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

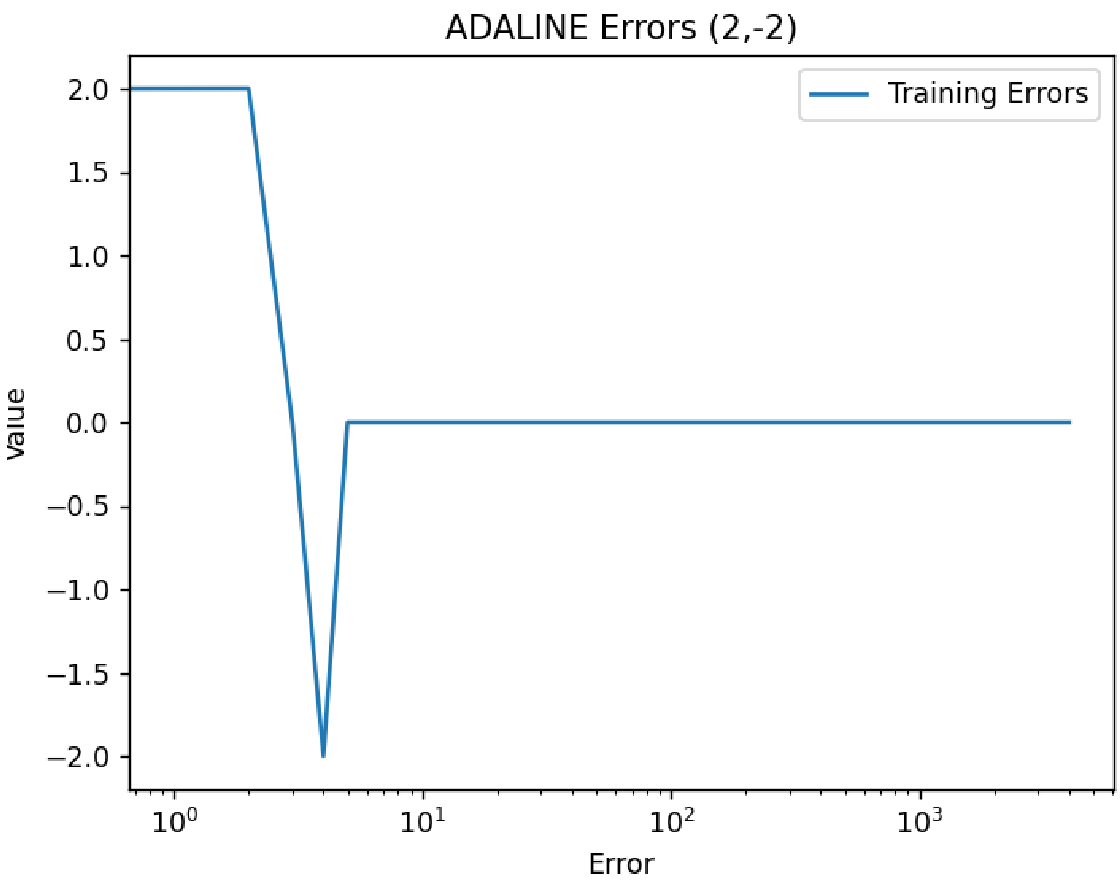
Random Weights [[1] [1] [1]] after training

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):
```

```python
        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)})
```

```python
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

```python
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):
```

```python
    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])
```

```python
    # 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])
```

```python
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: