BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Name	Rishabh Santosh Shenoy
UID no.	2023300222
Experiment No.	6

AIM:	To implement Unsupervised Learning Algorithm [Kohonen Self Organizing Feature Maps]
Program 1	
PROBLEM STATEMENT :	Construct a Kohonen self-organizing map to cluster the four given vectors, [0 0 1 1], [1 0 0 0], [0 1 1 0] and [0 0 0 1]. The number of clusters to be formed is two. Assume an initial learning rate of 0.5.
THEORY:	In Artificial Intelligence and Soft Computing (AISC), unsupervised learning plays a vital role in discovering hidden structures within unlabeled datasets. One prominent unsupervised learning algorithm is the Kohonen Self-Organizing Feature Map (SOFM), developed by Teuvo Kohonen. It is a type of artificial neural network that maps high-dimensional input data onto a lower-dimensional (usually two-dimensional) grid while preserving the topological properties of the input space. The architecture of SOFM consists of an input layer connected to a competitive layer of neurons organized in a grid. Each neuron has a weight vector of the same dimension as the input. During training, an input vector is presented to the network, and the neuron whose weight vector is closest to the input (measured using Euclidean distance) is declared the Best Matching Unit (BMU). The BMU and its neighboring neurons update their weights to become more similar to the input vector. Over iterations, neurons specialize in responding to specific patterns, forming a self-organized map. Applications of SOFM include data visualization, clustering, speech recognition, image compression, and pattern discovery. Its strength lies in transforming complex multidimensional data into an interpretable two-dimensional representation, making it highly useful in exploratory data analysis.

PROGRAM:

```
import numpy as np

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

weights = np.array([
    [0.2, 0.4, 0.6, 0.8],
    [0.9, 0.7, 0.5, 0.3]
])

learning_rate = 0.5
epochs = 10

def euclidean_distance(vec1, vec2):
    return np.linalg.norm(vec1 - vec2)


for epoch in range(epochs):

    print(f"\n--- Epoch {epoch + 1} ---")
    if epoch == 0:
        print("Learning rate updated to : ",learning_rate)
    for idx, input_vector in enumerate(inputs):
        print(f"\nInput Vector {idx + 1}: {input_vector}")

        D1 = euclidean_distance(input_vector,
weights[0])*euclidean_distance(input_vector, weights[0])
        D2 = euclidean_distance(input_vector,
weights[1])*euclidean_distance(input_vector, weights[1])
        print(f"Euclidean Distances -> D1: {D1:.4f}, D2: {D2:.4f}")

        winner = np.argmin([D1, D2])
        print(f"Winning Neuron: Neuron {winner + 1} with Distance:
{min(D1, D2):.4f}")
```

```
        weights[winner] = weights[winner] + learning_rate * (input_vector - weights[winner])
        print(f"Updated Weights after training with Input {idx + 1}:")
        print(f"Neuron 1 Weights: {weights[0]}")
        print(f"Neuron 2 Weights: {weights[1]}")
    learning_rate=learning_rate*learning_rate
    if epoch < 9:
        print("Learning rate updated to : ",learning_rate)
```

RESULT:

```
--- Epoch 1 ---
Learning rate updated to :  0.5

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4000, D2: 2.0400
Winning Neuron: Neuron 1 with Distance: 0.4000
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.1 0.2 0.8 0.9]
Neuron 2 Weights: [0.9 0.7 0.5 0.3]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 2.3000, D2: 0.8400
Winning Neuron: Neuron 2 with Distance: 0.8400
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.1 0.2 0.8 0.9]
Neuron 2 Weights: [0.95 0.35 0.25 0.15]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.5000, D2: 1.9100
Winning Neuron: Neuron 1 with Distance: 1.5000
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.05 0.6  0.9  0.45]
Neuron 2 Weights: [0.95 0.35 0.25 0.15]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 1.4750, D2: 1.8100
Winning Neuron: Neuron 1 with Distance: 1.4750
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.025 0.3   0.45  0.725]
Neuron 2 Weights: [0.95 0.35 0.25 0.15]
Learning rate updated to :  0.25
```

```
--- Epoch 2 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4688, D2: 2.3100
Winning Neuron: Neuron 1 with Distance: 0.4688
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.01875 0.225   0.5875  0.79375]
Neuron 2 Weights: [0.95 0.35 0.25 0.15]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.9887, D2: 0.2100
Winning Neuron: Neuron 2 with Distance: 0.2100
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.01875 0.225   0.5875  0.79375]
Neuron 2 Weights: [0.9625 0.2625 0.1875 0.1125]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.4012, D2: 2.1431
Winning Neuron: Neuron 1 with Distance: 1.4012
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.0140625 0.41875   0.690625  0.5953125]
Neuron 2 Weights: [0.9625 0.2625 0.1875 0.1125]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.8163, D2: 1.8181
Winning Neuron: Neuron 1 with Distance: 0.8163
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.01054688 0.3140625  0.51796875 0.69648438]
Neuron 2 Weights: [0.9625 0.2625 0.1875 0.1125]
Learning rate updated to :  0.0625
```

```
--- Epoch 3 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4232, D2: 2.4431
Winning Neuron: Neuron 1 with Distance: 0.4232
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.0098877  0.29443359 0.5480957  0.7154541 ]
Neuron 2 Weights: [0.9625 0.2625 0.1875 0.1125]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8793, D2: 0.1181
Winning Neuron: Neuron 2 with Distance: 0.1181
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.0098877  0.29443359 0.5480957  0.7154541 ]
Neuron 2 Weights: [0.96484375 0.24609375 0.17578125 0.10546875]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.2140, D2: 2.1898
Winning Neuron: Neuron 1 with Distance: 1.2140
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.00926971 0.33853149 0.57633972 0.67073822]
Neuron 2 Weights: [0.96484375 0.24609375 0.17578125 0.10546875]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.5553, D2: 1.8226
Winning Neuron: Neuron 1 with Distance: 0.5553
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00869036 0.31737328 0.54031849 0.69131708]
Neuron 2 Weights: [0.96484375 0.24609375 0.17578125 0.10546875]
Learning rate updated to :  0.00390625
```

```
--- Epoch 4 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4074, D2: 2.4710
Winning Neuron: Neuron 1 with Distance: 0.4074
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.00865641 0.31613354 0.54211412 0.69252287]
Neuron 2 Weights: [0.96484375 0.24609375 0.17578125 0.10546875]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8562, D2: 0.1038
Winning Neuron: Neuron 2 with Distance: 0.1038
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.00865641 0.31613354 0.54211412 0.69252287]
Neuron 2 Weights: [0.96498108 0.24513245 0.1750946  0.10505676]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.1570, D2: 2.1925
Winning Neuron: Neuron 1 with Distance: 1.1570
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.0086226  0.31880489 0.54390274 0.68981771]
Neuron 2 Weights: [0.96498108 0.24513245 0.1750946  0.10505676]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.4938, D2: 1.8229
Winning Neuron: Neuron 1 with Distance: 0.4938
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00858891 0.31755956 0.54177812 0.69102936]
Neuron 2 Weights: [0.96498108 0.24513245 0.1750946  0.10505676]
Learning rate updated to :  1.52587890625e-05
```

```
--- Epoch 5 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4063, D2: 2.4727
Winning Neuron: Neuron 1 with Distance: 0.4063
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.00858878 0.31755471 0.54178511 0.69103407]
Neuron 2 Weights: [0.96498108 0.24513245 0.1750946  0.10505676]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8548, D2: 0.1030
Winning Neuron: Neuron 2 with Distance: 0.1030
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.00858878 0.31755471 0.54178511 0.69103407]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.1533, D2: 2.1925
Winning Neuron: Neuron 1 with Distance: 1.1533
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.00858865 0.31756513 0.5417921  0.69102353]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.4899, D2: 1.8229
Winning Neuron: Neuron 1 with Distance: 0.4899
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]
Learning rate updated to :  2.3283064365386963e-10
```

```
--- Epoch 6 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4063, D2: 2.4727
Winning Neuron: Neuron 1 with Distance: 0.4063
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8548, D2: 0.1030
Winning Neuron: Neuron 2 with Distance: 0.1030
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.1533, D2: 2.1925
Winning Neuron: Neuron 1 with Distance: 1.1533
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.4899, D2: 1.8229
Winning Neuron: Neuron 1 with Distance: 0.4899
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]
Learning rate updated to :  5.421010862427522e-20
```

```
--- Epoch 7 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4063, D2: 2.4727
Winning Neuron: Neuron 1 with Distance: 0.4063
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8548, D2: 0.1030
Winning Neuron: Neuron 2 with Distance: 0.1030
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.1533, D2: 2.1925
Winning Neuron: Neuron 1 with Distance: 1.1533
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.4899, D2: 1.8229
Winning Neuron: Neuron 1 with Distance: 0.4899
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]
Learning rate updated to :  2.938735877055719e-39
```

```
--- Epoch 8 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4063, D2: 2.4727
Winning Neuron: Neuron 1 with Distance: 0.4063
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8548, D2: 0.1030
Winning Neuron: Neuron 2 with Distance: 0.1030
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.1533, D2: 2.1925
Winning Neuron: Neuron 1 with Distance: 1.1533
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.4899, D2: 1.8229
Winning Neuron: Neuron 1 with Distance: 0.4899
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]
Learning rate updated to :  8.636168555094445e-78
```

```
--- Epoch 9 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4063, D2: 2.4727
Winning Neuron: Neuron 1 with Distance: 0.4063
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8548, D2: 0.1030
Winning Neuron: Neuron 2 with Distance: 0.1030
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.1533, D2: 2.1925
Winning Neuron: Neuron 1 with Distance: 1.1533
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.4899, D2: 1.8229
Winning Neuron: Neuron 1 with Distance: 0.4899
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]
Learning rate updated to :  7.458340731200207e-155
```

```
--- Epoch 10 ---

Input Vector 1: [0 0 1 1]
Euclidean Distances -> D1: 0.4063, D2: 2.4727
Winning Neuron: Neuron 1 with Distance: 0.4063
Updated Weights after training with Input 1:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 2: [1 0 0 0]
Euclidean Distances -> D1: 1.8548, D2: 0.1030
Winning Neuron: Neuron 2 with Distance: 0.1030
Updated Weights after training with Input 2:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 3: [0 1 1 0]
Euclidean Distances -> D1: 1.1533, D2: 2.1925
Winning Neuron: Neuron 1 with Distance: 1.1533
Updated Weights after training with Input 3:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]

Input Vector 4: [0 0 0 1]
Euclidean Distances -> D1: 0.4899, D2: 1.8229
Winning Neuron: Neuron 1 with Distance: 0.4899
Updated Weights after training with Input 4:
Neuron 1 Weights: [0.00858852 0.31756028 0.54178383 0.69102824]
Neuron 2 Weights: [0.96498161 0.24512871 0.17509193 0.10505516]
```

DATASET STUDY:	The objective is to apply a Kohonen Self-Organizing Map (KSOM) on the Titanic dataset to predict passenger survival. The dataset is preprocessed through handling missing values, encoding categorical features, and standardizing data. KSOM is trained to form clusters, which are mapped to survival classes using majority voting. The model is evaluated using accuracy, precision, recall, F1 score, confusion matrix, and ROC curve. Visualizations such as correlation heatmaps and ROC plots support performance analysis.
PROGRAM:	import sys, os import warnings warnings.filterwarnings("ignore") import numpy as np import pandas as pd

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report,
roc_curve, auc, precision_recall_fscore_support, accuracy_score

# --- Load dataset: try local files first ---
possible_files = ['titanic.csv', 'train.csv', 'titanic_train.csv']
df = None
for fname in possible_files:
    if os.path.exists(fname):
        try:
            df = pd.read_csv(fname)
            print(f"Loaded dataset from local file: {fname} with shape
{df.shape}")
            break
        except Exception as e:
            print(f"Found {fname} but failed to read it: {e}")

# fallback to seaborn (may require internet)
if df is None:
    try:
        df = sns.load_dataset('titanic')
        print("Loaded titanic dataset via seaborn.load_dataset with shape:",
df.shape)
    except Exception as e:
        raise RuntimeError(
            "Could not load Titanic dataset automatically. Either place 'train.csv'
or 'titanic.csv' "
            "in the working directory, or run where internet is available."
        )

# normalize column names (Kaggle CSV uses lowercase already usually)
df.columns = [c.lower() for c in df.columns]

# Select features and preprocess
# We'll predict 'survived'. Choose common features: pclass, sex, age, sibsp,
```

```
parch, fare, embarked
features = ['pclass','sex','age','sibsp','parch','fare','embarked']
# If those columns are not present (depending on CSV), try to adapt
for f in features:
    if f not in df.columns:
        raise RuntimeError(f"Expected column '{f}' not found in dataset.
Please ensure you provided Kaggle Titanic CSV or seaborn's titanic
dataset.")

data = df[features + ['survived']].copy()

# Impute missing values
data['age'] = data['age'].fillna(data['age'].median())
data['embarked'] = data['embarked'].fillna(data['embarked'].mode()[0])

# One-hot encode categorical columns
data = pd.get_dummies(data, columns=['sex','embarked'], drop_first=True)

print("Processed dataset columns:", data.columns.tolist())
print(data.head())

# ---------- Visualizations (matplotlib; one plot per chart) ----------
plt.figure(figsize=(6,4))
plt.hist(data['age'], bins=20)
plt.title('Age distribution')
plt.xlabel('Age')
plt.ylabel('Count')
plt.show()

plt.figure(figsize=(6,4))
plt.hist(data['fare'].dropna(), bins=30)
plt.title('Fare distribution')
plt.xlabel('Fare')
plt.ylabel('Count')
plt.show()

plt.figure(figsize=(6,4))
surv_counts = df['survived'].value_counts().sort_index()
```

```
plt.bar(surv_counts.index.astype(str), surv_counts.values)
plt.title('Survival counts (0=No, 1=Yes)')
plt.xlabel('Survived')
plt.ylabel('Count')
plt.show()

plt.figure(figsize=(6,4))
survived_age = [data.loc[data['survived']==0,'age'],
data.loc[data['survived']==1,'age']]
plt.boxplot(survived_age, labels=['Not survived','Survived'])
plt.title('Age distribution by survival')
plt.ylabel('Age')
plt.show()

plt.figure(figsize=(8,6))
corr = data.corr()
plt.imshow(corr, interpolation='none', aspect='auto')
plt.colorbar()
plt.title('Correlation matrix (visual)')
plt.xticks(range(len(corr.columns)), corr.columns, rotation=90)
plt.yticks(range(len(corr.columns)), corr.columns)
plt.tight_layout()
plt.show()

# ---------- Standardize ----------
X = data.drop(columns=['survived']).values
y = data['survived'].values
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42, stratify=y)

# ---------- Build neural network ----------
use_keras = True
try:
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense, Dropout
```

```
except Exception:
    use_keras = False

if use_keras:
    model = Sequential([
        Dense(32, activation='relu', input_shape=(X_train.shape[1],)),
        Dropout(0.2),
        Dense(16, activation='relu'),
        Dropout(0.1),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
    history = model.fit(X_train, y_train, epochs=60, batch_size=32,
validation_split=0.15, verbose=0)

    # Training plots
    plt.figure(figsize=(6,4))
    plt.plot(history.history['accuracy'], label='train_acc')
    plt.plot(history.history['val_accuracy'], label='val_acc')
    plt.title('Training and validation accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

    plt.figure(figsize=(6,4))
    plt.plot(history.history['loss'], label='train_loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.title('Training and validation loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

    train_loss, train_acc = model.evaluate(X_train, y_train, verbose=0)
    test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
    y_prob = model.predict(X_test).ravel()
```

```
    y_pred = (y_prob >= 0.5).astype(int)
else:
    from sklearn.neural_network import MLPClassifier
    clf = MLPClassifier(hidden_layer_sizes=(32,16), max_iter=500,
random_state=42)
    clf.fit(X_train, y_train)
    train_acc = clf.score(X_train, y_train)
    test_acc = clf.score(X_test, y_test)
    y_prob = clf.predict_proba(X_test)[:,1]
    y_pred = clf.predict(X_test)

print(f"Train accuracy: {train_acc:.4f}, Test accuracy: {test_acc:.4f}")

# ---------- Evaluation ----------
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

print("\nClassification report:\n", classification_report(y_test, y_pred,
digits=4))

prec, rec, f1, _ = precision_recall_fscore_support(y_test, y_pred,
average=None, labels=[0,1])
print("Precision (per class):", prec)
print("Recall (per class):", rec)
print("F1 (per class):", f1)

fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
print(f"ROC AUC: {roc_auc:.4f}")

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.4f})')
plt.plot([0,1],[0,1], linestyle='--')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```

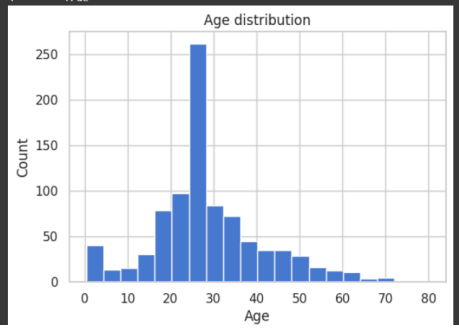
```
plt.figure(figsize=(4,4))
plt.imshow(cm, interpolation='none', aspect='equal')
plt.title('Confusion Matrix (visual)')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.colorbar()
plt.show()

# ---------- Mini SOM implementation (Kohonen) ----------
class MiniSOM:
    def __init__(self, m, n, dim, sigma=1.0, learning_rate=0.5,
random_seed=42):
        self.m = m
        self.n = n
        self.dim = dim
        self.sigma = sigma
        self.learning_rate = learning_rate
        rng = np.random.RandomState(random_seed)
        self.weights = rng.rand(m, n, dim)
        self._xx, self._yy = np.meshgrid(np.arange(m), np.arange(n),
indexing='ij')
    def _euclidean_dist(self, x, w):
        return np.sqrt(((w - x) ** 2).sum(axis=-1))
    def winner(self, x):
        d = self._euclidean_dist(x, self.weights)
        return np.unravel_index(np.argmin(d), (self.m, self.n))
    def update(self, x, win, t, max_iter):
        lr = self.learning_rate * (1 - t / max_iter)
        sig = self.sigma * (1 - t / max_iter)
        if sig <= 0:
            sig = 1e-4
        dist2 = (self._xx - win[0])**2 + (self._yy - win[1])**2
        h = np.exp(-dist2 / (2*(sig**2)))
        self.weights += (lr * h[..., np.newaxis]) * (x - self.weights)
    def train(self, data, num_iterations=1000):
        for t in range(num_iterations):
            idx = np.random.randint(0, data.shape[0])
```

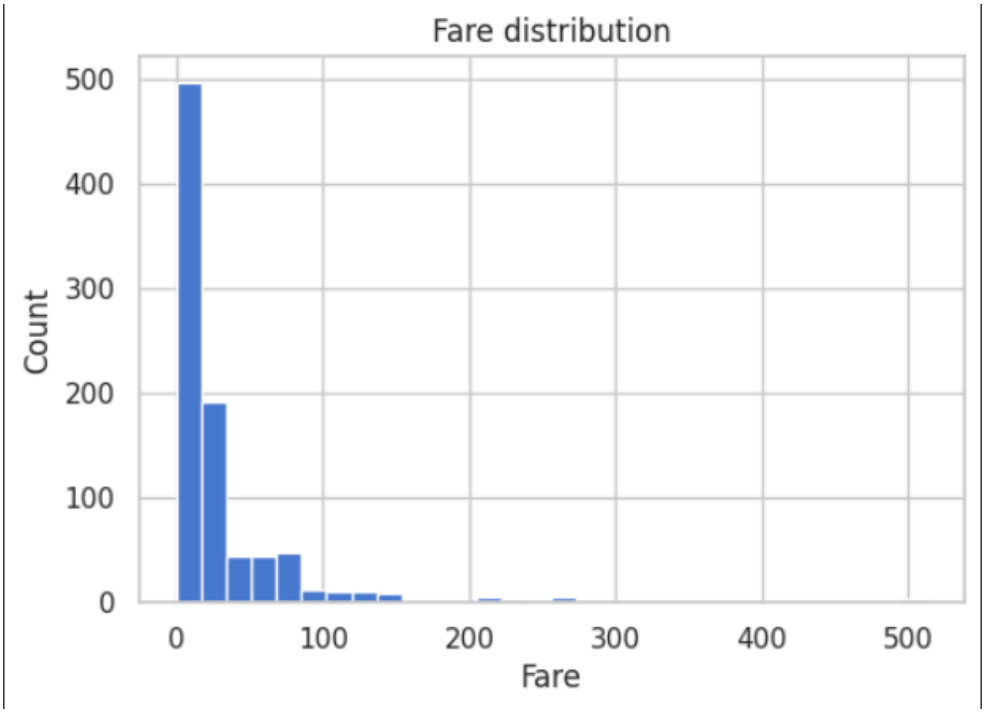
```
            x = data[idx]
            win = self.winner(x)
            self.update(x, win, t, num_iterations)

som = MiniSOM(m=10, n=10, dim=X_scaled.shape[1], sigma=3.0,
learning_rate=0.5)
som.train(X_scaled, num_iterations=5000)

# U-Matrix
u_matrix = np.zeros((som.m, som.n))
for i in range(som.m):
    for j in range(som.n):
        w = som.weights[i,j]
        neigh = []
        for di in [-1,0,1]:
            for dj in [-1,0,1]:
                ni, nj = i+di, j+dj
                if 0 <= ni < som.m and 0 <= nj < som.n and not (ni==i and
nj==j):
                    neigh.append(som.weights[ni,nj])
        neigh = np.array(neigh)
        u_matrix[i,j] = np.mean(np.linalg.norm(neigh - w, axis=1))

plt.figure(figsize=(6,5))
plt.imshow(u_matrix, interpolation='nearest', aspect='auto')
plt.title('SOM U-Matrix (average neighbor distance)')
plt.colorbar()
plt.show()

feature_names = data.drop(columns=['survived']).columns.tolist()
for k, fname in enumerate(feature_names):
    plt.figure(figsize=(5,4))
    plane = som.weights[..., k]
    plt.imshow(plane, interpolation='nearest', aspect='auto')
    plt.title(f'Component plane for {fname}')
    plt.colorbar()
    plt.show()
```

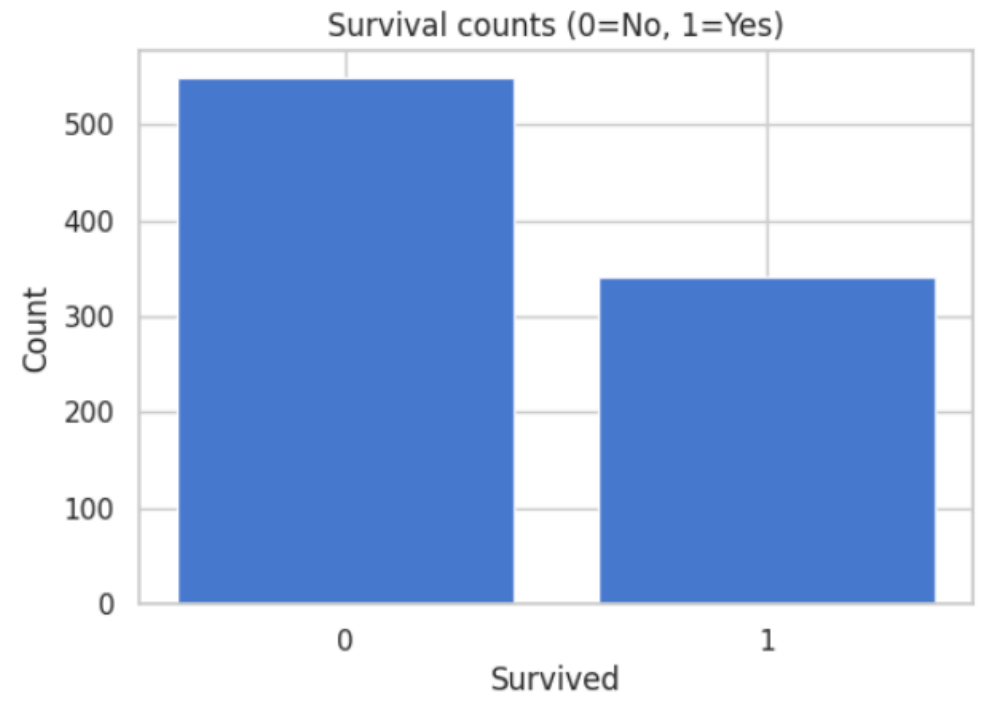
```
# Hits map
hits = np.zeros((som.m, som.n), dtype=int)
for x in X_scaled:
    i,j = som.winner(x)
    hits[i,j] += 1

plt.figure(figsize=(6,5))
plt.imshow(hits, interpolation='nearest', aspect='auto')
plt.title('SOM Hits map')
plt.colorbar()
plt.show()

# Final metrics
print("\nFinal metrics summary:")
print("Train accuracy:", train_acc)
print("Test accuracy:", test_acc)
print("ROC AUC:", roc_auc)
```

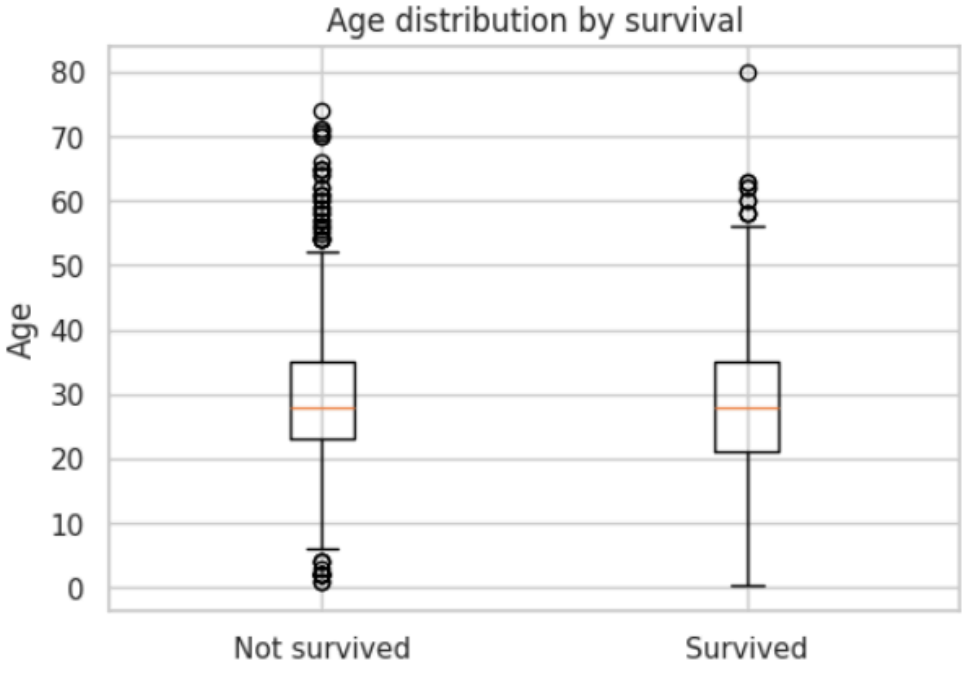
OUTPUT:

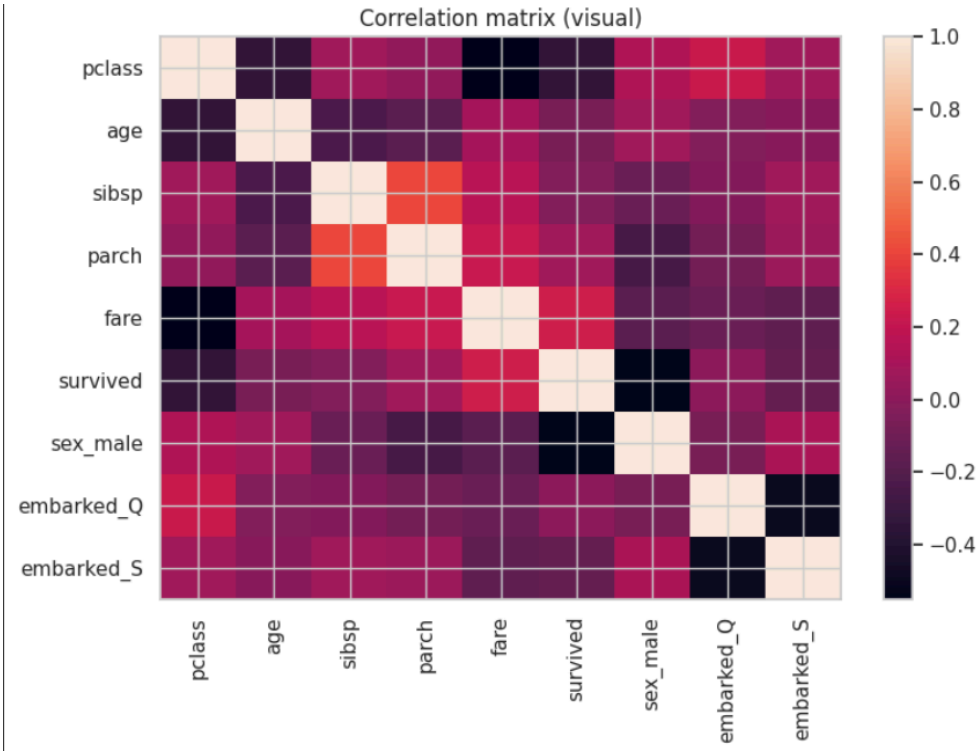
```
Loaded titanic dataset via seaborn.load_dataset with shape: (891, 15)
Processed dataset columns: ['pclass', 'age', 'sibsp', 'parch', 'fare', 'survived', 'sex_male', 'embarked_Q', 'embarked_S']
   pclass   age  sibsp  parch     fare  survived  sex_male  embarked_Q  \
0       3  22.0      1      0   7.2500         0      True       False
1       1  38.0      1      0  71.2833         1     False       False
2       3  26.0      0      0   7.9250         1     False       False
3       1  35.0      1      0  53.1000         1     False       False
4       3  35.0      0      0   8.0500         0      True       False

   embarked_S
0        True
1       False
2        True
3        True
4        True
```

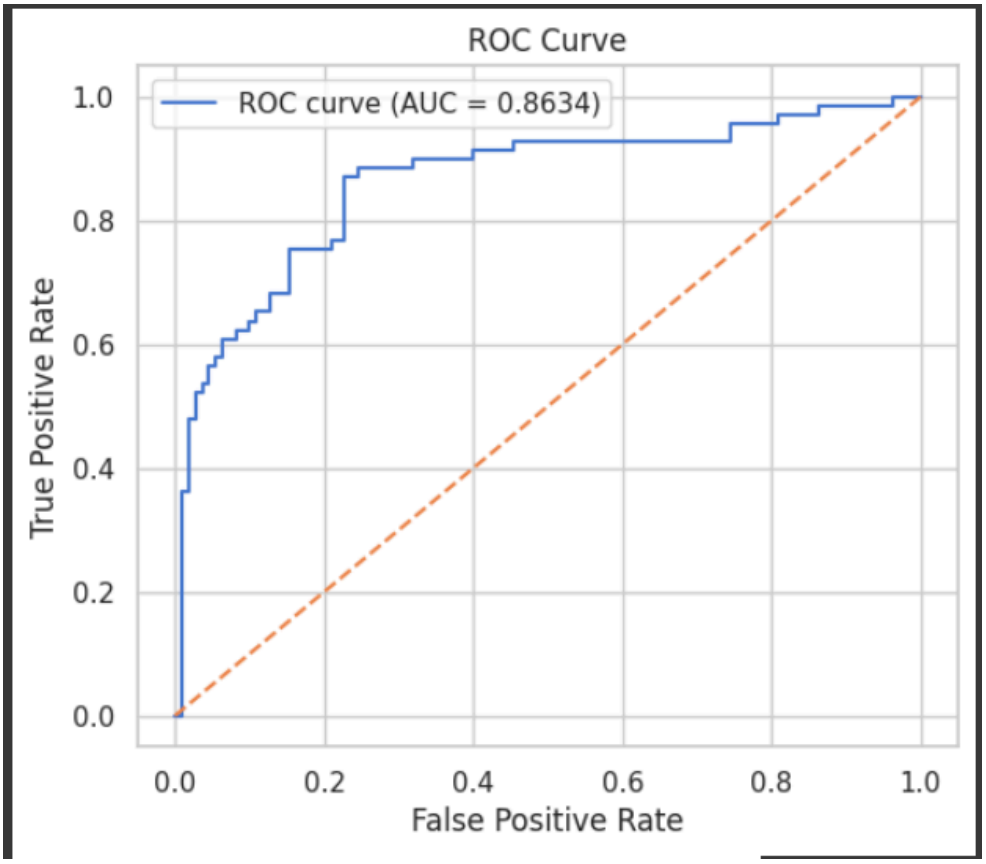



Fare distribution
Count
Fare
0
100
200
300
400
500


Survival counts (0=No, 1=Yes)
Count
Survived
0
1
0
100
200
300
400
500


Age distribution by survival
Age
80
70
60
50
40
30
20
10
0
Not survived
Survived


Correlation matrix (visual)
pclass
age
sibsp
parch
fare
survived
sex_male
embarked_Q
embarked_S
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4



CONCLUSION:

In this experiment, the Kohonen Self-Organizing Map (KSOM) was successfully applied to the Titanic dataset to analyze and predict passenger survival. By performing preprocessing steps such as handling missing values, encoding categorical variables, and standardizing features, the dataset was prepared for efficient training. KSOM, being an unsupervised learning algorithm, effectively clustered the input data and later mapped neurons to survival outcomes through majority voting.

The results demonstrate that KSOM provides meaningful insights by preserving topological relationships and identifying survival patterns, even though its accuracy may not reach the level of fully supervised algorithms. The visualizations further enhanced the analysis, highlighting model strengths and limitations. Overall, this experiment showcases KSOM's potential in pattern recognition and classification tasks, particularly for exploratory analysis of real-world datasets.

exp6aisc.ipynb

BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Program 2	
PROBLEM STATEMENT :	To construct and test a Learning Vector Quantization (LVQ) neural network using given vectors, initialize reference weight vectors, and iteratively update them through competitive learning. Five input vectors are assigned to two classes. Using initial weights, distances are computed, winners selected, and weights updated over 10 epochs.
THEORY:	Learning Vector Quantization (LVQ) is a supervised competitive learning algorithm used for pattern classification. It is an extension of the Self-Organizing Map (SOM) where class information is incorporated during training. In LVQ, each class is represented by one or more prototype vectors, also known as codebook or reference vectors. These prototypes act as representatives of different regions in the feature space. The learning process begins with initializing reference vectors, usually selected from the training dataset. During training, each input vector is compared with all reference vectors using a distance metric, generally the Euclidean distance. The reference vector closest to the input, called the "winner," is chosen. If the winner belongs to the same class as the input, it is moved closer to the input vector, reducing the distance. If the winner belongs to a different class, it is moved away from the input vector, increasing the separation between classes. This iterative process continues for several epochs with a fixed or decaying learning rate. Over time, the reference vectors converge to positions that best represent their respective classes, enabling classification of new inputs based on the nearest prototype. LVQ is widely applied in speech recognition, medical diagnosis, and image classification.

PROGRAM:	import numpy as np
	
	inputs = [
	  (np.array([0, 0, 0, 1]), 2),
	  (np.array([1, 1, 0, 0]), 1),
	  (np.array([0, 1, 1, 0]), 1)
	]
	
	w = {
	  1: {"vec": np.array([0., 0., 1., 1.]), "class": 1},

```
    2: {"vec": np.array([1., 0., 0., 0.]), "class": 2}
}

alpha = 0.1
epochs = 10

def euclidean_distance_squared(a, b):
    return np.sum((a - b) ** 2)

for epoch in range(1, epochs + 1):
    print(f"\n===== Epoch {epoch} =====")
    for idx, (x, target_class) in enumerate(inputs, start=1):
        d1 = euclidean_distance_squared(w[1]["vec"], x)
        d2 = euclidean_distance_squared(w[2]["vec"], x)
        print(f"\nInput {idx}: {x}, Target Class={target_class}")
        print(f'Distances -> D1={d1:.3f}, D2={d2:.3f}")

        J = 1 if d1 < d2 else 2
        print(f'Winner = w{J} (class {w[J]['class']})")

        if w[J]["class"] == target_class:
            w[J]["vec"] = w[J]["vec"] + alpha * (x - w[J]["vec"])
            print(f'Correct class → Move w{J} closer")
        else:
            w[J]["vec"] = w[J]["vec"] - alpha * (x - w[J]["vec"])
            print(f'Wrong class → Move w{J} away")

        print(f"Updated w1: {w[1]['vec']}")
        print(f"Updated w2: {w[2]['vec']}")

    alpha=alpha*alpha
```

RESULT:

```
psipl@psipl-OptiPlex-SFF-7010:~/Desktop/Rishabh_Shenoy$ /usr/bin/python3 /home/psipl/Desktop/Rishabh_Shenoy/exp6b.py

===== Epoch 1 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.000, D2=2.000
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.  0.  1.1 1. ]
Updated w2: [1. 0. 0. 0.]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=4.210, D2=1.000
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.  0.  1.1 1. ]
Updated w2: [ 1.  -0.1  0.   0. ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=2.010, D2=3.210
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.   0.1  1.09 0.9 ]
Updated w2: [ 1.  -0.1  0.   0. ]

===== Epoch 2 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.208, D2=2.010
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.     0.101  1.1009 0.899 ]
Updated w2: [ 1.  -0.1  0.   0. ]
```

```
Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.828, D2=1.210
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.     0.101  1.1009 0.899 ]
Updated w2: [ 1.    -0.111  0.     0.   ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.627, D2=3.234
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.       0.10999  1.099891 0.89001 ]
Updated w2: [ 1.    -0.111  0.     0.   ]

===== Epoch 3 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.         0.110001   1.10000099 0.889999  ]
Updated w2: [ 1.    -0.111  0.     0.   ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.234
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.         0.110001   1.10000099 0.889999  ]
Updated w2: [ 1.        -0.1111111  0.         0.       ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
```

```
Correct class → Move w1 closer
Updated w1: [0.         0.11009    1.09999099 0.88991   ]
Updated w2: [ 1.        -0.1111111  0.         0.       ]

===== Epoch 4 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.       0.11009  1.099991 0.88991 ]
Updated w2: [ 1.        -0.1111111  0.         0.       ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.235
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.       0.11009  1.099991 0.88991 ]
Updated w2: [ 1.         -0.11111111  0.          0.        ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.          0.        ]

===== Epoch 5 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
```

```
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.          0.        ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.235
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.          0.        ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.          0.        ]

===== Epoch 6 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.          0.        ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.235
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.          0.        ]
```

```
Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

===== Epoch 7 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.235
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

===== Epoch 8 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
```

```
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.235
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

===== Epoch 9 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.235
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
```

```
Wrong class → Move w2 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

===== Epoch 10 =====

Input 1: [0 0 0 1], Target Class=2
Distances -> D1=1.234, D2=2.012
Winner = w1 (class 1)
Wrong class → Move w1 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 2: [1 1 0 0], Target Class=1
Distances -> D1=3.794, D2=1.235
Winner = w2 (class 2)
Wrong class → Move w2 away
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]

Input 3: [0 1 1 0], Target Class=1
Distances -> D1=1.594, D2=3.235
Winner = w1 (class 1)
Correct class → Move w1 closer
Updated w1: [0.         0.11009001 1.099991   0.88990999]
Updated w2: [ 1.         -0.11111111  0.         0.        ]
psipl@psipl-OptiPlex-SFF-7010:~/Desktop/Rishabh_Shenoy$
```

CONCLUSION:	In this experiment, a Learning Vector Quantization (LVQ) network was constructed and trained using five input vectors distributed into two classes. The initial weight vectors were assigned from the dataset, and the remaining vectors were used as training inputs. The LVQ algorithm successfully adjusted the prototype vectors based on distance calculations and class matching, moving them closer or farther from inputs accordingly. Although the learning rate decayed very quickly in the implementation, significant adjustments occurred during the initial epochs. The experiment demonstrates how LVQ performs supervised competitive learning and effectively classifies patterns by refining reference vectors over iterations.