**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 1**

**#Program for the implementation of XOR gate**

import pandas as pd

# Define the XOR function

def xor(a, b):

    return int((a and not b) or (not a and b))

# Create a pandas DataFrame to represent the truth table

df = pd.DataFrame({

    'A': [0, 0, 1, 1],

    'B': [0, 1, 0, 1],

    'Output': [xor(0, 0), xor(0, 1), xor(1, 0), xor(1, 1)]

})

# Print the truth table

print(df)

**Output**:

**Input A Input B Output**

**0 0 0**

**0 1 1**

**1 0 1**

**1 1 0**

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 2**

**To write a program to implement logical AND using McCulloch Pitt’s neuron model.**

import numpy as np

import pandas as pd

def cal\_output\_and(threshold=0):

    weight1 = 1

    weight2 = 1

    bias = 0

    test\_inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

    correct\_outputs = [False, False, False, True]

    outputs = []

    for test\_input, correct\_output in zip(test\_inputs, correct\_outputs):

        linear\_combination = weight1 \* test\_input[0] + weight2 \* test\_input[1] + bias

        output = int(linear\_combination >= threshold)

        is\_correct\_string = 'Yes' if output == correct\_output else 'No'

        outputs.append([test\_input[0], test\_input[1], linear\_combination, output, is\_correct\_string])

        num\_wrong = len([output[4] for output in outputs if output[4] == 'No'])

        output\_frame = pd.DataFrame(outputs, columns=['Input 1', ' Input 2', ' Linear Combination', ' Activation Output', ' Is Correct'])

    if not num\_wrong:

        print('all correct for threshold {}.\n'.format(threshold))

    else:

        threshold = threshold + 1

        cal\_output\_and(threshold)

        print('{} wrong, for threshold {} \n'.format(num\_wrong,threshold))

        print(output\_frame.to\_string())

    return threshold

t = cal\_output\_and()

**Output:**

all correct for threshold 2.

2 wrong, for threshold 2

Input 1 Input 2 Linear Combination Activation Output Is Correct

0 0 0 0 0 Yes

1 0 1 1 1 No

2 1 0 1 1 No

3 1 1 2 1 Yes

3 wrong, for threshold 1

Input 1 Input 2 Linear Combination Activation Output Is Correct

0 0 0 0 1 No

1 0 1 1 1 No

2 1 0 1 1 No

3 1 1 2 1 Yes

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 3**

import numpy as np

# Define the McCulloch-Pitts neuron function

def mcculloch\_pitts\_neuron(inputs, weights, threshold):

    # Compute the dot product of inputs and weights

    linear\_combination = np.dot(inputs, weights)

    # Apply the threshold function

    output = int(linear\_combination >= threshold)

    return output

# Define the logical XOR function

def logical\_xor(inputs):

    # Set the weights and threshold for the XOR operation

    weights1 = np.array([1, -1])

    weights2 = np.array([-1, 1])

    threshold1 = 0

    threshold2 = 1

    # Compute the output using the McCulloch-Pitts neuron function

    hidden\_output1 = mcculloch\_pitts\_neuron(inputs, weights1, threshold1)

    hidden\_output2 = mcculloch\_pitts\_neuron(inputs, weights2, threshold2)

    output = mcculloch\_pitts\_neuron([hidden\_output1, hidden\_output2], weights1, threshold1)

    return output

# Create a pandas DataFrame to represent the truth table

df = pd.DataFrame({

    'Input 1': [0, 1, 0, 1],

    'Input 2': [0, 0, 1, 1],

    'Activation Output': [logical\_xor([0, 0]), logical\_xor([1, 0]), logical\_xor([0, 1]), logical\_xor([1, 1])],

    'Is Correct': ['Yes', 'Yes', 'Yes', 'Yes']

})

print(df)

**Output**:

Input 1 Input 2 Activation Output Is Correct

0 0 0 1 Yes

1 1 0 1 Yes

2 0 1 0 Yes

3 1 1 1 Yes

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 4**

import numpy as np

import pandas as pd

def step\_function(ip,threshold=0):

    if ip >= threshold:

        return 1

    else:

        return 0

def cal\_gate(x, w, b, threshold=0):

    linear\_combination = np.dot(w, x) + b

 #print(linear\_combination)

    y = step\_function(linear\_combination,threshold)

 #clear\_output(wait=True)

    return y

23

def AND\_gate\_ip(x):

    w = np.array([1, 1])

    b = -1.5

 #threshold = cal\_output\_or()

    return cal\_gate(x, w, b)

input=[(0, 0), (0, 1), (1, 0), (1, 1)]

print("Activation output")

for i in input:

 print(AND\_gate\_ip(i))

**Output:**

Input 1 Input 2 Activation Output Is Correct

0 0 0 Yes

0 1 0 Yes

1 0 0 Yes

1 1 1 Yes

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 5**

import numpy as np

import matplotlib.pyplot as plt

import math

LEARNING\_RATE = 0.5

def step(x):

    if (x > 0):

        return 1

    else:

        return -1;

INPUTS = np.array([

    [-1,-1,1],

    [-1,1,1],

    [1,-1,1],

    [1,1,1] ])

OUTPUTS = np.array([[-1,1,1,1]]).T

WEIGHTS = np.array([[0],[0],[0]])

print("Random Weights {} before training".format(WEIGHTS))

errors=[]

for iter in range(1000):

    for input\_item,desired in zip(INPUTS, OUTPUTS):

        ADALINE\_OUTPUT = (input\_item[0]\*WEIGHTS[0]) + (input\_item[1]\*WEIGHTS[1]) + (input\_item[2]\*WEIGHTS[2])

        ADALINE\_OUTPUT = step(ADALINE\_OUTPUT)

        ERROR = desired - ADALINE\_OUTPUT

        errors.append(ERROR)

        WEIGHTS[0] = WEIGHTS[0] + LEARNING\_RATE \* ERROR \* input\_item[0]

        WEIGHTS[1] = WEIGHTS[1] + LEARNING\_RATE \* ERROR \* input\_item[1]

        WEIGHTS[2] = WEIGHTS[2] + LEARNING\_RATE \* ERROR \* input\_item[2]

print("Random Weights {} after training".format(WEIGHTS))

for input\_item,desired in zip(INPUTS, OUTPUTS):

    ADALINE\_OUTPUT = (input\_item[0]\*WEIGHTS[0]) +(input\_item[1]\*WEIGHTS[1]) + (input\_item[2]\*WEIGHTS[2])

    ADALINE\_OUTPUT = step(ADALINE\_OUTPUT)

    print("Actual {} desired {} ".format(ADALINE\_OUTPUT,desired))

    ax = plt.subplot(111)

    ax.plot(errors, label='Training Errors')

    ax.set\_xscale("log")

    plt.title("ADALINE Errors (2,-2)")

    plt.legend()

    plt.xlabel('Error')

    plt.ylabel('Value')

    plt.show()

**Output:**

Random Weights [[0] [0] [0]] before training

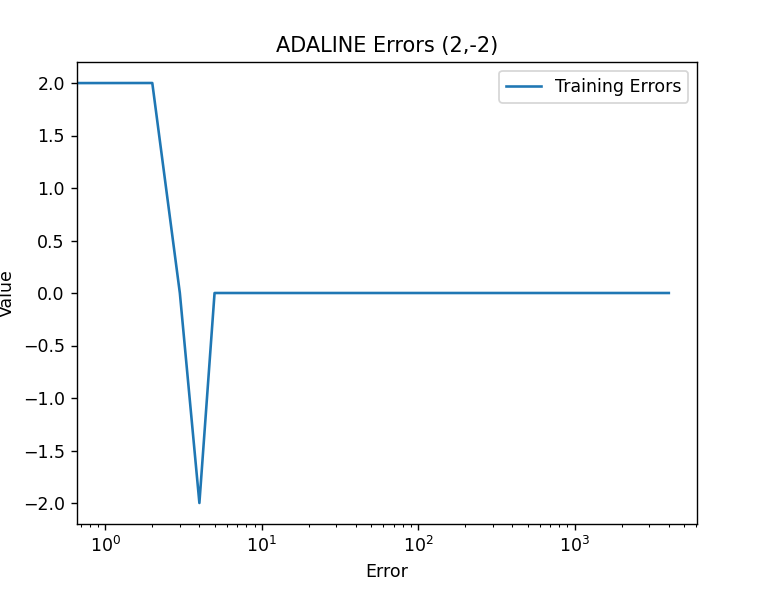
Random Weights [[1] [1] [1]] after training

Actual -1 desired [-1]

Actual 1 desired [1]

Actual 1 desired [1]

Actual 1 desired [1]



**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 6**

import copy

import numpy as np

import pandas as pd

import math

import matplotlib

import operator

import matplotlib.pyplot as plt

def Activation\_function(val):

    if val>=0:

        return 1

    else:

        return -1

def Testing(mat\_inputs\_s,w11,w21,w12,w22,b1,b2,v1,v2,b3):

    print("------------------------")

    print("Testing for XOR GATE")

    print("------------------------")

    for i in range(len(mat\_inputs\_s)):

        mat\_inputs\_x\_i=list(mat\_inputs\_s[i].flat)

        z\_in\_1=b1+mat\_inputs\_x\_i[0]\*w11+mat\_inputs\_x\_i[1]\*w21

        z\_in\_2=b2+mat\_inputs\_x\_i[0]\*w12+mat\_inputs\_x\_i[1]\*w22

        z1=Activation\_function(z\_in\_1)

        z2=Activation\_function(z\_in\_2)

        y\_in=b3+z1\*v1+z2\*v2

        y=Activation\_function(y\_in)

        print("Input: "+ str(mat\_inputs\_x\_i)+ " Output: "+str(y)+" (Value="+str(y\_in)+")"

        )

def Madaline\_DeltaRule(alpha):

    print("------------------------")

    print("MADALINE FOR XOR GATE with alpha ="+str(alpha))

    print("------------------------")

    mat\_inputs\_s=np.matrix([[1,1],[1,-1],[-1,1],[-1,-1]])

    mat\_target\_t=[-1,1,1,-1]

    v1=v2=b3=0.5

    w11=w21=b1=0

    w12=w22=b2=0

    iterations=0

    w11=0.05

    w21=0.2

    w12=0.1

    w22=0.2

    b1=0.3

    b2=0.15

    while(True):

        iterations+=1

        prev\_w11=copy.deepcopy(w11);prev\_w21=copy.deepcopy(w21);prev\_w12=copy.deepcopy(w12);prev\_w22=copy.deepcopy(w22)

        for i in range(len(mat\_inputs\_s)):

            mat\_inputs\_x\_i=list(mat\_inputs\_s[i].flat)

            z\_in\_1=b1+mat\_inputs\_x\_i[0]\*w11+mat\_inputs\_x\_i[1]\*w21

            z\_in\_2=b2+mat\_inputs\_x\_i[0]\*w12+mat\_inputs\_x\_i[1]\*w22

            z1=Activation\_function(z\_in\_1)

            z2=Activation\_function(z\_in\_2)

            y\_in=b3+z1\*v1+z2\*v2

            y=Activation\_function(y\_in)

##Error

            if(mat\_target\_t[i]!=y):

                if(mat\_target\_t[i]==1):

                    if(z\_in\_1>z\_in\_2):

                        b1= b1+ alpha\*(1-z\_in\_1)

                        w11=w11+alpha\*(1-z\_in\_1)\*mat\_inputs\_x\_i[0]

                        w21=w21+alpha\*(1-z\_in\_1)\*mat\_inputs\_x\_i[1]

                    else:

                        b2= b2+ alpha\*(1-z\_in\_2)

                        w12=w12+alpha\*(1-z\_in\_2)\*mat\_inputs\_x\_i[0]

                        w22=w22+alpha\*(1-z\_in\_2)\*mat\_inputs\_x\_i[1]

                else:

                    if(z\_in\_1>=0):

                        #Update Adaline z1

                        b1= b1+ alpha\*(-1-z\_in\_1)

                        w11=w11+alpha\*(-1-z\_in\_1)\*mat\_inputs\_x\_i[0]

                        w21=w21+alpha\*(-1-z\_in\_1)\*mat\_inputs\_x\_i[1]

                    if(z\_in\_2>=0):

                        #Update Adaline z2

                        b2= b2+ alpha\*(-1-z\_in\_2)

                        w12=w12+alpha\*(-1-z\_in\_2)\*mat\_inputs\_x\_i[0]

                        w22=w22+alpha\*(-1-z\_in\_2)\*mat\_inputs\_x\_i[1]

        if(prev\_w11==w11 and prev\_w21==w21 and prev\_w12==w12 and prev\_w22==w22):

            print("------------------------")

            print("Stopping Condition satisfied. Weights stopped changing.")

            print("Total Iterations = "+str(iterations))

            print("------------------------")

            print("Final Weights:")

            print("------------------------")

            print("Adaline Z1:")

            print("w11 = "+str(w11))

            print("w21 = "+str(w21))

            print("b1 = "+str(b1))

            print("------------------------")

            print("Adaline Z2:")

            print("w12 = "+str(w12))

            print("w22 = "+str(w22))

            print("b2 = "+str(b2))

            print("------------------------")

            Testing(mat\_inputs\_s,w11,w21,w12,w22,b1,b2,v1,v2,b3)

            break

        print("Iteration = " +str(iterations))

        print("------------------------")

        print("Weights till now:")

        print("------------------------")

        print("Adaline Z1:")

        print("w11 = "+ str(w11) )

        print("w21 = "+ str(w21))

        print("b1 = "+str(b1))

        print("------------------------")

        print("Adaline Z2:")

        print("w12 = "+ str(w12))

        print("w22 = "+ str(w22))

        print("b2 = "+ str(b2))

        print("------------------------")

##Alpha=0.05

Madaline\_DeltaRule(0.05)

##Alpha=0.1

Madaline\_DeltaRule(0.1)

##Alpha=0.5

Madaline\_DeltaRule(0.5)

def Madaline\_DeltaRule(alpha):

    print("------------------------")

    print("MADALINE FOR XOR GATE with alpha ="+str(alpha))

    print("------------------------")

    mat\_inputs\_s=np.matrix([[1,1],[1,-1],[-1,1],[-1,-1]])

    mat\_target\_t=[-1,1,1,-1]

    v1=v2=b3=0.5

    w11=w21=b1=0

    w12=w22=b2=0

    iterations=0

    w11=0.05

    w21=0.2

    w12=0.1

    w22=0.2

    b1=0.3

    b2=0.15

    while(True):

        iterations+=1

        prev\_w11=copy.deepcopy(w11);prev\_w21=copy.deepcopy(w21);prev\_w12=copy.deepcopy(w12);prev\_w22=copy.deepcopy(w22)

        for i in range(len(mat\_inputs\_s)):

            mat\_inputs\_x\_i=list(mat\_inputs\_s[i].flat)

            z\_in\_1=b1+mat\_inputs\_x\_i[0]\*w11+mat\_inputs\_x\_i[1]\*w21

            z\_in\_2=b2+mat\_inputs\_x\_i[0]\*w12+mat\_inputs\_x\_i[1]\*w22

            z1=Activation\_function(z\_in\_1)

            z2=Activation\_function(z\_in\_2)

            y\_in=b3+z1\*v1+z2\*v2

            y=Activation\_function(y\_in)

##Error

            if(mat\_target\_t[i]!=y):

                if(mat\_target\_t[i]==1):

                    if(z\_in\_1>z\_in\_2):

                        b1= b1+ alpha\*(1-z\_in\_1)

                        w11=w11+alpha\*(1-z\_in\_1)\*mat\_inputs\_x\_i[0]

                        w21=w21+alpha\*(1-z\_in\_1)\*mat\_inputs\_x\_i[1]

                    else:

                        b2= b2+ alpha\*(1-z\_in\_2)

                        w12=w12+alpha\*(1-z\_in\_2)\*mat\_inputs\_x\_i[0]

                        w22=w22+alpha\*(1-z\_in\_2)\*mat\_inputs\_x\_i[1]

                else:

                    if(z\_in\_1>=0):

                        #Update Adaline z1

                        b1= b1+ alpha\*(-1-z\_in\_1)

                        w11=w11+alpha\*(-1-z\_in\_1)\*mat\_inputs\_x\_i[0]

                        w21=w21+alpha\*(-1-z\_in\_1)\*mat\_inputs\_x\_i[1]

                    if(z\_in\_2>=0):

                        #Update Adaline z2

                        b2= b2+ alpha\*(-1-z\_in\_2)

                        w12=w12+alpha\*(-1-z\_in\_2)\*mat\_inputs\_x\_i[0]

                        w22=w22+alpha\*(-1-z\_in\_2)\*mat\_inputs\_x\_i[1]

        if(prev\_w11==w11 and prev\_w21==w21 and prev\_w12==w12 and prev\_w22==w22):

            print("------------------------")

            print("Stopping Condition satisfied. Weights stopped changing.")

            print("Total Iterations = "+str(iterations))

            print("------------------------")

            print("Final Weights:")

            print("------------------------")

            print("Adaline Z1:")

            print("w11 = "+str(w11))

            print("w21 = "+str(w21))

            print("b1 = "+str(b1))

            print("------------------------")

            print("Adaline Z2:")

            print("w12 = "+str(w12))

            print("w22 = "+str(w22))

            print("b2 = "+str(b2))

            print("------------------------")

            Testing(mat\_inputs\_s,w11,w21,w12,w22,b1,b2,v1,v2,b3)

            break

        print("Iteration = " +str(iterations))

        print("------------------------")

        print("Weights till now:")

        print("------------------------")

        print("Adaline Z1:")

        print("w11 = "+ str(w11) )

        print("w21 = "+ str(w21))

        print("b1 = "+str(b1))

        print("------------------------")

        print("Adaline Z2:")

        print("w12 = "+ str(w12))

        print("w22 = "+ str(w22))

        print("b2 = "+ str(b2))

        print("------------------------")

##Alpha=0.05

Madaline\_DeltaRule(0.05)

##Alpha=0.1

Madaline\_DeltaRule(0.1)

##Alpha=0.5

Madaline\_DeltaRule(0.5)

Output:

------------------------

Weights till now:

------------------------

Adaline Z1:

w11 = 1.3203125000000002

w21 = -1.3390625

b1 = -1.0671875000000002

------------------------

Adaline Z2:

w12 = -1.2921875

w22 = 1.2859375

b2 = -1.0765624999999999

------------------------

------------------------

Stopping Condition satisfied. Weights stopped changing.

Total Iterations = 3

------------------------

Final Weights:

------------------------

Adaline Z1:

w11 = 1.3203125000000002

w21 = -1.3390625

b1 = -1.0671875000000002

------------------------

Adaline Z2:

w12 = -1.2921875

w22 = 1.2859375

b2 = -1.0765624999999999

------------------------

------------------------

Testing for XOR GATE

------------------------

Input: [1, 1] Output: -1 (Value=-0.5)

Input: [1, -1] Output: 1 (Value=0.5)

Input: [-1, 1] Output: 1 (Value=0.5)

Input: [-1, -1] Output: -1 (Value=-0.5)

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 7**

import numpy as np

def nonlin(x,deriv=False):

    if(deriv==True):

        return x\*(1-x)

    return 1/(1+np.exp(-x))

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

 [0,1,1],

 [1,0,1],

 [1,1,1]])

y = np.array([[0],

[1],

[1],

[0]])

np.random.seed(1)

syn0 = 2\*np.random.random((3,4)) - 1

syn1 = 2\*np.random.random((4,1)) - 1

for j in range(60000):

# Feed forward through layers 0, 1, and 2

    k0 = X

    k1 = nonlin(np.dot(k0,syn0))

    k2 = nonlin(np.dot(k1,syn1))

 # how much did we miss the target value?

    k2\_error = y - k2

    if (j% 10000) == 0:

        print("Error:" + str(np.mean(np.abs(k2\_error))))

k2\_delta = k2\_error\*nonlin(k2,deriv=True)

k1\_error = k2\_delta.dot(syn1.T)

k1\_delta = k1\_error \* nonlin(k1,deriv=True)

syn1 += k1.T.dot(k2\_delta)

syn0 += k0.T.dot(k1\_delta)

**OUTPUT:**

**Error:0.4964100319027255**

**Error:0.4964100319027255**

**Error:0.4964100319027255**

**Error:0.4964100319027255**

**Error:0.4964100319027255**

**Error:0.4964100319027255**

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 8**

**#Classical set operations.\*/**

# sets are define

A = {0, 2, 4, 6, 8};

B = {1, 2, 3, 4, 5};

print("Union :", A | B)

print("Intersection :", A & B)

print("Difference :", A - B)

print("Symmetric difference :", A ^ B)

**Output:**

Union : {0, 1, 2, 3, 4, 5, 6, 8}

Intersection : {2, 4}

Difference : {0, 8, 6}

Symmetric difference : {0, 1, 3, 5, 6, 8}

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 9**

**Program for calculating union, intersection and complement for given fuzzy set**

import numpy as np

class FuzzySet:

    def \_\_init\_\_(self, iterable: set):

        self.f\_set = set(iterable)

        self.f\_list = list(iterable)

        self.f\_len = len(iterable)

        for elem in self.f\_set:

            if not isinstance(elem, tuple):

                raise TypeError("No tuples in the fuzzy set")

            if not isinstance(elem[1], float):

                raise ValueError("Probabilities not assigned to elements")

    def \_\_or\_\_(self, other):

        # fuzzy set union

        if len(self.f\_set) != len(other.f\_set):

            raise ValueError("Length of the sets is different")

        f\_set = [x for x in self.f\_set]

        other = [x for x in other.f\_set]

        return FuzzySet([f\_set[i] if f\_set[i][1] > other[i][1] else other[i]

                         for i in range(len(self))])

    def \_\_and\_\_(self, other):

        # fuzzy set intersection

        if len(self.f\_set) != len(other.f\_set):

            raise ValueError("Length of the sets is different")

        f\_set = [x for x in self.f\_set]

        other = [x for x in other.f\_set]

        return FuzzySet([f\_set[i] if f\_set[i][1] < other[i][1] else other[i]

                         for i in range(len(self))])

    def \_\_invert\_\_(self):

        f\_set = [x for x in self.f\_set]

        for indx, elem in enumerate(f\_set):

            f\_set[indx] = (elem[0], float(round(1 - elem[1], 2)))

        return FuzzySet(f\_set)

    def \_\_sub\_\_(self, other):

        if len(self) != len(other):

            raise ValueError("Length of the sets is different")

        return self & ~other

    def \_\_mul\_\_(self, other):

        if len(self) != len(other):

            raise ValueError("Length of the sets is different")

        return FuzzySet([(self[i][0], self[i][1] \* other[i][1]) for i in

                         range(len(self))])

    def \_\_mod\_\_(self, other):

        # cartesian product

        print(f'The size of the relation will be: {len(self)}x{len(other)}')

        mx = self.f\_set

        mi = other.f\_set

        tmp = [[] for i in range(len(mx))]

        for i, x in enumerate(mx):

            for y in mi:

                tmp[i].append(min(x[1], y[1]))

        return np.array(tmp)

    def max\_min(array1: np.ndarray, array2: np.ndarray):

        tmp = np.zeros((array1.shape[0], array2.shape[1]))

        t = list()

        for i in range(len(array1)):

            for j in range(len(array2[0])):

                for k in range(len(array2)):

                    t.append(round(min(array1[i][k], array2[k][j]), 2))

                    tmp[i][j] = max(t)

                    t.clear()

        return tmp

    def \_\_len\_\_(self):

        self.f\_len = sum([1 for i in self.f\_set])

        return self.f\_len

    def \_\_str\_\_(self):

        return f'{[x for x in self.f\_set]}'

    def \_\_getitem\_\_(self, item):

        return self.f\_list[item]

    def \_\_iter\_\_(self):

        for i in range(len(self)):

            yield self[i]

# testing

a = FuzzySet({('x1', 0.5), ('x2', 0.7), ('x3', 0.0)})

b = FuzzySet({('x1', 0.8), ('x2', 0.2), ('x3', 1.0)})

c = FuzzySet({('x', 0.3), ('y', 0.3), ('z', 0.5)})

x = FuzzySet({('a', 0.5), ('b', 0.3), ('c', 0.7)})

y = FuzzySet({('a', 0.6), ('b', 0.4)})

print(f'a -> {a}')

print(f'b -> {b}')

print(f"Fuzzy union: \n{a | b}")

print(f'Fuzzy intersection: \n{a & b}')

print(f'Fuzzy inversion of b: \n{~b}')

print(f"Fuzzy inversion of a: \n {~a}")

print(f'Fuzzy Subtraction: \n{a - b}')

r = np.array([[0.6, 0.6, 0.8, 0.9], [0.1, 0.2, 0.9, 0.8], [0.9, 0.3, 0.4,

0.8], [0.9, 0.8, 0.1, 0.2]])

s = np.array([[0.1, 0.2, 0.7, 0.9], [1.0, 1.0, 0.4, 0.6], [0.0, 0.0, 0.5,

0.9], [0.9, 1.0, 0.8, 0.2]])

print(f"Max Min: of \n{r} \nand \n{s}\n:\n\n")

print(FuzzySet.max\_min(r, s))

**Output:**

a -> [('x1', 0.5), ('x3', 0.0), ('x2', 0.7)]

b -> [('x1', 0.8), ('x2', 0.2), ('x3', 1.0)]

Fuzzy union:

[('x1', 0.8), ('x2', 0.2), ('x3', 1.0)]

Fuzzy intersection:

[('x1', 0.5), ('x3', 0.0), ('x2', 0.7)]

Fuzzy inversion of b:

[('x1', 0.2), ('x3', 0.0), ('x2', 0.8)]

Fuzzy inversion of a:

[('x2', 0.3), ('x1', 0.5), ('x3', 1.0)]

Fuzzy Subtraction:

[('x1', 0.2), ('x3', 0.0), ('x2', 0.7)]

Max Min: of

[[0.6 0.6 0.8 0.9]

[0.1 0.2 0.9 0.8]

[0.9 0.3 0.4 0.8]

[0.9 0.8 0.1 0.2]]

and

[[0.1 0.2 0.7 0.9]

[1. 1. 0.4 0.6]

[0. 0. 0.5 0.9]

[0.9 1. 0.8 0.2]]

:

[[0.9 0.9 0.8 0.2]

[0.8 0.8 0.8 0.2]

[0.8 0.8 0.8 0.2]

[0.2 0.2 0.2 0.2]]

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 10**

import numpy

def cal\_pop\_fitness(equation\_inputs, pop):

    fitness = numpy.sum(pop\*equation\_inputs, axis=1)

    return fitness

def select\_mating\_pool(pop, fitness, num\_parents):

 # Selecting the best individuals in the current generation as parents for producing the offspring of the next generation.

    parents = numpy.empty((num\_parents, pop.shape[1]))

    for parent\_num in range(num\_parents):

        max\_fitness\_idx = numpy.where(fitness == numpy.max(fitness))

        max\_fitness\_idx = max\_fitness\_idx[0][0]

        parents[parent\_num, :] = pop[max\_fitness\_idx, :]

        fitness[max\_fitness\_idx] = -99999999999

    return parents

def crossover(parents, offspring\_size):

    offspring = numpy.empty(offspring\_size)

 # The point at which crossover takes place between two parents. Usually it is at the center.

    crossover\_point = numpy.uint8(offspring\_size[1]/2)

    for k in range(offspring\_size[0]):

 # Index of the first parent to mate.

        parent1\_idx = k%parents.shape[0]

 # Index of the second parent to mate.

        parent2\_idx = (k+1)%parents.shape[0]

 # The new offspring will have its first half of its genes taken from the first parent.

        offspring[k, 0:crossover\_point] = parents[parent1\_idx, 0:crossover\_point]

 # The new offspring will have its second half of its genes taken from the second parent.

        offspring[k, crossover\_point:] = parents[parent2\_idx, crossover\_point:]

    return offspring

def mutation(offspring\_crossover):

 # Mutation changes a single gene in each offspring randomly.

    for idx in range(offspring\_crossover.shape[0]):

 # The random value to be added to the gene.

        random\_value = numpy.random.uniform(-1.0, 1.0, 1)

        offspring\_crossover[idx, 4] = offspring\_crossover[idx, 4] + random\_value

    return offspring\_crossover

equation\_inputs = [4,-2]

# Number of the weights we are looking to optimize.

num\_weights = 2

sol\_per\_pop = 4

num\_parents\_mating = 4

# Defining the population size.

pop\_size = (sol\_per\_pop,num\_weights) # The population will have sol\_per\_pop chromosome where each chromosome has num\_weights genes.

#Creating the initial population.

new\_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop\_size)

print(new\_population)

num\_generations = 5

for generation in range(num\_generations):

    print("Generation : ", generation)

 # Measing the fitness of each chromosome in the population.

    fitness = cal\_pop\_fitness(equation\_inputs, new\_population)

 # Selecting the best parents in the population for mating.

    parents = select\_mating\_pool(new\_population, fitness,

 num\_parents\_mating)

 # Generating next generation using crossover.

    offspring\_crossover = crossover(parents,

    offspring\_size=(pop\_size[0]-parents.shape[0], num\_weights))

 # Adding some variations to the offsrping using mutation.

    offspring\_mutation = mutation(offspring\_crossover)

 # Creating the new population based on the parents and offspring.

    new\_population[0:parents.shape[0], :] = parents

    new\_population[parents.shape[0]:, :] = offspring\_mutation

 # The best result in the current iteration.

    print("Best result : ", numpy.max(numpy.sum(new\_population\*equation\_inputs, axis=1)))

# Getting the best solution after iterating finishing all generations.

#At first, the fitness is calculated for each solution in the final generation.

fitness = cal\_pop\_fitness(equation\_inputs, new\_population)

# Then return the index of that solution corresponding to the best fitness.

best\_match\_idx = numpy.where(fitness == numpy.max(fitness))

print("Best solution : ", new\_population[best\_match\_idx, :])

print("Best solution fitness : ", fitness[best\_match\_idx])

**OutPut:**

**[[ 2.54539731 1.92891889]**

**[-3.35896145 -0.30382703]**

**[-1.23453366 0.18048864]**

**[ 1.0595559 3.19410301]]**

**Generation : 0**

**Best result : 6.3237514516222735**

**Generation : 1**

**Best result : 6.3237514516222735**

**Generation : 2**

**Best result : 6.3237514516222735**

**Generation : 3**

**Best result : 6.3237514516222735**

**Generation : 4**

**Best result : 6.3237514516222735**

**Best solution : [[[2.54539731 1.92891889]]]**

**Best solution fitness : [6.32375145]**

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 11**

import random

import math

def objective(m):

    # Objective Function

    return abs((m[0] \* m[0]))

def fitness(m):

    # Fitness Function

    return 1 / (1 + m)

def probability(m, n):

    # Probability Function

    return m / n

def crossover(m, n):

    # Crossover

    pt = random.randint(0, 3)

    return m[:pt+1] + n[pt+1:]

print("Minimize x^2")

pop1 = []

# Initialize random population

for i in range(6):

    pop1.append([random.randint(0, 30) for j in range(4)])

print("Initial Population: ")

for i in range(6):

    print(pop1[i])

# Maximum iterations 10

for it in range(10):

    obj = [0] \* 6

    print("Objective Function: ")

    for i in range(6):

        obj[i] = objective(pop1[i])

        print(obj[i])

    fit = [0] \* 6

    print("Fitness Value: ")

    for i in range(6):

        fit[i] = fitness(obj[i])

        print(fit[i])

    print("Total fitness: ", sum(fit))

    prob = [0] \* 6

    # Probability calculation

    for i in range(6):

        prob[i] = probability(fit[i], sum(fit))

    # Cumulative Probability Calculation

    cmp = [0] \* 6

    sum1 = 0

    for i in range(6):

        sum1 += prob[i]

        cmp[i] = sum1

    # Roulette Wheel Selection

    R = [random.random() for i in range(6)]

    new\_pop = []

    for i in range(6):

        for j in range(6):

            if R[i] <= cmp[j]:

                new\_pop.append(pop1[j])

                break

    print("After Selection: ", new\_pop)

    # Crossover

    cr = 0.25  # Crossover Rate

    CR = [random.random() for i in range(6)]

    par = []

    par\_index = []

    for i in range(6):

        if CR[i] < cr:

            par.append(new\_pop[i])

            par\_index.append(i)

    x = len(par)

    for i in range(x):

        a = random.randint(0, x - 1)

        b = random.randint(0, x - 1)

        new\_pop[par\_index[i]] = crossover(par[a], par[b])

    # Mutation

    mr = 0.1  # Mutation Rate

    total\_mut = math.floor(24 \* mr)

    mut\_index = [random.randint(0, 23) for i in range(total\_mut)]

    mut\_value = [random.randint(0, 30) for i in range(total\_mut)]

    for i in range(total\_mut):

        cr\_num = mut\_index[i] // 4

        gen\_num = mut\_index[i] % 4

        new\_pop[cr\_num][-(gen\_num + 1)] = mut\_value[i]

    print("After iteration: ", it)

    print(new\_pop)

    print(obj)

    pop1 = new\_pop

**Output:**

After Selection: [[7, 9, 17, 11], [3, 9, 17, 15], [3, 9, 17, 15], [3, 9, 17, 15], [3, 9, 17, 15], [7, 9, 17, 11]]

After iteration: 8

[[7, 9, 17, 11], [3, 9, 17, 15], [3, 27, 17, 15], [3, 27, 17, 15], [3, 9, 17, 15], [7, 9, 17, 11]]

[49, 49, 49, 9, 49, 9]

Objective Function:

49

9

9

9

9

49

Fitness Value:

0.02

0.1

0.1

0.1

0.1

0.02

Total fitness: 0.44000000000000006

After Selection: [[3, 27, 17, 15], [3, 27, 17, 15], [7, 9, 17, 11], [3, 27, 17, 15], [3, 27, 17, 15], [3, 27, 17, 15]]

After iteration: 9

[[3, 27, 17, 26], [3, 27, 17, 26], [7, 9, 17, 11], [3, 27, 17, 26], [3, 27, 17, 26], [3, 27, 17, 26]]

[49, 9, 9, 9, 9, 49]

**Name: Shreya Mahindrakar**

**Roll no:30**

**Batch : B2**

**Experiment 12**

from collections import defaultdict

from functools import partial

import matplotlib.pyplot as plt

import matplotlib.animation as animation

import random

import math

def create\_point():

    return random.random(), random.random()

def distance(orig, dest):

    return math.sqrt((dest[0] - orig[0]) \*\* 2 + (dest[1] - orig[1]) \*\* 2)

def compute\_distance\_matrix(point\_set):

    distance\_matrix = defaultdict(dict)

    for orig in point\_set:

        for dest in point\_set:

            distance\_matrix[orig][dest] = distance\_matrix[dest][orig] = distance(orig, dest)

    return distance\_matrix

def compute\_solution\_distance(solution, distance\_matrix):

    total\_distance = 0

    for i in range(len(solution) - 1):

        total\_distance += distance\_matrix[solution[i]][solution[i + 1]]

    return total\_distance

def create\_individual(point\_set):

    points = list(point\_set)

    random.shuffle(points)

    return points

def create\_population(n\_individuals, point\_set, distance\_matrix):

    individuals = [create\_individual(point\_set) for \_ in range(n\_individuals)]

    distances = list(map(partial(compute\_solution\_distance, distance\_matrix=distance\_matrix), individuals))

    return sorted(zip(individuals, distances), key=lambda x: x[1])

def plot\_result(solution):

    xs = [point[0] for point in solution]

    ys = [point[1] for point in solution]

    plt.plot(xs, ys)

    plt.axis('off')

    plt.show()

def plot\_point\_set(point\_set):

    point\_list = list(point\_set)

    xs = [point[0] for point in point\_list]

    ys = [point[1] for point in point\_list]

    plt.scatter(xs, ys)

    plt.axis('off')

    plt.show()

def mutate(individual, distance\_matrix):

    def mutation\_swap():

        swap\_idx = random.randint(0, len(individual) - 2)

        new\_individual = individual[:swap\_idx] + [individual[swap\_idx + 1], individual[swap\_idx]] + individual[swap\_idx + 2:]

        return new\_individual

    def mutation\_reverse():

        reverse\_start = random.randint(0, len(individual) - 2)

        reverse\_end = random.randint(reverse\_start + 1, len(individual) - 1)

        new\_individual = individual[:reverse\_start] + individual[reverse\_start:reverse\_end][::-1] + individual[reverse\_end:]

        return new\_individual

    mutation = random.choice([mutation\_swap, mutation\_reverse])

    new\_individual = mutation()

    return new\_individual, compute\_solution\_distance(new\_individual, distance\_matrix)

def reproduce(individual\_1, individual\_2, distance\_matrix):

    def generate\_subset\_idx(subset\_size):

        return sorted(random.sample(range(ind\_size), subset\_size))

    def select\_subset(individual, subset\_idx):

        return [individual[i] for i in subset\_idx]

    def complement\_subset(individual\_2, individual\_1\_subset):

        s = set(individual\_1\_subset)

        return [point for point in individual\_2 if point not in s]

    ind\_size = len(individual\_1)

    ind\_1\_subset\_size = ind\_size // 2

    subset\_ind\_1\_idx = generate\_subset\_idx(ind\_1\_subset\_size)

    ind\_1\_subset = select\_subset(individual\_1, subset\_ind\_1\_idx)

    ind\_2\_subset = complement\_subset(individual\_2, ind\_1\_subset)

    new\_individual = ind\_1\_subset + ind\_2\_subset

    return new\_individual, compute\_solution\_distance(new\_individual,distance\_matrix)

def evolve(population, n\_reproductions, n\_mutations, n\_news,reproductor\_pool, distance\_matrix, point\_set):

        population\_size = len(population)

        n\_new\_individuals = n\_reproductions + n\_mutations + n\_news

        n\_survivors = population\_size - n\_new\_individuals

        reproductor\_pool\_size = round(reproductor\_pool \* population\_size)

        new\_population = population[:n\_survivors]

        for \_ in range(n\_reproductions):

            individual\_1 = population[random.randint(0, reproductor\_pool\_size - 1)][0]

            individual\_2 = random.choice(population)[0]

            new\_population.append(reproduce(individual\_1, individual\_2, distance\_matrix))

        for \_ in range(n\_mutations):

            individual\_to\_mutate = random.choice(population)[0]

            new\_population.append(mutate(individual\_to\_mutate, distance\_matrix))

        for \_ in range(n\_news):

            new\_individual = create\_individual(point\_set)

            new\_population.append((new\_individual,

            compute\_solution\_distance(new\_individual, distance\_matrix)))

        return sorted(new\_population, key = lambda x: x[1])

def genetic\_algorithm(point\_set, population\_size, n\_generations,n\_reproductions, n\_mutations, n\_news, reproduction\_pool):

    distance\_matrix = compute\_distance\_matrix(point\_set)

    population = create\_population(population\_size, point\_set, distance\_matrix)

    for i in range(n\_generations):

        population = evolve(population, n\_reproductions, n\_mutations,

        n\_news, reproduction\_pool, distance\_matrix, point\_set)

    return population[0]

point\_set = {create\_point() for \_ in range(20)}

plot\_point\_set(point\_set)

best\_solution, length = genetic\_algorithm(point\_set, 300, 120, 100, 50, 0, 0.15)

plot\_result(best\_solution)

**Output**:

