**MLP**:

import numpy as np

import matplotlib.pyplot as plt

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

def mlp\_xor(inputs, targets, learning\_rate, epochs):

    input\_neurons = 2

    hidden\_neurons = 2

    output\_neurons = 1

    # Initialize weights with random values

    hidden\_weights = np.random.uniform(size=(input\_neurons, hidden\_neurons))

    hidden\_bias = np.random.uniform(size=(1, hidden\_neurons))

    output\_weights = np.random.uniform(size=(hidden\_neurons, output\_neurons))

    output\_bias = np.random.uniform(size=(1, output\_neurons))

    errors = []

    for epoch in range(epochs):

        total\_error = 0

        for i in range(len(inputs)):

            # Forward propagation

            hidden\_layer\_input = np.dot(inputs[i], hidden\_weights) + hidden\_bias

hidden\_layer\_output = sigmoid(hidden\_layer\_input)

            output\_layer\_input = np.dot(hidden\_layer\_output, output\_weights) + output\_bias

            predicted\_output = sigmoid(output\_layer\_input)

            # Calculate error

            error = targets[i] - predicted\_output

            total\_error += abs(error[0])

            # Backpropagation

            delta\_output = error \* sigmoid\_derivative(predicted\_output)

            error\_hidden = delta\_output.dot(output\_weights.T)

            delta\_hidden = error\_hidden \* sigmoid\_derivative(hidden\_layer\_output)

            # Update weights and biases

            output\_weights += hidden\_layer\_output.reshape(-1,1).dot(delta\_output) \* learning\_rate

            output\_bias += delta\_output \* learning\_rate

            hidden\_weights += inputs[i].reshape(-1,1).dot(delta\_hidden) \* learning\_rate

            hidden\_bias += delta\_hidden \* learning\_rate

        errors.append(total\_error / len(inputs))

    return hidden\_weights, hidden\_bias, output\_weights, output\_bias, errors

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

#Binary

targets\_binary = np.array([[0], [1], [1], [0]])

#Bipolar

targets\_bipolar = np.array([[-1], [1], [1], [-1]])

learning\_rate = 0.5

epochs = 50000

hidden\_weights\_b, hidden\_bias\_b, output\_weights\_b, output\_bias\_b, errors\_b = mlp\_xor(inputs, targets\_binary, learning\_rate, epochs)

hidden\_weights\_bp, hidden\_bias\_bp, output\_weights\_bp, output\_bias\_bp, errors\_bp = mlp\_xor(inputs, targets\_bipolar, learning\_rate, epochs)

#Plot the error curve

plt.plot(errors\_b, label='Binary')

plt.plot(errors\_bp, label='Bipolar')

plt.xlabel('Epochs')

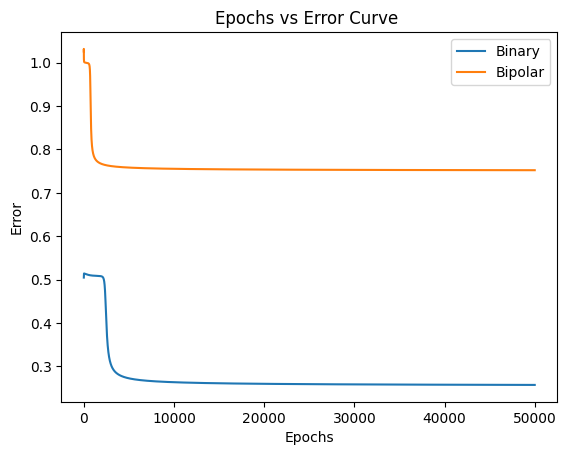
plt.ylabel('Error')

plt.title('Epochs vs Error Curve ')

plt.legend()

plt.show()

**OUTPUT**:



**HEBB RULE:**

import numpy as np

def hebb\_rule(inputs, targets, learning\_rate, epochs):

    num\_inputs = len(inputs[0])

    weights = np.zeros(num\_inputs)

    bias = 0

print("Initial weights:", weights)

    print("Initial bias:", bias)

for epoch in range(epochs):

        print(f"\nEpoch {epoch + 1}:")

        for i in range(len(inputs)):

            weighted\_sum = np.dot(inputs[i], weights) + bias

            #Hebb rule weight update

            weights = weights + learning\_rate \* targets[i] \* inputs[i]

            bias = bias + learning\_rate \* targets[i]

            print(f"  Input: {inputs[i]}, Target: {targets[i]}, Weights: {weights}, Bias: {bias}")

    return weights, bias

# Example usage (AND gate)

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

targets = np.array([0, 0, 0, 1])

learning\_rate = 0.2

epochs = 3  #for better output formatting

weights, bias = hebb\_rule(inputs, targets, learning\_rate, epochs)

print("\nFinal weights:", weights)

print("Final bias:", bias)

**OUTPUT**:

Initial weights: [0. 0.]

Initial bias: 0

Epoch 1:

Input: [0 0], Target: 0, Weights: [0. 0.], Bias: 0.0

Input: [0 1], Target: 0, Weights: [0. 0.], Bias: 0.0

Input: [1 0], Target: 0, Weights: [0. 0.], Bias: 0.0

Input: [1 1], Target: 1, Weights: [0.5 0.5], Bias: 0.5

Epoch 2:

Input: [0 0], Target: 0, Weights: [0.5 0.5], Bias: 0.5

Input: [0 1], Target: 0, Weights: [0.5 0.5], Bias: 0.5

Input: [1 0], Target: 0, Weights: [0.5 0.5], Bias: 0.5

Input: [1 1], Target: 1, Weights: [1. 1.], Bias: 1.0

Epoch 3:

Input: [0 0], Target: 0, Weights: [1. 1.], Bias: 1.0

Input: [0 1], Target: 0, Weights: [1. 1.], Bias: 1.0

Input: [1 0], Target: 0, Weights: [1. 1.], Bias: 1.0

Input: [1 1], Target: 1, Weights: [1.5 1.5], Bias: 1.5

Epoch 4:

Input: [0 0], Target: 0, Weights: [1.5 1.5], Bias: 1.5

Input: [0 1], Target: 0, Weights: [1.5 1.5], Bias: 1.5

Input: [1 0], Target: 0, Weights: [1.5 1.5], Bias: 1.5

Input: [1 1], Target: 1, Weights: [2. 2.], Bias: 2.0

Final weights: [2. 2.]

Final bias: 2.0

**SFS**:

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.metrics import accuracy\_score

import scipy.io

# Suppress warnings

import warnings

warnings.filterwarnings("ignore")

# Load the dataset

data=pd.read\_csv("/content/lymphography.data")

X=data.drop(["3"],axis=1)

y=data["3"]

# Split 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)

# Define the model

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

# Evaluate model before feature selection

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

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

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

print(f"Initial model accuracy (before feature selection): {initial\_score:.4f}")

# Function for Sequential Forward Selection (SFS) with fewer features

def forward\_selection(X, y, model, max\_features=5, scoring='accuracy', cv=5, min\_improvement=0.001):

    remaining\_features = list(X.columns)

    selected\_features = []

    best\_score = 0

    iteration = 0

    while remaining\_features and len(selected\_features) < max\_features:

        scores = []

        for feature in remaining\_features:

            current\_features = selected\_features + [feature]

            X\_selected = X[current\_features]

            score = cross\_val\_score(model, X\_selected, y, cv=cv, scoring=scoring).mean()

            scores.append((score, feature))

        best\_score\_feature = max(scores, key=lambda x: x[0])

        if best\_score\_feature[0] - best\_score < min\_improvement:

            # Stop if improvement is below the threshold

            print(f"No significant improvement; stopping at iteration {iteration}")

            break

        best\_score = best\_score\_feature[0]

        selected\_features.append(best\_score\_feature[1])

        remaining\_features.remove(best\_score\_feature[1])

        iteration += 1

        print(f"Selected feature: {best\_score\_feature[1]}, Cross-validated score: {best\_score:.4f}")

    return selected\_features

# Function for Sequential Backward Selection (SBS) with fewer features

def backward\_selection(X, y, model, min\_features=5, scoring='accuracy', cv=5, min\_improvement=0.001):

    selected\_features = list(X.columns)

    best\_score = cross\_val\_score(model, X[selected\_features], y, cv=cv, scoring=scoring).mean()

    iteration = 0

    while len(selected\_features) > min\_features:

        scores = []

        for feature in selected\_features:

            current\_features = [f for f in selected\_features if f != feature]

            X\_selected = X[current\_features]

            score = cross\_val\_score(model, X\_selected, y, cv=cv, scoring=scoring).mean()

            scores.append((score, feature))

        best\_score\_feature = max(scores, key=lambda x: x[0])

        if best\_score - best\_score\_feature[0] > min\_improvement:

            # Stop if removing features degrades performance significantly

            print(f"Performance drop detected; stopping at iteration {iteration}")

            break

        best\_score = best\_score\_feature[0]

        selected\_features.remove(best\_score\_feature[1])

        iteration += 1

        print(f"Removed feature: {best\_score\_feature[1]}, Cross-validated score: {best\_score:.4f}")

    return selected\_features

# Apply Forward Selection

print("\nPerforming Sequential Forward Selection (SFS):")

sfs\_selected\_features = forward\_selection(X\_train, y\_train, model, max\_features=5)

# Train and evaluate the model after SFS

X\_train\_sfs = X\_train[sfs\_selected\_features]

X\_test\_sfs = X\_test[sfs\_selected\_features]

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

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

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

print(f"\nModel accuracy after SFS: {sfs\_score:.4f}")

# Apply Backward Selection

print("\nPerforming Sequential Backward Selection (SBS):")

sbs\_selected\_features = backward\_selection(X\_train, y\_train, model, min\_features=5)

# Train and evaluate the model after SBS

X\_train\_sbs = X\_train[sbs\_selected\_features]

X\_test\_sbs = X\_test[sbs\_selected\_features]

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

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

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

print(f"\nModel accuracy after SBS: {sbs\_score:.4f}")

**OUTPUT**

Initial model accuracy (before feature selection): 0.8444

Performing Sequential Forward Selection (SFS):

Selected feature: 4.1, Cross-validated score: 0.7552

Selected feature: 2.5, Cross-validated score: 0.8048

Selected feature: 1.1, Cross-validated score: 0.8248

Selected feature: 2.6, Cross-validated score: 0.8343

No significant improvement; stopping at iteration 4

Model accuracy after SFS: 0.7556

Performing Sequential Backward Selection (SBS):

Removed feature: 2.1, Cross-validated score: 0.8624

Performance drop detected; stopping at iteration 1

Model accuracy after SBS: 0.8667