**Practical 1 a: Design a simple linear neural network model**

x=float(input("Enter value of x:"))

w=float(input("Enter value of weight w:"))

b=float(input("Enter value of bias b:"))

net = int(w\*x+b)

if(net<0):

    out=0

elif((net>=0)&(net<=1)):

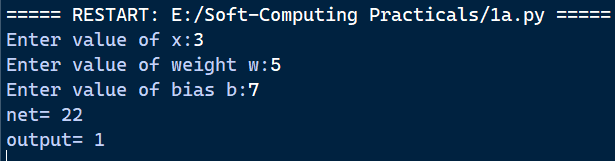
    out =net

else:

    out=1

print("net=",net)

print("output=",out)

**Output:**

**Practical 1 b: Calculate the output of neural net using both binary and bipolar sigmoidal function**

n = int(input("Enter number of elements : "))

print("Enter the inputs")

inputs = []

for i in range(0, n):

    ele = float(input())

    inputs.append(ele)

print(inputs)

print("Enter the weights")

weights = []

for i in range(0, n):

    ele = float(input())

    weights.append(ele)

print(weights)

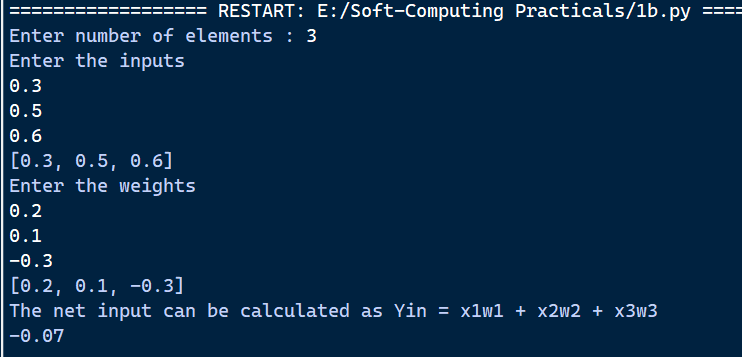
print("The net input can be calculated as Yin = x1w1 + x2w2 + x3w3")

Yin = []

for i in range(0, n):

    Yin.append(inputs[i] \* weights[i])

print(round(sum(Yin), 3))

**Output:**

**Practical 2 a: Implement AND/NOT function using McCulloch-Pits neuron (use binary data representation).**

num\_ip = int(input("Enter the number of inputs : "))

w1 = 1

w2 = 1

print("For the ", num\_ip, " inputs calculate the net input using yin = x1w1 + x2w2 ")

x1 = []

x2 = []

for j in range(0, num\_ip):

    ele1 = int(input("x1 = "))

    ele2 = int(input("x2 = "))

    x1.append(ele1)

    x2.append(ele2)

print("x1 = ", x1)

print("x2 = ", x2)

n = x1 \* w1

m = x2 \* w2

Yin = []

for i in range(0, num\_ip):

    Yin.append(n[i] + m[i])

print("Yin = ", Yin)

Yin = []

for i in range(0, num\_ip):

    Yin.append(m[i] - n[i])

print("After assuming one weight as excitatory and the other as inhibitory Yin = ", Yin)

Y = []

for i in range(0, num\_ip):

    if Yin[i] >= 1:

        ele = 1

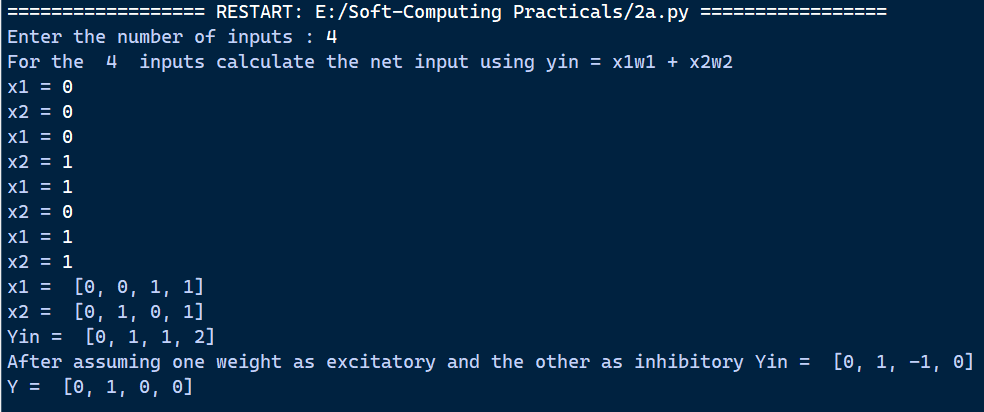
        Y.append(ele)

    if Yin[i] < 1:

        ele = 0

        Y.append(ele)

print("Y = ", Y)

**Output:**

**Practical 2 b: Generate XOR function using McCulloch-Pitts neural net**

import numpy as np

print('Enter weights')

w11 = int(input('Weight w11 = '))

w12 = int(input('Weight w12 = '))

w21 = int(input('Weight w21 = '))

w22 = int(input('Weight w22 = '))

v1 = int(input('Weight v1 = '))

v2 = int(input('Weight v2 = '))

print('Enter Threshold Value')

theta = int(input('Theta = '))

x1 = np.array([0, 0, 1, 1])

x2 = np.array([0, 1, 0, 1])

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

con = 1

y1 = np.zeros((4,))

y2 = np.zeros((4,))

y = np.zeros((4,))

while con == 1:

    zin1 = np.zeros((4,))

    zin2 = np.zeros((4,))

    zin1 = x1 \* w11 + x2 \* w21

    zin2 = x1 \* w12 + x2 \* w22

    print("z1", zin1)

    print("z2", zin2)

    for i in range(0, 4):

        if zin1[i] >= theta:

            y1[i] = 1

        else:

            y1[i] = 0

        if zin2[i] >= theta:

            y2[i] = 1

        else:

            y2[i] = 0

    yin = np.array([])

    yin = y1 \* v1 + y2 \* v2

    for i in range(0, 4):

        if yin[i] >= theta:

            y[i] = 1

        else:

            y[i] = 0

    print("yin", yin)

    print('Output of Net')

    y = y.astype(int)

    print("y", y)

    print("z", z)

    if np.array\_equal(y, z):

        con = 0

    else:

        print("Net is not learning. Enter another set of weights and Threshold value")

        w11 = int(input("Weight w11 = "))

        w12 = int(input("Weight w12 = "))

        w21 = int(input("Weight w21 = "))

        w22 = int(input("Weight w22 = "))

        v1 = int(input("Weight v1 = "))

        v2 = int(input("Weight v2 = "))

        theta = int(input("Theta = "))

print("McCulloch-Pitts Net for XOR function")

print("Weights of Neuron Z1")

print(w11,w12)

print("Weights of Neuron Z2")

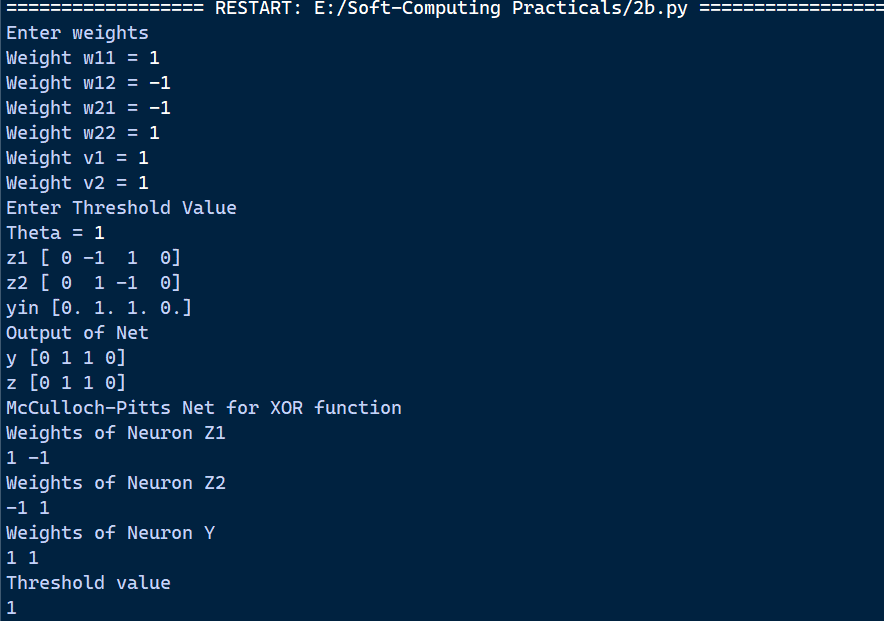
print(w12,w22)

print("Weights of Neuron Y")

print(v1,v2)

print("Threshold value")

print(theta)

**Output:**

**Practical 3 a: Write a program to implement Hebb’s rule**

import numpy as np

x1 = np.array([1, 1, 1, -1, 1, -1, 1, 1, 1])

x2 = np.array([1, 1, 1, 1, -1, 1, 1, 1, 1])

b = 0

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

wtold = np.zeros((9,))

wtnew = np.zeros((9,))

wtnew = wtnew.astype(int)

wtold = wtold.astype(int)

bais = 0

print("First input with target = 1")

for i in range(0, 9):

    wtold[i] = wtold[i] + x1[i] \* y[0]

wtnew = wtold

b = b + y[0]

print("New weights =", wtnew)

print("Bias value =", b)

print("Second input with target = -1")

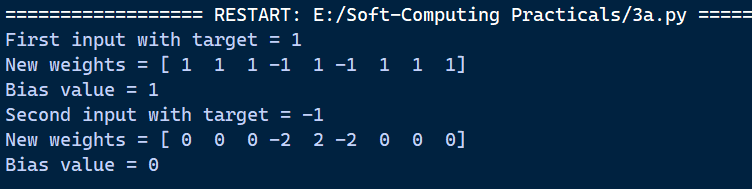
for i in range(0, 9):

    wtnew[i] = wtold[i] + x2[i] \* y[1]

b = b + y[1]

print("New weights =", wtnew)

print("Bias value =", b)

**Output:**

**Practical 3 b: Write a program to implement of delta rule**

import numpy as np

import time

np.set\_printoptions(precision=2)

x = np.zeros((3,))

weights = np.zeros((3,))

desired = np.zeros((3,))

actual = np.zeros((3,))

for i in range(0, 3):

    x[i] = float(input("Initial inputs:"))

for i in range(0, 3):

    weights[i] = float(input("Initial weights:"))

for i in range(0, 3):

    desired[i] = float(input("Desired output:"))

a = float(input("Enter learning rate:"))

actual = x \* weights

print("actual", actual)

print("desired", desired)

while True:

    if np.array\_equal(desired, actual):

        break

    else:

        for i in range(0, 3):

            weights[i] = weights[i] + a \* (desired[i] - actual[i])

        actual = x \* weights

        print("weights", weights)

        print("actual", actual)

        print("desired", desired)

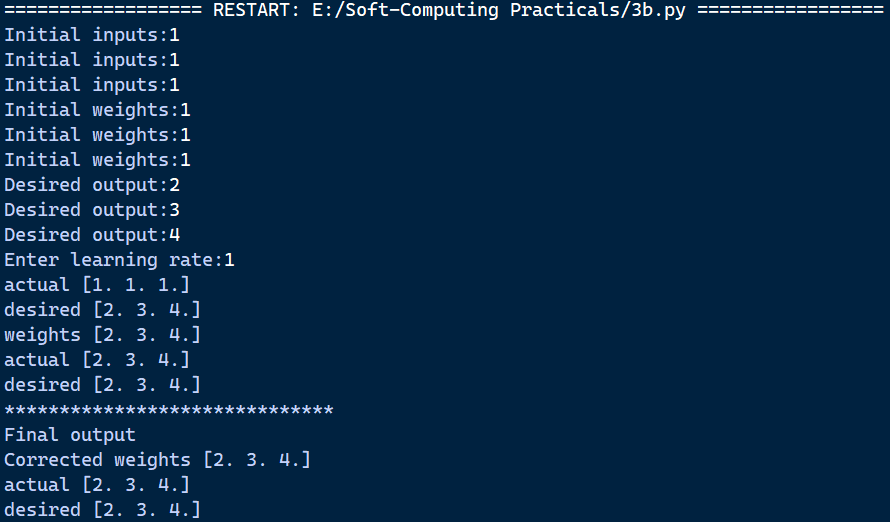
        print("\*" \* 30)

print("Final output")

print("Corrected weights", weights)

print("actual", actual)

print("desired", desired)

**Output:**

**Practical 4 a: Write a program for Back Propagation Algorithm**

import numpy as np

import decimal

import math

np.set\_printoptions(precision=2)

v1 = np.array([0.6, 0.3])

v2 = np.array([-0.1, 0.4])

w = np.array([-0.2, 0.4, 0.1])

b1 = 0.3

b2 = 0.5

x1 = 0

x2 = 1

alpha = 0.25

print("calculate net input to z1 layer")

zin1 = round(b1 + x1 \* v1[0] + x2 \* v2[0], 4)

print("z1=", round(zin1, 3))

print("calculate net input to z2 layer")

zin2 = round(b2 + x1 \* v1[1] + x2 \* v2[1], 4)

print("z2=", round(zin2, 4))

print("Apply activation function to calculate output")

z1 = 1 / (1 + math.exp(-zin1))

z1 = round(z1, 4)

z2 = 1 / (1 + math.exp(-zin2))

z2 = round(z2, 4)

print("z1=", z1)

print("z2=", z2)

print("calculate net input to output layer")

yin = w[0] + z1 \* w[1] + z2 \* w[2]

print("yin=", yin)

print("calculate net output")

y = 1 / (1 + math.exp(-yin))

print("y=", y)

fyin = y \* (1 - y)

dk = (1 - y) \* fyin

print("dk", dk)

dw1 = alpha \* dk \* z1

dw2 = alpha \* dk \* z2

dw0 = alpha \* dk

print("compute error portion in delta")

din1 = dk \* w[1]

din2 = dk \* w[2]

print("din1=", din1)

print("din2=", din2)

print("error in delta")

fzin1 = z1 \* (1 - z1)

print("fzin1", fzin1)

d1 = din1 \* fzin1

fzin2 = z2 \* (1 - z2)

print("fzin2", fzin2)

d2 = din2 \* fzin2

print("d1=", d1)

print("d2=", d2)

print("Changes in weights between input and hidden layer")

dv11 = alpha \* d1 \* x1

print("dv11=", dv11)

dv21 = alpha \* d1 \* x2

print("dv21=", dv21)

dv01 = alpha \* d1

print("dv01=", dv01)

dv12 = alpha \* d2 \* x1

print("dv12=", dv12)

dv22 = alpha \* d2 \* x2

print("dv22=", dv22)

dv02 = alpha \* d2

print("dv02=", dv02)

print("Final weights of network")

v1[0] = v1[0] + dv11

v1[1] = v1[1] + dv12

print("v=", v1)

v2[0] = v2[0] + dv21

v2[1] = v2[1] + dv22

print("v2", v2)

w[1] = w[1] + dw1

w[2] = w[2] + dw2

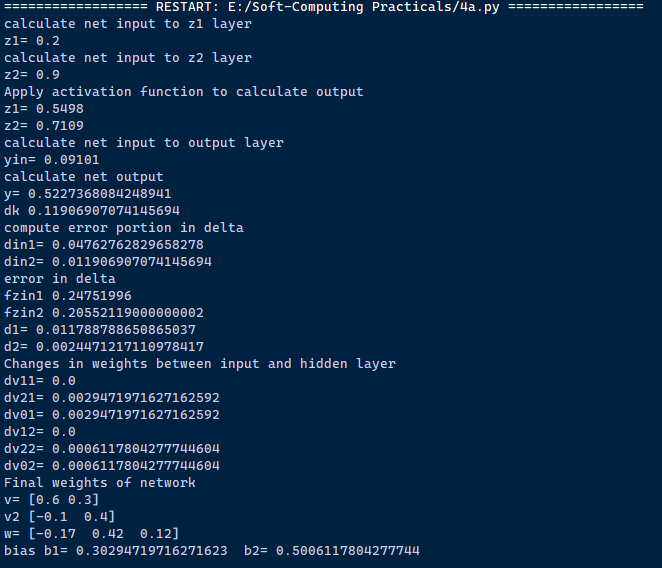
b1 = b1 + dv01

b2 = b2 + dv02

w[0] = w[0] + dw0

print("w=", w)

print("bias b1=", b1, " b2=", b2)

**Output:**

**Practical 4 b: Write a Program for Error Back Propagation Algorithm (Ebpa) Learning**

import math

a0=-1

t=-1

w10=float(input("Enter weight first network : "))

b10=float(input("Enter base first network : "))

w20=float(input("Enter weight second network : "))

b20=float(input("Enter base second network : "))

c=float(input("Enter learning coefficient : "))

n1=float(w10\*c+b10)

a1=math.tanh(n1)

n2=float(w20\*a1+b20)

a2=math.tanh(float(n2))

e=t-a2

s2=-2\*(1-a2\*a2)\*e

s1=(1-a1\*a1)\*w20\*s2

w21=w20-(c\*s2\*a1)

w11=w10-(c\*s1\*a0)

b21=b20-(c\*s2)

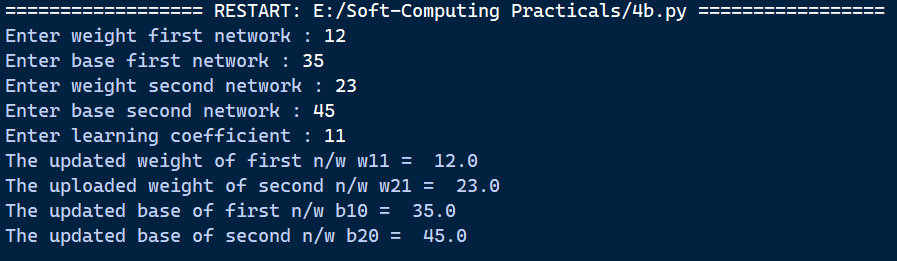
b11=b10-(c\*s1)

print("The updated weight of first n/w w11 = ",w11)

print("The uploaded weight of second n/w w21 = ",w21)

print("The updated base of first n/w b10 = ",b10)

print("The updated base of second n/w b20 = ",b20)

**Output:**

**Practical 5 b: Write a program for Radial Basis function**

from scipy import \*

from scipy.linalg import norm, pinv

from matplotlib import pyplot as plt

import numpy as np

class RBF:

    def \_\_init\_\_(self, indim, numCenters, outdim):

        self.indim = indim

        self.outdim = outdim

        self.numCenters = numCenters

        self.centers = [np.random.uniform(-1, 1, indim) for i in range(numCenters)]

        self.beta = 8

        self.W = np.random.random((self.numCenters, self.outdim))

    def \_basisfunc(self, c, d):

        assert len(d) == self.indim

        return np.exp(-self.beta \* norm(c - d)\*\*2)

    def \_calcAct(self, X):

        G = np.zeros((X.shape[0], self.numCenters), float)

        for ci, c in enumerate(self.centers):

            for xi, x in enumerate(X):

                G[xi, ci] = self.\_basisfunc(c, x)

        return G

    def train(self, X, Y):

        rnd\_idx = np.random.permutation(X.shape[0])[:self.numCenters]

        self.centers = [X[i, :] for i in rnd\_idx]

        G = self.\_calcAct(X)

        self.W = np.dot(pinv(G), Y)

    def test(self, X):

        G = self.\_calcAct(X)

        Y = np.dot(G, self.W)

        return Y

if \_\_name\_\_ == '\_\_main\_\_':

    n = 100

    x = np.mgrid[-1:1:complex(0, n)].reshape(n, 1)

    y = np.sin(3 \* (x + 0.5)\*\*3 - 1)

    rbf = RBF(1, 10, 1)

    rbf.train(x, y)

    z = rbf.test(x)

    plt.figure(figsize=(12, 8))

    plt.plot(x, y, 'k-')

    plt.plot(x, z, 'r-', linewidth=2)

    plt.plot(rbf.centers, np.zeros(rbf.numCenters), 'gs')

    for c in rbf.centers:

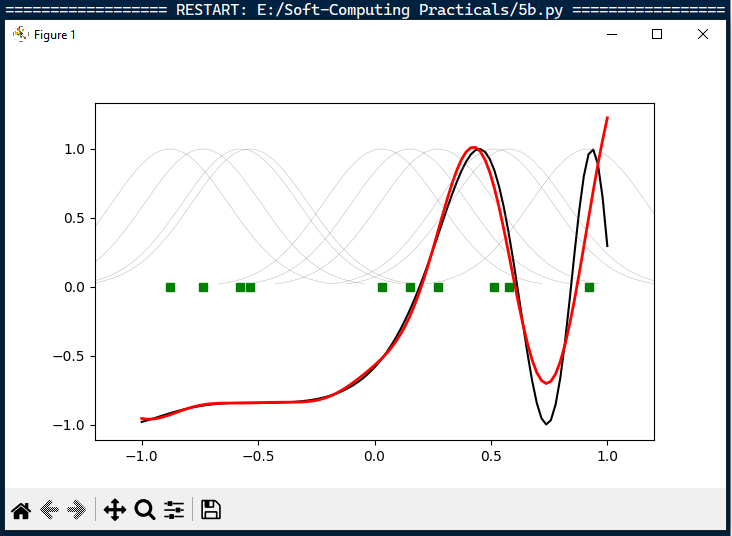
        cx = np.arange(c - 0.7, c + 0.7, 0.01)

        cy = [rbf.\_basisfunc(np.array([cx\_]), np.array([c])) for cx\_ in cx]

        plt.plot(cx, cy, '-', color='gray', linewidth=0.2)

    plt.xlim(-1.2, 1.2)

    plt.show()

**Output:**

**Practical 6 a: Self-Organizing Maps**

import numpy as np

import matplotlib.pyplot as plt

from minisom import MiniSom

colors = np.array(

    [[0., 0., 0.],

     [0., 0., 1.],

     [0., 0., 0.5],

     [0.125, 0.529, 1.0],

     [0.33, 0.4, 0.67],

     [0.6, 0.5, 1.0],

     [0., 1., 0.],

     [1., 0., 0.],

     [0., 1., 1.],

     [1., 0., 1.],

     [1., 1., 0.],

     [1., 1., 1.],

     [0.33, 0.33, 0.33],

     [0.5, 0.5, 0.5],

     [0.66, 0.66, 0.66]]

)

color\_names = [

    'black', 'blue', 'darkblue', 'skyblue',

    'greyblue', 'lilac', 'green', 'red',

    'cyan', 'violet', 'yellow', 'white',

    'darkgrey', 'mediumgrey', 'lightgrey'

]

som = MiniSom(30, 20, 3, sigma=1.0, learning\_rate=0.05)

som.train(colors, 400)

plt.imshow(som.distance\_map().T, origin='lower')

plt.title('Color SOM')

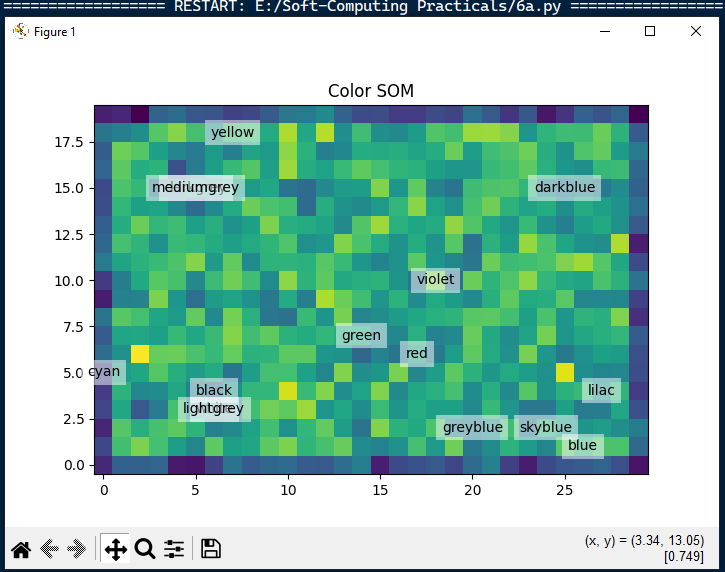
for i, color in enumerate(colors):

    w = som.winner(color)

    plt.text(w[0], w[1], color\_names[i], ha='center', va='center',

             bbox=dict(facecolor='white', alpha=0.5, lw=0))

plt.show()

**Output:**

**Practical 7 a: Line Separation**

import numpy as np

import matplotlib.pyplot as plt

def create\_distance\_function(a, b, c):

    def distance(x, y):

        nom = a \* x + b \* y + c

        if nom == 0:

            pos = 0

        elif (nom < 0 and b < 0) or (nom > 0 and b > 0):

            pos = -1

        else:

            pos = 1

        return (np.absolute(nom) / np.sqrt(a \*\* 2 + b \*\* 2), pos)

    return distance

points = [(3.5, 1.8), (1.1, 3.9)]

fig, ax = plt.subplots()

ax.set\_xlabel("sweetness")

ax.set\_ylabel("sourness")

ax.set\_xlim([-1, 6])

ax.set\_ylim([-1, 8])

X = np.arange(-0.5, 5, 0.1)

size = 10

for index, (x, y) in enumerate(points):

    if index == 0:

        ax.plot(x, y, "o", color="darkorange", markersize=size)

    else:

        ax.plot(x, y, "o", color="yellow", markersize=size)

step = 0.05

for x in np.arange(0, 1 + step, step):

    slope = np.tan(np.arccos(x))

    dist4line1 = create\_distance\_function(slope, -1, 0)

    Y = slope \* X

    results = [dist4line1(\*point) for point in points]

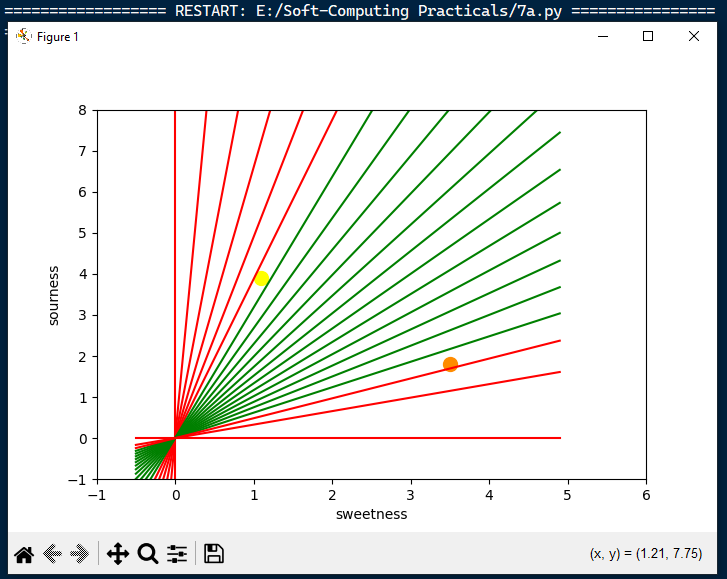
    if results[0][1] != results[1][1]:

        ax.plot(X, Y, "g-")

    else:

        ax.plot(X, Y, "r-")

plt.show()

**Output:**

**Practical 8 a: Membership and Identity operators in, not in**

def overlapping(list1, list2):

    c = 0

    d = 0

    for i in list1:

        c += 1

    for i in list2:

        d += 1

    for i in range(0, c):

        for j in range(0, d):

            if list1[i] == list2[j]:

                return 1

    return 0

# First case: no overlapping elements

list1 = [1, 2, 3, 4, 5]

list2 = [6, 7, 8, 9]

if overlapping(list1, list2):

    print("overlapping")

else:

    print("not overlapping")

# Second case: overlapping elements

list1 = [1, 2, 3, 4, 5]

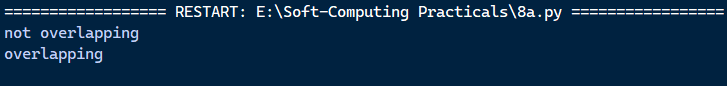
list2 = [4, 6, 7, 8, 9]

if overlapping(list1, list2):

    print("overlapping")

else:

    print("not overlapping")

**Output: **

**Practical 8 b: Membership and Identity Operators is, is not**

x = 5

if (type(x) is int):

    print("true")

else:

    print("false")

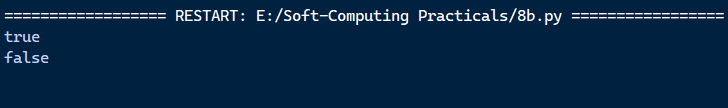
x = 5

if (type(x) is not int):

    print("true")

else:

    print("false")

**Output:**

**Practical 9 a: Find the ratios using fuzzy logic**

from fuzzywuzzy import fuzz

from fuzzywuzzy import process

s1 = "I love fuzzysforfuzzys"

s2 = "I am loving fuzzysforfuzzys"

print("FuzzyWuzzy Ratio:", fuzz.ratio(s1, s2))

print("FuzzyWuzzy Partial Ratio:", fuzz.partial\_ratio(s1, s2))

print("FuzzyWuzzy Token Sort Ratio:", fuzz.token\_sort\_ratio(s1, s2))

print("FuzzyWuzzy Token Set Ratio:", fuzz.token\_set\_ratio(s1, s2))

print("FuzzyWuzzy W Ratio:", fuzz.WRatio(s1, s2), '\n\n')

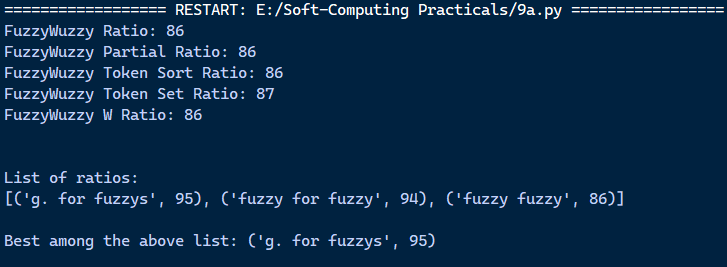
query = 'fuzzys for fuzzys'

choices = ['fuzzy for fuzzy', 'fuzzy fuzzy', 'g. for fuzzys']

print("List of ratios:")

print(process.extract(query, choices), '\n')

print("Best among the above list:", process.extractOne(query, choices))

**Output:**

**Practical 9 b: Solve Tipping Problem using fuzzy logic**

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership functions

quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')

service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')

tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3)

quality.automf(3)

service.automf(3)

# Custom membership functions for the 'tip' variable

tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])

tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])

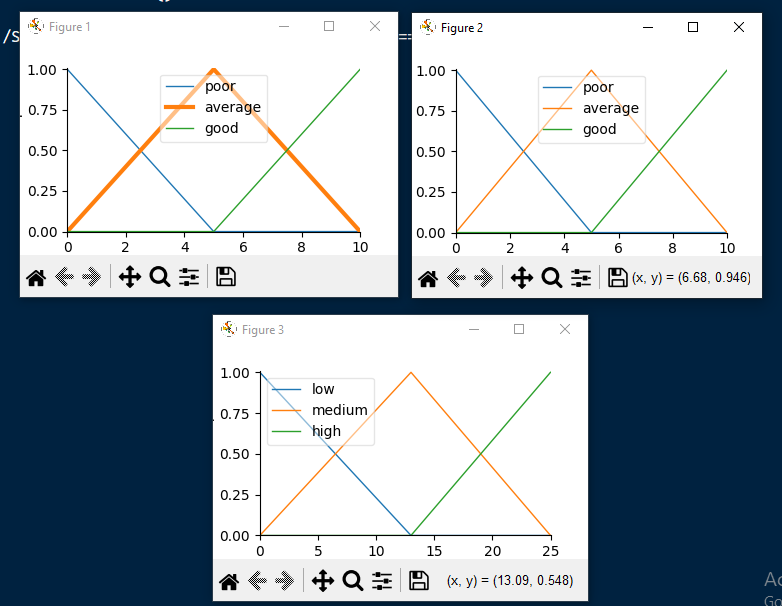
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

# Visualizing the membership functions for quality, service, and tip

quality['average'].view()

service.view()

tip.view()

**Output:**

**Practical 10 a: Implementation of simple genetic algorithm**

import random

POPULATION\_SIZE = 100

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890, .-;:\_!"#%&/()=?@${[]}'''

TARGET = "I love GeeksforGeeks"

class Individual(object):

    def \_\_init\_\_(self, chromosome):

        self.chromosome = chromosome

        self.fitness = self.cal\_fitness()

    @classmethod

    def mutated\_genes(self):

        global GENES

        gene = random.choice(GENES)

        return gene

    @classmethod

    def create\_gnome(self):

        global TARGET

        gnome\_len = len(TARGET)

        return [self.mutated\_genes() for \_ in range(gnome\_len)]

    def mate(self, par2):

        child\_chromosome = []

        for gp1, gp2 in zip(self.chromosome, par2.chromosome):

            prob = random.random()

            if prob < 0.45:

                child\_chromosome.append(gp1)

            elif prob < 0.90:

                child\_chromosome.append(gp2)

            else:

                child\_chromosome.append(self.mutated\_genes())

        return Individual(child\_chromosome)

    def cal\_fitness(self):

        global TARGET

        fitness = 0

        for gs, gt in zip(self.chromosome, TARGET):

            if gs != gt:

                fitness += 1

        return fitness

def main():

    global POPULATION\_SIZE

    generation = 1

    found = False

    population = []

    # Initializing population

    for \_ in range(POPULATION\_SIZE):

        gnome = Individual.create\_gnome()

        population.append(Individual(gnome))

    while not found:

        population = sorted(population, key=lambda x: x.fitness)

        if population[0].fitness <= 0:

            found = True

            break

        new\_generation = []

        s = int((10 \* POPULATION\_SIZE) / 100)

        new\_generation.extend(population[:s])

        s = int((90 \* POPULATION\_SIZE) / 100)

        for \_ in range(s):

            parent1 = random.choice(population[:50])

            parent2 = random.choice(population[:50])

            child = parent1.mate(parent2)

            new\_generation.append(child)

        population = new\_generation

        # Print every 10 generations

        if generation % 10 == 0:

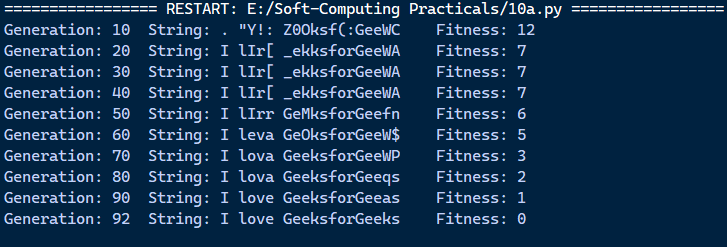
            print(f"Generation: {generation}\tString: {''.join(population[0].chromosome)}\tFitness: {population[0].fitness}")

        generation += 1

    print(f"Generation: {generation}\tString: {''.join(population[0].chromosome)}\tFitness: {population[0].fitness}")

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Output:**

**Practical 10 b: Create two classes: City and Fitness using Genetic algorithm**

import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt

from tkinter import Tk, Canvas, Frame, BOTH, Text

import math

class City:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def distance(self, city):

        xDis = abs(self.x - city.x)

        yDis = abs(self.y - city.y)

        distance = np.sqrt((xDis \*\* 2) + (yDis \*\* 2))

        return distance

    def \_\_repr\_\_(self):

        return "(" + str(self.x) + "," + str(self.y) + ")"

class Fitness:

    def \_\_init\_\_(self, route):

        self.route = route

        self.distance = 0

        self.fitness = 0.0

    def routeDistance(self):

        if self.distance == 0:

            pathDistance = 0

            for i in range(0, len(self.route)):

                fromCity = self.route[i]

                toCity = None

                if i + 1 < len(self.route):

                    toCity = self.route[i + 1]

                else:

                    toCity = self.route[0]

                pathDistance += fromCity.distance(toCity)

            self.distance = pathDistance

        return self.distance

    def routeFitness(self):

        if self.fitness == 0:

            self.fitness = 1 / float(self.routeDistance())

        return self.fitness

def createRoute(cityList):

    route = random.sample(cityList, len(cityList))

    return route

def initialPopulation(popSize, cityList):

    population = []

    for i in range(0, popSize):

        population.append(createRoute(cityList))

    return population

def rankRoutes(population):

    fitnessResults = {}

    for i in range(0, len(population)):

        fitnessResults[i] = Fitness(population[i]).routeFitness()

    return sorted(fitnessResults.items(), key=operator.itemgetter(1), reverse=True)

def selection(popRanked, eliteSize):

    selectionResults = []

    df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])

    df['cum\_sum'] = df.Fitness.cumsum()

    df['cum\_perc'] = 100 \* df.cum\_sum / df.Fitness.sum()

    for i in range(0, eliteSize):

        selectionResults.append(popRanked[i][0])

    for i in range(0, len(popRanked) - eliteSize):

        pick = 100 \* random.random()

        for i in range(0, len(popRanked)):

            if pick <= df.iat[i, 3]:

                selectionResults.append(popRanked[i][0])

                break

    return selectionResults

def matingPool(population, selectionResults):

    matingpool = []

    for i in range(0, len(selectionResults)):

        index = selectionResults[i]

        matingpool.append(population[index])

    return matingpool

def breed(parent1, parent2):

    child = []

    childP1 = []

    childP2 = []

    geneA = int(random.random() \* len(parent1))

    geneB = int(random.random() \* len(parent1))

    startGene = min(geneA, geneB)

    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):

        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2

    return child

def breedPopulation(matingpool, eliteSize):

    children = []

    length = len(matingpool) - eliteSize

    pool = random.sample(matingpool, len(matingpool))

    for i in range(0, eliteSize):

        children.append(matingpool[i])

    for i in range(0, length):

        child = breed(pool[i], pool[len(matingpool) - i - 1])

        children.append(child)

    return children

def mutate(individual, mutationRate):

    for swapped in range(len(individual)):

        if(random.random() < mutationRate):

            swapWith = int(random.random() \* len(individual))

            city1 = individual[swapped]

            city2 = individual[swapWith]

            individual[swapped] = city2

            individual[swapWith] = city1

    return individual

def mutatePopulation(population, mutationRate):

    mutatedPop = []

    for ind in range(0, len(population)):

        mutatedInd = mutate(population[ind], mutationRate)

        mutatedPop.append(mutatedInd)

    return mutatedPop

def nextGeneration(currentGen, eliteSize, mutationRate):

    popRanked = rankRoutes(currentGen)

    selectionResults = selection(popRanked, eliteSize)

    matingpool = matingPool(currentGen, selectionResults)

    children = breedPopulation(matingpool, eliteSize)

    nextGeneration = mutatePopulation(children, mutationRate)

    return nextGeneration

def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):

    pop = initialPopulation(popSize, population)

    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):

        pop = nextGeneration(pop, eliteSize, mutationRate)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))

    bestRouteIndex = rankRoutes(pop)[0][0]

    bestRoute = pop[bestRouteIndex]

    return bestRoute

def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations):

    pop = initialPopulation(popSize, population)

    progress = []

    progress.append(1 / rankRoutes(pop)[0][1])

    for i in range(0, generations):

        pop = nextGeneration(pop, eliteSize, mutationRate)

        progress.append(1 / rankRoutes(pop)[0][1])

    plt.plot(progress)

    plt.ylabel('Distance')

    plt.xlabel('Generation')

    plt.show()

def main():

    cityList = []

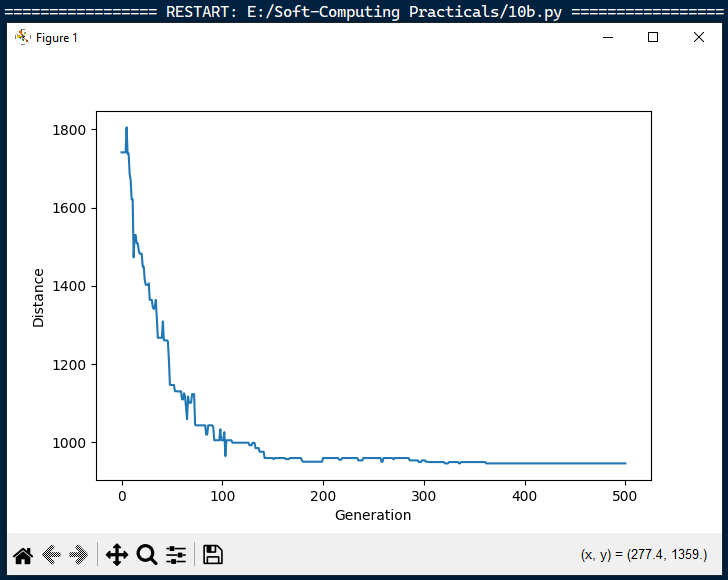
    for i in range(0, 25):

        cityList.append(City(x=int(random.random() \* 200), y=int(random.random() \* 200)))

    geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Output:**