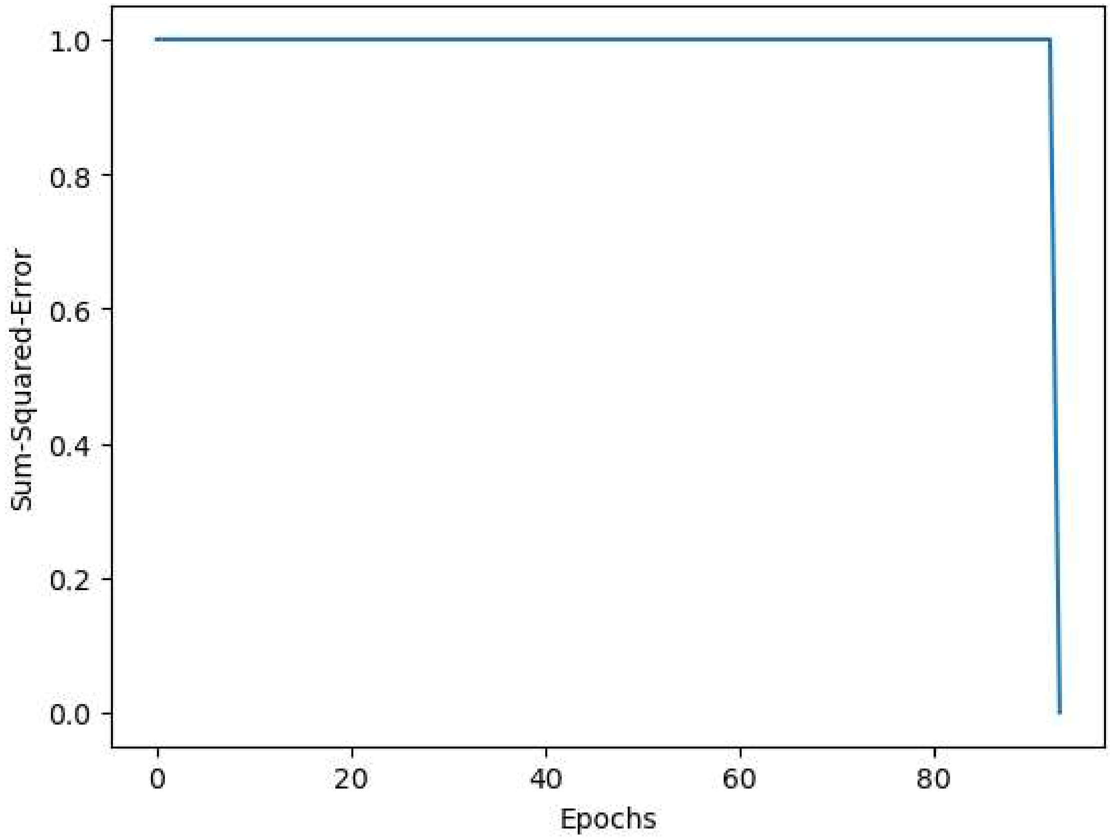
In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
```

In [6]:
```
#A1
def activation(x):
    if x >= 0:
        return 1
    else:
        return 0

def perceptron(train_data, W0, W1, b, alpha, thresh):
    errors = []
    epoch = 0
    w0_upd = 0
    w1_upd = 0
    b_upd = 0
    converge = False

    while not converge and epoch < 1000:
        total_error = 0
        for i in range(len(train_data)):
            x0 = train_data[i][0]
            x1 = train_data[i][1]
            target_output = train_data[i][2]
            pred_output = activation(W0*x0 + W1*x1 + b)
            error = target_output - pred_output
            total_error += error**2
            w0_upd = alpha * error * x0
            w1_upd = alpha * error * x1
            b_upd = alpha * error
            W0 += w0_upd
            W1 += w1_upd
            b += b_upd
        errors.append(total_error)
        epoch += 1
        if total_error <= thresh:
            converge = True

    return W0, W1, b, errors

train_data = [(0,0,0), (0,1,0), (1,0,0), (1,1,1)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = 0.05
thresh = 0.002
```

```
W0, W1, b, errors = perceptron(train_data, W0, W1, b, alpha, thresh)

print("Number of epochs needed:", len(errors))
plt.plot(errors)
plt.xlabel('Epochs')
plt.ylabel('Sum-Squared-Error')
plt.show()
```

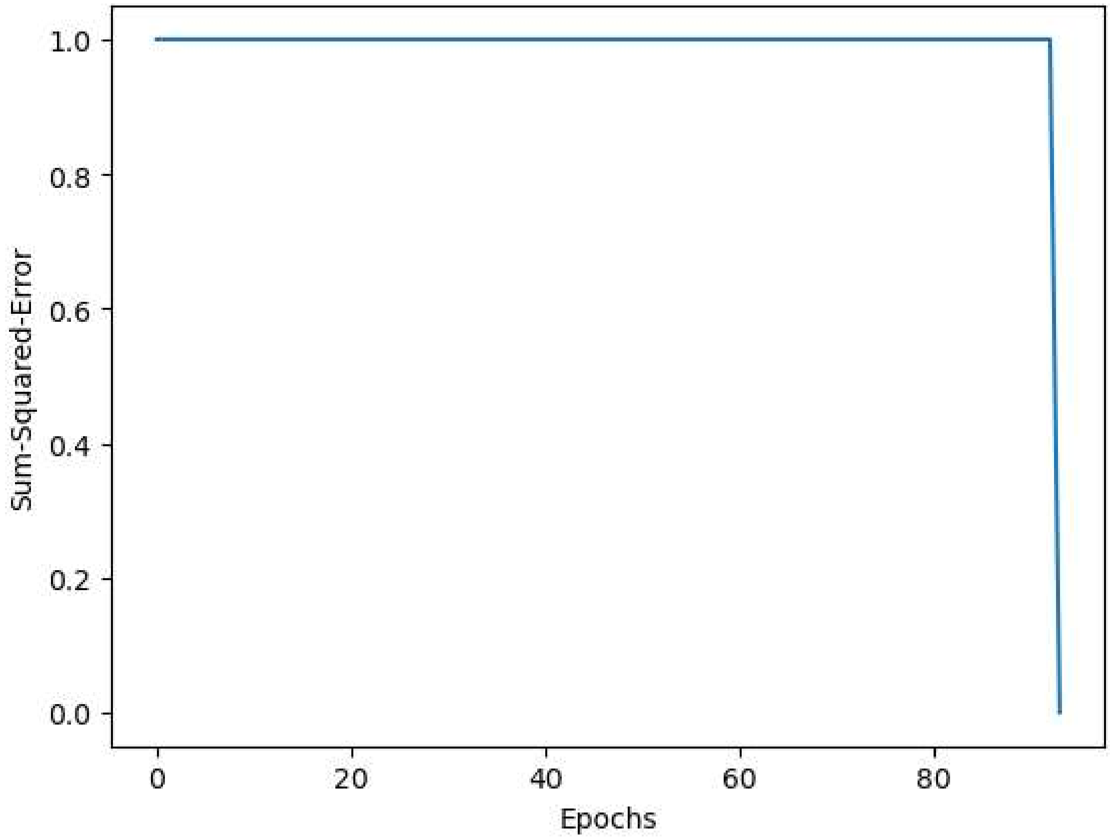
```
Number of epochs needed: 94
```



In [21]:
```
#A2
def bipolar(x):
    if x >= 0:
        return 1
    else:
        return 0

def perceptron(train_data, W0, W1, b, alpha, thresh):
    errors = []
    epoch = 0
    w0_upd = 0
    w1_upd = 0
    b_upd = 0
    converge = False

    while not converge and epoch < 1000:
        total_error = 0
        for i in range(len(train_data)):
            x0 = train_data[i][0]
            x1 = train_data[i][1]
            target_output = train_data[i][2]
            pred_output = bipolar(W0*x0 + W1*x1 + b)
            error = target_output - pred_output
            total_error += error**2
            w0_upd = alpha * error * x0
            w1_upd = alpha * error * x1
            b_upd = alpha * error
            W0 += w0_upd
            W1 += w1_upd
            b += b_upd
        errors.append(total_error)
        epoch += 1
        if total_error <= thresh:
            converge = True

    return W0, W1, b, errors

train_data = [(0,0,0), (0,1,0), (1,0,0), (1,1,1)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = 0.05
thresh = 0.002
```

```
W0, W1, b, errors = perceptron(train_data, W0, W1, b, alpha, thresh)

print("Number of epochs needed:", len(errors))
plt.plot(errors)
plt.xlabel('Epochs')
plt.ylabel('Sum-Squared-Error')
plt.show()
```

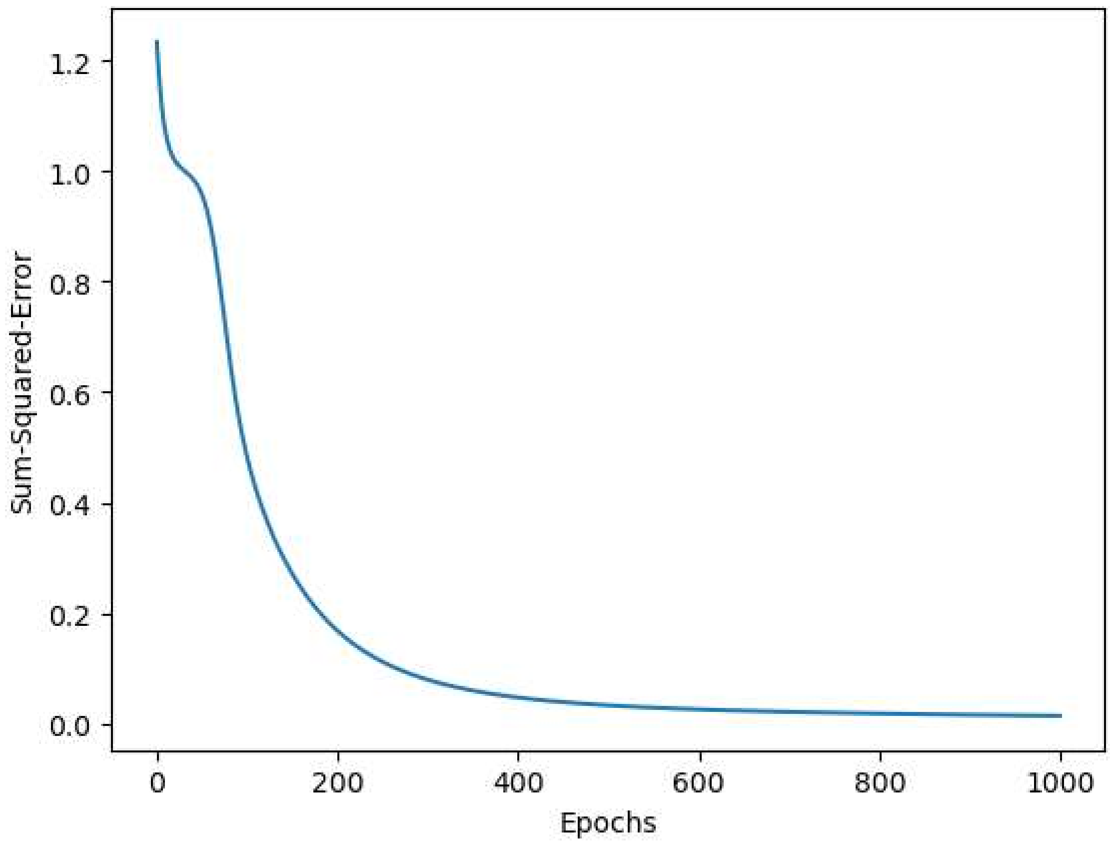
```
Number of epochs needed: 94
```



In [22]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def perceptron(train_data, W0, W1, b, alpha, thresh):
    errors = []
    epoch = 0
    w0_upd = 0
    w1_upd = 0
    b_upd = 0
    converge = False

    while not converge and epoch < 1000:
        total_error = 0
        for i in range(len(train_data)):
            x0 = train_data[i][0]
            x1 = train_data[i][1]
            target_output = train_data[i][2]
            pred_output = sigmoid(W0*x0 + W1*x1 + b)
            error = target_output - pred_output
            total_error += error**2
            w0_upd = alpha * error * x0
            w1_upd = alpha * error * x1
            b_upd = alpha * error
            W0 += w0_upd
            W1 += w1_upd
            b += b_upd
        errors.append(total_error)
        epoch += 1
        if total_error <= thresh:
            converge = True

    return W0, W1, b, errors

train_data = [(0,0,0), (0,1,0), (1,0,0), (1,1,1)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = 0.05
thresh = 0.002

W0, W1, b, errors = perceptron(train_data, W0, W1, b, alpha, thresh)

print("Number of epochs needed:", len(errors))
```

```
plt.plot(errors)
plt.xlabel('Epochs')
plt.ylabel('Sum-Squared-Error')
plt.show()
```

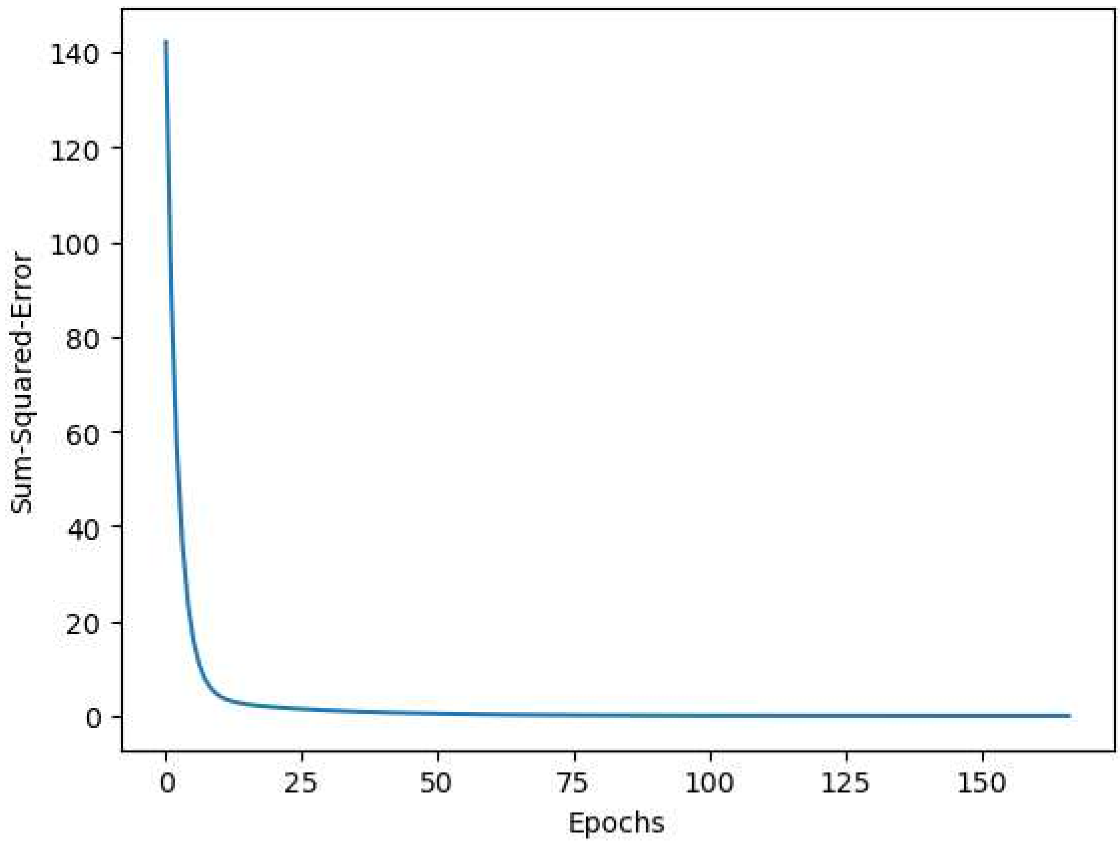
```
Number of epochs needed: 1000
```



In [8]:
```
def relu(x):
    return max(0, x)

def perceptron(train_data, W0, W1, b, alpha, thresh):
    errors = []
    epoch = 0
    w0_upd = 0
    w1_upd = 0
    b_upd = 0
    converge = False

    while not converge and epoch < 1000:
        total_error = 0
        for i in range(len(train_data)):
            x0 = train_data[i][0]
            x1 = train_data[i][1]
            target_output = train_data[i][2]
            pred_output = relu(W0*x0 + W1*x1 + b)
            error = target_output - pred_output
            total_error += error**2
            w0_upd = alpha * error * x0
            w1_upd = alpha * error * x1
            b_upd = alpha * error
            W0 += w0_upd
            W1 += w1_upd
            b += b_upd
        errors.append(total_error)
        epoch += 1
        if total_error <= thresh:
            converge = True

    return W0, W1, b, errors

train_data = [(0,0,0), (0,1,0), (1,0,0), (1,1,1)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = 0.05
thresh = 0.002

W0, W1, b, errors = perceptron(train_data, W0, W1, b, alpha, thresh)

print("Number of epochs needed:", len(errors))
```

```
plt.plot(errors)
plt.xlabel('Epochs')
plt.ylabel('Sum-Squared-Error')
plt.show()
```

```
Number of epochs needed: 167
```

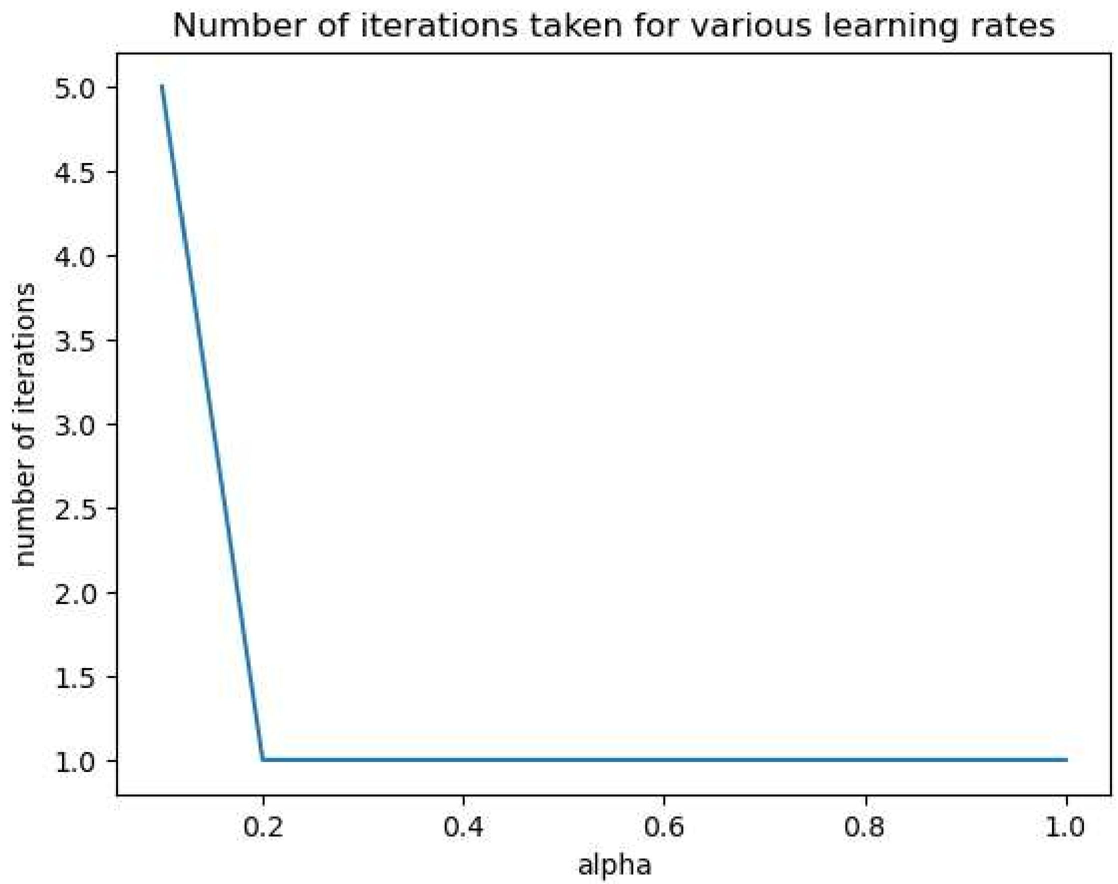


In [23]:
```
#A3
def activation(x):
    if x >= 0:
        return 1
    else:
        return 0

def perceptron(train_data, W0, W1, b, alpha, thresh):

    no_iters = []

    for i in alpha:

        epoch = 0
        errors = []
        w0_upd = 0
        w1_upd = 0
        b_upd = 0
        converge = False

        while not converge and epoch < 1000:
            total_error = 0
            for i in range(len(train_data)):
                x0 = train_data[i][0]
                x1 = train_data[i][1]
                target_output = train_data[i][2]
                pred_output = activation(W0*x0 + W1*x1 + b)
                error = target_output - pred_output
                total_error += error**2
                w0_upd = i * error * x0
                w1_upd = i * error * x1
                b_upd = i * error
                W0 += w0_upd
                W1 += w1_upd
                b += b_upd
            errors.append(total_error)
            epoch += 1
            if total_error <= thresh:
                converge = True

        no_iters.append(len(errors))

    return no_iters
```

```
train_data = [(0,0,0), (0,1,0), (1,0,0), (1,1,1)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
thresh = 0.002

no_iters = perceptron(train_data, W0, W1, b, alpha, thresh)

plt.plot(alpha, no_iters)
plt.xlabel("alpha")
plt.ylabel("number of iterations")
plt.title("Number of iterations taken for various learning rates")
```

Out[23]:

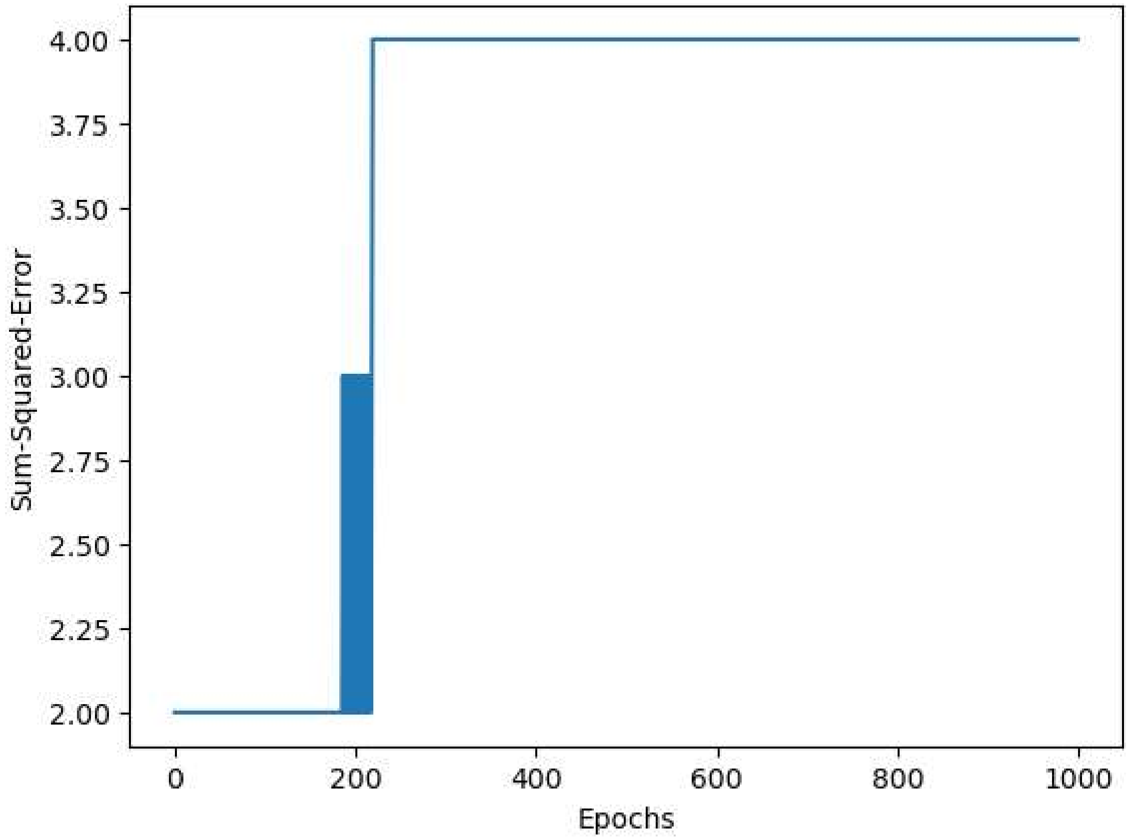
```
Text(0.5, 1.0, 'Number of iterations taken for various learning rates')
```


Number of iterations taken for various learning rates
number of iterations
alpha
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.2
0.4
0.6
0.8
1.0

In [25]:
```
#A4
def bipolar(x):
    if x >= 0:
        return 1
    else:
        return 0

def perceptron(train_data, W0, W1, b, alpha, thresh):
    errors = []
    epoch = 0
    w0_upd = 0
    w1_upd = 0
    b_upd = 0
    converge = False

    while not converge and epoch < 1000:
        total_error = 0
        for i in range(len(train_data)):
            x0 = train_data[i][0]
            x1 = train_data[i][1]
            target_output = train_data[i][2]
            pred_output = bipolar(W0*x0 + W1*x1 + b)
            error = target_output - pred_output
            total_error += error**2
            w0_upd = alpha * error * x0
            w1_upd = alpha * error * x1
            b_upd = alpha * error
            W0 += w0_upd
            W1 += w1_upd
            b += b_upd
        errors.append(total_error)
        epoch += 1
        if total_error <= thresh:
            converge = True

    return W0, W1, b, errors

train_data = [(0,0,0), (0,1,1), (1,0,1), (1,1,0)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = 0.05
thresh = 0.002
```

```
W0, W1, b, errors = perceptron(train_data, W0, W1, b, alpha, thresh)

print("Number of epochs needed:", len(errors))
plt.plot(errors)
plt.xlabel('Epochs')
plt.ylabel('Sum-Squared-Error')
plt.show()
```

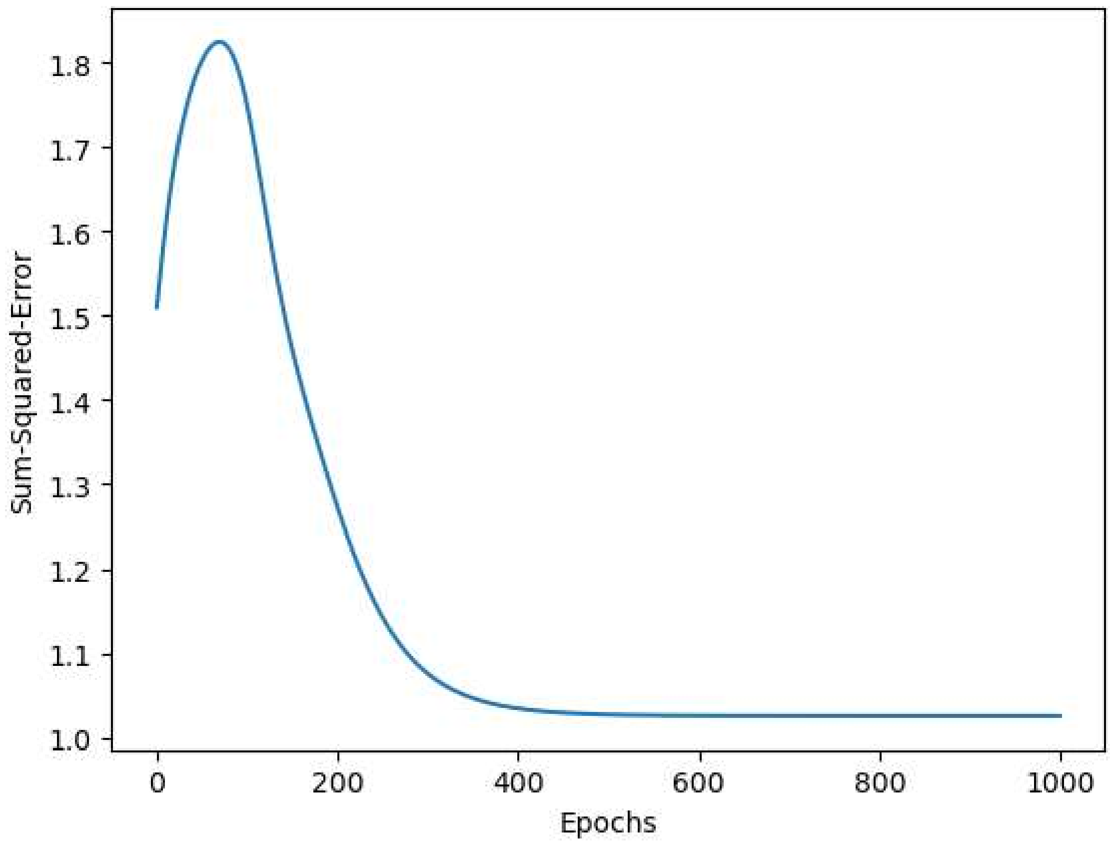
```
Number of epochs needed: 1000
```



In [12]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def perceptron(train_data, W0, W1, b, alpha, thresh):
    errors = []
    epoch = 0
    w0_upd = 0
    w1_upd = 0
    b_upd = 0
    converge = False

    while not converge and epoch < 1000:
        total_error = 0
        for i in range(len(train_data)):
            x0 = train_data[i][0]
            x1 = train_data[i][1]
            target_output = train_data[i][2]
            pred_output = sigmoid(W0*x0 + W1*x1 + b)
            error = target_output - pred_output
            total_error += error**2
            w0_upd = alpha * error * x0
            w1_upd = alpha * error * x1
            b_upd = alpha * error
            W0 += w0_upd
            W1 += w1_upd
            b += b_upd
        errors.append(total_error)
        epoch += 1
        if total_error <= thresh:
            converge = True

    return W0, W1, b, errors

train_data = [(0,0,0), (0,1,1), (1,0,1), (1,1,0)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = 0.05
thresh = 0.002

W0, W1, b, errors = perceptron(train_data, W0, W1, b, alpha, thresh)

print("Number of epochs needed:", len(errors))
```

```
plt.plot(errors)
plt.xlabel('Epochs')
plt.ylabel('Sum-Squared-Error')
plt.show()
```

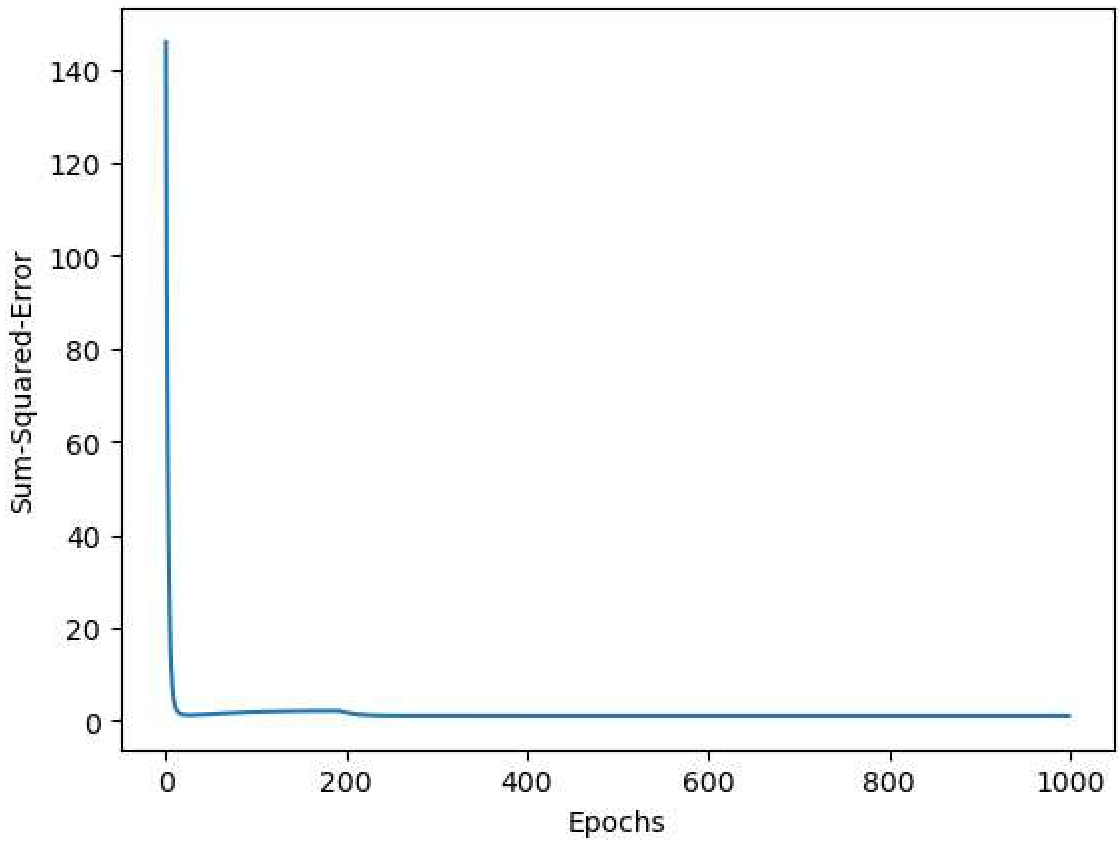
```
Number of epochs needed: 1000
```



In [14]:
```
def relu(x):
    return max(0, x)

def perceptron(train_data, W0, W1, b, alpha, thresh):
    errors = []
    epoch = 0
    w0_upd = 0
    w1_upd = 0
    b_upd = 0
    converge = False

    while not converge and epoch < 1000:
        total_error = 0
        for i in range(len(train_data)):
            x0 = train_data[i][0]
            x1 = train_data[i][1]
            target_output = train_data[i][2]
            pred_output = relu(W0*x0 + W1*x1 + b)
            error = target_output - pred_output
            total_error += error**2
            w0_upd = alpha * error * x0
            w1_upd = alpha * error * x1
            b_upd = alpha * error
            W0 += w0_upd
            W1 += w1_upd
            b += b_upd
        errors.append(total_error)
        epoch += 1
        if total_error <= thresh:
            converge = True

    return W0, W1, b, errors

train_data = [(0,0,0), (0,1,1), (1,0,1), (1,1,0)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = 0.05
thresh = 0.002

W0, W1, b, errors = perceptron(train_data, W0, W1, b, alpha, thresh)

print("Number of epochs needed:", len(errors))
```

```
plt.plot(errors)
plt.xlabel('Epochs')
plt.ylabel('Sum-Squared-Error')
plt.show()
```

```
Number of epochs needed: 1000
```

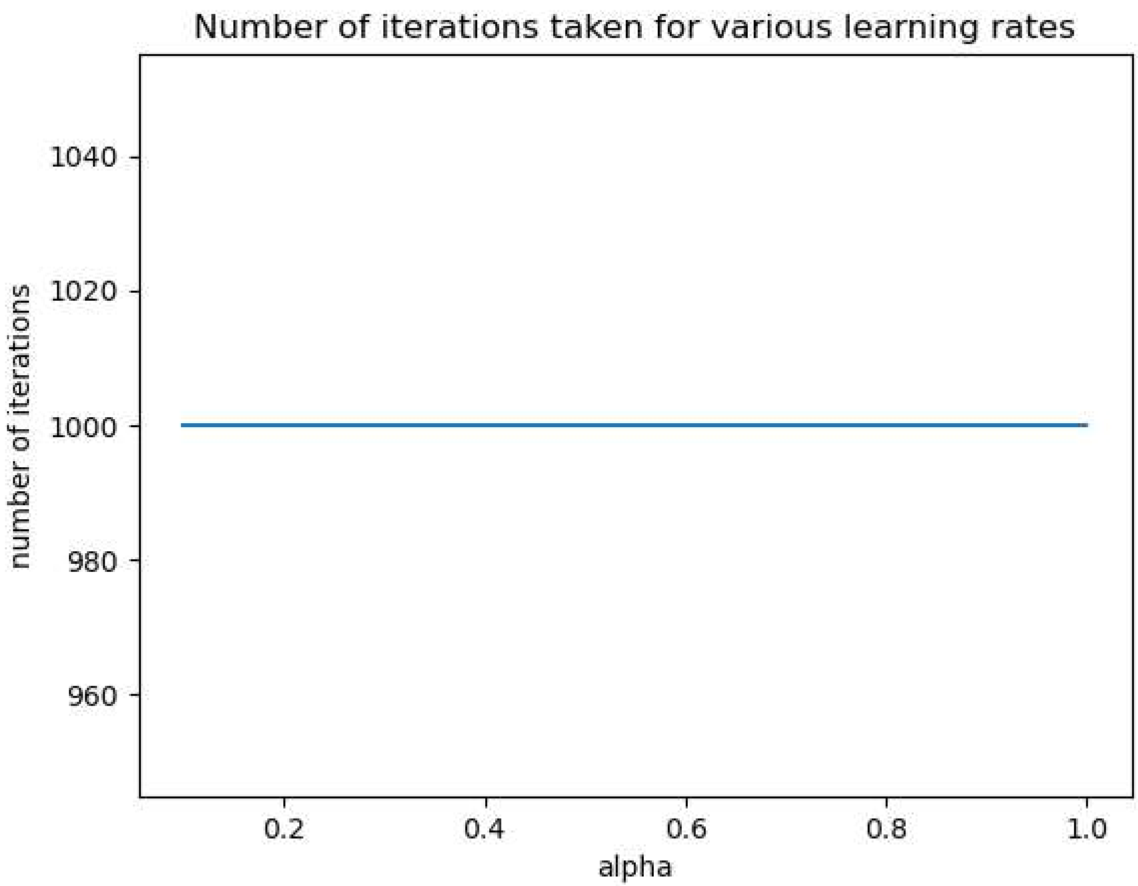


In [18]:
```
def activation(x):
    if x >= 0:
        return 1
    else:
        return 0

def perceptron(train_data, W0, W1, b, alpha, thresh):

    no_of_iters = []

    for i in alpha:

        epoch = 0
        errors = []
        w0_upd = 0
        w1_upd = 0
        b_upd = 0
        converge = False

        while not converge and epoch < 1000:
            total_error = 0
            for i in range(len(train_data)):
                x0 = train_data[i][0]
                x1 = train_data[i][1]
                target_output = train_data[i][2]
                pred_output = activation(W0*x0 + W1*x1 + b)
                error = target_output - pred_output
                total_error += error**2
                w0_upd = i * error * x0
                w1_upd = i * error * x1
                b_upd = i * error
                W0 += w0_upd
                W1 += w1_upd
                b += b_upd
            errors.append(total_error)
            epoch += 1
            if total_error <= thresh:
                converge = True

        no_of_iters.append(len(errors))

    return no_of_iters
```

```
train_data = [(0,0,0), (0,1,1), (1,0,1), (1,1,0)]
W0 = 10
W1 = 0.2
b = -0.75
alpha = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
thresh = 0.002

no_of_iters = perceptron(train_data, W0, W1, b, alpha, thresh)

plt.plot(alpha, no_of_iters)
plt.xlabel("alpha")
plt.ylabel("number of iterations")
plt.title("Number of iterations taken for various learning rates")
```

Out[18]:

```
Text(0.5, 1.0, 'Number of iterations taken for various learning rates')
```



In [27]:
```
#A5
import numpy as np
ip1_data = np.array([[20, 6, 2, 368],
              [16, 3, 6, 289],
              [27, 6, 2, 393],
              [19, 1, 2, 110],
              [24, 4, 2, 280],
              [22, 1, 5, 167],
              [15, 4, 2, 271],
              [18, 4, 2, 274],
              [21, 1, 4, 148],
              [16, 2, 4, 198]])

ip_data = np.array([1, 1, 1, 0, 1, 0, 1, 1, 0, 0])
wts = np.array([0.5, 0.5, 1, 0.5])
learning_rate = 0.1
def sigmoid_func(z):
    return 1 / (1 + np.exp(-z))

def perceptron(ip1_data, ip_data, wts, learning_rate, sigmoid_func):
    errors = []
    converge = False
    epoch = 0

    while not converge and epoch < 1000:
        error = 0

        for i in range(len(ip1_data)):
            op = np.dot(ip1_data[i], w)
            pred_val = sigmoid_func(op)
            delta = learning_rate * (ip_data[i] - pred_val) * pred_val * (1 - pred_val)
            wts += delta * ip1_data[i]
            error += delta ** 2

        errors.append(error)
        epoch += 1

        if error <= 0.002:
            converged = True

    return wts, epoch
wts_trained, epochs = perceptron(ip1_data, ip_data, wts, learning_rate, sigmoid_func)
```

```
print('Trained Weights:', wts_trained)
print('Number of Epochs:', epochs)
new_transaction = np.array([16, 3, 6, 289])
output = sigmoid_func(np.dot(new_transaction, wts_trained))
if output >= 0.5:
    print('High Value Transaction')
else:
    print('Low Value Transaction')
```

```
Trained Weights: [0.5 0.5 1.  0.5]
Number of Epochs: 1000
High Value Transaction
```

In [29]:
```
#A6
import numpy as np
ip1_data = np.array([[20, 6, 2, 368],
              [16, 3, 6, 289],
              [27, 6, 2, 393],
              [19, 1, 2, 110],
              [24, 4, 2, 280],
              [22, 1, 5, 167],
              [15, 4, 2, 271],
              [18, 4, 2, 274],
              [21, 1, 4, 148],
              [16, 2, 4, 198]])

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

wts = np.array([0.5, 0.5, 1, 0.5])
learning_rate = 0.1

def sigmoid(z):
    return 1 / (1 + np.exp(-z))


def perceptron(ip1_data, ip_data, wts, learning_rate, sigmoid):
    errors = []
    converge = False
    epoch = 0

    while not converge and epoch < 1000:
        error = 0

        for i in range(len(ip1_data)):
            op = np.dot(ip1_data[i], w)
            pred_val = sigmoid(op)
            delta = learning_rate * (ip_data[i] - pred_val) * pred_val * (1 - pred_val)
            wts += delta * ip1_data[i]
            error += delta ** 2

        errors.append(error)
        epoch += 1

        if error <= 0.002:
            converged = True
```

```
    return wts, epoch


wts_trained, epochs = perceptron(ip1_data, ip_data, wts, learning_rate, sigmoid)


print('Trained Weights:', wts_trained)
print('Number of Epochs:', epochs)

ip1_data_pinv = np.linalg.pinv(ip1_data)
wts_pinv = np.dot(ip1_data_pinv,ip_data )

print('Trained Weights from Perceptron Learning:', wts_trained)
print('Weights from Matrix Pseudo-Inverse:', wts_pinv)
```

```
Trained Weights: [0.5 0.5 1.  0.5]
Number of Epochs: 1000
Trained Weights from Perceptron Learning: [0.5 0.5 1.  0.5]
Weights from Matrix Pseudo-Inverse: [-0.04822325 -1.02587409 -0.3882899   0.0241747 ]
```

In [31]:
```
data_f = pd.read_excel("LabSession5Data.xlsx")
data_f
```

Out[31]:

	Customer	Candies (#)	Mangoes (Kg)	Milk Packets (#)	Payment (Rs)	High Value (Tx)	Unnamed: 6	Unnamed: 7	Unnamed: 8	Unnamed: 9	...	Unnamed: 12	Unnamed: 13	Unna
0	C_1	20	6	2	386	Yes	NaN	NaN	NaN	NaN	...	NaN	NaN	
1	C_2	16	3	6	289	Yes	NaN	NaN	NaN	NaN	...	NaN	NaN	
2	C_3	27	6	2	393	Yes	NaN	NaN	NaN	NaN	...	NaN	NaN	
3	C_4	19	1	2	110	No	NaN	NaN	NaN	NaN	...	NaN	NaN	
4	C_5	24	4	2	280	Yes	NaN	NaN	NaN	NaN	...	NaN	NaN	
5	C_6	22	1	5	167	No	NaN	NaN	NaN	NaN	...	NaN	NaN	
6	C_7	15	4	2	271	Yes	NaN	NaN	NaN	NaN	...	NaN	NaN	
7	C_8	18	4	2	274	Yes	NaN	NaN	NaN	NaN	...	NaN	NaN	
8	C_9	21	1	4	148	No	NaN	NaN	NaN	NaN	...	NaN	NaN	
9	C_10	16	2	4	198	No	NaN	NaN	NaN	NaN	...	NaN	NaN	

10 rows × 22 columns

In [32]:
```
data1 = data.iloc[:,1:5]
data1
```

Out[32]:

	Candies (#)	Mangoes (Kg)	Milk Packets (#)	Payment (Rs)
0	20	6	2	386
1	16	3	6	289
2	27	6	2	393
3	19	1	2	110
4	24	4	2	280
5	22	1	5	167
6	15	4	2	271
7	18	4	2	274
8	21	1	4	148
9	16	2	4	198

In [33]:
```
pseudo_inverse = np.linalg.pinv(data1)
print("Pseudo inverse is",pseudo_inverse)
```

```
Pseudo inverse is [[-0.01158602 -0.03328061  0.00992701  0.0309081   0.01893411  0.01257157
  -0.00872551  0.00049436  0.01868374 -0.00579619]
 [ 0.00809324 -0.03931864  0.02004214  0.01022259  0.01645572 -0.01682076
   0.00109285  0.00621381 -0.00780631 -0.0171085 ]
 [-0.02400235  0.12210231 -0.06177958 -0.03305478 -0.05136901  0.05064536
  -0.00279828 -0.01898852  0.02274531  0.05261889]
 [ 0.00150006  0.00203556  0.00021249 -0.0018353  -0.00064341 -0.00095362
   0.00101203  0.00046022 -0.00124752  0.00037604]]
```

In [42]:
```
#A7
ip1 = np.array([[0, 0],
                [0, 1],
                [1, 0],
                [1, 1]])

ip2 = np.array([[0], [0], [0], [1]])

def sigmoid(g):
    return 1 / (1 + np.exp(-g))

def sigmoid_derivative(g):
    return sigmoid(g) * (1 - sigmoid(g))

def neural_network(ip1, ip2, learning_rate):
    #initial weights and biases
    w1 = np.array([[0.1, 0.2], [0.3, 0.4]])
    b1 = np.array([[0.5], [0.6]])
    w2 = np.array([[0.7], [0.8]])
    b2 = np.array([[0.9]])

    epochs = 1000
    error_threshold = 0.002
    for i in range(epochs):

        g1 = np.dot(ip1, w1) + b1.T
        op1 = sigmoid_activation(g1)
        g2 = np.dot(op1, w2) + b2.T
        ip2_pred = sigmoid(g2)

        error = ip2 - ip2_pred
        d2 = error * sigmoid_derivative(g2)
        d1 = np.dot(d2, w2.T) * sigmoid_derivative(g1)
        w2 += learning_rate * np.dot(op1.T, d2)
        b2 += learning_rate * np.sum(d2, axis=0, keepdims=True).T
        w1 += learning_rate * np.dot(ip1.T, d1)
        b1 += learning_rate * np.sum(d1, axis=0, keepdims=True).T

        mse = np.mean(error ** 2)

        if mse <= error_threshold:
            break
```

```
    return w1, b1, w2, b2, i+1


w1, b1, w2, b2, epochs = neural_network(ip1, ip2, 0.05)


print('Trained Weights and Biases of Hidden Layer:', w1, b1)
print('Trained Weights and Biases of Output Layer:', w2, b2)
print('Number of Epochs:', epochs)
```

```
Trained Weights and Biases of Hidden Layer: [[-0.16274875  0.10536218]
 [ 0.03045954  0.30465139]] [[0.44635021]
 [0.52036229]]
Trained Weights and Biases of Output Layer: [[-0.29914351]
 [ 0.03514045]] [[-0.92324186]]
Number of Epochs: 1000
```

In [36]:
```
#A8
ip1 = np.array([[0, 0],
                [0, 1],
                [1, 0],
                [1, 1]])

ip2 = np.array([[0], [1], [1], [0]])

def sigmoid(g):
    return 1 / (1 + np.exp(-g))

def sigmoid_derivative(g):
    return sigmoid_func(g) * (1 - sigmoid_func(g))

def neural_network(ip1, ip2, lr_rate):
    #initial weights and biases
    w1 = np.array([[0.1, 0.2], [0.3, 0.4]])
    b1 = np.array([[0.5], [0.6]])
    w2 = np.array([[0.7], [0.8]])
    b2 = np.array([[0.9]])


    epochs = 1000
    error_threshold = 0.002

    for i in range(epochs):

        g1 = np.dot(ip1, w1) + b1.T
        op1 = sigmoid_activation(g1)
        g2 = np.dot(op1, w2) + b2.T
        ip2_pred = sigmoid(g2)

        error = ip2 - ip2_pred
        d2 = error * sigmoid_derivative(g2)
        d1 = np.dot(d2, w2.T) * sigmoid_derivative(g1)
        w2 += learning_rate * np.dot(op1.T, d2)
        b2 += learning_rate * np.sum(d2, axis=0, keepdims=True).T
        w1 += learning_rate * np.dot(ip1.T, d1)
        b1 += learning_rate * np.sum(d1, axis=0, keepdims=True).T

        mserror = np.mean(error ** 2)
        if mserror <= error_threshold:
            break
```

```
    return w1, b1, w2, b2, i+1


w1, b1, w2, b2, epochs = neural_network(ip1, ip2, 0.05)


print('Trained Weights and Biases of Hidden Layer:', w1, b1)
print('Trained Weights and Biases of Output Layer:', w2, b2)
print('Number of Epochs:', epochs)
```

```
Trained Weights and Biases of Hidden Layer: [[0.07268746 0.20986447]
 [0.2629408  0.38294729]] [[0.41573035]
 [0.51056756]]
Trained Weights and Biases of Output Layer: [[0.04150609]
 [0.14919327]] [[-0.128267]]
Number of Epochs: 1000
```

In [37]:
```
#A9
v10,v20 = 0.01,0.4
v11,v12,v21,v22 = 10,0.2,-0.75,0.09
w10,w20 = 0.11,0.41
w11,w12,w21,w22 = -20,0.1,-1.2,0.7
learning_rate = 0.05
bias = [1,1,1,1]
x1 = [0,0,1,1]
x2 = [0,1,0,1]
op_a = [1,0,0,1]
op_a1 = [0,1,1,0]
op_a2 = [1,0,0,1]
op_pre = 0
h1,h2 = 0,0
op_pred1, op_pred2 = 0, 0
iterations=0
while (iterations < 2500):
    print("Epoch",iterations+1)
    for i in range(0,len(bias)):
        h1 = bias[i] * v10 + x1[i] * v11 + x2[i] * v21
        h2 = bias[i] * v20 + x1[i] * v12 + x2[i] * v22
        op_pred1 = 1/(1+ np.exp(-h1))
        op_pred2 = 1/(1+ np.exp(-h2))
        op_pred01 = 1/(1+ np.exp(-(w10 + op_pred1 * w11 + op_pred2 * w21)))
        op_pred02 = 1/(1+ np.exp(-(w20 + op_pred1 * w12 + op_pred2 * w22)))
        if (op_pred01 == op_a[i]):
            print("The Output 1: ")
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\n""h1 = ",op_pred1,"\n""h2
            continue
        else:
            derivative = op_pred01*(1-op_pred01)
            delk = derivative*(-op_pred01 + op_a1[i])
            del1 = op_pred1*(1-op_pred1)*(w11)*delk
            del2 = op_pred2*(1-op_pred2)*(w21)*delk
            w10 = w10 + learning_rate * delk * 1
            w11 = w11 + learning_rate * delk * op_pred1
            w12 = w21 + learning_rate * delk * op_pred2
            v10,v20 = (v10 + learning_rate*del1*bias[i]),(v20 + learning_rate*del2*bias[i])
            v11,v12,v21,v22 = (v11 + learning_rate*del1*x1[i]),(v12 + learning_rate*del2*x1[i]),(v21 + le
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\n""h1 unit = ",op_pred1,"\n
        if (op_pred02 == op_a[i]):
            print("The Output 2: ")
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\n""h1 = ",op_pred1,"\n""h2
```

```
            continue
        else:
            derivative = op_pred02*(1-op_pred02)
            delk = derivative*(-op_pred02 + op_a2[i])
            del1 = op_pred1*(1-op_pred1)*(w12)*delk
            del2 = op_pred2*(1-op_pred2)*(w22)*delk
            w20 = w20 + learning_rate * delk * 1
            w21 = w21 + learning_rate * delk * op_pred1
            w22 = w22 + learning_rate * delk * op_pred2
            v10,v20 = (v10 + learning_rate*del1*bias[i]),(v20 + learning_rate*del2*bias[i])
            v11,v12,v21,v22 = (v11 + learning_rate*del1*x1[i]),(v12 + learning_rate*del2*x1[i]),(v21 + le
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\n""h1 unit = ",op_pred1,"\n

    iterations=iterations+1
    if abs(op_pred01 - op_a1[i]) < 0.002 and abs(op_pred02 - op_a2[i]) < 0.002:
        print("The error is ",abs( op_pred02 - op_a2[i]),abs( op_pred01 - op_a1[i]))
        break
    else:
        continue
```

```
Epoch 1

bias =  1
x1 =  0
x2 =  0
h1 unit =  0.5024999791668749
h2 unit= 0.598687660112452

bias =  1
x1 =  0
x2 =  0
h1 unit =  0.5024999791668749
h2 unit = 0.598687660112452

bias =  1
x1 =  0
x2 =  1
h1 unit =  0.3228047544296615
h2 unit= 0.6202268665825541
```

In [39]:
```
#A10
from sklearn.neural_network import MLPClassifier
X1 = [[0, 0], [0, 1], [1, 0], [1, 1]]
Y1 = [0, 0, 0, 1]
cf_and = MLPClassifier(hidden_layer_sizes=(2,), activation='logistic', solver='lbfgs', random_state=1)
cf_and.fit(X1, Y1)
accuracy_1 = clf_and.score(X1, Y1)
predictions_1 = clf_and.predict(X1)
print("AND Gate Results:")
print("Actual:", Y1)
print("Predicted:", predictions_1)
print("Accuracy is:", accuracy_1)

X2 = [[0, 0], [0, 1], [1, 0], [1, 1]]
Y2 = [0, 1, 1, 0]
cf_xor = MLPClassifier(hidden_layer_sizes=(2,), activation='logistic', solver='lbfgs', random_state=1)
cf_xor.fit(X2, Y2)
accuracy_2 = cf_xor.score(X2, Y2)
predictions_2 = cf_xor.predict(X2)
print("\nXOR Gate Results:")
print("Actual:", Y2)
print("Predicted:", predictions_2)
print("Accuracy is:", accuracy_2)
```

```
AND Gate Results:
Actual: [0, 0, 0, 1]
Predicted: [0 0 0 1]
Accuracy is: 1.0

XOR Gate Results:
Actual: [0, 1, 1, 0]
Predicted: [0 1 0 1]
Accuracy is: 0.5
```

In [41]:
```
#A11
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
from sklearn.impute import SimpleImputer

data = pd.read_csv('HAM10000_metadata.csv')
label_encoder = LabelEncoder()
data['sex'] = label_encoder.fit_transform(data['sex'])
X = data[['age', 'sex']]
y = data['dx']

imputer = SimpleImputer(strategy='most_frequent')
X = imputer.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
cf = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=1000, random_state=42)
cf.fit(X_train, y_train)
y_pred = cf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

```
Accuracy: 67.55%
```

In [ ]: