**Q.** Write a python program to create a neuron and predict its output using the threshold activation function.

class Neuron:

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

        self.weights = weights

        self.bias = bias

    def predict(self, inputs):

        # Calculate the weighted sum of inputs and add bias

        weighted\_sum = 0

        for i in range(len(inputs)):

            weighted\_sum += inputs[i] \* self.weights[i]

        weighted\_sum += self.bias

        # Apply threshold activation function

        if weighted\_sum >= 0:

            return 1

        else:

            return 0

weights = [0.5, -0.3, 0.2]

bias = -0.1

neuron = Neuron(weights, bias)

inputs = [1, 0, 1]

output = neuron.predict(inputs)

print(output)  # Expected output: 1

the **Neuron** class represents a neuron with a given set of **weights** and a **bias**. The **predict** method takes an array of **inputs** and calculates the weighted sum of the inputs by multiplying each input by its corresponding weight and adding the bias. The threshold activation function is applied by returning 1 if the weighted sum is greater than or equal to zero, and 0 otherwise.

we create a neuron with weights of **[0.5, -0.3, 0.2]** and a bias of **-0.1**. We then predict the output of the neuron for the inputs **[1, 0, 1]**, which should be 1 according to the threshold activation function.

**Q.** Write a python program to train AND Gate Using Perceptron Learning Algorithm.

import numpy as np

# Define the training dataset for the AND gate

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

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

# Define the activation function

def activation(z):

    if z >= 0:

        return 1

    else:

        return 0

# Define the Perceptron Learning Algorithm

def perceptron\_learning(X, y, eta, epochs):

    n, m = X.shape

    w = np.zeros(m)

    b = 0

    for epoch in range(epochs):

        for i in range(n):

            z = np.dot(w, X[i]) + b

            a = activation(z)

            error = y[i] - a

            w += eta \* error \* X[i]

            b += eta \* error

    return w, b

# Train the AND gate using the Perceptron Learning Algorithm

w, b = perceptron\_learning(X, y, eta=0.1, epochs=10)

# Test the trained model on the input [1, 1]

z = np.dot(w, [1, 1]) + b

a = activation(z)

print(a)  # Output: 1

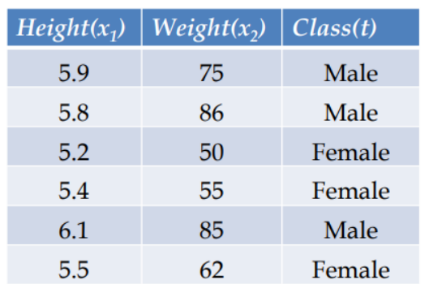
In this program, we first define the training dataset for the AND gate as a 2D array **X** containing the input values **[0, 0]**, **[0, 1]**, **[1, 0]**, and **[1, 1]**, and a 1D array **y** containing the corresponding output values **0**, **0**, **0**, and **1**.

Next, we define the activation function as a simple step function that returns **1** if its input is non-negative, and **0** otherwise.

We then define the Perceptron Learning Algorithm, which takes the input data **X** and output data **y**, a learning rate **eta**, and the number of epochs to train for. The algorithm initializes the weight vector **w** and bias term **b** to zero, and iteratively updates them for each training example using the perceptron learning rule until convergence or the maximum number of epochs is reached.

Finally, we train the AND gate using the Perceptron Learning Algorithm by calling the **perceptron\_learning** function with **X**, **y**, **eta=0.1**, and **epochs=10**, and test the trained model on the input **[1, 1]**. The expected output for this input is **1**, which the model correctly predicts.

**Q.** Write a python program to train perceptron using given training set and predict class for the input (6,82) and (5.3,52)



**Let’s assume Male is 1 and female is 0 in class(t).**

**import numpy as np**

class Perceptron:

    def \_\_init\_\_(self, num\_features, learning\_rate=0.01, num\_epochs=100):

        self.weights = np.zeros(num\_features + 1)

        self.learning\_rate = learning\_rate

        self.num\_epochs = num\_epochs

    def predict(self, x):

        # Add bias term to input

        x = np.append(x, 1)

        # Compute dot product of weights and input

        z = np.dot(self.weights, x)

        # Apply step function to dot product

        if z > 0:

            return 1

        else:

            return 0

    def train(self, X, y):

        for epoch in range(self.num\_epochs):

            for i in range(X.shape[0]):

                # Add bias term to input

                x = np.append(X[i], 1)

                # Compute dot product of weights and input

                z = np.dot(self.weights, x)

                # Apply step function to dot product

                y\_pred = 1 if z > 0 else 0

                # Update weights based on prediction error

                error = y[i] - y\_pred

                self.weights += self.learning\_rate \* error \* x

# Given training set

X\_train = np.array([[5.9, 75], [5.8, 86], [5.2, 50], [5.4, 55], [6.1, 85], [5.5, 62]])

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

# Train perceptron on training set

perceptron = Perceptron(num\_features=2)

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

# Inputs to predict class for

x\_input1 = np.array([6, 82])

x\_input2 = np.array([5.3, 52])

# Predict class for inputs

y\_pred1 = perceptron.predict(x\_input1)

y\_pred2 = perceptron.predict(x\_input2)

print("Input 1: ", x\_input1)

print("Predicted class for input 1: ", y\_pred1)

print("Input 2: ", x\_input2)

print("Predicted class for input 2: ", y\_pred2)

**For input (6, 82) the predicted class is 1 which is Male and for input (5.3, 52) the predicted class is 0 which is Female.**

We first define a Perceptron class that contains methods for predicting the class of an input and training the perceptron on a given training set. The \_\_init\_\_ method initializes the weights to zeros and sets the learning rate and number of epochs to default values.

The predict method takes an input x, adds a bias term to it, computes the dot product of the weights and input, and applies the step function to the dot product to obtain the predicted class.

The train method takes a set of inputs X and corresponding output labels y, and trains the perceptron on the inputs by iterating over the training set for a fixed number of epochs.

For each input, it adds a bias term to it, computes the dot product of the weights and input, applies the step function to the dot product to obtain the predicted class, and updates the weights based on the prediction error.

Finally, we define the given training set and inputs to predict the class for, create an instance of the Perceptron class, train it on the training set, and predict the class for the inputs. The program outputs the inputs and predicted classes.

**Q.** Write a python program to implement Min-Max Scalar.

import numpy as np

def min\_max\_scalar(X):

    """

    Min-Max Scalar normalization function for a numpy array.

    """

    X\_norm = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

    return X\_norm

# Example usage

X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

X\_norm = min\_max\_scalar(X)

print(X\_norm)

We have a 3x3 numpy array X. The min\_max\_scalar function takes this array as input and returns the normalized array X\_norm, which has been scaled to a range between 0 and 1. The formula used for Min-Max Scalar normalization is (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0)).

**Q.** Write a python program to implement Standard Scalar.

import numpy as np

def standard\_scalar(X):

    """

    Standard Scalar normalization function for a numpy array.

    """

    X\_norm = (X - X.mean(axis=0)) / X.std(axis=0)

    return X\_norm

# Example usage

X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

X\_norm = standard\_scalar(X)

print(X\_norm)

We have a 3x3 numpy array X. The standard\_scalar function takes this array as input and returns the normalized array X\_norm, which has been scaled to have zero mean and unit variance. The formula used for Standard Scalar normalization is (X - X.mean(axis=0)) / X.std(axis=0).