array).

A: Coefficients of the constraints (2D array).

b: Right-hand side values of the constraints (1D array).

Returns:

Solution vector (1D array) and optimal value.

"""

m, n = A.shape # Number of constraints and variables

tableau = np.hstack([A, np.eye(m), b.reshape(-1, 1)]) # Augmented matrix with identity for slack variables

c = np.hstack([c, np.zeros(m)]) # Add zeros for slack variable costs

# Phase 1: Set up the initial tableau

while True:

# Find the most negative value in the last row (excluding the right-hand side column)

pivot\_col = np.argmin(tableau[-1, :-1])

# If all values in the objective row are >= 0, we are done

if tableau[-1, pivot\_col] >= 0:

break

# Calculate the ratios to determine the pivot row

ratios = tableau[:-1, -1] / tableau[:-1, pivot\_col]

pivot\_row = np.argmin(ratios)

# Perform row operations to make the pivot element 1 and other elements 0 in the pivot column

pivot = tableau[pivot\_row, pivot\_col]

tableau[pivot\_row, :] /= pivot

for i in range(m + 1):

if i != pivot\_row:

tableau[i, :] -= tableau[i, pivot\_col] \* tableau[pivot\_row, :]

# Phase 2: Solve the original problem

while True:

# Find the most negative value in the objective row (c row)

pivot\_col = np.argmin(tableau[-1, :-1])

# If no negative values are present, optimal solution is found

if tableau[-1, pivot\_col] >= 0:

break

# Calculate the ratios for the pivot row selection

ratios = tableau[:-1, -1] / tableau[:-1, pivot\_col]

pivot\_row = np.argmin(ratios)

# Perform row operations to make the pivot element 1 and other elements 0 in the pivot column

pivot = tableau[pivot\_row, pivot\_col]

tableau[pivot\_row, :] /= pivot

for i in range(m + 1):

if i != pivot\_row:

tableau[i, :] -= tableau[i, pivot\_col] \* tableau[pivot\_row, :]

# Extract the optimal solution and the optimal value

solution = np.zeros(n)

for i in range(m):

if np.all(tableau[i, :-1] == 0):

continue

basic\_variable = np.argmax(tableau[i, :-1] == 1)

if basic\_variable != -1:

solution[basic\_variable] = tableau[i, -1]

optimal\_value = tableau[-1, -1]

return solution, optimal\_value

# Example problem:

# Maximize Z = 3x1 + 2x2

# Subject to:

# x1 + x2 <= 4

# 2x1 + x2 <= 5

# x1, x2 >= 0

c = np.array([3, 2]) # Coefficients of the objective function

A = np.array([[1, 1], [2, 1]]) # Coefficients of the constraints

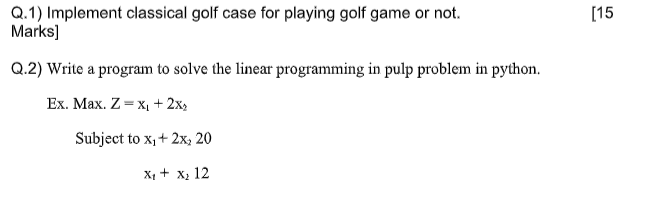
b = np.array([4, 5]) # Right-hand side values

solution, optimal\_value = simplex(c, A, b)

print("Optimal solution:", solution)

print("Optimal value of the objective function:", optimal\_value)

**Practical 7**



**Q1. Classical golf game**

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

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

from sklearn.tree import plot\_tree

import matplotlib.pyplot as plt

# Step 1: Create the dataset

data = {

'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny', 'Sunny', 'Rainy'],

'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild'],

'Humidity': ['High', 'High', 'High', 'High', 'Low', 'Low', 'Low', 'High', 'Low', 'Low'],

'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Weak', 'Weak'],

'Play Golf': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes']

}

# Step 2: Convert the dataset to a pandas DataFrame

df = pd.DataFrame(data)

# Step 3: Preprocess the data (Convert categorical variables to numerical)

df['Outlook'] = df['Outlook'].map({'Sunny': 0, 'Overcast': 1, 'Rainy': 2})

df['Temperature'] = df['Temperature'].map({'Hot': 0, 'Mild': 1, 'Cool': 2})

df['Humidity'] = df['Humidity'].map({'High': 0, 'Low': 1})

df['Wind'] = df['Wind'].map({'Weak': 0, 'Strong': 1})

df['Play Golf'] = df['Play Golf'].map({'No': 0, 'Yes': 1})

# Step 4: Define the features (X) and the target (y)

X = df[['Outlook', 'Temperature', 'Humidity', 'Wind']] # Features

y = df['Play Golf'] # Target

# Step 5: Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Step 6: Initialize the Decision Tree classifier

dtree = DecisionTreeClassifier(random\_state=42)

# Step 7: Train the model

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

# Step 8: Make predictions

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

# Step 9: Evaluate the model

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

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

# Print the results

print("Accuracy:", accuracy)

print("Classification Report:")

print(class\_report)

# Step 10: Visualize the decision tree

plt.figure(figsize=(12,8))

plot\_tree(dtree, feature\_names=X.columns, class\_names=['No', 'Yes'], filled=True)

plt.title("Decision Tree for Play Golf Prediction")

plt.show()

**Q2. Write a program to solve the linear programming in pulp problem in python.**

**Ex. Max. Z = x 1 + 2x 2**

**Subject to x 1 + 2x 2 20**

**x 1 + x 2 12**

# Importing PuLP library

from pulp import \*

# Step 1: Define the Linear Program

lp\_problem = LpProblem("Maximize\_Z", LpMaximize)

# Step 2: Define the decision variables

x1 = LpVariable('x1', lowBound=0) # x1 >= 0

x2 = LpVariable('x2', lowBound=0) # x2 >= 0

# Step 3: Define the objective function

lp\_problem += x1 + 2\*x2, "Objective Function"

# Step 4: Define the constraints

lp\_problem += x1 + 2\*x2 <= 20, "Constraint 1"

lp\_problem += x1 + x2 <= 12, "Constraint 2"

# Step 5: Solve the problem

lp\_problem.solve()

# Step 6: Display the results

print("Status:", LpStatus[lp\_problem.status])

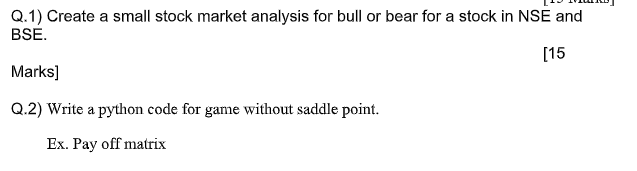
print("Optimal Solution:")

print(f"x1 = {x1.varValue}")

print(f"x2 = {x2.varValue}")

print(f"Maximized Z = {value(lp\_problem.objective)}")

**Practical 8**



**Q1. Stock market analysis**

pip install yfinance pandas matplotlib

# Install required libraries first (if not already installed)

# pip install yfinance pandas matplotlib

import yfinance as yf

import pandas as pd

import matplotlib.pyplot as plt

# Step 1: Fetch stock data from Yahoo Finance

# For example, fetching data for the stock 'RELIANCE.NS' from NSE

stock\_symbol = 'RELIANCE.NS' # Use NSE symbol

stock\_data = yf.download(stock\_symbol, start="2020-01-01", end="2024-01-01")

# Step 2: Calculate Moving Averages (50-day and 200-day)

stock\_data['50-day MA'] = stock\_data['Close'].rolling(window=50).mean()

stock\_data['200-day MA'] = stock\_data['Close'].rolling(window=200).mean()

# Step 3: Plot the stock data with Moving Averages

plt.figure(figsize=(10, 6))

plt.plot(stock\_data['Close'], label="Stock Price", color='blue')

plt.plot(stock\_data['50-day MA'], label="50-Day Moving Average", color='green')

plt.plot(stock\_data['200-day MA'], label="200-Day Moving Average", color='red')

# Step 4: Analyze Bull or Bear Market

if stock\_data['50-day MA'][-1] > stock\_data['200-day MA'][-1]:

market\_condition = "Bull Market"

else:

market\_condition = "Bear Market"

plt.title(f"{stock\_symbol} - {market\_condition}")

plt.xlabel("Date")

plt.ylabel("Stock Price")

plt.legend(loc="best")

plt.grid(True)

plt.show()

# Step 5: Display the market condition

print(f"Based on the analysis, the current market condition for {stock\_symbol} is: {market\_condition}")

**Q2. Write a python code for game without saddle point.**

**Ex. Pay off matrix**

from scipy.optimize import linprog

# Payoff matrix for Player A

payoff\_matrix = [

[3, 4, 6], # Player A's payoffs for R1

[2, 1, 5], # Player A's payoffs for R2

[0, 3, 4] # Player A's payoffs for R3

]

# Number of strategies for Player A and Player B

num\_rows = len(payoff\_matrix) # 3 strategies for Player A

num\_cols = len(payoff\_matrix[0]) # 3 strategies for Player B

# Step 1: Solve for Player B's mixed strategy (probabilities over columns)

# Let x1, x2, x3 be the probabilities of Player B choosing C1, C2, C3

# Objective: Maximize the expected payoff for Player A

# Player A's expected payoff for each row:

# Expected payoff = p1 \* payoff\_matrix[0][0] + p2 \* payoff\_matrix[0][1] + p3 \* payoff\_matrix[0][2]

# Step 2: Formulate the linear program to minimize/maximize

c = [-1] \* num\_cols # Coefficients (we are maximizing, so we negate the matrix)

A\_eq = [[1] \* num\_cols] # Sum of probabilities should equal 1

b\_eq = [1] # Probabilities sum to 1

# Bounds for each variable (probabilities between 0 and 1)

bounds = [(0, 1) for \_ in range(num\_cols)]

# Solve the linear program using SciPy

result = linprog(c, A\_eq=A\_eq, b\_eq=b\_eq, bounds=bounds, method='highs')

# Step 3: Get Player B's mixed strategy probabilities

probabilities\_B = result.x

# Step 4: Find Player A's optimal mixed strategy

# Now that we have Player B's strategy, we can find Player A's optimal mixed strategy.

# Find Player A's expected payoffs based on Player B's strategy

expected\_payoffs\_A = [sum(payoff\_matrix[i][j] \* probabilities\_B[j] for j in range(num\_cols)) for i in range(num\_rows)]

# Step 5: Find the probabilities for Player A to play each strategy

# Player A will choose probabilities (y1, y2, y3) that maximize their expected payoff

# We will use linear programming again to find Player A's mixed strategy

# Coefficients are the expected payoffs

c\_A = [-1 \* x for x in expected\_payoffs\_A] # Maximize expected payoffs for Player A

A\_eq\_A = [[1] \* num\_rows] # Probabilities sum to 1

b\_eq\_A = [1]

bounds\_A = [(0, 1) for \_ in range(num\_rows)] # Each probability must be between 0 and 1

# Solve the linear program for Player A

result\_A = linprog(c\_A, A\_eq=A\_eq\_A, b\_eq=b\_eq\_A, bounds=bounds\_A, method='highs')

# Step 6: Get Player A's mixed strategy probabilities

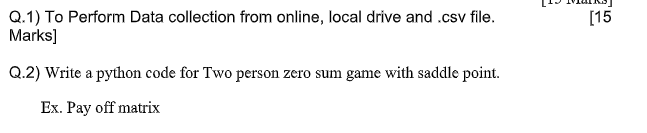
probabilities\_A = result\_A.x

# Step 7: Print the results

print("Player A's Mixed Strategy Probabilities: ", probabilities\_A)

print("Player B's Mixed Strategy Probabilities: ", probabilities\_B)

**Practical 9**



**Q2. Two person zero sum game with saddle point**

import numpy as np

def find\_saddle\_point(payoff\_matrix):

# Convert the payoff matrix to a numpy array

matrix = np.array(payoff\_matrix)

# Step 1: Find the row minima

row\_minima = matrix.min(axis=1)

# Step 2: Find the column maxima

column\_maxima = matrix.max(axis=0)

# Step 3: Find the saddle point (if any)

saddle\_points = []

for i in range(matrix.shape[0]): # For each row

for j in range(matrix.shape[1]): # For each column

if matrix[i, j] == row\_minima[i] and matrix[i, j] == column\_maxima[j]:

saddle\_points.append((i, j, matrix[i, j])) # Add the position and value of saddle point

return saddle\_points

# Define the payoff matrix for Player A (Row player)

payoff\_matrix = [

[3, 4, 6], # Player A's payoffs for R1

[2, 1, 5], # Player A's payoffs for R2

[0, 3, 4] # Player A's payoffs for R3

]

# Step 1: Find saddle point(s)

saddle\_points = find\_saddle\_point(payoff\_matrix)

# Step 2: Print the results

if saddle\_points:

print("Saddle Point(s) Found:")

for sp in saddle\_points:

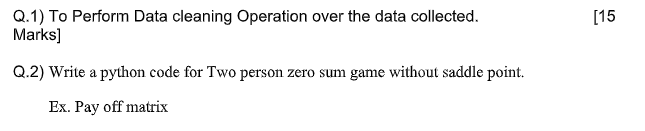
row, col, value = sp

print(f"Saddle point at Row {row+1}, Column {col+1} with value {value}")

else:

print("No saddle point exists in the payoff matrix.")

**Practical 10**



**Q1. Data cleaning**

import pandas as pd

import numpy as np

# Step 1: Load the Data

# Let's assume the dataset is in a CSV file named 'data.csv'

df = pd.read\_csv('data.csv')

# Display the first few rows of the dataset

print("Original Dataset:")

print(df.head())

# Step 2: Handle Missing Values

# 2.1 Check for missing values

missing\_values = df.isnull().sum()

print("\nMissing Values Count:")

print(missing\_values)

# 2.2 Fill missing values with a specific value (e.g., mean, median, or mode)

df.fillna(df.mean(), inplace=True) # You can replace NaN with the column mean

# Alternatively, drop rows with missing values

# df.dropna(inplace=True)

# Step 3: Remove Duplicates

# Check for duplicate rows

duplicates = df.duplicated().sum()

print(f"\nNumber of duplicate rows: {duplicates}")

# Remove duplicates

df.drop\_duplicates(inplace=True)

# Step 4: Handle Outliers

# Use Z-score to detect outliers (example: for columns 'col1', 'col2')

from scipy.stats import zscore

z\_scores = np.abs(zscore(df[['col1', 'col2']]))

df\_no\_outliers = df[(z\_scores < 3).all(axis=1)] # Keep rows where Z-score is less than 3

# Alternatively, use IQR to detect outliers

Q1 = df[['col1', 'col2']].quantile(0.25)

Q3 = df[['col1', 'col2']].quantile(0.75)

IQR = Q3 - Q1

df\_no\_outliers = df[~((df[['col1', 'col2']] < (Q1 - 1.5 \* IQR)) | (df[['col1', 'col2']] > (Q3 + 1.5 \* IQR))).any(axis=1)]

# Step 5: Correct Data Types

# Convert columns to appropriate data types

df['col1'] = df['col1'].astype('float64')

df['date\_col'] = pd.to\_datetime(df['date\_col'], errors='coerce')

# Step 6: Normalize or Standardize Data (if necessary)

from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Normalization (scale values between 0 and 1)

scaler = MinMaxScaler()

df[['col1', 'col2']] = scaler.fit\_transform(df[['col1', 'col2']])

# Standardization (scale values to have mean 0 and variance 1)

standardizer = StandardScaler()

df[['col1', 'col2']] = standardizer.fit\_transform(df[['col1', 'col2']])

# Final Cleaned Data

print("\nCleaned Dataset:")

print(df.head())

# Save cleaned data to a new CSV file

df.to\_csv('cleaned\_data.csv', index=False)

**Q2. Q.2) Write a python code for Two person zero sum game without saddle point.**

import numpy as np

from scipy.optimize import linprog

# Payoff matrix for Player A (row player)

payoff\_matrix = np.array([[4, 2, 3], # Payoffs for R1

[3, 5, 1], # Payoffs for R2

[2, 4, 6]]) # Payoffs for R3

# Step 1: Solve for Player A's mixed strategy using linear programming

# Player A wants to maximize the minimum expected payoff

# We will set up a linear program to maximize Player A's expected payoff.

# Number of strategies for Player A (rows)

n\_strategies\_A = len(payoff\_matrix)

# Objective: Maximize the expected payoff

c = [-1] \* n\_strategies\_A # Coefficients for the objective function (negative for maximization)

# Constraints: The probabilities should sum to 1 (for Player A's mixed strategy)

A\_eq = np.ones((1, n\_strategies\_A)) # Each row player's probability must sum to 1

b\_eq = np.array([1])

# Bounds: The probability for each strategy should be between 0 and 1

x\_bounds = [(0, 1)] \* n\_strategies\_A

# Solving the linear programming problem for Player A's strategy

res = linprog(c, A\_eq=A\_eq, b\_eq=b\_eq, bounds=x\_bounds, method='highs')

# Player A's optimal mixed strategy

player\_A\_strategy = res.x

# Step 2: Solve for Player B's mixed strategy (minimizing Player A's expected payoff)

# Player B's goal is to minimize the expected payoff for Player A, which is equivalent to maximizing Player B's expected payoff.

# Objective: Minimize the expected payoff for Player A

c\_B = [1] \* payoff\_matrix.shape[1] # Player B's strategy vector (positive for minimization)

# Constraints: The probabilities for Player B must sum to 1

A\_eq\_B = np.ones((1, payoff\_matrix.shape[1])) # Each column player's probability must sum to 1

b\_eq\_B = np.array([1])

# Bounds: The probability for each strategy should be between 0 and 1

x\_bounds\_B = [(0, 1)] \* payoff\_matrix.shape[1]

# Solve the linear program for Player B's strategy

res\_B = linprog(c\_B, A\_eq=A\_eq\_B, b\_eq=b\_eq\_B, bounds=x\_bounds\_B, method='highs')

# Player B's optimal mixed strategy

player\_B\_strategy = res\_B.x

# Step 3: Calculate the expected payoff for Player A using their mixed strategy

expected\_payoff\_A = np.dot(player\_A\_strategy, np.dot(payoff\_matrix, player\_B\_strategy))

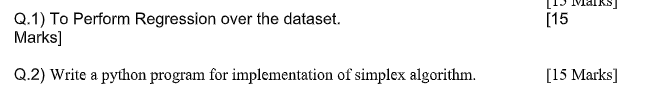
# Display the results

print("Optimal Mixed Strategy for Player A:", player\_A\_strategy)

print("Optimal Mixed Strategy for Player B:", player\_B\_strategy)

print("Expected Payoff for Player A:", expected\_payoff\_A)

**Practical 11**



**Q1.Regression**

import pandas as pd

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

# Load dataset (replace 'your\_dataset.csv' with your actual dataset file)

data = pd.read\_csv('your\_dataset.csv')

# Display the first few rows of the dataset

print("Dataset preview:")

print(data.head())

# Define features (X) and target variable (y)

# Replace 'feature\_column' and 'target\_column' with your actual column names

X = data[['feature\_column']] # Use double brackets for a DataFrame (for single feature)

y = data['target\_column']

# Split 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 Linear Regression model and fit it to the training data

model = LinearRegression()

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

# Predict target values for the testing set

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

# Print model coefficients

print("Coefficient:", model.coef\_)

print("Intercept:", model.intercept\_)

# Evaluate the model

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

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

print("Mean Squared Error:", mse)

print("R^2 Score:", r2)

# Plot the results

plt.figure(figsize=(10, 6))

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

plt.plot(X\_test, y\_pred, color='red', label='Predicted')

plt.xlabel('Feature')

plt.ylabel('Target')

plt.title('Linear Regression Model')

plt.legend()

plt.show()

**Q2.simplex algorithm**

import numpy as np

def simplex(c, A, b):

"""

Simplex algorithm implementation for Linear Programming problems.

Parameters:

c: Coefficients of the objective function (1D array).

A: Coefficients of the constraints (2D array).

b: Right-hand side values of the constraints (1D array).

Returns:

Solution vector (1D array) and optimal value.

"""

m, n = A.shape # Number of constraints and variables

tableau = np.hstack([A, np.eye(m), b.reshape(-1, 1)]) # Augmented matrix with identity for slack variables

c = np.hstack([c, np.zeros(m)]) # Add zeros for slack variable costs

# Phase 1: Set up the initial tableau

while True:

# Find the most negative value in the last row (excluding the right-hand side column)

pivot\_col = np.argmin(tableau[-1, :-1])

# If all values in the objective row are >= 0, we are done

if tableau[-1, pivot\_col] >= 0:

break

# Calculate the ratios to determine the pivot row

ratios = tableau[:-1, -1] / tableau[:-1, pivot\_col]

pivot\_row = np.argmin(ratios)

# Perform row operations to make the pivot element 1 and other elements 0 in the pivot column

pivot = tableau[pivot\_row, pivot\_col]

tableau[pivot\_row, :] /= pivot

for i in range(m + 1):

if i != pivot\_row:

tableau[i, :] -= tableau[i, pivot\_col] \* tableau[pivot\_row, :]

# Phase 2: Solve the original problem

while True:

# Find the most negative value in the objective row (c row)

pivot\_col = np.argmin(tableau[-1, :-1])

# If no negative values are present, optimal solution is found

if tableau[-1, pivot\_col] >= 0:

break

# Calculate the ratios for the pivot row selection

ratios = tableau[:-1, -1] / tableau[:-1, pivot\_col]

pivot\_row = np.argmin(ratios)

# Perform row operations to make the pivot element 1 and other elements 0 in the pivot column

pivot = tableau[pivot\_row, pivot\_col]

tableau[pivot\_row, :] /= pivot

for i in range(m + 1):

if i != pivot\_row:

tableau[i, :] -= tableau[i, pivot\_col] \* tableau[pivot\_row, :]

# Extract the optimal solution and the optimal value

solution = np.zeros(n)

for i in range(m):

if np.all(tableau[i, :-1] == 0):

continue

basic\_variable = np.argmax(tableau[i, :-1] == 1)

if basic\_variable != -1:

solution[basic\_variable] = tableau[i, -1]

optimal\_value = tableau[-1, -1]

return solution, optimal\_value

# Example problem:

# Maximize Z = 3x1 + 2x2

# Subject to:

# x1 + x2 <= 4

# 2x1 + x2 <= 5

# x1, x2 >= 0

c = np.array([3, 2]) # Coefficients of the objective function

A = np.array([[1, 1], [2, 1]]) # Coefficients of the constraints

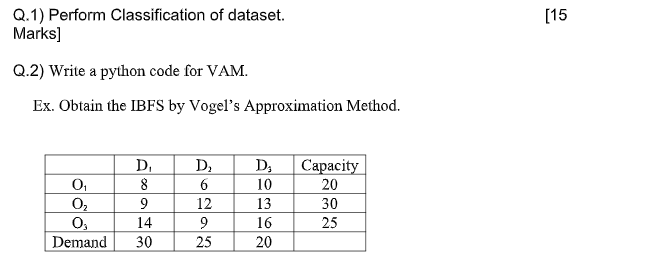
b = np.array([4, 5]) # Right-hand side values

solution, optimal\_value = simplex(c, A, b)

print("Optimal solution:", solution)

print("Optimal value of the objective function:", optimal\_value)

**Practical 12**



**Q2. IBFS by VAM**

import numpy as np

def vogels\_approximation\_method(cost\_matrix, supply, demand):

# Initialize variables

m, n = cost\_matrix.shape

allocation = np.zeros\_like(cost\_matrix)

while np.sum(supply) > 0 and np.sum(demand) > 0:

# Calculate penalties for rows

row\_penalties = []

for i in range(m):

if supply[i] > 0:

row\_sorted = sorted([(cost\_matrix[i, j], j) for j in range(n) if demand[j] > 0], key=lambda x: x[0])

if len(row\_sorted) > 1:

penalty = row\_sorted[1][0] - row\_sorted[0][0]

else:

penalty = row\_sorted[0][0]

row\_penalties.append((penalty, i))

else:

row\_penalties.append((float('inf'), i))

# Calculate penalties for columns

col\_penalties = []

for j in range(n):

if demand[j] > 0:

col\_sorted = sorted([(cost\_matrix[i, j], i) for i in range(m) if supply[i] > 0], key=lambda x: x[0])

if len(col\_sorted) > 1:

penalty = col\_sorted[1][0] - col\_sorted[0][0]

else:

penalty = col\_sorted[0][0]

col\_penalties.append((penalty, j))

else:

col\_penalties.append((float('inf'), j))

# Find row or column with highest penalty

max\_row\_penalty, row\_idx = max(row\_penalties)

max\_col\_penalty, col\_idx = max(col\_penalties)

if max\_row\_penalty >= max\_col\_penalty:

# Process row with the highest penalty

sorted\_row = sorted([(cost\_matrix[row\_idx, j], j) for j in range(n) if demand[j] > 0], key=lambda x: x[0])

min\_cost, min\_cost\_col = sorted\_row[0]

allocation[row\_idx, min\_cost\_col] = min(supply[row\_idx], demand[min\_cost\_col])

supply[row\_idx] -= allocation[row\_idx, min\_cost\_col]

demand[min\_cost\_col] -= allocation[row\_idx, min\_cost\_col]

else:

# Process column with the highest penalty

sorted\_col = sorted([(cost\_matrix[i, col\_idx], i) for i in range(m) if supply[i] > 0], key=lambda x: x[0])

min\_cost, min\_cost\_row = sorted\_col[0]

allocation[min\_cost\_row, col\_idx] = min(supply[min\_cost\_row], demand[col\_idx])

supply[min\_cost\_row] -= allocation[min\_cost\_row, col\_idx]

demand[col\_idx] -= allocation[min\_cost\_row, col\_idx]

return allocation

# Cost matrix (D1, D2, D3)

cost\_matrix = np.array([[8, 6, 10], # O1

[9, 12, 13], # O2

[14, 9, 16]]) # O3

# Supply and demand

supply = np.array([20, 30, 25])

demand = np.array([30, 25, 20])

# Call the function to get the allocation

allocation = vogels\_approximation\_method(cost\_matrix, supply, demand)

# Print the results

print("Initial Basic Feasible Solution (Allocation):")

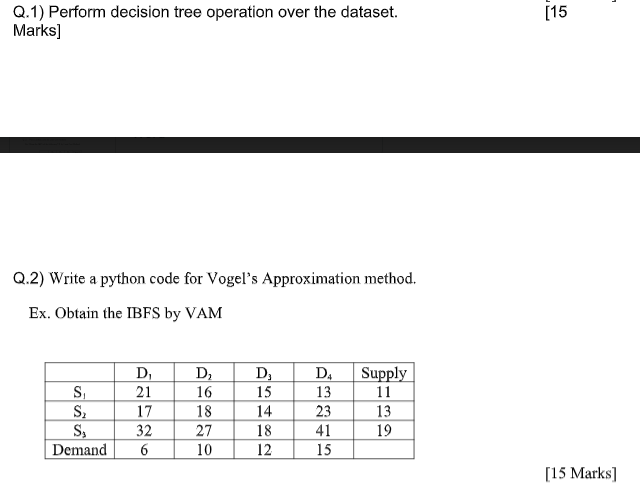
print(allocation)

# Calculate the total cost

total\_cost = np.sum(allocation \* cost\_matrix)

print(f"Total Cost: {total\_cost}")

**Practical 13**



Q1.

Q2.IBFS by vam

import numpy as np

def vogels\_approximation\_method(cost\_matrix, supply, demand):

# Initialize the allocation matrix with zeros

m, n = cost\_matrix.shape

allocation = np.zeros\_like(cost\_matrix)

while np.sum(supply) > 0 and np.sum(demand) > 0:

# Calculate penalties for each row

row\_penalties = []

for i in range(m):

if supply[i] > 0:

row\_sorted = sorted([(cost\_matrix[i, j], j) for j in range(n) if demand[j] > 0], key=lambda x: x[0])

if len(row\_sorted) > 1:

penalty = row\_sorted[1][0] - row\_sorted[0][0]

else:

penalty = row\_sorted[0][0]

row\_penalties.append((penalty, i))

else:

row\_penalties.append((float('inf'), i))

# Calculate penalties for each column

col\_penalties = []

for j in range(n):

if demand[j] > 0:

col\_sorted = sorted([(cost\_matrix[i, j], i) for i in range(m) if supply[i] > 0], key=lambda x: x[0])

if len(col\_sorted) > 1:

penalty = col\_sorted[1][0] - col\_sorted[0][0]

else:

penalty = col\_sorted[0][0]

col\_penalties.append((penalty, j))

else:

col\_penalties.append((float('inf'), j))

# Find row or column with the highest penalty

max\_row\_penalty, row\_idx = max(row\_penalties)

max\_col\_penalty, col\_idx = max(col\_penalties)

if max\_row\_penalty >= max\_col\_penalty:

# Process row with the highest penalty

sorted\_row = sorted([(cost\_matrix[row\_idx, j], j) for j in range(n) if demand[j] > 0], key=lambda x: x[0])

min\_cost, min\_cost\_col = sorted\_row[0]

allocation[row\_idx, min\_cost\_col] = min(supply[row\_idx], demand[min\_cost\_col])

supply[row\_idx] -= allocation[row\_idx, min\_cost\_col]

demand[min\_cost\_col] -= allocation[row\_idx, min\_cost\_col]

else:

# Process column with the highest penalty

sorted\_col = sorted([(cost\_matrix[i, col\_idx], i) for i in range(m) if supply[i] > 0], key=lambda x: x[0])

min\_cost, min\_cost\_row = sorted\_col[0]

allocation[min\_cost\_row, col\_idx] = min(supply[min\_cost\_row], demand[col\_idx])

supply[min\_cost\_row] -= allocation[min\_cost\_row, col\_idx]

demand[col\_idx] -= allocation[min\_cost\_row, col\_idx]

return allocation

# Given cost matrix, supply, and demand

cost\_matrix = np.array([[21, 16, 15, 13], # S1

[17, 18, 14, 23], # S2

[32, 27, 18, 41]]) # S3

supply = np.array([11, 13, 19])

demand = np.array([6, 10, 12, 15])

# Call the function to get the allocation

allocation = vogels\_approximation\_method(cost\_matrix, supply, demand)

# Print the results

print("Initial Basic Feasible Solution (Allocation):")

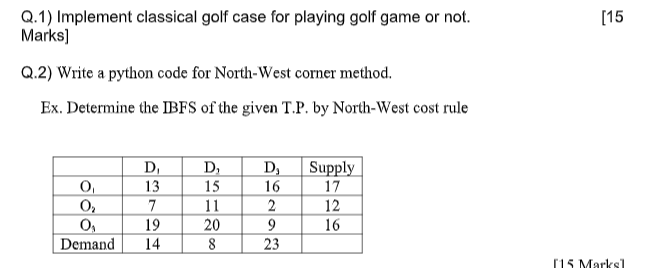
print(allocation)

# Calculate the total transportation cost

total\_cost = np.sum(allocation \* cost\_matrix)

print(f"Total Transportation Cost: {total\_cost}")

**Practical 14**



**Q2. IBFS by north west**

import numpy as np

def north\_west\_corner\_method(cost\_matrix, supply, demand):

# Initialize the allocation matrix with zeros

m, n = cost\_matrix.shape

allocation = np.zeros\_like(cost\_matrix)

# Initialize row and column pointers

i, j = 0, 0

while i < m and j < n:

# Find the minimum of supply[i] and demand[j]

allocated = min(supply[i], demand[j])

allocation[i, j] = allocated

# Update supply and demand

supply[i] -= allocated

demand[j] -= allocated

# Move to the next cell

if supply[i] == 0:

i += 1

if demand[j] == 0:

j += 1

return allocation

# Given cost matrix, supply, and demand

cost\_matrix = np.array([[13, 15, 16], # O1

[7, 11, 2], # O2

[19, 20, 9]]) # O3

supply = np.array([17, 12, 16])

demand = np.array([14, 8, 23])

# Call the function to get the allocation

allocation = north\_west\_corner\_method(cost\_matrix, supply.copy(), demand.copy())

# Print the results

print("Initial Basic Feasible Solution (Allocation):")

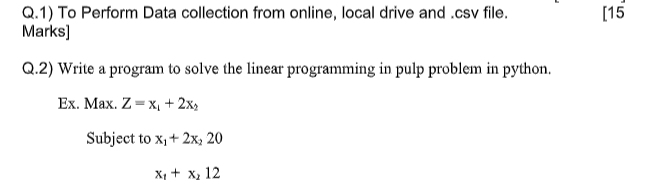
print(allocation)

# Calculate the total transportation cost

total\_cost = np.sum(allocation \* cost\_matrix)

print(f"Total Transportation Cost: {total\_cost}")

**Practical 15**



**Q2.**

# Import the PuLP library

from pulp import LpMaximize, LpProblem, LpVariable

# Step 1: Define the Linear Program (Maximization Problem)

lp\_problem = LpProblem("Maximize\_Z", LpMaximize)

# Step 2: Define the decision variables

x1 = LpVariable('x1', lowBound=0) # x1 >= 0

x2 = LpVariable('x2', lowBound=0) # x2 >= 0

# Step 3: Define the objective function

lp\_problem += x1 + 2 \* x2 # Maximize Z = x1 + 2x2

# Step 4: Define the constraints

lp\_problem += x1 + 2 \* x2 <= 20 # Constraint: x1 + 2x2 <= 20

lp\_problem += x1 + x2 <= 12 # Constraint: x1 + x2 <= 12

# Step 5: Solve the problem

lp\_problem.solve()

# Step 6: Print the results

print("Status:", lp\_problem.status)

print("Optimal value of x1:", x1.varValue)

print("Optimal value of x2:", x2.varValue)

print("Maximum Z value:", lp\_problem.objective.value())