Exp1

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, r2\_score

# Generate sample data

np.random.seed(42)

X = 2 \* np.random.rand(100, 1)

y = 4 + 3 \* X + np.random.randn(100, 1)

# Split the data

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)

# Make predictions

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

# Evaluate the model

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f'Coefficients: {model.coef\_}')

print(f'Intercept: {model.intercept\_}')

print(f'Mean squared error: {mse:.2f}')

print(f'R² score: {r2:.2f}')

# Plot the results

plt.scatter(X\_test, y\_test, color='blue', label='Actual data')

plt.plot(X\_test, y\_pred, color='red', linewidth=2, label='Prediction')

plt.xlabel('X')

plt.ylabel('y')

plt.title('Linear Regression')

plt.legend()

plt.show()

Exp2

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import make\_classification

# Generate sample classification data

X, y = make\_classification(n\_samples=100, n\_features=2, n\_redundant=0,

n\_informative=2, random\_state=42, n\_clusters\_per\_class=1)

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Train the model

model = LogisticRegression(random\_state=42, max\_iter=1000)

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

# Make predictions

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

y\_prob = model.predict\_proba(X\_test)

# Evaluate the model

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

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

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

print('Confusion Matrix:')

print(conf\_matrix)

print('Classification Report:')

print(classification\_report(y\_test, y\_pred))

# Plot the decision boundary

plt.figure(figsize=(10, 6))

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, cmap='viridis', marker='o', edgecolor='k')

# Create a mesh grid

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.1),

np.arange(y\_min, y\_max, 0.1))

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Logistic Regression Decision Boundary')

plt.colorbar()

plt.show()

Exp3

import numpy as np

import matplotlib.pyplot as plt

from sklearn import svm

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, confusion\_matrix

from sklearn.datasets import make\_classification

# Generate sample classification data

X, y = make\_classification(n\_samples=100, n\_features=2, n\_redundant=0,

n\_informative=2, random\_state=42, n\_clusters\_per\_class=1)

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Train the model

model = svm.SVC(kernel='linear', C=1, probability=True)

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

# Make predictions

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

# Evaluate the model

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

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

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

print('Confusion Matrix:')

print(conf\_matrix)

# Plot the decision boundary

plt.figure(figsize=(10, 6))

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, cmap='viridis', marker='o', edgecolor='k')

# Create a mesh grid

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.1),

np.arange(y\_min, y\_max, 0.1))

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('SVM Decision Boundary')

# Plot support vectors

plt.scatter(model.support\_vectors\_[:, 0], model.support\_vectors\_[:, 1],

s=100, linewidth=1, facecolors='none', edgecolors='black')

plt.show()

Exp4

import numpy as np

import matplotlib.pyplot as plt

class HebbianLearning:

def \_\_init\_\_(self, input\_size):

self.weights = np.zeros(input\_size)

def train(self, X, y, learning\_rate=0.1, epochs=100):

n\_samples = X.shape[0]

for epoch in range(epochs):

for i in range(n\_samples):

# Hebbian update rule: Δw = η \* x \* y

self.weights += learning\_rate \* X[i] \* y[i]

def predict(self, X):

# Simple dot product for prediction

return np.sign(np.dot(X, self.weights))

# Generate sample data for binary classification

np.random.seed(42)

X = np.random.rand(100, 2) \* 2 - 1

y = np.where((X[:, 0] + X[:, 1]) > 0, 1, -1)

# Train Hebbian model

hebbian = HebbianLearning(input\_size=2)

hebbian.train(X, y, learning\_rate=0.01, epochs=100)

# Plot the results

plt.figure(figsize=(10, 6))

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', marker='o', edgecolor='k')

# Plot the decision boundary

x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

y\_min, y\_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100),

np.linspace(y\_min, y\_max, 100))

Z = np.sign(xx \* hebbian.weights[0] + yy \* hebbian.weights[1])

plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Hebbian Learning Decision Boundary')

plt.colorbar()

plt.show()

print(f'Learned weights: {hebbian.weights}')

Exp5

import numpy as np

import matplotlib.pyplot as plt

from sklearn.mixture import GaussianMixture

from sklearn.datasets import make\_blobs

# Generate sample data

n\_samples = 300

X, y\_true = make\_blobs(n\_samples=n\_samples, centers=3, cluster\_std=0.5, random\_state=42)

# Implement EM algorithm using GaussianMixture

gmm = GaussianMixture(n\_components=3, random\_state=42)

gmm.fit(X)

# Make predictions

y\_pred = gmm.predict(X)

# Plot results

plt.figure(figsize=(10, 6))

plt.scatter(X[:, 0], X[:, 1], c=y\_pred, cmap='viridis', marker='o', edgecolor='k')

plt.scatter(gmm.means\_[:, 0], gmm.means\_[:, 1], c='red', s=200, marker='X')

# Draw ellipses for each Gaussian component

from matplotlib.patches import Ellipse

def draw\_ellipse(position, covariance, ax=None, \*\*kwargs):

"""Draw an ellipse with a given position and covariance"""

ax = ax or plt.gca()

# Convert covariance to principal axes

if covariance.shape == (2, 2):

U, s, Vt = np.linalg.svd(covariance)

angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))

width, height = 2 \* np.sqrt(s)

else:

angle = 0

width, height = 2 \* np.sqrt(covariance)

# Draw the Ellipse

for nsig in range(1, 4):

ax.add\_patch(Ellipse(position, nsig \* width, nsig \* height,

angle, \*\*kwargs))

for pos, covar in zip(gmm.means\_, gmm.covariances\_):

draw\_ellipse(pos, covar, alpha=0.2, color='blue')

plt.title('EM Algorithm: Gaussian Mixture Model Clustering')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.show()

print(f'Cluster means: \n{gmm.means\_}')

print(f'Cluster weights: \n{gmm.weights\_}')

Exp6

import numpy as np

import matplotlib.pyplot as plt

class McCullochPittsNeuron:

def \_\_init\_\_(self, weights, threshold):

self.weights = weights

self.threshold = threshold

def activate(self, inputs):

# Calculate weighted sum

weighted\_sum = np.dot(inputs, self.weights)

# Step activation function

return 1 if weighted\_sum >= self.threshold else 0

# Demonstrate logical AND function

and\_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=2)

# Demonstrate logical OR function

or\_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=1)

# Demonstrate logical NOT function (for one input)

not\_neuron = McCullochPittsNeuron(weights=[-1], threshold=0)

# Test inputs

inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

not\_inputs = [0, 1]

# Evaluate the neurons

print("McCulloch-Pitts Neuron Demonstrations:")

print("\nAND Gate:")

for input\_pair in inputs:

output = and\_neuron.activate(input\_pair)

print(f"Input: {input\_pair}, Output: {output}")

print("\nOR Gate:")

for input\_pair in inputs:

output = or\_neuron.activate(input\_pair)

print(f"Input: {input\_pair}, Output: {output}")

print("\nNOT Gate:")

for input\_val in not\_inputs:

output = not\_neuron.activate([input\_val])

print(f"Input: {input\_val}, Output: {output}")

# Visualize the decision boundaries for AND and OR

plt.figure(figsize=(12, 5))

# AND gate

plt.subplot(1, 2, 1)

x = np.linspace(-0.5, 1.5, 100)

y = (and\_neuron.threshold - and\_neuron.weights[0] \* x) / and\_neuron.weights[1]

plt.plot(x, y, 'r-', label='Decision Boundary')

plt.scatter([0, 0, 1, 1], [0, 1, 0, 1], c=['blue', 'blue', 'blue', 'red'])

plt.annotate('(0,0)=0', (0, 0), xytext=(-0.1, -0.1))

plt.annotate('(0,1)=0', (0, 1), xytext=(-0.1, 1.1))

plt.annotate('(1,0)=0', (1, 0), xytext=(1.1, -0.1))

plt.annotate('(1,1)=1', (1, 1), xytext=(1.1, 1.1))

plt.xlim(-0.5, 1.5)

plt.ylim(-0.5, 1.5)

plt.xlabel('Input 1')

plt.ylabel('Input 2')

plt.title('AND Gate')

plt.grid(True)

# OR gate

plt.subplot(1, 2, 2)

y = (or\_neuron.threshold - or\_neuron.weights[0] \* x) / or\_neuron.weights[1]

plt.plot(x, y, 'r-', label='Decision Boundary')

plt.scatter([0, 0, 1, 1], [0, 1, 0, 1], c=['blue', 'red', 'red', 'red'])

plt.annotate('(0,0)=0', (0, 0), xytext=(-0.1, -0.1))

plt.annotate('(0,1)=1', (0, 1), xytext=(-0.1, 1.1))

plt.annotate('(1,0)=1', (1, 0), xytext=(1.1, -0.1))

plt.annotate('(1,1)=1', (1, 1), xytext=(1.1, 1.1))

plt.xlim(-0.5, 1.5)

plt.ylim(-0.5, 1.5)

plt.xlabel('Input 1')

plt.ylabel('Input 2')

plt.title('OR Gate')

plt.grid(True)

plt.tight\_layout()

plt.show()

Exp7

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

class Perceptron:

def \_\_init\_\_(self, learning\_rate=0.01, n\_iterations=1000):

self.lr = learning\_rate

self.n\_iterations = n\_iterations

self.weights = None

self.bias = None

def fit(self, X, y):

n\_samples, n\_features = X.shape

# Initialize weights and bias

self.weights = np.zeros(n\_features)

self.bias = 0

# Convert y to bipolar if not already (-1, 1)

y\_ = np.where(y <= 0, -1, 1)

# Weight update history for visualization

self.weight\_history = []

# Training

for \_ in range(self.n\_iterations):

for idx, x\_i in enumerate(X):

linear\_output = np.dot(x\_i, self.weights) + self.bias

y\_predicted = 1 if linear\_output >= 0 else -1

# Perceptron update rule

if y\_predicted != y\_[idx]:

self.weights += self.lr \* y\_[idx] \* x\_i

self.bias += self.lr \* y\_[idx]

self.weight\_history.append((self.weights.copy(), self.bias))

return self

def predict(self, X):

linear\_output = np.dot(X, self.weights) + self.bias

return np.where(linear\_output >= 0, 1, 0)

# Generate linearly separable data

np.random.seed(42)

X = np.random.randn(100, 2)

y = np.where(X[:, 0] + X[:, 1] > 0, 1, 0) # A simple linear boundary

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train the perceptron

perceptron = Perceptron(learning\_rate=0.01, n\_iterations=1000)

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

# Make predictions

y\_pred = perceptron.predict(X\_test)

# Evaluate the model

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

print(f"Perceptron accuracy: {accuracy:.2f}")

# Plot the decision boundary

plt.figure(figsize=(10, 6))

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', marker='o', edgecolor='k')

# Create a mesh grid

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.1),

np.arange(y\_min, y\_max, 0.1))

# Predict labels for all mesh grid points

Z = perceptron.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Single Layer Perceptron Decision Boundary')

# Plot the decision line: w1\*x1 + w2\*x2 + b = 0

# Rearranged as x2 = (-w1\*x1 - b) / w2

slope = -perceptron.weights[0] / perceptron.weights[1]

intercept = -perceptron.bias / perceptron.weights[1]

x\_line = np.linspace(x\_min, x\_max, 100)

y\_line = slope \* x\_line + intercept

plt.plot(x\_line, y\_line, 'r-', label=f'Decision Boundary: {slope:.2f}\*x + {intercept:.2f}')

plt.legend()

plt.show()