```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score, classification_report
from sklearn import tree
```

--------- Data Collection ---------

```
# 1. Online Data Collection using Seaborn (Example: 'mpg' dataset)
online_data = sns.load_dataset('mpg').dropna()  # Loading mpg dataset from Seaborn (fuel consumption data)
print("Online Data Head:\n", online_data.head())

# 2. Local Data Collection (Assume 'car_prices.csv' is a file on local drive)
# Example CSV file structure: 'price', 'mileage', 'year', 'horsepower'
# local_data = pd.read_csv('car_prices.csv')  # Uncomment this line if you have the CSV file locally
# For demonstration purposes, let’s create synthetic data for local collection
local_data = pd.DataFrame({
    'price': np.random.randint(5000, 30000, 50),
    'mileage': np.random.uniform(10, 30, 50),
    'year': np.random.randint(2000, 2022, 50),
    'horsepower': np.random.randint(100, 300, 50)
})

# 3. Example CSV File loading (Ensure you have a CSV file or use the generated `local_data`)
# file_data = pd.read_csv('path_to_your_file.csv') # Uncomment if using your own dataset
```

--------- Data Visualization ---------

```
# 1. 2D Scatter Plot (Price vs Mileage)
plt.figure(figsize=(8,6))
plt.scatter(local_data['mileage'], local_data['price'], c='blue', alpha=0.5)
plt.title('Price vs Mileage')
plt.xlabel('Mileage (mpg)')
plt.ylabel('Price ($)')
plt.show()

# 2. 3D Scatter Plot (Price, Mileage, Horsepower)
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(local_data['mileage'], local_data['horsepower'], local_data['price'], c='red')
ax.set_xlabel('Mileage (mpg)')
ax.set_ylabel('Horsepower')
ax.set_zlabel('Price ($)')
plt.title('3D Scatter Plot: Price, Mileage, Horsepower')
plt.show()
```

```python
# --------- Regression ---------

# Linear Regression: Predict Price based on Mileage and Horsepower
X_reg = local_data[['mileage', 'horsepower']]  # Features
y_reg = local_data['price']                    # Target: Price

# Splitting the data for training and testing
X_train, X_test, y_train, y_test = train_test_split(X_reg, y_reg, test_size=0.2, random_state=42)

# Linear Regression Model
reg_model = LinearRegression()
reg_model.fit(X_train, y_train)

# Predictions
y_pred_reg = reg_model.predict(X_test)

# Performance Metrics
print("Regression - Mean Squared Error:", mean_squared_error(y_test, y_pred_reg))
print("Regression - R-squared:", r2_score(y_test, y_pred_reg))

# --------- Classification (Decision Tree) ---------

# Classifying high vs low price (Example: Above or Below median price)
local_data['price_class'] = (local_data['price'] > local_data['price'].median()).astype(int)

X_class = local_data[['mileage', 'horsepower']]  # Features
y_class = local_data['price_class']              # Target: Binary classification (high/low price)

# Splitting the data for training and testing
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(X_class, y_class, test_size=0.2,
random_state=42)

# Decision Tree Classifier
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train_class, y_train_class)

# Predictions
y_pred_class = dt_model.predict(X_test_class)

# Performance Metrics for Classification
print("\nClassification - Accuracy:", accuracy_score(y_test_class, y_pred_class))
print("Classification - Classification Report:\n", classification_report(y_test_class, y_pred_class))

# Visualize Decision Tree
plt.figure(figsize=(12,8))
tree.plot_tree(dt_model, feature_names=['Mileage', 'Horsepower'], class_names=['Low Price', 'High Price'],
filled=True)
plt.title('Decision Tree Classifier - Price Classification')
plt.show()
```

Implement a classical golf case for playing golf game or not.

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn import tree

# --------- 1. Data Preparation ---------

# Example dataset: Features include Outlook, Temperature, Humidity, Wind and the Target is PlayGolf (Yes/No)
data = {
    'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rain', 'Rain', 'Rain', 'Overcast', 'Sunny', 'Sunny', 'Rain', 'Sunny', 'Overcast',
'Overcast', 'Rain'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild', 'Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Low', 'Low', 'Low', 'High', 'Low', 'Low', 'High', 'Low', 'Low', 'High'],
    'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Weak', 'Strong', 'Weak', 'Strong',
'Weak', 'Strong'],
    'PlayGolf': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']
}
# Convert the dataset into a pandas DataFrame
df = pd.DataFrame(data)
# --------- 2. Data Preprocessing ---------
# Convert categorical variables to numerical values (encoding)
df_encoded = df.copy()
df_encoded['Outlook'] = df_encoded['Outlook'].map({'Sunny': 0, 'Overcast': 1, 'Rain': 2})
df_encoded['Temperature'] = df_encoded['Temperature'].map({'Hot': 0, 'Mild': 1, 'Cool': 2})
df_encoded['Humidity'] = df_encoded['Humidity'].map({'High': 0, 'Low': 1})
df_encoded['Wind'] = df_encoded['Wind'].map({'Weak': 0, 'Strong': 1})
df_encoded['PlayGolf'] = df_encoded['PlayGolf'].map({'No': 0, 'Yes': 1})
# --------- 3. Splitting Data ---------
X = df_encoded.drop('PlayGolf', axis=1)  # Features
y = df_encoded['PlayGolf']  # Target
# Split 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)
# --------- 4. Train the Decision Tree Classifier ---------
# Create and fit the Decision Tree Classifier
clf = DecisionTreeClassifier(random_state=42)
clf.fit(X_train, y_train)
# --------- 5. Make Predictions ---------
y_pred = clf.predict(X_test)
# --------- 6. Model Evaluation ---------
# Accuracy and Classification Report
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
# --------- 7. Visualize the Decision Tree ---------
plt.figure(figsize=(12,8))
tree.plot_tree(clf, feature_names=X.columns, class_names=['No', 'Yes'], filled=True)
plt.title("Decision Tree for Playing Golf Prediction")
plt.show()
```

Create a small stock market analysis for bull or bear for a stock in NSE and BSE.

```
import yfinance as yf
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 LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
# --------- 1. Data Collection ---------
# Download historical stock data from Yahoo Finance for a stock (e.g., TATAMOTORS) from NSE/BSE
stock_symbol = 'TATAMOTORS.NS'  # TATAMOTORS in NSE (use the respective stock symbol for BSE or other stocks)
data = yf.download(stock_symbol, start="2023-01-01", end="2024-01-01")
# Show the data head
print("Stock Data Head:\n", data.head())
# --------- 2. Data Preprocessing ---------
# Create moving averages (short-term and long-term) and calculate daily returns
data['Short_MA'] = data['Close'].rolling(window=20).mean()  # 20-day moving average
data['Long_MA'] = data['Close'].rolling(window=50).mean()  # 50-day moving average
data['Daily_Return'] = data['Close'].pct_change()  # Daily percentage return
# Define "Bull" and "Bear" as a classification based on moving averages
# Bull Market: Short MA > Long MA (uptrend), Bear Market: Short MA < Long MA (downtrend)
data['Market_State'] = np.where(data['Short_MA'] > data['Long_MA'], 1, 0)  # 1: Bull, 0: Bear
# Drop rows with NaN values (due to rolling window)
data = data.dropna()
# --------- 3. Visualization ---------
# Plot Closing Price, Short and Long Moving Averages
plt.figure(figsize=(12, 6))
plt.plot(data['Close'], label='Closing Price', color='blue')
plt.plot(data['Short_MA'], label='20-Day MA', color='green')
plt.plot(data['Long_MA'], label='50-Day MA', color='red')
plt.title(f'{stock_symbol} - Bull vs Bear Market')
plt.legend()
plt.show()
# --------- 4. Prepare Data for Classification ---------
# Features: Short MA, Long MA, Daily Return
X = data[['Short_MA', 'Long_MA', 'Daily_Return']]
# Target: Market State (1: Bull, 0: Bear)
y = data['Market_State']
# 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)
# --------- 5. Logistic Regression Model ---------
# Fit Logistic Regression Model
model = LogisticRegression()
model.fit(X_train, y_train)
# Predict the market state (Bull or Bear)
y_pred = model.predict(X_test)
# --------- 6. Evaluate the Model ---------
# Evaluate the model using Accuracy and Classification Report
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
# --------- 7. Model Visualization ---------
# Visualize the model's prediction vs actual
plt.figure(figsize=(12, 6))
plt.plot(data.index[-len(y_test):], y_test, label='Actual Market State', color='blue', alpha=0.6)
plt.plot(data.index[-len(y_test):], y_pred, label='Predicted Market State', color='red', linestyle='--')
plt.title(f'{stock_symbol} - Bull vs Bear Market Prediction')
```

```
plt.legend()
plt.show()
```

To Perform Data cleaning Operation over the data collected simple 5-6 line program

```
import pandas as pd
# Example DataFrame (replace this with your actual data)
data = pd.DataFrame({
    'Column1': [1, 2, 3, 4, None, 6, 7],
    'Column2': [None, 2, 3, 4, 5, 6, 7],
    'Column3': ['A', 'B', 'C', 'D', 'E', 'F', 'F']
})
# 1. Handle missing values (fill with the mean for numeric columns)
data['Column1'].fillna(data['Column1'].mean(), inplace=True)
data['Column2'].fillna(data['Column2'].mean(), inplace=True)
# 2. Remove duplicates
data.drop_duplicates(inplace=True)
# 3. Convert data types (e.g., Column3 should be categorical)
data['Column3'] = data['Column3'].astype('category')
# 4. Check data after cleaning
print(data)
```

Practical -1 Implementation of simplex algorithm

```
from scipy.optimize import linprog
# Objective function coefficients
c = [-3, -5]  # Maximization of 3x + 5y, so we use -3 and -5 for minimization
# Coefficients of the inequality constraints (Ax <= b)
A = [[1, 0], [0, 2], [3, 2]]
b = [4, 12, 18]
# Boundaries of the decision variables (x, y >= 0)
x_bounds = (0, None)
y_bounds = (0, None)
# Solving the linear programming problem using the simplex method
result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method='simplex')
# Output the results
print(f"Optimal value: {-result.fun}")
print(f"Decision variables: {result.x}")
```

OUTPUT:

Optimal value: 15.0
Decision variables: [2. 6.]

Practical -2 Linear Programming using PuLP in Python

```
import pulp
# Create a linear programming problem (Maximization)
lp_problem = pulp.LpProblem("Maximize_Profit", pulp.LpMaximize)
# Decision variables
x = pulp.LpVariable('x', lowBound=0)  # x >= 0
y = pulp.LpVariable('y', lowBound=0)  # y >= 0
# Objective function
lp_problem += 3 * x + 5 * y, "Maximize profit"
# Constraints
lp_problem += x <= 4
lp_problem += 2 * y <= 12
lp_problem += 3 * x + 2 * y <= 18
# Solve the problem
lp_problem.solve()
# Output the results
print(f"Status: {pulp.LpStatus[lp_problem.status]}")
print(f"Optimal value: {pulp.value(lp_problem.objective)}")
print(f"Decision variables: x = {x.varValue}, y = {y.varValue}")
```

OUTPUT:

Status: Optimal
Optimal value: 15.0
Decision variables: x = 2.0, y = 6.0

Practical -3 LPP by calling solve() method

```
from scipy.optimize import linprog
# Objective function coefficients (for minimization)
c = [-1, -2]  # Maximization, so we use negative values for minimization
# Coefficients for inequality constraints (Ax <= b)
A = [[2, 1], [1, 1], [1, 0]]
b = [20, 16, 8]
# Boundaries for decision variables (x, y >= 0)
bounds = [(0, None), (0, None)]
# Solve the LPP using the default method (Simplex)
solution = linprog(c, A_ub=A, b_ub=b, bounds=bounds, method="simplex")
# Output the results
print(f"Optimal value: {-solution.fun}")
print(f"Decision variables: {solution.x}")
```

OUTPUT:

```
Optimal value: 14.0
Decision variables: [4.  8.]
```

Practical -4 PP model by declaring decision variables, list, objective function, and constraints

```
from pulp import LpMaximize, LpProblem, LpVariable
model = LpProblem("Maximize_Profit", LpMaximize)
x1 = LpVariable("x1", lowBound=0)  # x1 >= 0
x2 = LpVariable("x2", lowBound=0)  # x2 >= 0
# Objective function
model += 40 * x1 + 30 * x2, "Maximize Revenue"
# Constraints
model += 2 * x1 + x2 <= 60, "Material Constraint"
model += x1 + x2 <= 40, "Labor Constraint"
model += x1 <= 20, "Production Constraint"
# Solve the model
model.solve()
# Output results
print(f"Status: {pulp.LpStatus[model.status]}")
print(f"Optimal value: {pulp.value(model.objective)}")
print(f"x1 = {x1.varValue}, x2 = {x2.varValue}")
```

OUTPUT:

```
Status: Optimal
Optimal value: 1600.0
x1 = 20.0, x2 = 20.0
```

Practical -5 North-West corner method

```
def north_west_corner_method(c, b, A):
    # Initialize the transportation table
    T = [[0 for _ in range(len(b))] for _ in range(len(c))]

    # Initialize the remaining supply and demand
    remaining_supply = [c[i] for i in range(len(c))]
    remaining_demand = [b[i] for i in range(len(b))]

    # Start with the northwest corner
    i, j = 0, 0

    while i < len(c) and j < len(b):
        # Find the minimum of the remaining supply and demand
        min_remaining = min(remaining_supply[i], remaining_demand[j])

        # Update the transportation table
        T[i][j] = min_remaining

        # Update the remaining supply and demand
        remaining_supply[i] -= min_remaining
        remaining_demand[j] -= min_remaining

        # Move to the next cell
        if remaining_supply[i] == 0 and remaining_demand[j] == 0:
            i += 1
            j += 1
        elif remaining_supply[i] == 0:
            i += 1
        elif remaining_demand[j] == 0:
            j += 1

    return T

# Example input
supply = [20, 30, 25]
demand = [10, 40, 25]
cost = [[2, 3, 1], [5, 4, 8], [5, 6, 8]]

# Call the function
result = north_west_corner_method(supply, demand, cost)

# Print the result
for row in result:
    print(row)
```

OUTPUT

```
[10, 10, 0]
[0, 30, 0]
[0, 0, 25]
```

Practical -6 Least Cost Method

```
def least_cost_method(supply, demand, cost):
    rows = len(supply)
    cols = len(demand)
    allocation = [[0] * cols for _ in range(rows)]

    i = 0
    j = 0

    while i < rows and j < cols:
        if supply[i] < demand[j]:
            allocation[i][j] = supply[i]
            demand[j] -= supply[i]
            supply[i] = 0
            i += 1
        else:
            allocation[i][j] = demand[j]
            supply[i] -= demand[j]
            demand[j] = 0
            j += 1

    return allocation

def calculate_total_cost(allocation, cost):
    total_cost = 0
    for i in range(len(allocation)):
        for j in range(len(allocation[0])):
            total_cost += allocation[i][j] * cost[i][j]
    return total_cost

# Example usage
supply = [20, 30, 25]  # Supply for each source
demand = [30, 25, 20]  # Demand for each destination
cost = [[8, 6, 10],    # Cost matrix
        [9, 12, 13],
        [14, 9, 16]]

allocation = least_cost_method(supply, demand, cost)

print("Allocation Matrix:")
for row in allocation:
    print(row)

total_cost = calculate_total_cost(allocation, cost)
print(f"\nTotal Transportation Cost: {total_cost}")
```

OUTPUT:

```
Allocation Matrix:
[20, 0, 0]
[10, 20, 0]
[0, 5, 20]

Total Transportation Cost: 855
```

Practical -7 VAM

```
def vogels_approximation_method(supply, demand, cost):
    rows = len(supply)
    cols = len(demand)
    allocation = [[0] * cols for _ in range(rows)]

    while sum(supply) > 0 and sum(demand) > 0:
        # Calculate row and column penalties
        row_penalty = [max(cost[i]) - sorted(cost[i])[0] if supply[i] > 0 else float('inf') for i in
range(rows)]
        col_penalty = [max([cost[i][j] for i in range(rows)]) - min([cost[i][j] for i in range(rows)]) if
demand[j] > 0 else float('inf') for j in range(cols)]

        # Find the row/column with the highest penalty
        if min(row_penalty) <= min(col_penalty):
            i = row_penalty.index(min(row_penalty))
            j = cost[i].index(min(cost[i]))
        else:
            j = col_penalty.index(min(col_penalty))
            i = min(range(rows), key=lambda x: cost[x][j] if supply[x] > 0 else float('inf'))

        # Allocate as much as possible
        allocation_amount = min(supply[i], demand[j])
        allocation[i][j] = allocation_amount
        supply[i] -= allocation_amount
        demand[j] -= allocation_amount

    return allocation

def calculate_total_cost(allocation, cost):
    total_cost = 0
    for i in range(len(allocation)):
        for j in range(len(allocation[0])):
            total_cost += allocation[i][j] * cost[i][j]
    return total_cost

# Example usage
supply = [20, 30, 25]  # Supply for each source
demand = [30, 25, 20]  # Demand for each destination
cost = [[8, 6, 10],    # Cost matrix
        [9, 12, 13],
        [14, 9, 16]]

allocation = vogels_approximation_method(supply, demand, cost)

print("Allocation Matrix:")
for row in allocation:
    print(row)
total_cost = calculate_total_cost(allocation, cost)
print(f"\nTotal Transportation Cost: {total_cost}")
```

OUTPUT:
Allocation Matrix:
[0, 20, 0]
[30, 0, 0]
[0, 5, 20] Total Transportation Cost: 755

Practical -8 Algebraic method for 2 by 2 game

```
import numpy as np
def solve_2x2_game(payoff_matrix):
    # Get elements from the payoff matrix
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]

    # Solving for player A's strategy
    # Player A's probabilities: p1 for row 1 (A1) and p2 for row 2 (A2)
    # To find the optimal strategies, solve:
    # p1 * a11 + p2 * a21 = v (expected payoff)
    # p1 * a12 + p2 * a22 = v (expected payoff)

    # Solve for p1 and p2 using the algebraic method
    v = (a11 * a22 - a12 * a21) / (a11 + a22 - a12 - a21)
    p1 = (a22 - a12) / (a11 + a22 - a12 - a21)
    p2 = 1 - p1  # Since p1 + p2 = 1

    # Solving for player B's strategy
    q1 = (a22 - a21) / (a11 + a22 - a12 - a21)
    q2 = 1 - q1  # Since q1 + q2 = 1

    return {
        "Player A's strategy": (p1, p2),
        "Player B's strategy": (q1, q2),
        "Value of the game (v)": v
    }

# Example usage
payoff_matrix = [[3, 2],  # Payoffs for player A
                 [1, 4]]

result = solve_2x2_game(payoff_matrix)

print("Optimal Strategies:")
print(result)
```

OUTPUT:
Optimal Strategies:
{"Player A's strategy": (0.5, 0.5), "Player B's strategy":
(0.75, 0.25), 'Value of the game (v)': 2.5}

Practical -9 Graphical method for solving 2 by 2 game

```
import numpy as np
import matplotlib.pyplot as plt
def graphical_method(payoff_matrix):
    # Get payoffs from the matrix
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]
    # Create an array of probabilities for Player A's first strategy (p1)
    p1_values = np.linspace(0, 1, 100)
    # Calculate expected payoffs for Player A's strategies A1 and A2
    payoff_A1 = p1_values * a11 + (1 - p1_values) * a12
    payoff_A2 = p1_values * a21 + (1 - p1_values) * a22
    # Find the intersection point (where the difference between payoffs is minimum)
    optimal_p1_index = np.argmin(abs(payoff_A1 - payoff_A2))
    optimal_p1 = p1_values[optimal_p1_index]
    optimal_value = payoff_A1[optimal_p1_index]
    # Plot the payoffs
    plt.plot(p1_values, payoff_A1, label="Payoff for A1")
    plt.plot(p1_values, payoff_A2, label="Payoff for A2")
    # Mark the optimal point
    plt.plot(optimal_p1, optimal_value, 'ro', label=f'Optimal p1: {optimal_p1:.2f}')

    # Add labels and legend
    plt.xlabel("Probability of Player A playing A1 (p1)")
    plt.ylabel("Expected Payoff")
    plt.title("Graphical Method for 2x2 Game")
    plt.legend()
    plt.grid(True)
    plt.show()
    # Calculate the optimal probability for Player B (complementary)
    optimal_p2 = 1 - optimal_p1
    return {
        "Player A's optimal strategy (p1)": optimal_p1,
        "Player A's complementary strategy (p2)": optimal_p2,
        "Value of the game (v)": optimal_value
    }
# Example usage
payoff_matrix = [[3, 2],  # Payoffs for Player A
                 [1, 4]]
result = graphical_method(payoff_matrix)
print("Optimal Strategy and Game Value:")
print(result)
```

OUTPUT:

Practical -10 Game without saddle point

```
def solve_game_without_saddle_point(payoff_matrix):
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]
    # Solve for Player A's strategy probabilities (p1 for A1, p2 for A2)
    # p1 * a11 + (1 - p1) * a12 = p1 * a21 + (1 - p1) * a22
    # Simplify the equation: (a11 - a12) * p1 + a12 = (a21 - a22) * p1 + a22
    p1 = (a22 - a12) / ((a11 - a12) - (a21 - a22))
    p2 = 1 - p1
    # Solve for Player B's strategy probabilities (q1 for B1, q2 for B2)
    # Similar logic as for Player A
    q1 = (a22 - a21) / ((a11 - a21) - (a12 - a22))
    q2 = 1 - q1
    return {
        "Player A's strategy": (p1, p2),
        "Player B's strategy": (q1, q2),
    }
# Example usage
payoff_matrix = [[1, 4],  # Payoff for Player A
                 [3, 2]]
result = solve_game_without_saddle_point(payoff_matrix)
print("Optimal Mixed Strategies:")
print(result)
```

OUTPUT:
Optimal Mixed Strategies:
{"Player A's strategy": (0.5, 0.5), "Player B's strategy": (0.25, 0.75)}

Practical -11 Two person zero sum game with saddle point

```
def find_saddle_point(payoff_matrix):
    # Find the minimum of each row (Player A's best worst-case scenario)
    row_minima = [min(row) for row in payoff_matrix]

    # Find the maximum of each column (Player B's worst best-case scenario)
    column_maxima = [max(col) for col in zip(*payoff_matrix)]

    # The saddle point exists if any of the row minima equals any of the column maxima
    saddle_points = []

    for i in range(len(payoff_matrix)):
        for j in range(len(payoff_matrix[i])):
            if payoff_matrix[i][j] == max(row_minima) and payoff_matrix[i][j] ==
min(column_maxima):
                saddle_points.append((i, j, payoff_matrix[i][j]))

    if saddle_points:
        return saddle_points
    else:
        return "No saddle point"

# Example usage
payoff_matrix = [[3, 2],  # Payoff for Player A
                 [1, 4]]

result = find_saddle_point(payoff_matrix)
```

```
if result != "No saddle point":
    print("Saddle Point(s) Found:")
    for point in result:
        print(f"Saddle point at row {point[0]+1}, column {point[1]+1} with value {point[2]}")
else:
    print(result)
```

OUTPUT:
No saddle point

Practical -12 Two person zero sum game without saddle point

```
def solve_zero_sum_game(payoff_matrix):
    # Extract the payoff matrix
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]

    # Calculate Player A's mixed strategy probabilities
    p1 = (a22 - a12) / ((a11 - a12) + (a22 - a21))
    p2 = 1 - p1

    # Calculate Player B's mixed strategy probabilities
    q1 = (a22 - a21) / ((a11 - a21) + (a22 - a12))
    q2 = 1 - q1

    # Calculate the value of the game
    v = p1 * a11 + p2 * a21

    return {
        "Player A's strategy": (p1, p2),
        "Player B's strategy": (q1, q2),
        "Value of the game": v
    }

# Example usage
payoff_matrix = [[1, 4],  # Payoff for Player A
                 [3, 2]]

result = solve_zero_sum_game(payoff_matrix)
print("Optimal Mixed Strategies and Game Value:")
print(f"Player A's strategy: {result['Player A\'s strategy']}")
print(f"Player B's strategy: {result['Player B\'s strategy']}")
print(f"Value of the game: {result['Value of the game']:.2f}")
```

OUTPUT:
Optimal Mixed Strategies and Game Value:
Player A's strategy: (0.5, 0.5)
Player B's strategy: (0.25, 0.75)
Value of the game: 2.0