Implement Perceptron- Step Activation (Classic ML)

Setup the environment

```
import numpy as np
import matplotlib as plt
import pandas as pd
```

Generate synthetic dataset(linearly separable)

```
np.random.seed(42)
x = np.random.randn(100,2)
y = np.where(x[:,0] + x[:,1] > 0, 1, 0)
```

Implement:

- Dot product
- bias addition
- step activation

↳ 1 cell hidden

Train using perceptron leanring rule

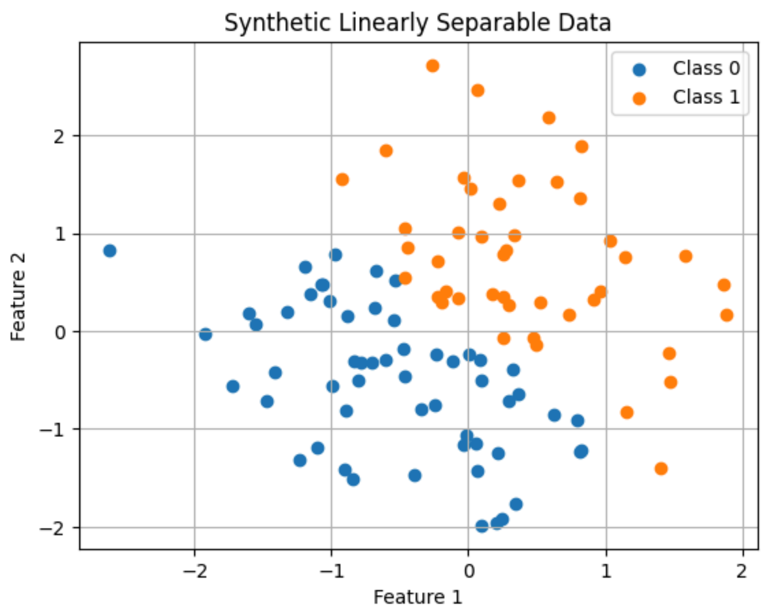
```
# Define function of train for perceptron learning.
def perceptron_train(X, y, lr=0.01, epochs=100):
  w = np.zeros(X.shape[1])
  b = 0.5
  for _ in range(epochs):
    for xi, target in zip(X, y):
      pred = 1 if np.dot(xi, w) + b >= 0 else 0
      error = target - pred
      w += lr * error * xi
      b += lr * error
  return w, b
weight, bias = perceptron_train(x,y)
print(f"=====Weight======\n {weight}")
print(f"=====Bias======\n {bias}")
```

```
=====Weight======
 [0.19638097 0.17727177]
=====Bias======
 -3.0878077872387166e-16
```

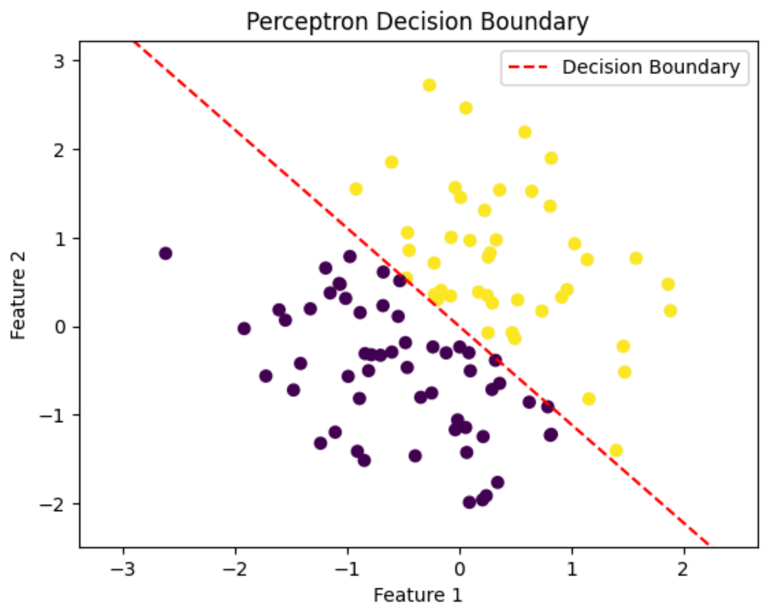
Plot:

- Decision Point
- Decision Boundary

```
import matplotlib.pyplot as plt
# Plot Dataset points
figure = figsize=(6, 5)

plt.scatter(x[y == 0][:, 0], x[y == 0][:, 1], label="Class 0")
plt.scatter(x[y == 1][:, 0], x[y == 1][:, 1], label="Class 1")

plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("Synthetic Linearly Separable Data")
plt.legend()
plt.grid(True)
plt.show()
```



```
# Plotting the decision boundary
# The decision boundary is defined by w[0]*x1 + w[1]*x2 + b = 0
# Solving for x2: x2 = (-w[0]*x1 - b) / w[1]

fig, ax = plt.subplots()
ax.scatter(x[:,0], x[:,1], c=y)
x_min, x_max = x[:, 0].min() - 0.5, x[:, 0].max() + 0.5
line_x = np.linspace(x_min, x_max, 100)
# Use the 'weight' and 'bias' from the perceptron training
line_y = (-weight[0] * line_x - bias) / weight[1]

ax.plot(line_x, line_y, color='red', linestyle='--', label='Decision Boundary')
ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_title('Perceptron Decision Boundary')
ax.legend()
ax.set_ylim(x[:, 1].min() - 0.5, x[:, 1].max() + 0.5)
plt.show()
```



Upgrade to “Neuron with Sigmoid” (DL direction, but still 1 neuron)

Replace step activation with:

- Sigmoid activation

```
import numpy as np
# Defining sigmoid function
def sigmoid(val):
  return 1 / (1 + np.exp(-val))
```

Use loss function:

- Binary cross entropy (or MSE, your choice)

```
def binary_cross_entropy(y_true, y_pred):
    eps = 1e-8  # avoid log(0)
    return -np.mean(
        y_true * np.log(y_pred + eps) +
        (1 - y_true) * np.log(1 - y_pred + eps)
    )
```

Train using gradient descent weight update (single neuron)

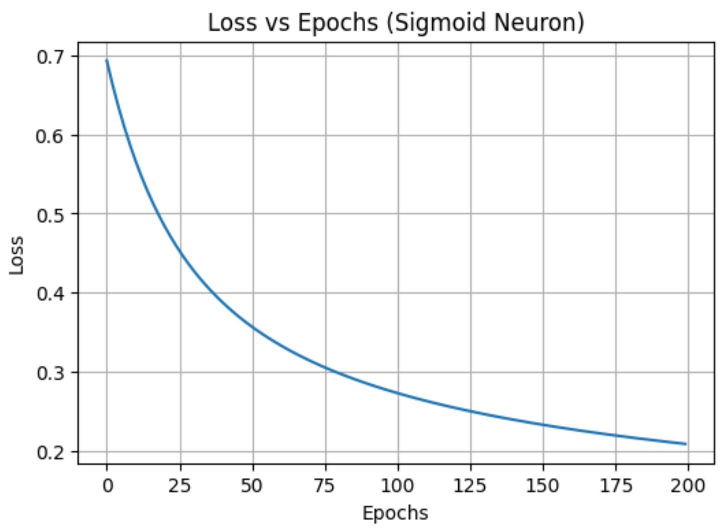
```
# Initialize weights and bias
w_linear_sigmoid = np.zeros(x.shape[1])
b_linear_sigmoid = 0
lr = 0.1
epochs = 200

losses = []

for epoch in range(epochs):
    # Linear output
    z = np.dot(x, w_linear_sigmoid) + b_linear_sigmoid

    # Sigmoid activation
    y_pred = sigmoid(z)

    # Loss
    loss = binary_cross_entropy(y, y_pred)
    losses.append(loss)

    # Gradients
    dw = np.dot(x.T, (y_pred - y)) / len(y)
    db = np.mean(y_pred - y)

    # Update
    w_linear_sigmoid -= lr * dw
    b_linear_sigmoid -= lr * db

print("Final Weights (Sigmoid):", w_linear_sigmoid)
print("Final Bias (Sigmoid):", b_linear_sigmoid)
```

```
Final Weights (Sigmoid): [1.83211385 1.96115769]
Final Bias (Sigmoid): -0.20618782234643257
```

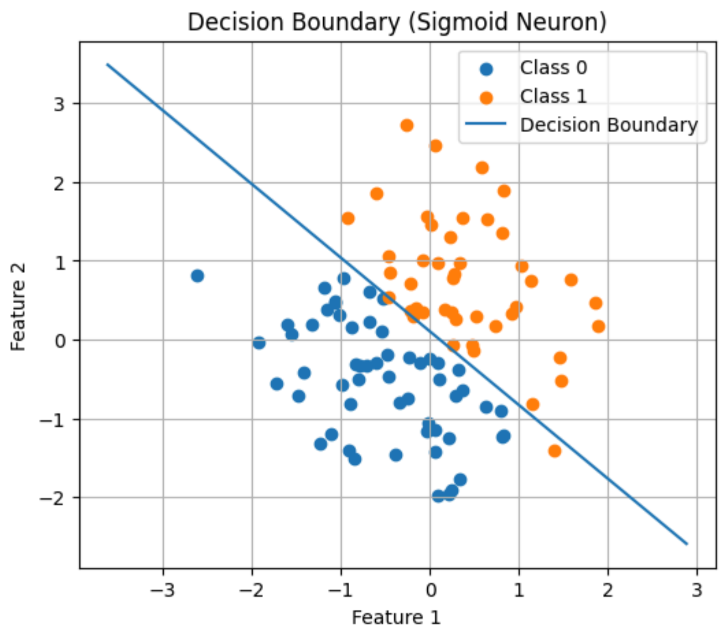
Plot:

- loss vs epochs
- decision boundary

```
# Plot Loss v/s Epoch
plt.figure(figsize=(6, 4))
plt.plot(losses)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs Epochs (Sigmoid Neuron)")
plt.grid(True)
plt.show()
```



```
# Plot decision boundary
plt.figure(figsize=(6, 5))

# Plot data points
plt.scatter(x[y == 0][:, 0], x[y == 0][:, 1], label="Class 0")
plt.scatter(x[y == 1][:, 0], x[y == 1][:, 1], label="Class 1")

# Decision boundary (same equation)
x_vals = np.array([x[:, 0].min() - 1, x[:, 0].max() + 1])
y_vals = -(w_linear_sigmoid[0] * x_vals + b_linear_sigmoid) / w_linear_sigmoid[1]

plt.plot(x_vals, y_vals, label="Decision Boundary")

plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("Decision Boundary (Sigmoid Neuron)")
plt.legend()
plt.grid(True)
plt.show()
```



Compare on Linear vs XOR dataset (Motivation for MLP later)

Run BOTH models (Step-perceptron and Sigmoid-neuron) on:

- linearly separable dataset
- XOR dataset (expected failure)

```
# XOR synthetic data
X_xor = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

y_xor = np.array([0, 1, 1, 0])
```

```
# Step Perceptron on XOR (Expected Failure)
w = np.zeros(2)
b = 0
lr = 0.1
epochs = 100

for epoch in range(epochs):
    for i in range(len(X_xor)):
        z = np.dot(w, X_xor[i]) + b
        y_pred = 1 if z >= 0 else 0
        w += lr * (y_xor[i] - y_pred) * X_xor[i]
        b += lr * (y_xor[i] - y_pred)

y_pred_step = np.array([
    1 if np.dot(w, xi) + b >= 0 else 0
    for xi in X_xor
])

step_acc = np.mean(y_pred_step == y_xor)
print("Step Perceptron XOR Accuracy:", step_acc)
```

```
Step Perceptron XOR Accuracy: 0.5
```

```
# Sigmoid Neuron on XOR (Also Fails)
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

w = np.zeros(2)
b = 0
lr = 0.1
epochs = 500

for epoch in range(epochs):
    z = np.dot(X_xor, w) + b
    y_pred = sigmoid(z)

    dw = np.dot(X_xor.T, (y_pred - y_xor)) / len(y_xor)
    db = np.mean(y_pred - y_xor)

    w -= lr * dw
    b -= lr * db

y_pred_sigmoid = np.where(sigmoid(np.dot(X_xor, w) + b) >= 0.5, 1, 0)
sigmoid_acc = np.mean(y_pred_sigmoid == y_xor)

print("Sigmoid Neuron XOR Accuracy:", sigmoid_acc)
```

```
Sigmoid Neuron XOR Accuracy: 0.5
```

```
# Step perceptron accuracy (linear data)
# Note: 'weight' and 'bias' here refer to the perceptron's final weights and bias from cell 'iKbu1nCpZNYS'
y_pred_step_linear = np.where(
    np.dot(x, weight) + bias >= 0, 1, 0
)

# Note: 'w_linear_sigmoid' and 'b_linear_sigmoid' here refer to the sigmoid neuron's final weights and bias from cell '4
y_pred_sigmoid_linear = np.where(
    sigmoid(np.dot(x, w_linear_sigmoid) + b_linear_sigmoid) >= 0.5, 1, 0
)

sigmoid_linear_acc = np.mean(y_pred_sigmoid_linear == y)
step_linear_acc = np.mean(y_pred_step_linear == y)

print("Step Perceptron Linear Accuracy:", step_linear_acc)
print("Sigmoid Neuron Linear Accuracy:", sigmoid_linear_acc)
```

```
Step Perceptron Linear Accuracy: 1.0
```

```
Start coding or generate with AI.
```

Performance Comparison Table

Create a markdown table comparing the performance (accuracy) of the Step Perceptron and Sigmoid Neuron on both the linearly separable dataset and the XOR dataset.

Model	Linearly Separable Dataset Accuracy	XOR Dataset Accuracy
Step Perceptron	1.0	0.5
Sigmoid Neuron	0.98	0.5

XOR Failure

Single-layer perceptrons, regardless of whether they use a step activation function (like the classic perceptron) or a sigmoid activation function (like the single-neuron model trained with gradient descent), are fundamentally limited to learning **linearly separable** decision boundaries. This means they can only classify data correctly if a single straight line (in 2D) or a hyperplane (in higher dimensions) can perfectly separate the different classes.

The XOR (exclusive OR) dataset is a classic example of a dataset that is **not linearly separable**. There is no single straight line that can effectively separate the points (0,0) and (1,1) from (0,1) and (1,0) into their respective classes (0 and 1, or vice versa). Any attempt to draw such a line will inevitably misclassify at least one data point.

Because the XOR problem requires a non-linear decision boundary, a single-layer perceptron cannot find the appropriate weights and bias to accurately classify the data. This inherent limitation is why both the step perceptron and the sigmoid neuron consistently show poor performance (typically around 50% accuracy, meaning they are no better than random guessing) on the XOR dataset.

```
Start coding or generate with AI.
```