VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Algorithms and AI Laboratory

(MCSL106)

Submitted by

Shreyas Bhat K (1BM24SCS14)

in partial fulfillment for the award of the degree of

Master OF Technology
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019

Dec-2024 to March-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled “Algorithms & AI Lab (MCSL106)” carried out by **Shreyas Bhat K (1BM24SCS14),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Master of Technology in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Algorithms & AI Lab (MCSL106) work prescribed for the said degree.

Dr.K.Panimozhi Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/shreyasbkgit/ailab.git

Program 1

Implement Naive Bayes models and Bayesian networks. (Demonstrate the diagnosis of heart patients using standard heart disease data set etc)

Algorithm:

- Load and preprocess the dataset (e.g., UCI Heart Disease dataset).
- Calculate prior probabilities for each class (disease present or not).
- Compute likelihood probabilities using conditional probability and the assumption of feature independence (for Naïve Bayes).
- Apply Bayes' Theorem to compute posterior probabilities.
- Classify new patient data based on the highest posterior probability

Code:

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score, classification_report

from pgmpy.models import BayesianModel

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.inference import VariableElimination

dataset_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data"

columns = ["age", "sex", "cp", "trestbps", "chol", "fbs", "restecg", "thalach", "exang", "oldpeak", "slope", "ca", "thal", "target"]

data = pd.read_csv(dataset_url, names=columns, na_values='?')

data.dropna(inplace=True)

data["target"] = (data["target"] > 0).astype(int)

X = data.drop(columns=["target"])
```

y = data["target"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

nb_model = GaussianNB()

nb_model.fit(X_train, y_train)

y_pred = nb_model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))

print(classification_report(y_test, y_pred))

model = BayesianModel([("age", "target"), ("sex", "target"), ("cp", "target"), ("chol", "target"), ("thal", "target")])

model.fit(data, estimator=MaximumLikelihoodEstimator)

infer = VariableElimination(model)

print("Probability of Heart Disease given cp=3:")

print(infer.query(variables=["target"], evidence={"cp": 3}))

Output

```
Accuracy: 0.9166666666666666
              precision    recall  f1-score   support

           0       0.90      0.97      0.93        36
           1       0.95      0.83      0.89        24

    accuracy                           0.92        60
   macro avg       0.92      0.90      0.91        60
weighted avg       0.92      0.92      0.92        60
```

Program 2

Implement a simple linear regression algorithm to predict a continuous target variable based on a given dataset.

Algorithm:

- Load dataset and preprocess it.
- Define the hypothesis function:Y=mX+b.
- Compute the cost function (Mean Squared Error).
- Use Gradient Descent or Least Squares to optimize m and b.
- Predict values for new inputs and visualize results.

Code:

```
import numpy as np

import pandas as pd

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

from sklearn.datasets import fetch_california_housing

data = fetch_california_housing()

df = pd.DataFrame(data.data, columns=data.feature_names)

df["Price"] = data.target

X = df[["MedInc"]]

y = df["Price"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = LinearRegression()
```

```
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error: {mse:.2f}")

print(f"R-squared: {r2:.2f}")
```

Output

```
Mean Squared Error: 0.71
R-squared: 0.46
```

Program 3

Develop a program to implement a Support Vector Machine for binary classification. Use a sample dataset and visualize the decision boundary.

Algorithm:

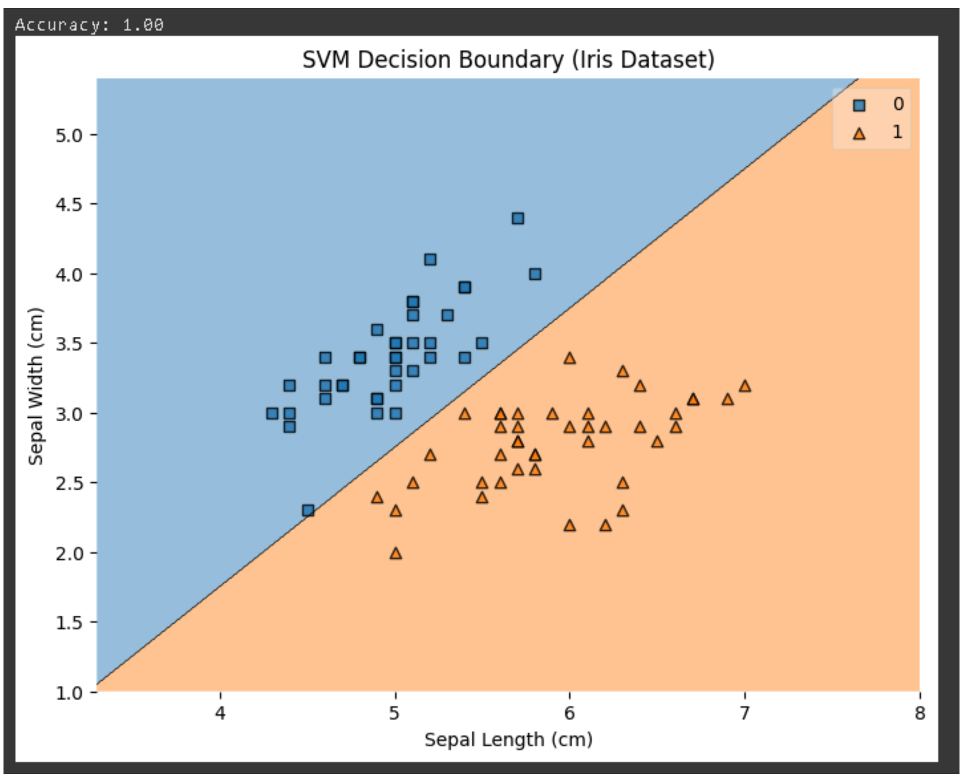
- Load and preprocess dataset.
- Define the SVM objective: maximize margin between two classes.
- Solve the optimization problem using techniques like the SMO algorithm.
- Use kernels (linear, polynomial, RBF) if needed.
- Plot the decision boundary using Matplotlib.

Code:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score

from mlxtend.plotting import plot_decision_regions

iris = datasets.load_iris()

df = pd.DataFrame(data=iris.data, columns=iris.feature_names)

df['target'] = iris.target

df = df[df['target'] != 2]

X = df[['sepal length (cm)', 'sepal width (cm)']].values

y = df['target'].values
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

svm_model = SVC(kernel='linear')

svm_model.fit(X_train, y_train)

y_pred = svm_model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy:.2f}')

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

plot_decision_regions(X_train, y_train, clf=svm_model)

plt.xlabel('Sepal Length (cm)')

plt.ylabel('Sepal Width (cm)')

plt.title('SVM Decision Boundary (Iris Dataset)')

plt.show()
```

Output



Program 4

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

Algorithm:

- Load dataset and preprocess it.
- Compute entropy and information gain for each feature.
- Select the feature with the highest information gain as the root node.
- Recursively split the dataset based on feature values until all instances belong to the same class.
- Use the trained tree to classify new data points.

Code:

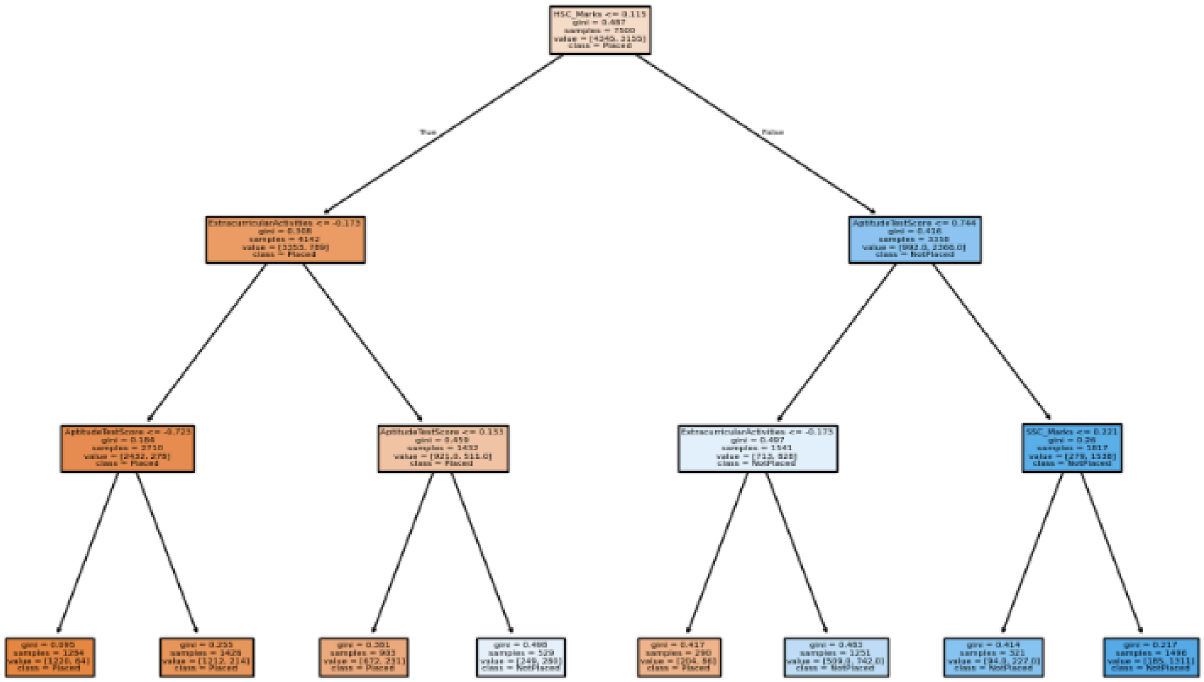
```
iimport pandas as pd

from google.colab import drive

drive.mount('/content/drive')

data=pd.read_csv('/content/drive/MyDrive/workshop/placementdata.csv')

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.tree import DecisionTreeClassifier, plot_tree

from sklearn.metrics import accuracy_score,confusion_matrix,classification_report

data.head()

data.duplicated().sum()

data.isnull().sum()

data.shape
```

```
data.info()

data.describe()

data['PlacementStatus'].value_counts().plot(kind='bar')

data['PlacementStatus'].value_counts()/len(data)*100

placement_plot=['Internships','Projects','Workshops/Certifications','ExtracurricularActivities','Placement
Training']

plt.figure(figsize=(15,12))

for i,graph in enumerate(placement_plot):

  plt.subplot(2,3,i+1)

  sns.countplot(x=graph,data=data,hue='PlacementStatus')

  plt.title(f'PLacement status based on {graph}')

num_col=data.select_dtypes('number')

num_col.columns.tolist()

non_num_col=data.select_dtypes('object')

non_num_col.columns.tolist()

data['PlacementStatus'].replace({'Placed':1,'NotPlaced':0},inplace=True)

data['PlacementStatus'].head()

from sklearn.preprocessing import LabelEncoder

le=LabelEncoder()

data['PlacementStatus']=le.fit_transform(data['PlacementStatus'])

data['ExtracurricularActivities']=le.fit_transform(data['ExtracurricularActivities'])

data['PlacementTraining']=le.fit_transform(data['PlacementTraining'])

data.info()

x=data.drop(['PlacementStatus','StudentID'],axis=1)

y=data['PlacementStatus']

x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.25,random_state=42)
```

```
scaler=StandardScaler()

scaler=StandardScaler()

x_train_scaled=scaler.fit_transform(x_train)

x_test_scaled=scaler.transform(x_test)

model=DecisionTreeClassifier()

model.fit(x_train_scaled,y_train)

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

plot_tree(model,filled=True,feature_names=x.columns,class_names=['Placed','NotPlaced'])

plt.show()

clf=DecisionTreeClassifier(max_depth=3,min_samples_leaf=5,min_samples_split=10)

clf.fit(x_train_scaled,y_train)

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

plot_tree(clf,filled=True,feature_names=x.columns,class_names=['Placed','NotPlaced'])

plt.show()

y_pred_model=model.predict(x_test_scaled)

y_pred_clf=clf.predict(x_test_scaled)

print(classification_report(y_test,y_pred_model))

print(classification_report(y_test,y_pred_clf))

importances=model.feature_importances_

feature_importances_df=pd.DataFrame({'feature':x_train.columns,'importance':importances})

feature_importances_df.sort_values(by='importance',ascending=False,inplace=True)

feature_importances_df

x=data.drop(['SoftSkillsRating','Workshops/Certifications','Internships','Projects','PlacementTraining'],a
xis=1)

print(x.head())

x=x.drop(['SoftSkillsRating','SoftSkillsRating','Workshops/Certifications','Internships','Projects','Placem
```

```
entTraining'],axis=1)
y=data['PlacementStatus']
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.25,random_state=42)
scaler=StandardScaler()
scaler=StandardScaler()
x_train_scaled=scaler.fit_transform(x_train)
x_test_scaled=scaler.transform(x_test)
clf=DecisionTreeClassifier(max_depth=3,min_samples_leaf=5,min_samples_split=10)
clf.fit(x_train_scaled,y_train)
plt.figure(figsize=(12,8))
plot_tree(clf,filled=True,feature_names=x.columns,class_names=['Placed','NotPlaced'])
plt.show()
y_pred_clf=clf.predict(x_test_scaled)
print(classification_report(y_test,y_pred_clf))
```

Output



HSC_Marks <= 0.115
class = Placed
True
False
ExtracurricularActivities <= -0.173
class = Placed
AptitudeTestScore <= 0.744
class = NotPlaced
AptitudeTestScore <= -0.723
class = Placed
AptitudeTestScore <= 0.153
class = Placed
ExtracurricularActivities <= -0.173
class = NotPlaced
SSC_Marks <= 0.221
class = NotPlaced
class = Placed
class = Placed
class = Placed
class = NotPlaced
class = Placed
class = NotPlaced
class = NotPlaced
class = NotPlaced

Program 5

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.

Algorithm:

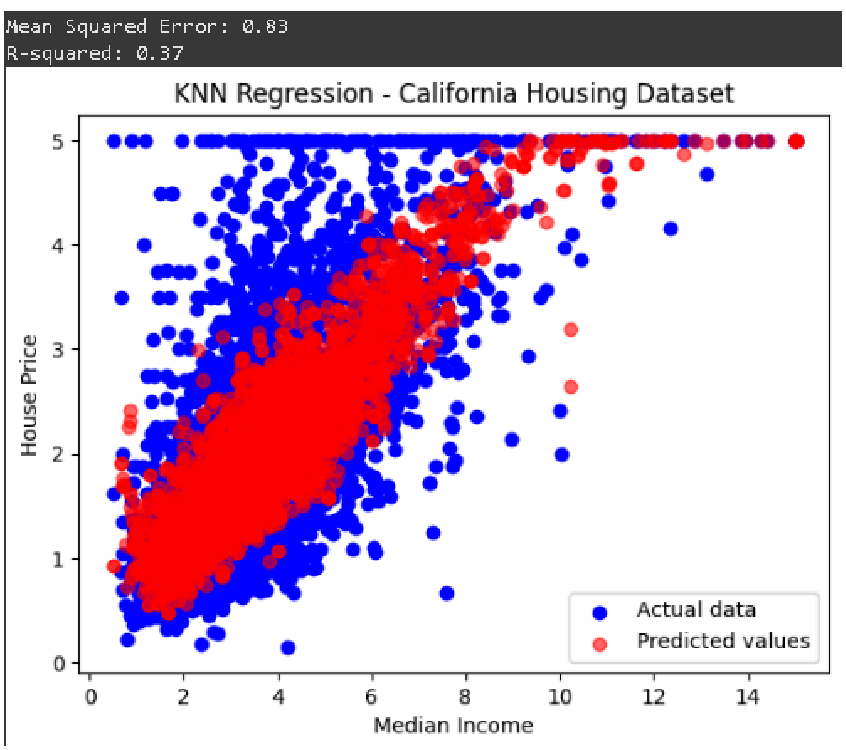
- Load and preprocess dataset.
- Choose the number of neighbors (K).
- Compute the Euclidean distance between the query point and all training samples.
- Select the K nearest neighbors.
- Compute the average of their target values to predict the output.

Code:

```
import numpy as np

import pandas as pd

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, r2_score

from sklearn.datasets import fetch_california_housing

data = fetch_california_housing()

df = pd.DataFrame(data.data, columns=data.feature_names)

df["Price"] = data.target  # Target variable

X = df[["MedInc"]]  # Median income as predictor

y = df["Price"]  # House price as target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

knn_model = KNeighborsRegressor(n_neighbors=5)

knn_model.fit(X_train, y_train)
```

```
y_pred = knn_model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error: {mse:.2f} ")

print(f"R-squared: {r2:.2f} ")

plt.scatter(X_test, y_test, color='blue', label='Actual data')

plt.scatter(X_test, y_pred, color='red', label='Predicted values', alpha=0.6)

plt.xlabel("Median Income")

plt.ylabel("House Price")

plt.title("KNN Regression - California Housing Dataset")

plt.legend()

plt.show()
```

Output



Program 6

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

Algorithm:

- Load the Iris dataset.
- Choose K.
- Compute the Euclidean distance between the query point and training points.
- Select K nearest neighbors.
- Assign the most common class among the neighbors to the query point.
- Print correct and incorrect predictions.

Code:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score, classification_report

iris = datasets.load_iris()

df = pd.DataFrame(data=iris.data, columns=iris.feature_names)

df['target'] = iris.target

X = df[iris.feature_names].values

y = df['target'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

knn_model = KNeighborsClassifier(n_neighbors=5)
```

```
knn_model.fit(X_train, y_train)

y_pred = knn_model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy:.2f}')

print(classification_report(y_test, y_pred))
```

Output

```
Accuracy: 1.00
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Program 7

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

Algorithm:

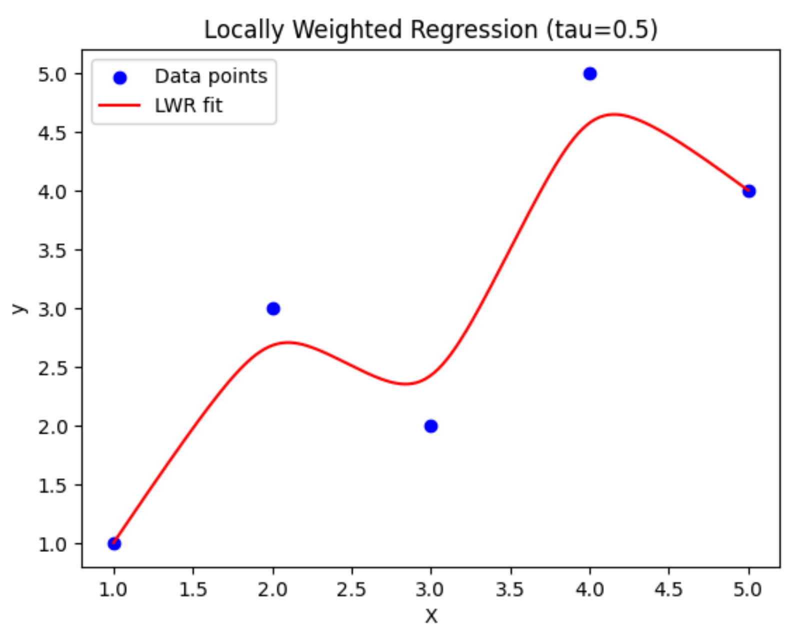
- Load and preprocess dataset.
- Define a weight function W(x)W(x)W(x) that assigns higher weights to closer points.
- Compute the weighted least squares estimate for a given query point.
- Predict and visualize the regression line.

Code:

```
import numpy as np

import matplotlib.pyplot as plt

def gaussian_kernel(x, x0, tau):

    return np.exp(-np.sum((x - x0)**2) / (2 * tau**2))

def compute_weights(X, x0, tau):

    m = X.shape[0]

    weights = np.zeros(m)

    for i in range(m):

        weights[i] = gaussian_kernel(X[i], x0, tau)

    return np.diag(weights)

def locally_weighted_regression(X, y, x0, tau):

    X_b = np.c_[np.ones((X.shape[0], 1)), X]  # Add intercept term

    x0_b = np.r_[1, x0]  # Add intercept term to the query point

    W = compute_weights(X, x0, tau)

    theta = np.linalg.inv(X_b.T @ W @ X_b) @ (X_b.T @ W @ y)

    return x0_b @ theta
```

```
def plot_lwr(X, y, tau):
    X_range = np.linspace(np.min(X), np.max(X), 300)
    y_pred = [locally_weighted_regression(X, y, x0, tau) for x0 in X_range]

    plt.scatter(X, y, color='blue', label='Data points')
    plt.plot(X_range, y_pred, color='red', label='LWR fit')
    plt.xlabel('X')
    plt.ylabel('y')
    plt.title(f'Locally Weighted Regression (tau={tau})')
    plt.legend()
    plt.show()
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([1, 3, 2, 5, 4])
plot_lwr(X, y, tau=0.5)
```

Output



Program 8

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

Algorithm:

- Initialize weights randomly.
- Forward propagation: Compute activations using weights.
- Compute error using a loss function (e.g., Mean Squared Error for regression, Cross-Entropy for classification).
- Backpropagation: Compute gradients and update weights using Gradient Descent.
- Repeat until convergence.

Code:

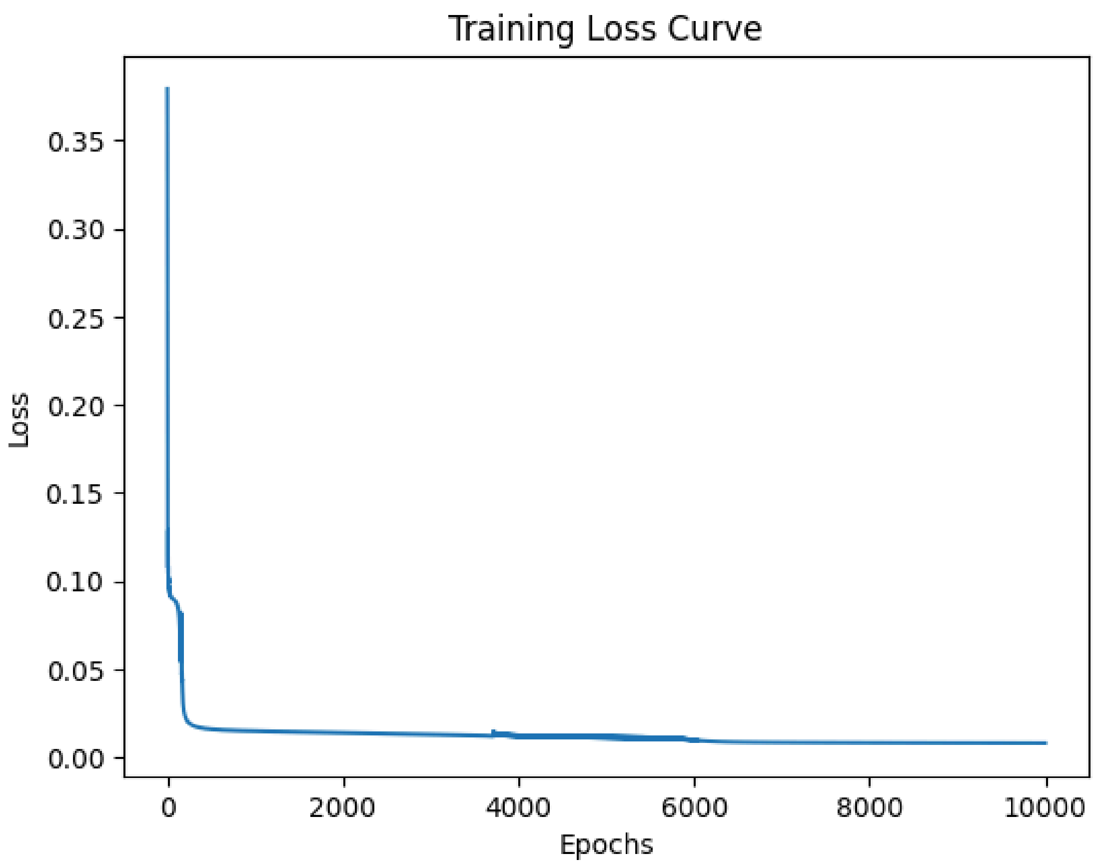
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_moons

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import OneHotEncoder

X, y = make_moons(n_samples=500, noise=0.2, random_state=42)

y = y.reshape(-1, 1)

encoder = OneHotEncoder(sparse_output=False)

y = encoder.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):

    return x * (1 - x)

input_size = X_train.shape[1]
```

```
hidden_size = 5
output_size = y_train.shape[1]
learning_rate = 0.1
epochs = 10000
np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
B1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
B2 = np.zeros((1, output_size))
losses = []
for epoch in range(epochs):
    Z1 = np.dot(X_train, W1) + B1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2) + B2
    A2 = sigmoid(Z2)
    loss = np.mean((A2 - y_train) ** 2)
    losses.append(loss)
    dA2 = A2 - y_train
    dZ2 = dA2 * sigmoid_derivative(A2)
    dW2 = np.dot(A1.T, dZ2)
    dB2 = np.sum(dZ2, axis=0, keepdims=True)
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * sigmoid_derivative(A1)
    dW1 = np.dot(X_train.T, dZ1)
    dB1 = np.sum(dZ1, axis=0, keepdims=True)
```

```
    W2 -= learning_rate * dW2
    B2 -= learning_rate * dB2
    W1 -= learning_rate * dW1
    B1 -= learning_rate * dB1
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")
plt.plot(losses)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training Loss Curve")
plt.show()
def predict(X):
    Z1 = np.dot(X, W1) + B1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2) + B2
    A2 = sigmoid(Z2)
    return np.argmax(A2, axis=1)
predictions = predict(X_test)
y_test_labels = np.argmax(y_test, axis=1)
accuracy = np.mean(predictions == y_test_labels)
print(f"Test Accuracy: {accuracy:.2f}")
```

Output

```
Epoch 0, Loss: 0.3792
Epoch 1000, Loss: 0.0149
Epoch 2000, Loss: 0.0140
Epoch 3000, Loss: 0.0130
Epoch 4000, Loss: 0.0121
Epoch 5000, Loss: 0.0112
Epoch 6000, Loss: 0.0098
Epoch 7000, Loss: 0.0085
Epoch 8000, Loss: 0.0084
Epoch 9000, Loss: 0.0082
```




```
Test Accuracy: 0.97
```

Program 9

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

Algorithm:

- Define the environment (states, actions, rewards).
- Initialize Q-table with zeros.
- For each episode:
- Select an action using an epsilon-greedy policy.
- Take the action, observe reward, and update the Q-value:

$$Q(s,a)=Q(s,a)+\alpha[r+\gamma \max Q(s',a')-Q(s,a)]$$

- Repeat until convergence and analyze agent performance.

Code:

```
import numpy as np

import random

import matplotlib.pyplot as plt

GRID_SIZE = 5

ACTIONS = ['up', 'down', 'left', 'right']

ACTION_MAP = {0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0, 1)}

GOAL_STATE = (4, 4)

PENALTY_STATE = (2, 2)

GAMMA = 0.9  # Discount factor

ALPHA = 0.1  # Learning rate

EPSILON = 0.1  # Exploration rate
```

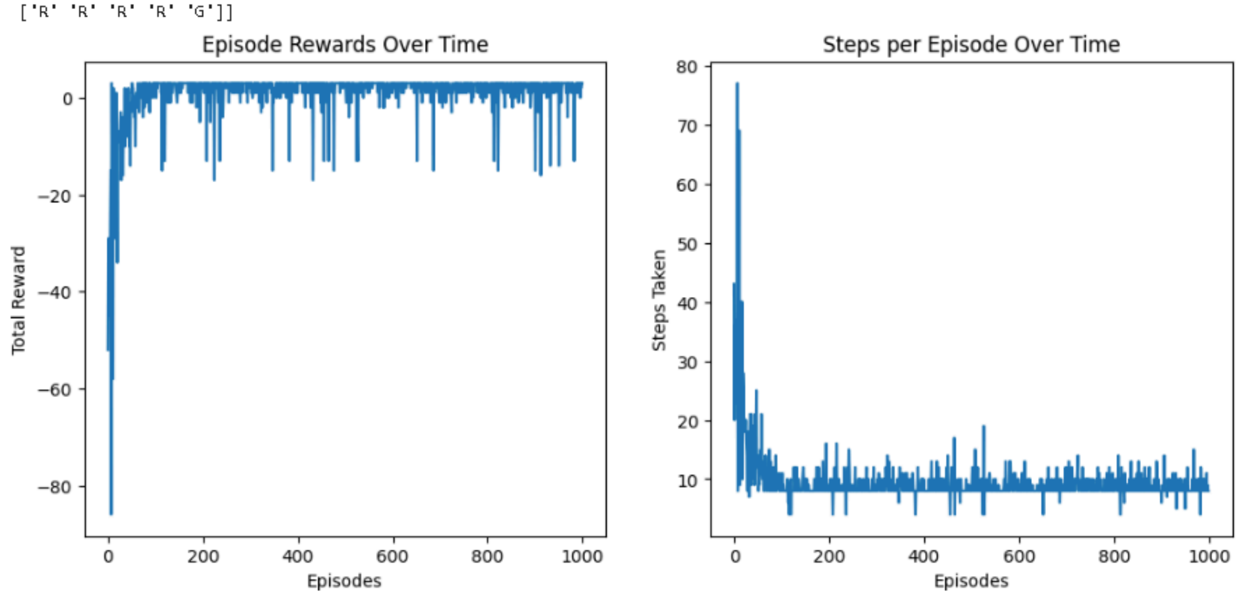
```
EPISODES = 1000
Q_table = np.zeros((GRID_SIZE, GRID_SIZE, len(ACTIONS)))
episode_rewards = []
episode_steps = []
def take_action(state, action):
    new_state = (max(0, min(GRID_SIZE - 1, state[0] + ACTION_MAP[action][0])),
                 max(0, min(GRID_SIZE - 1, state[1] + ACTION_MAP[action][1])))
    if new_state == GOAL_STATE:
        return new_state, 10  # Reward for reaching goal
    elif new_state == PENALTY_STATE:
        return new_state, -10  # Penalty state
    else:
        return new_state, -1  # Small penalty for each move
for episode in range(EPISODES):
    state = (0, 0)  # Start at top-left corner
    done = False
    total_reward = 0
    steps = 0
    while not done:
        # Choose action (ε-greedy policy)
        if random.uniform(0, 1) < EPSILON:
            action = random.randint(0, len(ACTIONS) - 1)  # Explore
        else:
            action = np.argmax(Q_table[state[0], state[1], :])  # Exploit
        new_state, reward = take_action(state, action)
```

```
        Q_table[state[0], state[1], action] += ALPHA * (
                reward + GAMMA * np.max(Q_table[new_state[0], new_state[1], :]) - Q_table[state[0],
state[1], action]
        )
        state = new_state
        total_reward += reward
        steps += 1
        if state == GOAL_STATE or state == PENALTY_STATE:
            done = True
    episode_rewards.append(total_reward)
    episode_steps.append(steps)
policy = np.full((GRID_SIZE, GRID_SIZE), 'X')
for i in range(GRID_SIZE):
    for j in range(GRID_SIZE):
        if (i, j) == GOAL_STATE:
            policy[i, j] = 'G'
        elif (i, j) == PENALTY_STATE:
            policy[i, j] = 'P'
        else:
            best_action = np.argmax(Q_table[i, j, :])
            policy[i, j] = ACTIONS[best_action][0].upper()
print("Optimal Policy:")
print(policy)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(episode_rewards)
```

```
plt.xlabel("Episodes")

plt.ylabel("Total Reward")

plt.title("Episode Rewards Over Time")

plt.subplot(1, 2, 2)

plt.plot(episode_steps)

plt.xlabel("Episodes")

plt.ylabel("Steps Taken")

plt.title("Steps per Episode Over Time")
```

Output

```
Optimal Policy:
[['R' 'D' 'D' 'D' 'D']
 ['R' 'R' 'R' 'R' 'D']
 ['R' 'D' 'P' 'R' 'D']
 ['R' 'D' 'D' 'D' 'D']
 ['R' 'R' 'R' 'R' 'G']]
```



Program 10

Write a python program
a. to perform tokenization by word and sentence using nltk.
b. to eliminate stop words using nltk.
c. to perform stemming using nltk.
d. to perform Parts of Speech tagging using nltk.

Algorithm:

a. Tokenization (Splitting text into words/sentences)

- Use nltk.word_tokenize(text) for word tokenization.
- Use nltk.sent_tokenize(text) for sentence tokenization.

b. Remove Stop Words

- Use nltk.corpus.stopwords to filter out common words like "the", "is", etc.

c. Stemming (Reducing words to root form)

- Use nltk.stem.PorterStemmer to transform words (e.g., "running" → "run").

d. POS Tagging (Assigning part-of-speech labels to words)

- Use nltk.pos_tag(tokens) to label words as nouns, verbs, etc.

Code:

```
user_input = input("Enter some text: ")

upper_input = user_input.upper()

upper_input

import nltk

from nltk.stem import PorterStemmer

from nltk.tokenize import word_tokenize

text=input("Enter some text: ")
```

```
text = text.lower()
words = word_tokenize(text)
ps = PorterStemmer()
stemmed_words = [ps.stem(w) for w in words]
stemmed_words
import nltk
from nltk.tokenize import word_tokenize
text=input("Enter some text: ")
text = text.lower()
words = word_tokenize(text)
pos_tags = nltk.pos_tag(words)
pos_tags
import spacy
nlp=spacy.load('en_core_web_sm')
doc=nlp('she saw a bear')
for word in doc:
  print (word.text,word.pos_)
```

Output

A.

```
Enter some text: Once upon a time, in a quaint village nestled between rolling hills and lush forests, there lived a young girl
named Aria. Aria had a special gift: she could communicate with animals. Every morning, she would wander into the forest, where
the birds would sing her songs, and the deer would share their secrets. One day, a mysterious creature appeared, and Aria's life
changed forever...
 ['onc',
  'upon',
  'a',
  'time',
  ',',
  'in',
  'a',
  'quaint',
  'villag',
  'nestl',
  'between',
```

B.

```
Enter some text: Once upon a time, in a quaint village nestled between rolling hills and lush forests, there lived a young girl
named Aria. Aria had a special gift: she could communicate with animals. Every morning, she would wander into the forest, where
the birds would sing her songs, and the deer would share their secrets. One day, a mysterious creature appeared, and Aria's life
changed forever...
 [('once', 'RB'),
  ('upon', 'IN'),
  ('a', 'DT'),
  ('time', 'NN'),
  (',', ','),
  ('in', 'IN'),
  ('a', 'DT'),
  ('quaint', 'NN'),
  ('village', 'NN'),
  ('nestled', 'VBD'),
  ('between', 'IN'),
  ('rolling', 'VBG'),
  ('hills', 'NNS'),
  ('and', 'CC'),
  ('lush', 'JJ'),
  ('forests', 'NNS'),
  (',', ','),
  ('there', 'EX'),
  ('lived', 'VBD'),
  ('a', 'DT'),
  ('young', 'JJ'),
  ('girl', 'NN'),
  ('named', 'VBN'),
  ('aria', 'NN'),
  ('.', '.'),
  ('aria', 'NN'),
  ('had', 'VBD'),
  ('a', 'DT'),
  ('special', 'JJ'),
```

C.

```
she PRON
saw VERB
a DET
bear NOUN
```