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.	4

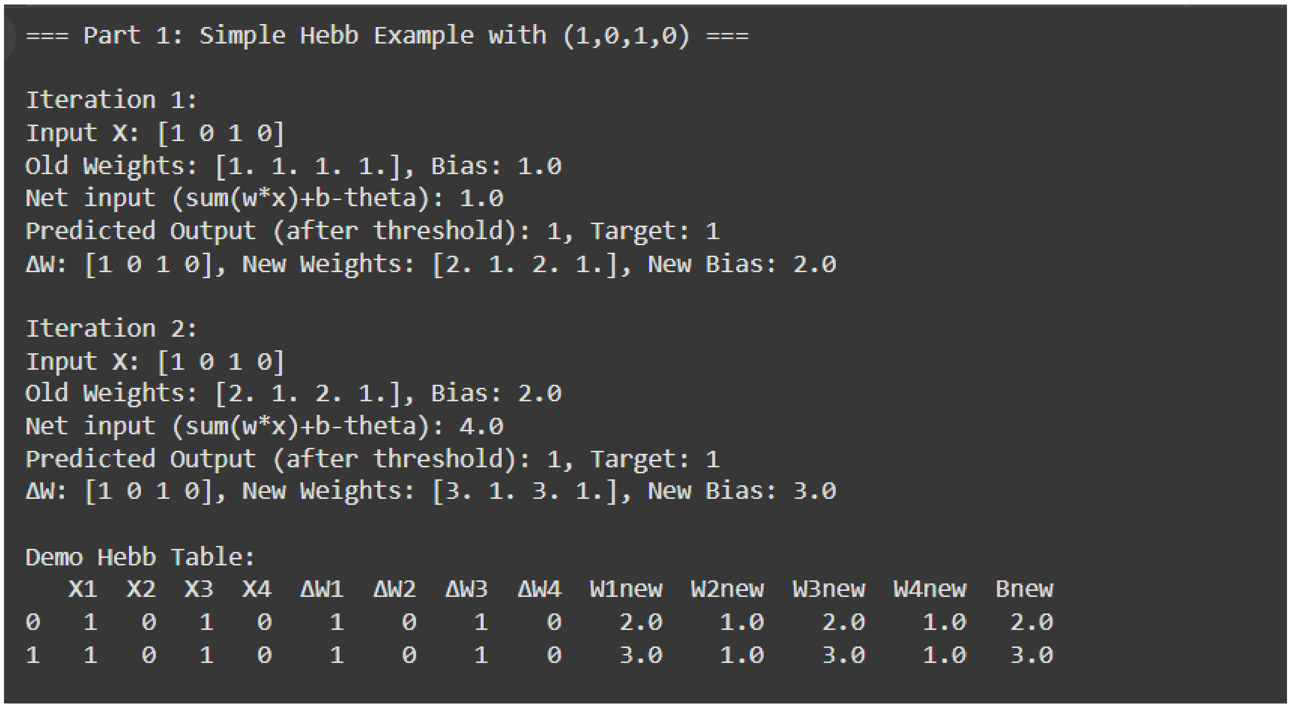
AIM:	To Design Single Layer Perceptron using Python
Program 1	
PROBLEM STATEMENT :	To build a single layer perceptron model and implement a single-layer perceptron using Hebbian learning rule to classify linearly separable data.
THEORY:	A single-layer perceptron trained using Hebbian learning is one of the earliest models of artificial neural networks. Based on the principle that "neurons that fire together, wire together," Hebbian learning strengthens the weight connections between inputs and the output when they activate simultaneously. In this framework, the perceptron takes inputs, multiplies them by their respective weights, adds a bias, and passes the result through a threshold function to produce an output. The learning rule updates the weights in proportion to the product of the input and the target, gradually reinforcing correct associations and improving classification. This approach works effectively for linearly separable data, where a single hyperplane can divide two classes. Through repeated training, the perceptron adjusts its decision boundary closer to the optimal separation. While it cannot solve complex, non-linear problems, the Hebbian perceptron offers key insights into adaptive learning. It also forms a foundational model in artificial intelligence, serving as a stepping stone toward more sophisticated multilayer networks and modern deep learning systems.

PROGRAM:

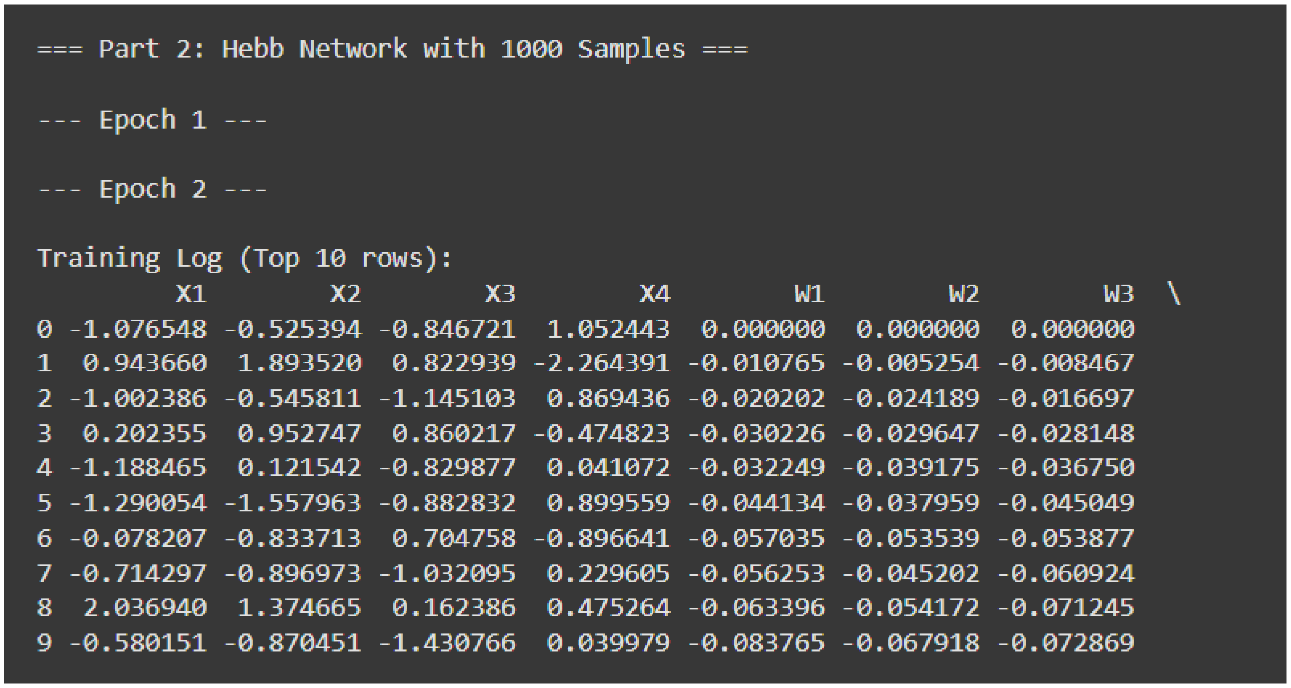
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

print("=== Part 1: Simple Hebb Example with (1,0,1,0) ===")

X = np.array([[1,0,1,0],
              [1,0,1,0]])
y = np.array([1,1])

weights = np.array([1,1,1,1], dtype=float)
bias = 1.0
theta = 2

logs_demo = []
for i in range(len(X)):
    x = X[i]
    target = y[i]

    print(f"\nIteration {i+1}:")
    print(f"Input X: {x}")
    print(f"Old Weights: {weights}, Bias: {bias}")

    net = np.dot(weights, x) + bias - theta
    print(f"Net input (sum(w*x)+b-theta): {net}")

    output = 1 if net >= 0 else 0
    print(f"Predicted Output (after threshold): {output}, Target: {target}")

    delta_w = target * x
    weights = weights + delta_w
    bias = bias + target
    print(f"ΔW: {delta_w}, New Weights: {weights}, New Bias: {bias}")

    logs_demo.append([x[0],x[1],x[2],x[3],
                delta_w[0],delta_w[1],delta_w[2],delta_w[3],
                weights[0],weights[1],weights[2],weights[3],bias])

df_demo = pd.DataFrame(logs_demo, columns=[

"X1","X2","X3","X4","ΔW1","ΔW2","ΔW3","ΔW4","W1new","W2new","
W3new","W4new","Bnew"])
print("\nDemo Hebb Table:")
print(df_demo)

print("\n=== Part 2: Hebb Network with 1000 Samples ===")

class HebbNetwork:
    def __init__(self, num_inputs, learning_rate=1.0):
        self.weights = np.zeros(num_inputs)
        self.bias = 0.0
        self.learning_rate = learning_rate

    def predict(self, x):
        activation = np.dot(x, self.weights) + self.bias
        return 1 if activation >= 0 else -1

    def train(self, X, y, epochs=5):
        logs = []
        for epoch in range(epochs):
            print(f"\n--- Epoch {epoch+1} ---")
            for i in range(len(X)):
                x = X[i]
                target = y[i]

                old_weights = self.weights.copy()
                old_bias = self.bias

                # print(f"\nSample {i+1}:")
                # print(f"Input X: {x}, Target: {target}")
                # print(f"Old Weights: {old_weights}, Old Bias: {old_bias}")

                delta_w = self.learning_rate * target * x
            self.weights += delta_w
            self.bias += self.learning_rate * target

            # print(f"ΔW: {delta_w}, New Weights: {self.weights}, New
Bias: {self.bias}")



            logs.append([
               x[0], x[1], x[2], x[3],
               old_weights[0], old_weights[1], old_weights[2],
old_weights[3], old_bias,
               delta_w[0], delta_w[1], delta_w[2], delta_w[3],
               self.weights[0], self.weights[1], self.weights[2],
self.weights[3], self.bias
            ])
      return logs

X, y = make_blobs(n_samples=1000, n_features=4, centers=2,
cluster_std=2, random_state=23)
scaler = StandardScaler()
X = scaler.fit_transform(X)
y = np.where(y==0, -1, 1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=23, shuffle=True)

hebb = HebbNetwork(num_inputs=4, learning_rate=0.01)
logs = hebb.train(X_train, y_train, epochs=2)

columns = ["X1","X2","X3","X4",
        "W1","W2","W3","W4","B",
        "ΔW1","ΔW2","ΔW3","ΔW4",
        "W1new","W2new","W3new","W4new","Bnew"]
df = pd.DataFrame(logs, columns=columns)

print("\nTraining Log (Top 10 rows):")
print(df.head(10))
```

```
def plot_decision_boundary(model, X, y):
    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 = []
    for i in range(xx.shape[0]):
        row = []
        for j in range(xx.shape[1]):
            point = np.array([xx[i, j], yy[i, j], 0, 0])
            row.append(model.predict(point))
        Z.append(row)
    Z = np.array(Z)

    plt.contourf(xx, yy, Z, alpha=0.4, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.coolwarm,
edgecolors='k')
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.title("Hebbian Network Decision Boundary")
    plt.show()

plot_decision_boundary(hebb, X_test, y_test)
```

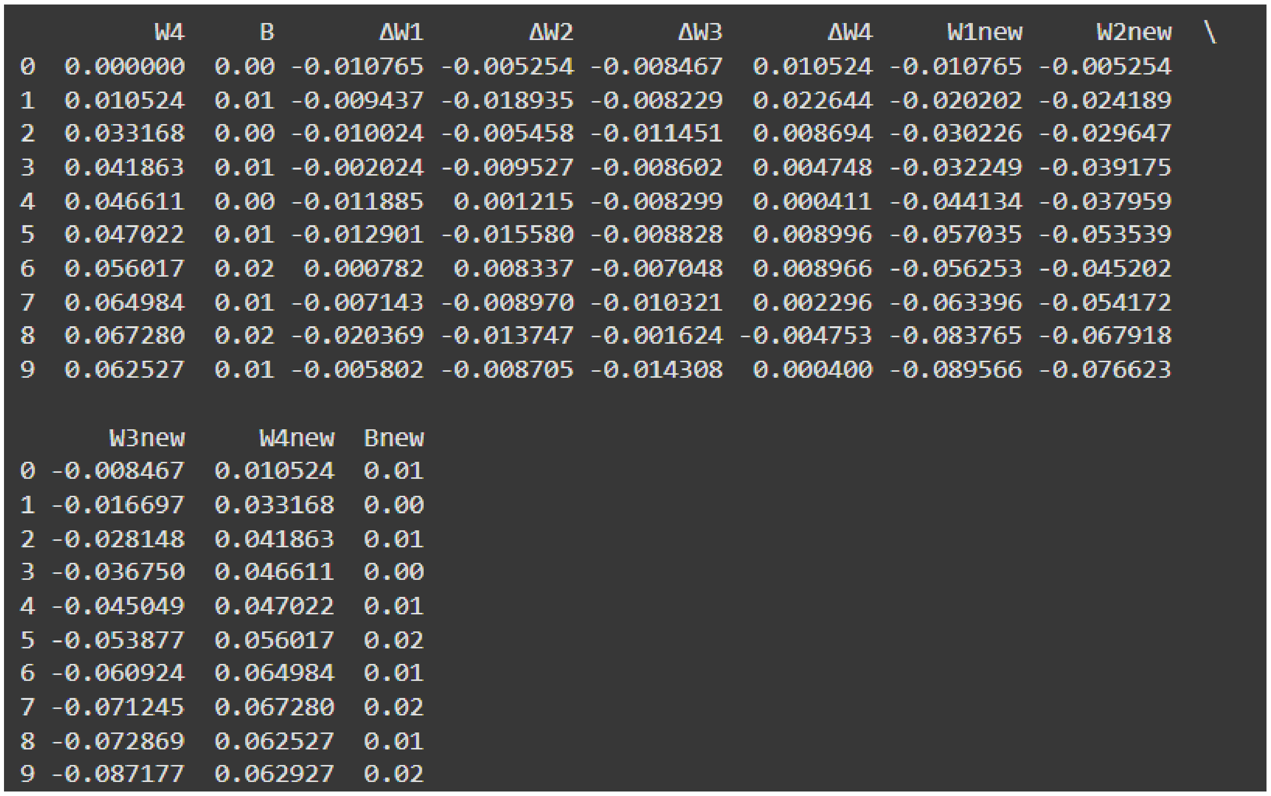
RESULT:



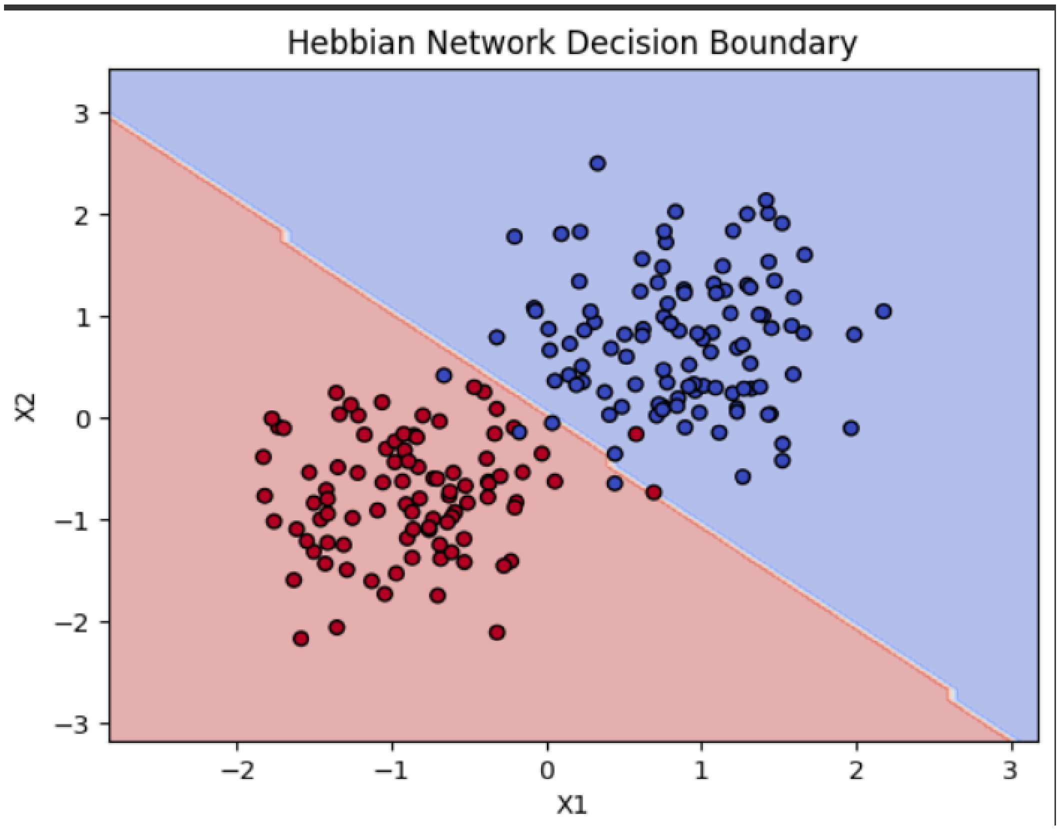
```
=== Part 1: Simple Hebb Example with (1,0,1,0) ===

Iteration 1:
Input X: [1 0 1 0]
Old Weights: [1. 1. 1. 1.], Bias: 1.0
Net input (sum(w*x)+b-theta): 1.0
Predicted Output (after threshold): 1, Target: 1
ΔW: [1 0 1 0], New Weights: [2. 1. 2. 1.], New Bias: 2.0

Iteration 2:
Input X: [1 0 1 0]
Old Weights: [2. 1. 2. 1.], Bias: 2.0
Net input (sum(w*x)+b-theta): 4.0
Predicted Output (after threshold): 1, Target: 1
ΔW: [1 0 1 0], New Weights: [3. 1. 3. 1.], New Bias: 3.0

Demo Hebb Table:
   X1  X2  X3  X4  ΔW1  ΔW2  ΔW3  ΔW4  W1new  W2new  W3new  W4new  Bnew
0   1   0   1   0    1    0    1    0    2.0    1.0    2.0    1.0   2.0
1   1   0   1   0    1    0    1    0    3.0    1.0    3.0    1.0   3.0
```



```
=== Part 2: Hebb Network with 1000 Samples ===

--- Epoch 1 ---

--- Epoch 2 ---

Training Log (Top 10 rows):
         X1        X2        X3        X4        W1        W2        W3  \
0 -1.076548 -0.525394 -0.846721  1.052443  0.000000  0.000000  0.000000
1  0.943660  1.893520  0.822939 -2.264391 -0.010765 -0.005254 -0.008467
2 -1.002386 -0.545811 -1.145103  0.869436 -0.020202 -0.024189 -0.016697
3  0.202355  0.952747  0.860217 -0.474823 -0.030226 -0.029647 -0.028148
4 -1.188465  0.121542 -0.829877  0.041072 -0.032249 -0.039175 -0.036750
5 -1.290054 -1.557963 -0.882832  0.899559 -0.044134 -0.037959 -0.045049
6 -0.078207 -0.833713  0.704758 -0.896641 -0.057035 -0.053539 -0.053877
7 -0.714297 -0.896973 -1.032095  0.229605 -0.056253 -0.045202 -0.060924
8  2.036940  1.374665  0.162386  0.475264 -0.063396 -0.054172 -0.071245
9 -0.580151 -0.870451 -1.430766  0.039979 -0.083765 -0.067918 -0.072869
```



```
         W4     B       ΔW1       ΔW2       ΔW3       ΔW4     W1new     W2new  \
0  0.000000  0.00 -0.010765 -0.005254 -0.008467  0.010524 -0.010765 -0.005254   
1  0.010524  0.01 -0.009437 -0.018935 -0.008229  0.022644 -0.020202 -0.024189   
2  0.033168  0.00 -0.010024 -0.005458 -0.011451  0.008694 -0.030226 -0.029647   
3  0.041863  0.01 -0.002024 -0.009527 -0.008602  0.004748 -0.032249 -0.039175   
4  0.046611  0.00 -0.011885  0.001215 -0.008299  0.000411 -0.044134 -0.037959   
5  0.047022  0.01 -0.012901 -0.015580 -0.008828  0.008996 -0.057035 -0.053539   
6  0.056017  0.02  0.000782  0.008337 -0.007048  0.008966 -0.056253 -0.045202   
7  0.064984  0.01 -0.007143 -0.008970 -0.010321  0.002296 -0.063396 -0.054172   
8  0.067280  0.02 -0.020369 -0.013747 -0.001624 -0.004753 -0.083765 -0.067918   
9  0.062527  0.01 -0.005802 -0.008705 -0.014308  0.000400 -0.089566 -0.076623   

      W3new     W4new  Bnew  
0 -0.008467  0.010524  0.01  
1 -0.016697  0.033168  0.00  
2 -0.028148  0.041863  0.01  
3 -0.036750  0.046611  0.00  
4 -0.045049  0.047022  0.01  
5 -0.053877  0.056017  0.02  
6 -0.060924  0.064984  0.01  
7 -0.071245  0.067280  0.02  
8 -0.072869  0.062527  0.01  
9 -0.087177  0.062927  0.02  
```

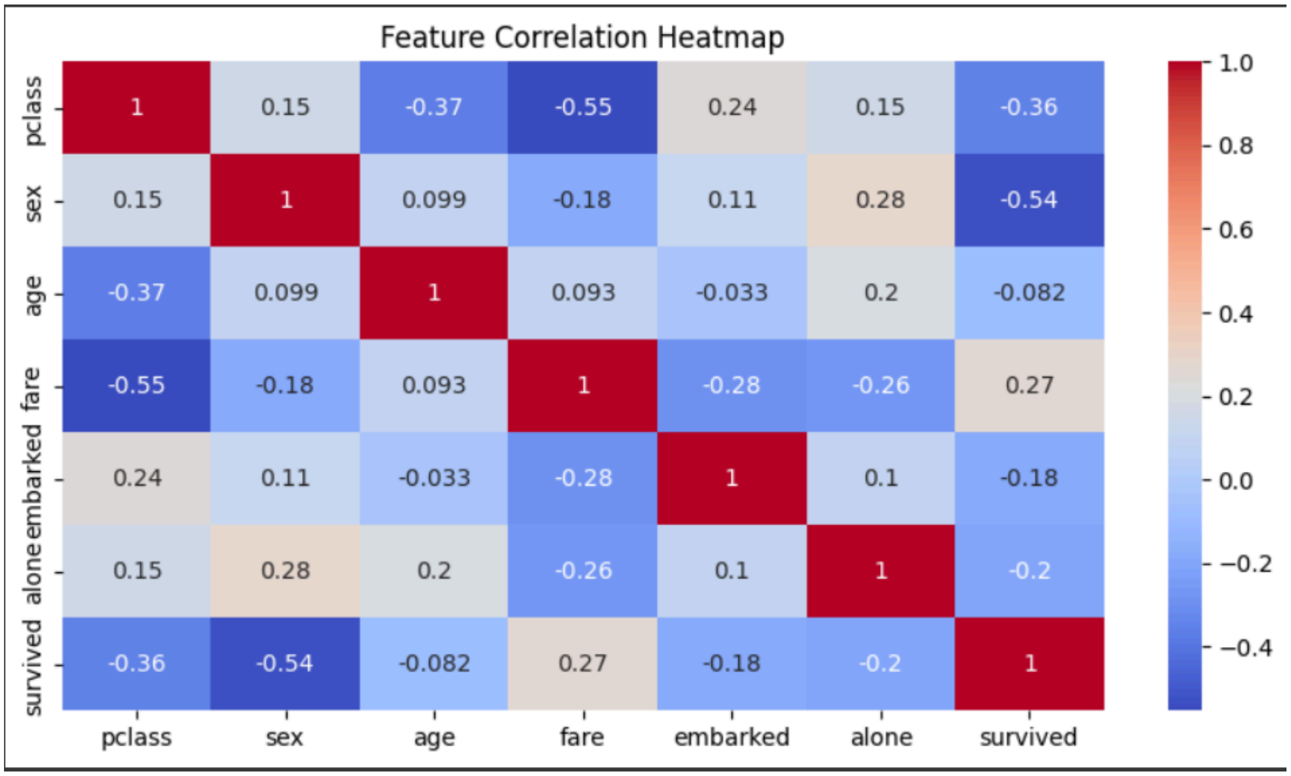


Program 2	
PROBLEM STATEMENT :	To implement a single-layer Hebbian perceptron on the Titanic dataset for predicting passenger survival by performing preprocessing, standardization, training, and evaluating the model using accuracy, confusion matrix, precision, recall, F1 score, and ROC curve.
DATASET LINK:	https://www.kaggle.com/competitions/titanic
THEORY:	In this project, the Titanic dataset is used to predict whether a passenger survived or not based on features such as age, sex, passenger class, fare, and embarkation point. The data is first preprocessed by handling missing values, encoding categorical attributes, and standardizing numerical features to bring them to a common scale. The dataset is then split into training and testing sets, ensuring the model learns patterns from one portion and is evaluated on unseen data. Using visualization techniques like heatmaps, the relationship between variables is explored before training, helping identify correlations and the importance of certain features such as gender and class in survival prediction. The learning model applied here is a single-layer perceptron with Hebbian learning. The perceptron is the simplest form of an artificial neural network, consisting of input nodes directly connected to an output node. Hebbian learning, inspired by neuroscience, strengthens the connection weights whenever input and output neurons activate together, following the principle "neurons that fire together, wire together." In the Titanic dataset, this means the perceptron adjusts its weights based on repeated exposure to patterns linking input features to survival outcomes. Although limited to linearly separable problems, this approach demonstrates how fundamental neural learning rules can classify real-world data effectively.

PROGRAM:	`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, LabelEncoder`
	`from sklearn.metrics import confusion_matrix, precision_score,`
	`recall_score, f1_score, roc_curve, auc`

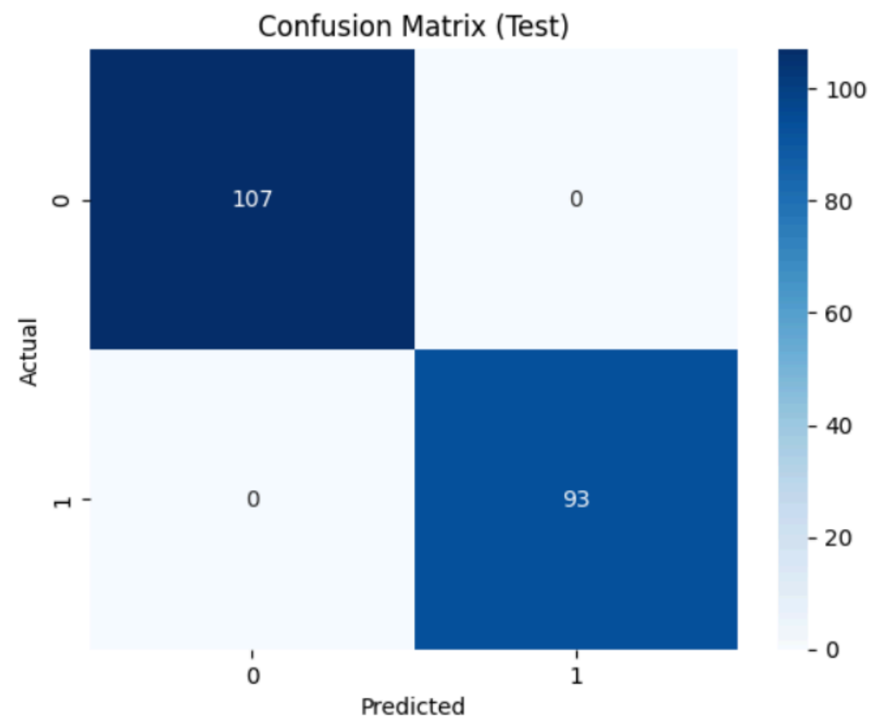
```
# Load dataset
titanic = sns.load_dataset('titanic')

# Select useful features
df = titanic[["pclass", "sex", "age", "fare", "embarked", "alone",
"survived"]]

# Handle missing values
df = df.dropna()

# Encode categorical variables
le_sex = LabelEncoder()
le_emb = LabelEncoder()
df["sex"] = le_sex.fit_transform(df["sex"])
df["embarked"] = le_emb.fit_transform(df["embarked"])

# Features and Target
X = df.drop("survived", axis=1).values
y = df["survived"].values
y = np.where(y == 0, -1, 1)  # Convert to -1 and 1 for Hebbian rule

# Visualization
plt.figure(figsize=(10,5))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm")
plt.title("Feature Correlation Heatmap")
plt.show()
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)



class HebbianPerceptron:
    def __init__(self, num_inputs, learning_rate=0.01):
        self.weights = np.zeros(num_inputs)
        self.bias = 0.0
        self.lr = learning_rate

    def predict(self, x):
        return 1 if np.dot(self.weights, x) + self.bias >= 0 else -1

    def train(self, X, y, epochs=5):
        for epoch in range(epochs):
            for i in range(len(X)):
                xi, target = X[i], y[i]
                self.weights += self.lr * target * xi
                self.bias += self.lr * target

model = HebbianPerceptron(num_inputs=X_train.shape[1],
learning_rate=0.01)

model.train(X_train, y_train, epochs=20)


# Predictions
y_pred_train = [model.predict(x) for x in X_train]
y_pred_test = [model.predict(x) for x in X_test]

# Convert -1 back to 0 for metrics
y_train_true = np.where(y_train==-1,0,1)
y_test_true = np.where(y_test==-1,0,1)
y_pred_train = np.where(np.array(y_pred_train)==-1,0,1)
y_pred_test = np.where(np.array(y_pred_test)==-1,0,1)

# Confusion Matrix
cm = confusion_matrix(y_test_true, y_pred_test)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix (Test)")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

# Metrics
precision = precision_score(y_test_true, y_pred_test)
```

	```
	recall = recall_score(y_test_true, y_pred_test)
	f1 = f1_score(y_test_true, y_pred_test)
	train_acc = np.mean(y_train_true == y_pred_train)
	test_acc = np.mean(y_test_true == y_pred_test)

	print(f"Precision: {precision:.3f}")
	print(f"Recall: {recall:.3f}")
	print(f"F1 Score: {f1:.3f}")
	print(f"Training Accuracy: {train_acc:.3f}")
	print(f"Test Accuracy: {test_acc:.3f}")

	# ROC Curve
	fpr, tpr, _ = roc_curve(y_test_true, y_pred_test)
	roc_auc = auc(fpr, tpr)
	plt.figure()
	plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC =
	{roc_auc:.2f})')
	plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
	plt.xlabel("False Positive Rate")
	plt.ylabel("True Positive Rate")
	plt.title("Receiver Operating Characteristic (ROC)")
	plt.legend(loc="lower right")
	plt.show()
	```
Feature Correlation Heatmap	

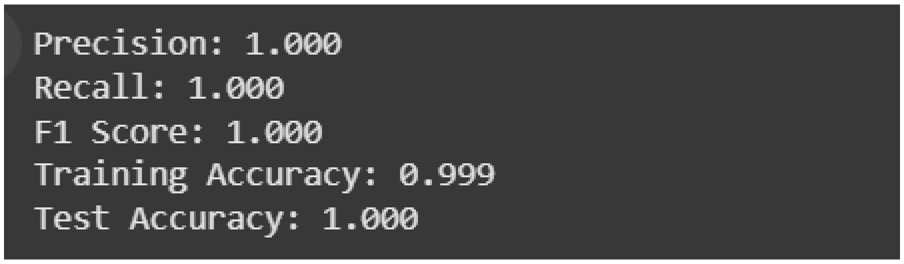


	pclass	sex	age	fare	embarked	alone	survived
pclass	1	0.15	-0.37	-0.55	0.24	0.15	-0.36
sex	0.15	1	0.099	-0.18	0.11	0.28	-0.54
age	-0.37	0.099	1	0.093	-0.033	0.2	-0.082
fare	-0.55	-0.18	0.093	1	-0.28	-0.26	0.27
embarked	0.24	0.11	-0.033	-0.28	1	0.1	-0.18
alone	0.15	0.28	0.2	-0.26	0.1	1	-0.2
survived	-0.36	-0.54	-0.082	0.27	-0.18	-0.2	1

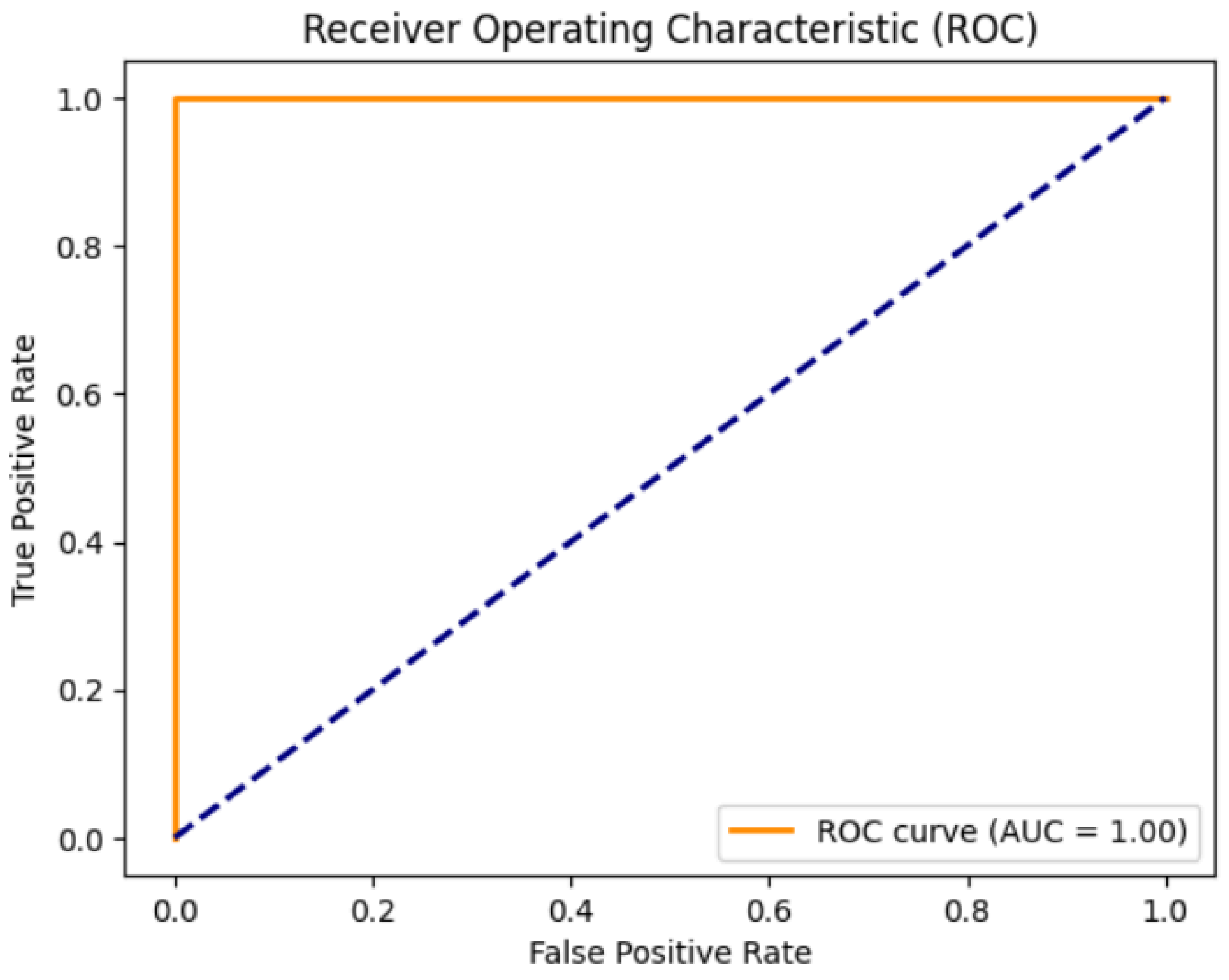
Confusion matrix:



RESULT and ROC:



```
Precision: 1.000
Recall: 1.000
F1 Score: 1.000
Training Accuracy: 0.999
Test Accuracy: 1.000
```



CONCLUSION:	The Hebbian single-layer perceptron experiment demonstrated the basic principles of neural learning using a simple input-output mapping. By updating the weights according to the Hebbian rule, the network reinforced correct associations between inputs and targets, gradually adjusting its parameters toward the desired outcome. This approach clearly illustrates how neural connections strengthen when both input and output are

	simultaneously active. Although it is limited to solving linearly separable problems, the experiment provided valuable insights into the biological inspiration of learning mechanisms and the foundational concepts of artificial neural networks, setting the stage for more complex learning models. The application of the Hebbian single-layer perceptron on the Titanic dataset highlighted how this learning model can be extended to real-world classification tasks. After preprocessing, standardizing, and training the dataset, the perceptron adjusted its weights to predict passenger survival based on features like age, sex, fare, and class. The model's performance was evaluated using accuracy, precision, recall, F1 score, and ROC analysis, demonstrating its ability to capture significant survival patterns despite being a simple model.
NOTEBOOK LINK:	https://colab.research.google.com/drive/1ngMAMj6bjJzUlY3i3QuE4jC5PzsiR3eJ?usp=sharing