# 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**

Certificate

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**

|  |  |  |  |
| --- | --- | --- | --- |
| **SL NO.** | **NAME OF THE PROGRAMS** | **PAGE**  **NO** | **DATE** |
| **1** | **Implement a simple linear regression algorithm to predict a continuous target variable based on a given dataset.** |  |  |
| **2** | **Develop a program to implement a Support Vector Machine for binary classification. Use a sample dataset and visualize**  **the decision boundary** |  |  |
| **3** | **Develop a simple case-based reasoning system that stores instances of past cases. Implement a retrieval method to find**  **the most similar cases and make predictions based on them.** |  |  |
| **4** | **Write a program to demonstrate the ID3 decision tree algorithm using an appropriate dataset for classification.** |  |  |
| **5** | **Build an Artificial Neural Network by implementing the Backpropagation algorithm and test it with suitable datasets.** |  |  |
| **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** |  |  |
| **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.** |  |  |
| **8** | **Implement the k-Nearest Neighbor algorithm to classify the Iris dataset, printing both correct and incorrect predictions.** |  |  |
| **9** | **Develop a program to implement the non-parametric Locally Weighted Regression algorithm, fitting data points and visualizing results.** |  |  |
| **10** | **Implement a Q-learning algorithm to navigate a simple grid environment, defining the reward structure and analyzing**  **agent performance.** |  |  |

## 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()

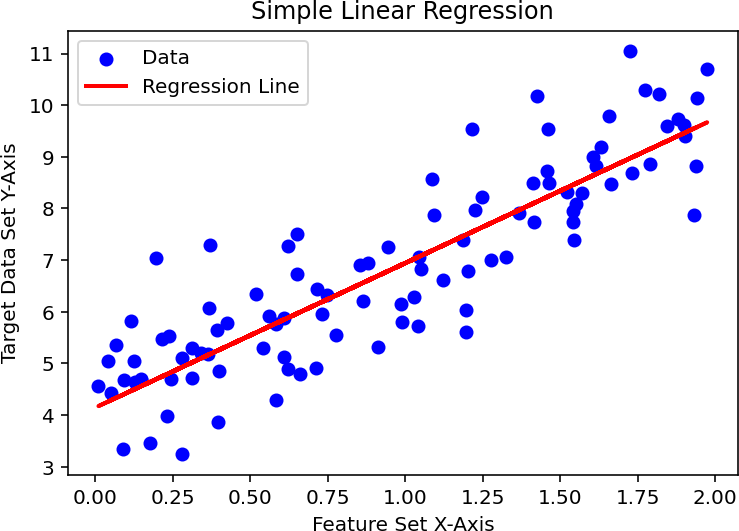
plt.show()

## 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)

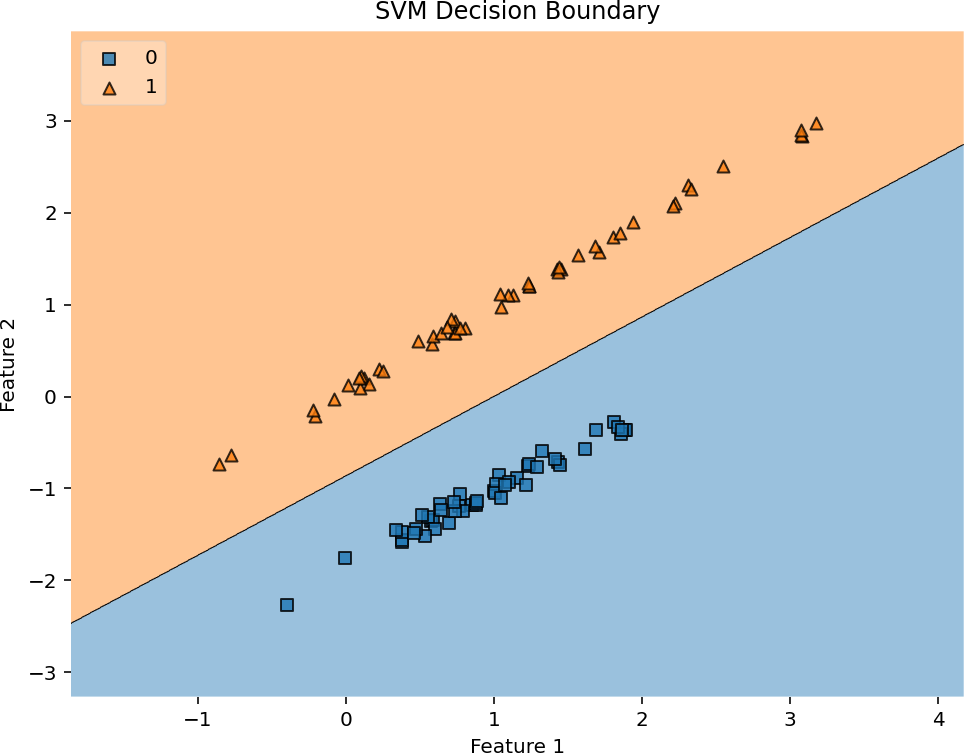
**# Plot decision boundary** plt.figure(figsize=(8, 6)) plot\_decision\_regions(X, y, clf=svm, legend=2) plt.xlabel("Feature 1")

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()

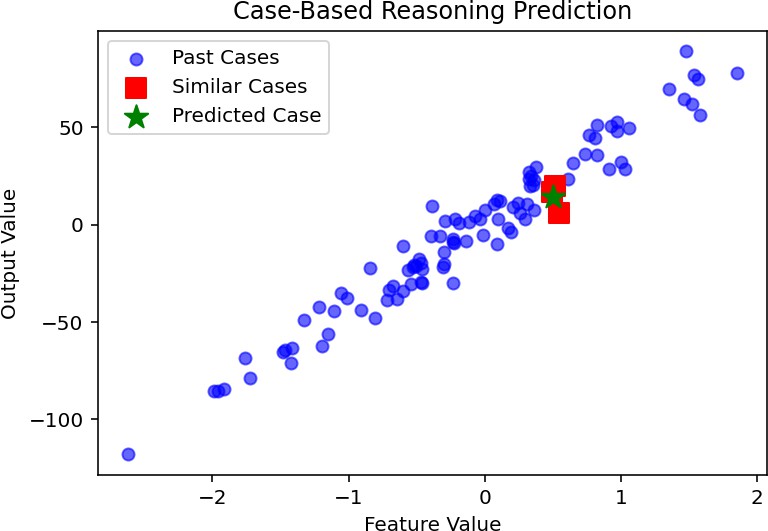
## Output:

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

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")

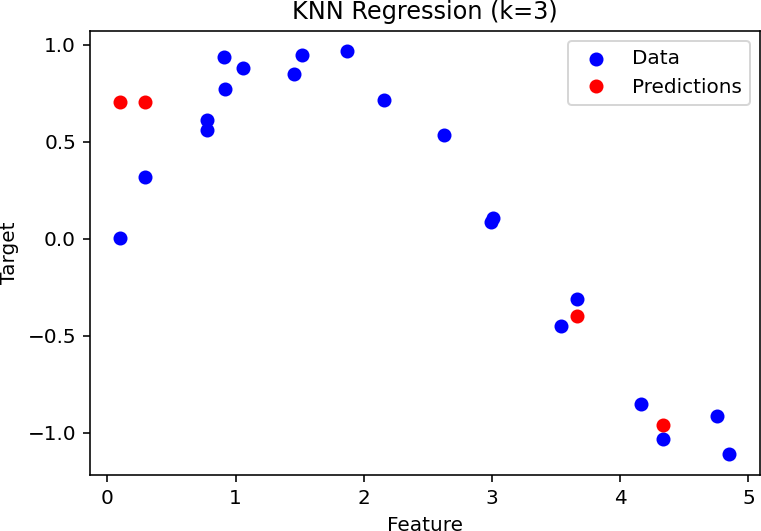
plt.ylabel("Target")

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)

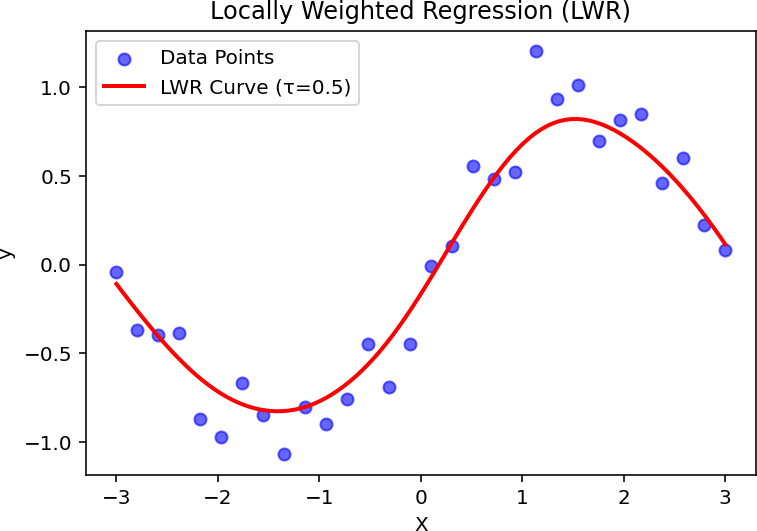
plt.plot(X\_test, y\_pred, label=f"LWR Curve (τ={tau})", color="red", linewidth=2) plt.xlabel("X")

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

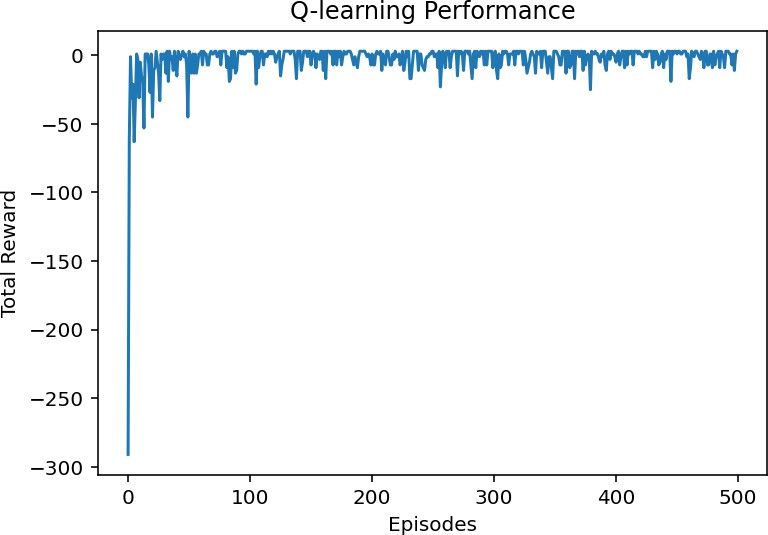
rewards = train()

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

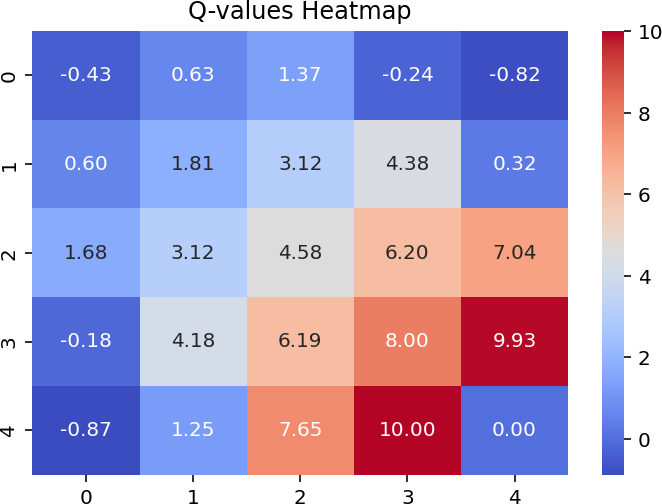
sns.heatmap(np.max(Q\_table, axis=2), annot=True, cmap="coolwarm", fmt=".2f") plt.title("Q-values Heatmap")

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