Visvesvaraya Technological University

“Jnana Sangama”, VTU-Campus, Belagavi-590018

2024 – 2025

LABORATORY JOURNAL

OF

ALGORITHMS & ARTIFICIAL INTELLIGENCE

LABORATORY (MCSL106)

Submitted By:

Swapnadeep Kapuri

2VX24SCS17

M.Tech in CSE

1st Sem

Department of Computer Science and Engineering

Visvesvaraya Technological University

"Jnana Sangama", VTU-Campus, Belagavi-590018

Department of Computer Science and Engineering

<u>Certificate</u>

This is to certify that **Mr. Swapnadeep Kapuri (2VX24SCS17)** has satisfactorily completed the Laboratory Experiments for **Algorithms and AI Laboratory (MCSL106)** during the academic year 2024-25.

Faculty In Charge	Course Coordinator	Chairperson
Dr. Shanmukhappa Angadi	**Dr. Shanmukhappa Angadi**	**Dr S.L.Deshpande**
Dr. Rashmi R Rachh		

Name of Examiner and Signature

1.__________________ __________________

2.__________________ __________________

INDEX

Program 1: Implement a simple linear regression algorithm to predict a continuous target variable based on a given data set.

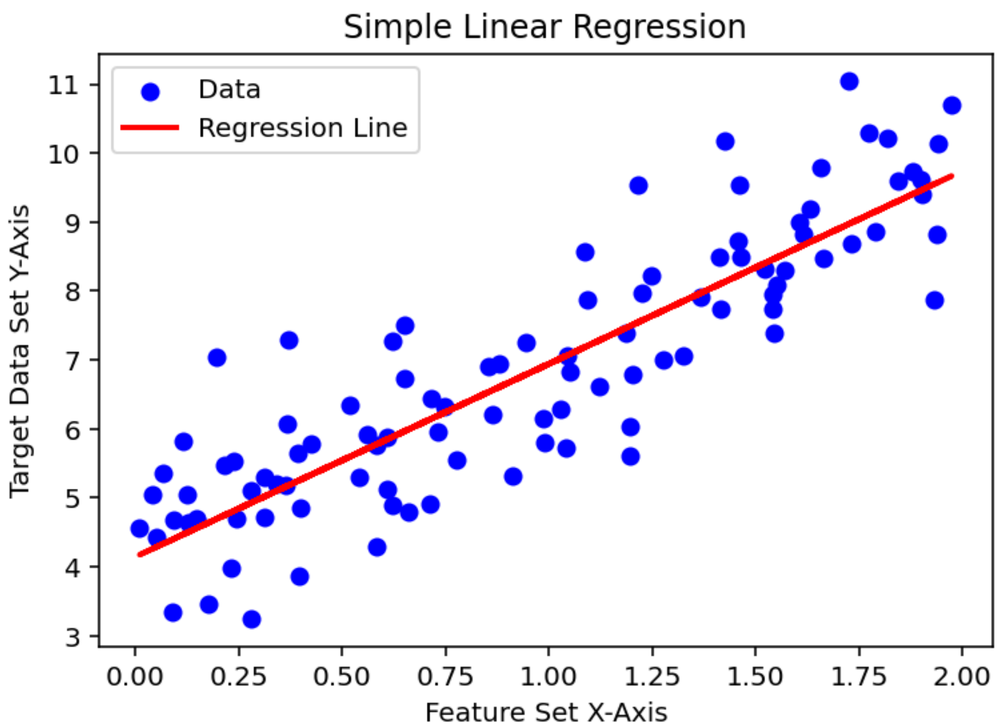
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Generate synthetic data
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# 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)

# Train the model
model = LinearRegression()
model.fit(X_train, y_train)

# Print model parameters
print(f"Intercept: {model.intercept_[0]}")
print(f"Coefficient: {model.coef_[0][0]}")

# Make predictions
y_pred = model.predict(X_test)

# Compute mean squared error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")

# Plot the results
plt.scatter(X, y, color='blue', label='Data')
plt.plot(X, model.predict(X), color='red', linewidth=2, label='Regression Line')
plt.xlabel('Feature Set X-Axis')
plt.ylabel('Target Data Set Y-Axis')
plt.title('Simple Linear Regression')
plt.legend()
plt.show()
```

Output:

Intercept: 4.142913319458566

Coefficient: 2.7993236574802762

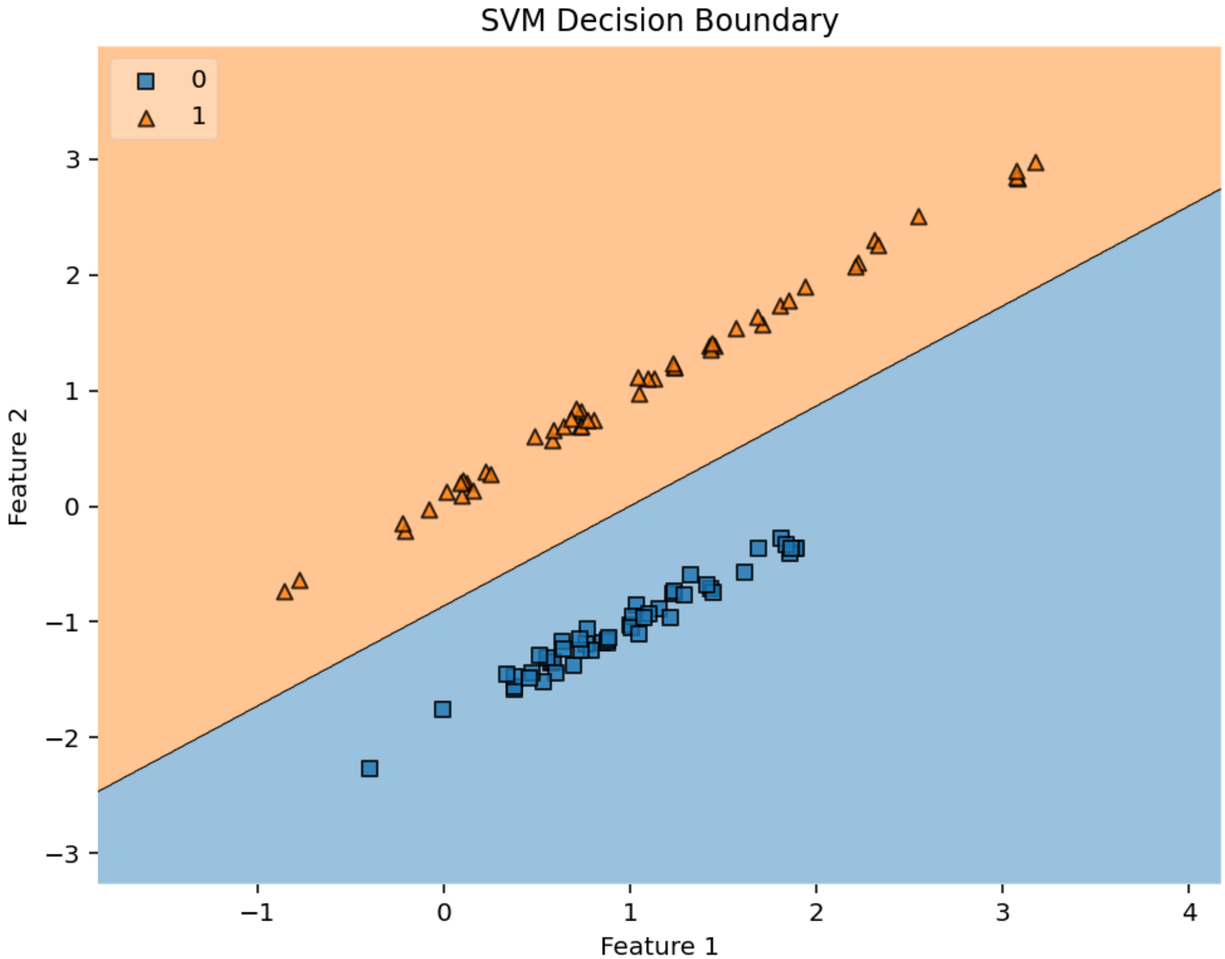
Mean Squared Error: 0.6536995137170021



Program 2: Develop a program to implement a Support Vector Machine for binary classification. Use a sample data set and visualize the decision boundary.

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from mlxtend.plotting import plot_decision_regions

# Generate a synthetic dataset
X, y = datasets.make_classification(n_samples=100, n_features=2,
                                    n_classes=2, n_clusters_per_class=1,
                                    n_redundant=0, random_state=42)

# Split 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)

# Train an SVM classifier
svm = SVC(kernel='linear', C=1.0, random_state=42)
svm.fit(X_train, y_train)

# Plot decision boundary
plt.figure(figsize=(8, 6))
plot_decision_regions(X, y, clf=svm, legend=2)
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("SVM Decision Boundary")
plt.show()

# Print model accuracy
accuracy = svm.score(X_test, y_test)
print(f"\nModel Accuracy: {accuracy * 100:.2f}% ")
```

Output:



Model Accuracy: 100.00%

Program 3: Develop a simple case-based reasoning system that stores instances of the past cases. Implement a retrieval method to find the most similar cases and make predictions based on them.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler

# Generate a sample dataset (100 past cases)
X, y = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Normalize for better distance calculation

# Case-Based Reasoning (CBR) Function
def retrieve_similar_cases(X_cases, y_cases, new_case, n_similar=3):
    """Finds the most similar past cases using Euclidean distance."""
    nbrs = NearestNeighbors(n_neighbors=n_similar, metric='euclidean').fit(X_cases)
    distances, indices = nbrs.kneighbors(new_case)  # FIXED: Removed extra brackets
    return indices[0], distances[0]

# New case to predict
new_case = np.array([[0.5]])  # Example new input
new_case_scaled = scaler.transform(new_case)  # Keeps it 2D

# Retrieve most similar cases
similar_indices, similar_distances = retrieve_similar_cases(X_scaled, y, new_case_scaled)
similar_cases = X[similar_indices]  # Original scale for visualization
predicted_value = np.mean(y[similar_indices])  # Average output of similar cases

# Output results
print(f"New Case Input: {new_case.flatten()[0]:.2f}")
print(f"Most Similar Cases (X values): {similar_cases.flatten()}")
print(f"Corresponding Outputs (y values): {y[similar_indices]}")
print(f"Predicted Output: {predicted_value:.2f}")
```

 # Plot cases and prediction

```
plt.scatter(X, y, label="Past Cases", color="blue", alpha=0.6)
plt.scatter(similar_cases, y[similar_indices], label="Similar Cases", color="red", marker="s", s=100)
plt.scatter(new_case, predicted_value, label="Predicted Case", color="green", marker="*", s=150)
plt.xlabel("Feature Value")
plt.ylabel("Output Value")
plt.title("Case-Based Reasoning Prediction")
plt.legend()
plt.show()
```

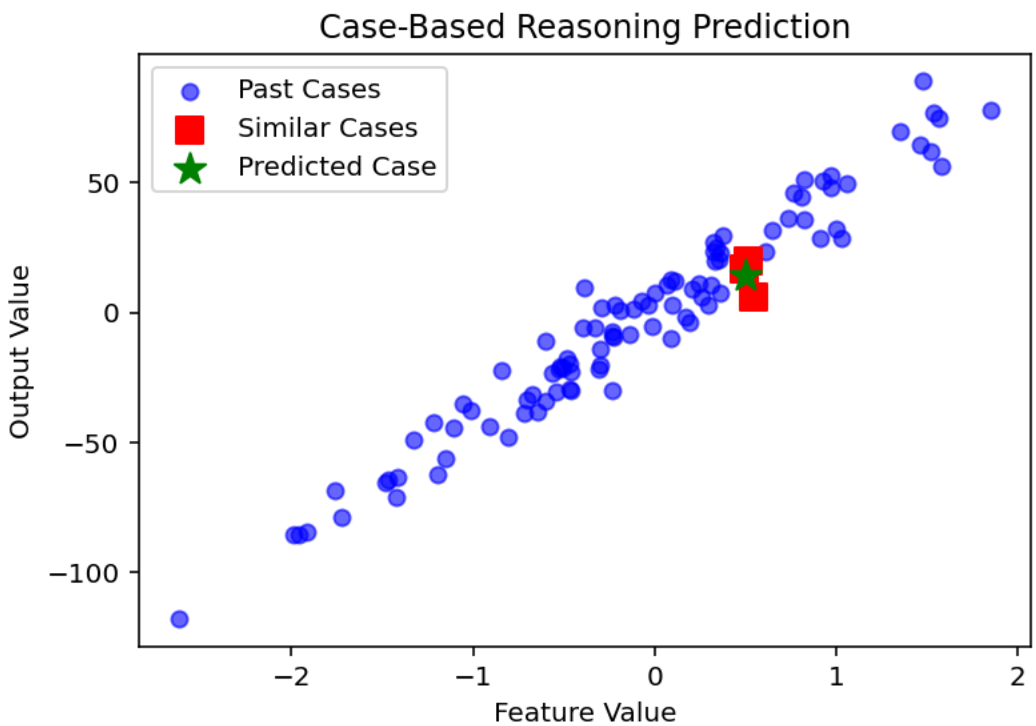
Output:

New Case Input: 0.50

Most Similar Cases (X values): [0.49671415 0.51326743 0.54256004]

Corresponding Outputs (y values): [16.77823077 20.05162924 5.91200699]

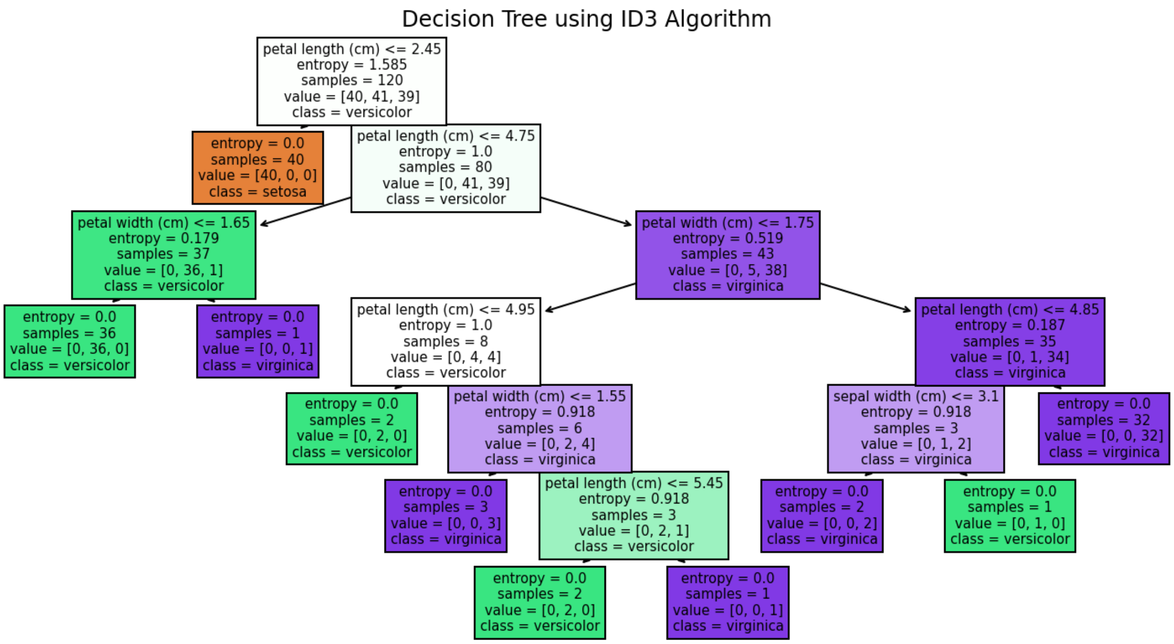
Predicted Output: 14.25



Program 4: Write a program to demonstrate the ID3 decision tree algorithm using an appropriate dataset for classification.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load dataset (Iris dataset for classification)
iris = load_iris()
X, y = iris.data, iris.target
feature_names = iris.feature_names
class_names = iris.target_names

# Split 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)

# Train Decision Tree Classifier using ID3 (entropy-based)
clf = DecisionTreeClassifier(criterion='entropy', random_state=42)
clf.fit(X_train, y_train)

# Plot the decision tree
plt.figure(figsize=(12, 6))
plot_tree(clf, feature_names=feature_names, class_names=class_names, filled=True)
plt.title("Decision Tree using ID3 Algorithm")
plt.show()

# Print model accuracy
accuracy = clf.score(X_test, y_test)
print(f"\nModel Accuracy: {accuracy * 100:.2f}% ")

# Print feature importance
feature_importance = pd.DataFrame({'Feature': feature_names, 'Importance': clf.feature_importances_})
print("\nFeature Importance:\n", feature_importance.sort_values(by='Importance', ascending=False))
```

Output:



Model Accuracy: 100.00%

Feature Importance:

	Feature	Importance
2	petal length (cm)	0.895406
3	petal width (cm)	0.090107
1	sepal width (cm)	0.014487
0	sepal length (cm)	0.000000

Program 5: Build an Artificial Neural Network by implementing the Backpropagation algorithm and test it with suitable datasets.

(**Note: TensorFlow module is still not compatible with Python 3.12.8!!! Use Python 3.10/3.11 or use Google Colab to work the below code)**

```
import numpy as np
from tensorflow import keras
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler

# Load dataset (Iris dataset for classification)
iris = load_iris()
X, y = iris.data, iris.target

# One-hot encode target labels (Fix: Use sparse_output instead of sparse)
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y.reshape(-1, 1))

# Split 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)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Build the Neural Network model
model = keras.Sequential([
    keras.layers.Dense(10, activation='relu', input_shape=(X_train.shape[1],)),  # Hidden Layer 1
    keras.layers.Dense(10, activation='relu'),  # Hidden Layer 2
    keras.layers.Dense(y.shape[1], activation='softmax')  # Output Layer (3 classes)
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=10, validation_data=(X_test, y_test), verbose=1)

# Evaluate model performance
```

```python
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"\nModel Accuracy on Test Data: {test_accuracy * 100:.2f}% ")

# Predict on test data
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Display some predictions
print("\nSample Predictions (True vs Predicted Labels):")
print(np.vstack((y_test_classes[:10], y_pred_classes[:10])).T)
```

Output:

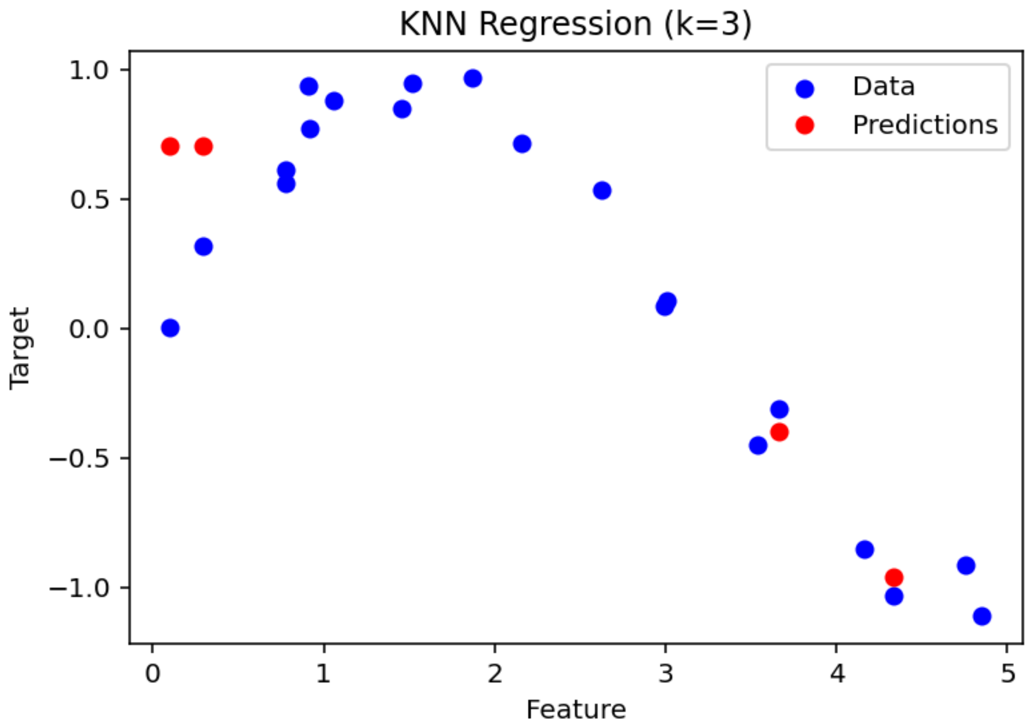
```
Model Accuracy on Test Data: 100.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step

Sample Predictions (True vs Predicted Labels):
[[1 1]
 [0 0]
 [2 2]
 [1 1]
 [1 1]
 [0 0]
 [1 1]
 [2 2]
 [1 1]
 [1 1]]
```

Program 6: Implement a KNN algorithm for regression tasks instead of classification. Use a small dataset, and predict continuous values based on the average of the nearest neighbors.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
# Generate a small dataset
np.random.seed(42)
X = np.sort(5 * np.random.rand(20, 1), axis=0) # Feature
y = np.sin(X).ravel() + np.random.normal(0, 0.1, X.shape[0]) # Target with noise
# Split 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)

# Implement KNN Regression
k = 3 # Number of neighbors
knn_regressor = KNeighborsRegressor(n_neighbors=k, weights='uniform')
knn_regressor.fit(X_train, y_train)
# Predict on test set
y_pred = knn_regressor.predict(X_test)

# Compute Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.4f}")

# Plot results
plt.scatter(X, y, color='blue', label='Data')
plt.plot(X_test, y_pred, 'ro', label='Predictions')
plt.xlabel("Feature")
plt.ylabel("Target")
plt.title(f"KNN Regression (k={k})")
plt.legend()
plt.show()
```

Output:

Mean Squared Error: 0.1636



Program 7: Create a program that calculates different distance metrics (Euclidean and Manhattan) between two points in a dataset. Allow the user to input two points and display the calculated distances.

```
import numpy as np
def euclidean_distance(point1, point2):
    """Compute Euclidean distance between two points."""
    return np.sqrt(np.sum((np.array(point1) - np.array(point2)) ** 2))

def manhattan_distance(point1, point2):
    """Compute Manhattan distance between two points."""
    return np.sum(np.abs(np.array(point1) - np.array(point2)))

def get_coordinates():
    """Get valid user input for coordinates."""
    while True:
        try:
            coords = list(map(float, input().strip().split()))
            return coords
        except ValueError:
            print("Invalid input! Please enter numeric values separated by spaces.")

def main():
    print("Enter the coordinates of two points (space-separated):")

    print("Point 1: ", end="")
    point1 = get_coordinates()

    print("Point 2: ", end="")
    point2 = get_coordinates()

    # Check if dimensions match
    if len(point1) != len(point2):
        print("Error: Points must have the same number of dimensions.")
        return

    # Calculate distances
    euclidean = euclidean_distance(point1, point2)
    manhattan = manhattan_distance(point1, point2)
```

```
    # Print results
    print("\nDistance Calculations:")
    print(f"Euclidean Distance  : {euclidean:.4f}")
    print(f"Manhattan Distance : {manhattan:.4f}")

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print(f"An error occurred: {e}")
```

Output:

```
Enter the coordinates of two points (space-separated):
Point 1: 3        4
Point 2: 7        1

Distance Calculations:
Euclidean Distance  : 5.0000
Manhattan Distance : 7.0000
```

Program 8: Implement the k-Nearest Neighbor algorithm to classify the Iris dataset, printing both correct and incorrect predictions.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split 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)

# Train k-NN classifier
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Make predictions
y_pred = knn.predict(X_test)

# Print correct and incorrect predictions
for i in range(len(y_test)):
    print(f"Predicted: {y_pred[i]}, Actual: {y_test[i]}, {'Correct' if y_pred[i] == y_test[i] else 'Incorrect'}")
```

Output:

```
Predicted: 1, Actual: 1, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 1, Actual: 1, Correct
Predicted: 1, Actual: 1, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 1, Actual: 1, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 1, Actual: 1, Correct
Predicted: 1, Actual: 1, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 1, Actual: 1, Correct
```

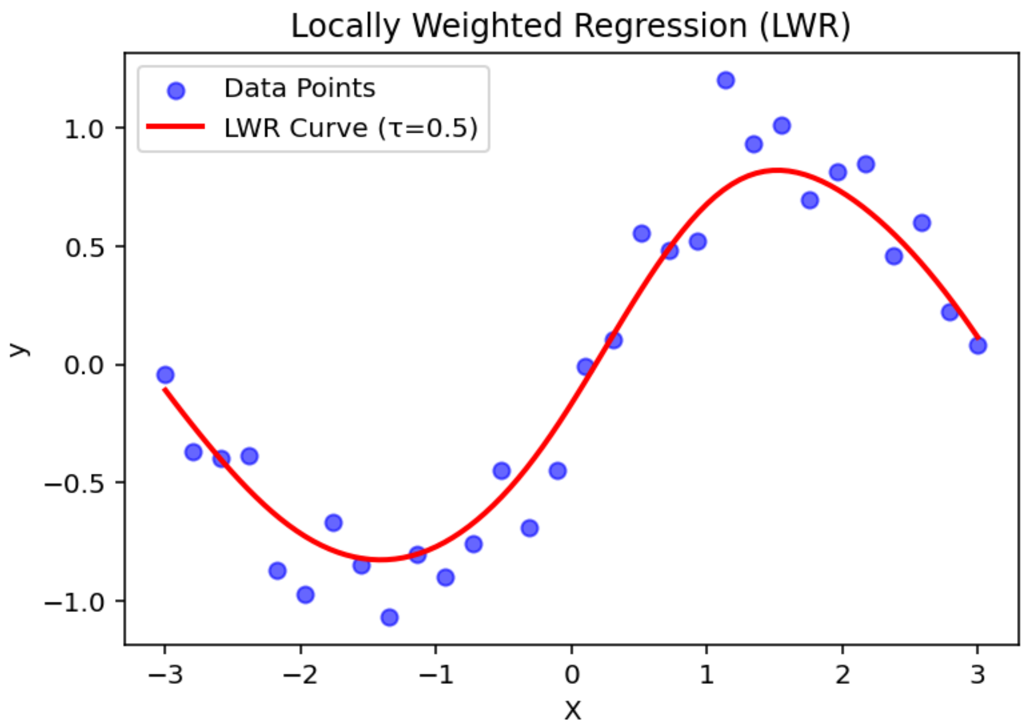
Predicted: 2, Actual: 2, Correct
Predicted: 1, Actual: 1, Correct
Predicted: 1, Actual: 1, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 2, Actual: 2, Correct
Predicted: 0, Actual: 0, Correct
Predicted: 0, Actual: 0, Correct

Program 9: Develop a program to implement the non-parametric Locally Weighted Regression Algorithm, fitting data points and visualizing results.

```
import numpy as np
import matplotlib.pyplot as plt

def kernel_weight(query_x, X, tau):
    """Compute weights using a Gaussian kernel."""
    query_x = np.array(query_x).reshape(1, -1)  # Ensure query_x is 2D
    distances = np.linalg.norm(X - query_x, axis=1) ** 2  # Compute squared distances
    weights = np.exp(-distances / (2 * tau ** 2))  # Gaussian kernel
    return np.diag(weights)  # Return diagonal weight matrix

def locally_weighted_regression(X, y, tau, query_points):
    """Perform Locally Weighted Regression."""
    X_bias = np.c_[np.ones(len(X)), X]  # Add bias term
    y_pred = []

    for query_x in query_points:
        query_x_bias = np.hstack(([1], query_x.ravel()))  # Ensure 1D array
        W = kernel_weight(query_x, X, tau)  # Compute weight matrix
        theta = np.linalg.pinv(X_bias.T @ W @ X_bias) @ (X_bias.T @ W @ y)  # Compute parameters
        y_pred.append(query_x_bias @ theta)  # Predict value

    return np.array(y_pred)

# Generate sample data
np.random.seed(42)
X = np.linspace(-3, 3, 30).reshape(-1, 1)  # Reshape X to be a 2D column vector
y = np.sin(X).flatten() + np.random.normal(scale=0.2, size=X.shape[0])  # True function with noise

# Define test points for visualization
X_test = np.linspace(-3, 3, 100).reshape(-1, 1)  # Reshape X_test to 2D

# Perform LWR with bandwidth parameter tau
tau = 0.5  # Smoothing parameter
y_pred = locally_weighted_regression(X, y, tau, X_test)

# Plot original data and LWR curve
plt.scatter(X, y, label="Data Points", color="blue", alpha=0.6)
```

```
plt.plot(X_test, y_pred, label=f"LWR Curve (τ={tau})", color="red", linewidth=2)
plt.xlabel("X")
plt.ylabel("y")
plt.title("Locally Weighted Regression (LWR)")
plt.legend()
plt.show()
```

Output:

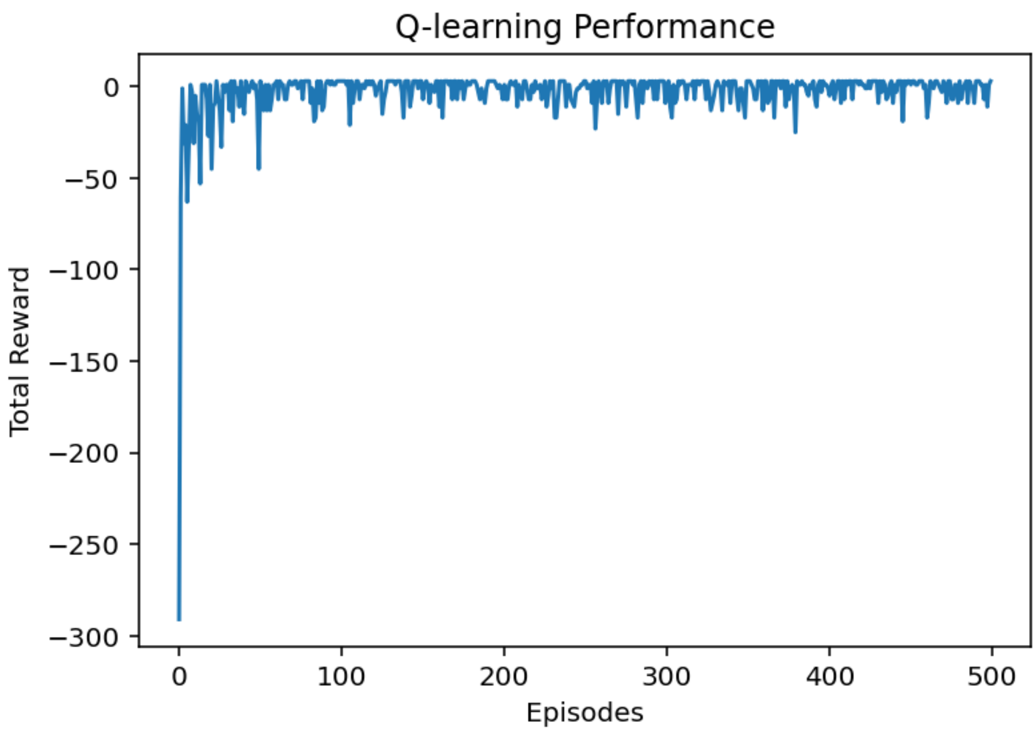


Program 10: Implement a Q-learning algorithm to navigate a simple grid environment, defining the reward structure and analyzing agent performance.

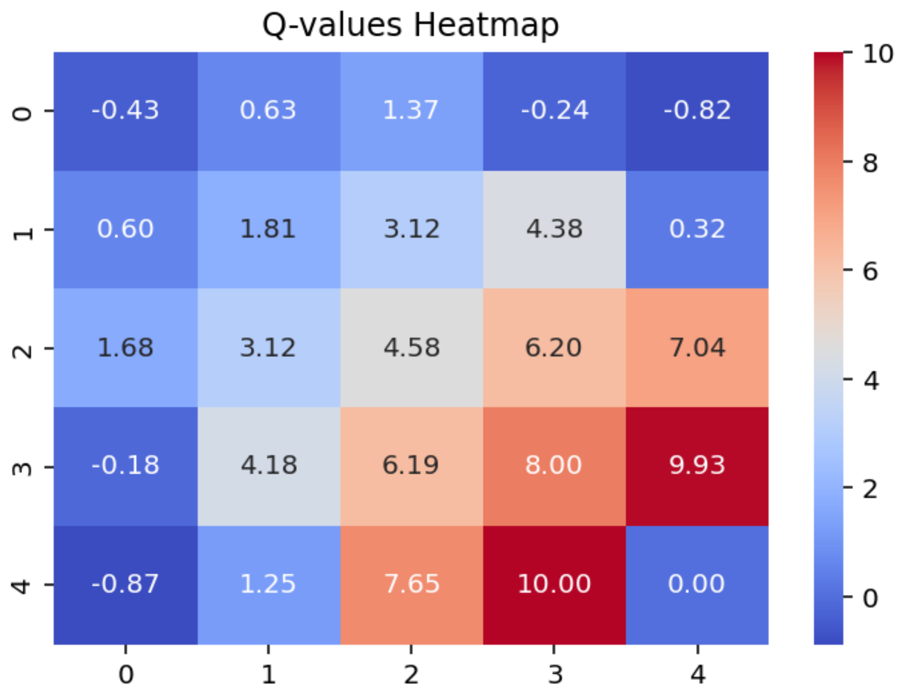
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Grid & Hyperparameters
GRID_SIZE, GAMMA, ALPHA, EPSILON, EPISODES = 5, 0.9, 0.1, 0.2, 500
ACTIONS = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
START, GOAL = (0, 0), (4, 4)
Q_table = np.zeros((GRID_SIZE, GRID_SIZE, len(ACTIONS)))

# Reward grid (-1 per step, +10 goal, -10 wall)
REWARD_GRID = np.full((GRID_SIZE, GRID_SIZE), -1)
REWARD_GRID[GOAL] = 10

def move(state, action):
    """Returns the next state & reward."""
    next_state = (state[0] + action[0], state[1] + action[1])
    return (state, -10) if not (0 <= next_state[0] < GRID_SIZE and 0 <= next_state[1] < GRID_SIZE) else
(next_state, REWARD_GRID[next_state])

def choose_action(state):
    """Epsilon-greedy action selection."""
    return np.random.choice(list(ACTIONS)) if np.random.rand() < EPSILON else
list(ACTIONS)[np.argmax(Q_table[state])]

def train():
    """Q-learning training loop."""
    rewards = []
    for episode in range(EPISODES):
        state, total_reward = START, 0
        while state != GOAL:
            action = choose_action(state)
            next_state, reward = move(state, ACTIONS[action])
            Q_table[state][list(ACTIONS).index(action)] += ALPHA * (reward + GAMMA *
np.max(Q_table[next_state]) - Q_table[state][list(ACTIONS).index(action)])
            state, total_reward = next_state, total_reward + reward
        rewards.append(total_reward)
    return rewards
```

```
# Run training & visualize results
rewards = train()

plt.plot(rewards)
plt.xlabel("Episodes")
plt.ylabel("Total Reward")
plt.title("Q-learning Performance")
plt.show()

sns.heatmap(np.max(Q_table, axis=2), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Q-values Heatmap")
plt.show()
```

Output:



(Note: The above graph chart is done by using pyplot library from Matplotlib module.)



(Note: The above heat map is done by using Seaborn library.)

Graph of Rewards Over Episodes

- **Early episodes**: Low rewards (random moves).
- **Later episodes**: Higher rewards (better paths).

Q-values Heatmap

- **Brighter values** show **stronger Q-values** (good moves).
- The **goal state has the highest Q-value**.

Python Requirements:

- Install Python 3.12.8 or 3.10
- Then install the following libraries/modules of python:
    - pip install spyder
    - pip install seaborn
    - pip install matplotlib
    - pip install scikit-learn
    - pip install mlxtend
    - pip install tensorflow
    - pip install numpy
    - pip install scipy
    - pip install pandas