# Practical No: 01

**A. Design a simple neural network model.**

**CODE:**

inputs = float(input("Enter the input: ")) weights = float(input("Enter the weight: ")) bias = float(input("Enter bias: ")) print(' ') yin = bias + (inputs \* weights) if yin < 0: out = 0 elif yin > 1:

out= 1 else:

out = yin print("Output is: ",out) print(' ') print('Ayush Patel : 53004230035')

**B. Calculate the output of neural net. using both binary and bipolar sigmoidal function.**

**CODE:**

import math n = int(input("Enter no. of elements: ")) yin = 0 for i in range(0,n):

x=float(input("x= ")) w=float(input("w= ")) yin = yin + (x\*w) b = float(input("b= ")) yin = yin + b print(' ') print("yin",yin) print(' ') binary\_sigmoidal = (1/(1+(math.e\*\*(-yin)))) print("Binary Sigmoidal= ",round(binary\_sigmoidal,3)) print(' ') bipolar\_sigmoidal = (2/(1+(math.e\*\*(-yin)))) - 1 print("Bipolar Sigmoidal= ",round(bipolar\_sigmoidal,3)) print(' ') print('Ayush Patel : 53004230035')

# Practical No: 02

**A. Generate AND/NOT function using McCulloch-Pitts neural network**.

**CODE:**

print("\nANDNOT function using MP\n") x1inputs = [1,1,0,0] x2inputs = [1,0,1,0] print("Considering all weights as excitatory"); w1 = [1,1,1,1]

w2 = [1,1,1,1] yin = [] print("x1","x2","yin") for i in range(0,4):

yin.append(x1inputs[i]\*w1[i] + x2inputs[i]\*w2[i]) print(x1inputs[i]," ",x2inputs[i]," ", yin[i]) print("\nConsidering all weights as excitatory"); w1 = [1,1,1,1] w2 = [-1,-1,-1,-1] yin = [] print("x1","x2","yin") for i in range(0,4):

yin.append(x1inputs[i]\*w1[i] + x2inputs[i]\*w2[i]) print(x1inputs[i]," ",x2inputs[i]," ", yin[i]) theta = 2\*1-1 print("Threshold -Theta =",theta) print("\nApplying Threshold ") y = [] for i in range(0,4): if(yin[i]>=theta): value = 1

y.append(value) else:

value = 0

y.append(value) print("x1","x2"," y") for i in range(0,4):

print(x1inputs[i]," ",x2inputs[i]," ",y[i]) print('\nAyush : 53004230035')

**B. Generate XOR function using McCulloch-Pitts neural net.**

**CODE:**

print("\nXOR function using McClloch-Pitts") x1inputs = [1,1,0,0] x2inputs = [1,0,1,0] print("Calculating z1 = x1w11 + x2w12") print("Considering one weight as exciatatory and other as inhibitory ") w11 = [1,1,1,1] w21 = [-1,-1,-1,-1] print("x1","x2","z1") z1 = [] for i in range(0,4):

z1.append(x1inputs[i]\*w11[i] + x2inputs[i]\*w21[i]) print(x1inputs[i]," ",x2inputs[i]," ",z1[i]) print("\nCalculating z1 = x1w21 + x2w22") print("Considering one weight as exciatatory and other as inhibitory ") w21 = [-1,-1,-1,-1] w22 = [1,1,1,1] print("x1","x2","z2") z2 = [] for i in range(0,4):

z2.append(x1inputs[i]\*w21[i] + x2inputs[i]\*w22[i]) print(x1inputs[i]," ",x2inputs[i]," ",z2[i]) print("\nApplying Threshold = 1 for z1 and z2") for i in range(0,4): if(z1[i]>=1): z1[i] = 1 else:

z1[i] = 0 if(z2[i]>=1): z2[i]=1 else:

z2[i]=0 print("z1","z2") for i in range(0,4): print(z1[i]," ",z2[i]," ") print("x1" , "x2" , "yin") yin = [] v1 = 1 v2 = 1 for i in range(0,4):

yin.append(z1[i]\*v1 + z2[i]\*v2) print(x1inputs[i]," ",x2inputs[i]," ",yin[i]) y=[] for i in range(0,4): if(yin[i]>=1):

y.append(1) else:

y.append(0) print("x1","x2","y") for i in range(0,4):

print(x1inputs[i]," ",x2inputs[i]," ",y[i]) print('\nAyush Patel : 53004230035')

# Practical No: 03

**A. Write a program to implement Hebb’s rule.**

**CODE:**

import numpy as np x1 = np.array([1,-1,-1,1,-1,-1,1,1,1,1]) x2 = np.array([1,-1,1,1,-1,1,1,1,1,1]) y = np.array([1,-1]) b = 0 wtold = np.zeros((9,)).astype(int) wtnew = np.zeros((9,)).astype(int) print("--",wtold) print("First input with target 1") for i in range(0,9):

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

wtold = wtnew 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) print('\nAyush Patel : 53004230035')

**B. Write a program to implement delta rule.**

**CODE:**

import numpy as np 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("Initial Desired: ")) a = float(input("Enter learning rate: ")) print("\nActual",actual) print("Desired\n",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('\nAyush Patel : 53004230035')

**Practical No: 04**

**A. Write a program for Backpropagation Algorithm**.

**CODE:**

import numpy as np

X=np.array(([2,9],[1,5],[3,6]),dtype=float)

Y=np.array(([92],[86],[89]),dtype=float)

#scale units

X=X/np.amax(X,axis=0) Y=Y/100; class NN(object): def \_\_init\_\_(self): self.inputsize=2 self.outputsize=1 self.hiddensize=3

self.W1=np.random.randn(self.inputsize,self.hiddensize)

self.W2=np.random.randn(self.hiddensize,self.outputsize) def forward(self,X):

self.z=np.dot(X,self.W1) self.z2=self.sigmoidal(self.z) self.z3=np.dot(self.z2,self.W2) op=self.sigmoidal(self.z3) return op; def sigmoidal(self,s):

return 1/(1+np.exp(-s)) obj=NN() op=obj.forward(X) print("actual output\n"+str(op)) print("expected output\n"+str(Y)) print('\nAyush Patel : 53004230035')

**B. Write a program for Error Backpropagation Algorithm.**

**CODE:**

import numpy as np

X=np.array(([2,9],[1,5],[3,6]),dtype=float)

Y=np.array(([92],[86],[89]),dtype=float)

X=X/np.amax(X,axis=0) Y=Y/100; class NN(object): def \_\_init\_\_(self): self.inputsize=2 self.outputsize=1 self.hiddensize=1

self.W1=np.random.randn(self.inputsize,self.hiddensize)

self.W2=np.random.randn(self.hiddensize,self.outputsize) def forward(self,X):

self.z=np.dot(X,self.W1) self.z2=self.sigmoidal(self.z) self.z3=np.dot(self.z2,self.W2) op=self.sigmoidal(self.z3) return op; def sigmoidal(self,s):

return 1/(1+np.exp(-s)) def sigmoidalprime(self,s):

return s\* (1-s) def backward(self,X,Y,o):

self.o\_error=Y-o self.o\_delta=self.o\_error \* self.sigmoidalprime(o) self.z2\_error=self.o\_delta.dot(self.W2.T) self.z2\_delta=self.z2\_error \* self.sigmoidalprime(self.z2) self.W1 = self.W1 + X.T.dot(self.z2\_delta) self.W2= self.W2+ self.z2.T.dot(self.o\_delta) def train(self,X,Y): o=self.forward(X) self.backward(X,Y,o) obj=NN() for i in range(2000): print("input"+str(X)) print("Actual output"+str(Y)) print("Predicted output"+str(obj.forward(X))) print("loss"+str(np.mean(np.square(Y-obj.forward(X))))) obj.train(X,Y)

print('\nAyush Patel: 53004230035')

# Practical No: 05

**A. Write a program for Hopfield Network .**

**CODE:**

import numpy as np def compute\_next\_state(state,weight):

next\_state = np.where(weight @ state>= 0, +1, -1) return next\_state def compute\_final\_state(initial\_state,weight,max\_iter=1000): previous\_state = initial\_state next\_state =compute\_next\_state(previous\_state,weight) is\_stable = np.all(previous\_state == next\_state)

n\_iter = 0 while(not is\_stable) and (n\_iter <= max\_iter):

previous\_state = next\_state; next\_state = compute\_next\_state(previous\_state,weight) is\_stable = np.all(previous\_state==next\_state) n\_iter +=1 return previous\_state, is\_stable,n\_iter initial\_state = np.array([+1,-1,-1,-1]) weight = np.array([ [0, -1, -1, +1],

[-1, 0, +1, -1],

[-1,+1, 0, -1], [+1,-1, -1, 0]]) final\_state, is\_stable, n\_iter = compute\_final\_state(initial\_state,weight) print("Final state",final\_state) print("is\_Stable",is\_stable) print('\nAyush Patel : 53004230035')

**B. Write a program for Radial Basis function.**

**CODE:**

**:** R Compiler

D <- matrix(c(-3,1,4), ncol=1) N <- length(D) rbf.gauss <- function(gamma=1.0) { function(x){ exp(-gamma \* norm(as.matrix(x),"F")^2)

} } xlim <- c(-5,7) print(N) print(xlim) plot(NULL,xlim=xlim,ylim=c(0,1.25), type = "n") points(D,rep(0,length(D)), col= 1:N,pch=19)

x.coord = seq(-7,7,length=250) gamma <- 1.5 for (i in 1:N){

points(x.coord, lapply(x.coord -

D[i,],rbf.gauss(gamma)),type="l",col=i)

}

# Practical No: 06

**A. Kohonen Self organizing map .**

**CODE:**

!pip install minisom import minisom from minisom import MiniSom import matplotlib.pyplot as plt data = [[ 0.80, 0.55, 0.22, 0.03],

[ 0.82, 0.50, 0.23, 0.03],

[ 0.80, 0.54, 0.22, 0.03],

[ 0.80, 0.53, 0.26, 0.03],

[ 0.79, 0.56, 0.22, 0.03],

[ 0.75, 0.60, 0.25, 0.03],

[ 0.77, 0.59, 0.22, 0.03]] som = MiniSom(6, 6, 4, sigma=0.3, learning\_rate=0.5) # initialization of 6x6 SOM som.train\_random(data, 100) # trains the SOM with 100 iterations plt.imshow(som.distance\_map())

print('\nAyush Patel : 53004230035')

**B. Adaptive Resonance Theory .**

**CODE:**

from \_\_future\_\_ import print\_function from \_\_future\_\_ import division import numpy as np class ART: def \_\_init\_\_(self, n=5, m=10, rho=.5):

self.F1 = np.ones(n) self.F2 = np.ones(m) self.Wf = np.random.random((m,n)) self.Wb = np.random.random((n,m)) self.rho = rho self.active = 0 def learn(self, X):

self.F2[...] = np.dot(self.Wf, X)

I = np.argsort(self.F2[:self.active].ravel())[::-1] for i in I:

d = (self.Wb[:,i]\*X).sum()/X.sum() if d >= self.rho: self.Wb[:,i] \*= X self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum()) return self.Wb[:,i], i if self.active < self.F2.size:

i = self.active self.Wb[:,i] \*= X self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum()) self.active += 1 return self.Wb[:,i], i return None,None if \_\_name\_\_ == '\_\_main\_\_':

np.random.seed(1) network = ART( 5, 10, rho=0.5) data = [" O ", " O O",

" O",

" O O",

" O",

" O O",

" O",

" OO O",

" OO ",

" OO O",

" OO ",

"OOO ",

"OO ",

"O ",

"OO ",

"OOO ",

"OOOO ",

"OOOOO",

"O ",

" O ",

" O ",

" O ",

" O",

" O O",

" OO O",

" OO ",

"OOO ",

"OO ",

"OOOO ",

"OOOOO"] X = np.zeros(len(data[0])) for i in range(len(data)): for j in range(len(data[i])):

X[j] = (data[i][j] == 'O') Z, k = network.learn(X) print("|%s|"%data[i],"-> class", k)

print('\nAyush Patel : 53004230035')

**Practical No: 07**

**A. Write a program for Linear Separation .**

**CODE:**

import numpy as np import matplotlib.pyplot as plt def create\_distance\_function(a, b, c):

""" 0 = ax + by + c """ def distance(x, y):

""" returns tuple (d, pos) d is the distance

If pos == -1 point is below the line,

0 on the line and +1 if above the line

""" 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("Sweet") ax.set\_ylabel("Sour") ax.set\_xlim([-1, 6]) ax.set\_ylim([-1, 8]) X = np.arange(-0.5, 5, 0.1) colors = ["r", ""] # for the samples 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, "oy", 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)

#print("x: ", x, "slope: ", slope)

Y = slope \* X

results = [] for point in points:

results.append(dist4line1(\*point))

#print(slope, results) if (results[0][1] != results[1][1]): ax.plot(X, Y, "g-") else: ax.plot(X, Y, "r-") plt.show() print('\nAyush Patel : 53004230035')

**B. Write a program for Hopfield Network model for associative memory .**

**CODE:**

!pip install neurodynex from neurodynex.hopfield\_network import network, pattern\_tools, plot\_tools import matplotlib.pyplot as plt pattern\_size = 5

# create an instance of the class HopfieldNetwork hopfield\_net = network.HopfieldNetwork(nr\_neurons= pattern\_size\*\*2)

# instantiate a pattern factory factory = pattern\_tools.PatternFactory(pattern\_size, pattern\_size)

# create a checkerboard pattern and add it to the pattern list checkerboard = factory.create\_checkerboard() pattern\_list = [checkerboard] # add random patterns to the list pattern\_list.extend(factory.create\_random\_pattern\_list(nr\_patt erns=3, on\_probability=0.5)) plot\_tools.plot\_pattern\_list(pattern\_list)

# how similar are the random patterns and the checkerboard? Check the overlaps overlap\_matrix = pattern\_tools.compute\_overlap\_matrix(pattern\_list) plot\_tools.plot\_overlap\_matrix(overlap\_matrix)

# let the hopfield network "learn" the patterns. Note: they are not stored

# explicitly but only network weights are updated !

hopfield\_net.store\_patterns(pattern\_list)

# create a noisy version of a pattern and use that to initialize the network noisy\_init\_state = pattern\_tools.flip\_n(checkerboard, nr\_of\_flips=4) hopfield\_net.set\_state\_from\_pattern(noisy\_init\_state) # from this initial state, let the network dynamics evolve. states = hopfield\_net.run\_with\_monitoring(nr\_steps=4) # each network state is a vector. reshape it to the same shape used to create the patterns.

states\_as\_patterns = factory.reshape\_patterns(states)

# plot the states of the network plot\_tools.plot\_state\_sequence\_and\_overlap(states\_as\_patterns, pattern\_list, reference\_idx=0, suptitle="Network dynamics")

# Practical No: 08

**A. Membership and Identity Operators | in, not in** .

**1- In Operator**

**CODE:**

list1=[] print("Enter 5 numbers: ") for i in range(0,5): v=int(input()) list1.append(v) list2=[] print("Enter 5 numbers: ") for i in range(0,5): v=int(input()) list2.append(v) flag=0 for i in list1: if i in list2: flag=1 if(flag==1):

print("The Lists Overlap") else:

print("The Lists do Not overlap")

print('\nAyush Patel : 53004230035')

**2- not In Operator**

**CODE:**

#Aim: Not in operator in python list1=[] c=int(input("Enter the number of elements that you want to insert in List: ")) for i in range(0,c): ele = int(input("Enter the element: ")) list1.append(ele) a = int(input("Enter the number that you want to find in List:

")) if a not in list1:

print("\nThe list does not contain", a ) else:

print("\nThe list contains", a ) print('\nAyush Patel : 53004230035')

**B. Membership and Identity Operators | is, is not**.

**1: Implement Membership and Identity Operators is.**

**CODE:**

details =[] name=input("Enter your name : ") details.append(name) age=float(input("Enter your exact age : ")) details.append(age) roll\_no=int(input("Enter your roll no : ")) details.append(roll\_no) for i in details:

print(i) print("Int = ",type(i) is int) print("Float = ",type(i) is float) print("String = ",type(i) is str) print()

print('\nAyush Patel : 53004230035')

**2: Implement Membership and Identity Operators is not.**

**CODE:**

details =[] name=input("Enter your name : ") details.append(name) age=float(input("Enter your exact age : ")) details.append(age) roll\_no=int(input("Enter your roll no : ")) details.append(roll\_no) print() for i in details:

print(i) print("Not Int = ",type(i) is not int) print("Not Float = ",type(i) is not float) print("Not String = ",type(i) is not str) print('\nAyush Patel : 53004230035')

# Practical No: 09

**A. Find ratios using Fuzzy logic**.

**CODE:**

!pip install fuzzywuzzy

from fuzzywuzzy import fuzz from fuzzywuzzy import process print('Ayush Patel - 53004230035') s1 = "I love fuzzysforfuzzys" s2 = "I am loving fuzzysforfuzzys" print ("FuzzyWuzzy Ratio:", fuzz.ratio(s1, s2)) print ("FuzzyWuzzy PartialRatio: ", fuzz.partial\_ratio(s1, s2)) print ("FuzzyWuzzy TokenSortRatio: ", fuzz.token\_sort\_ratio(s1, s2)) print ("FuzzyWuzzy TokenSetRatio: ", fuzz.token\_set\_ratio(s1, s2)) print ("FuzzyWuzzy WRatio: ", fuzz.WRatio(s1, s2),'\n\n')

# for process library, 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))

**B. Solve Tipping problem using Fuzzy logic**.

**CODE:**

!pip install fuzzywuzzy

!pip install -U scikit-fuzzy

import skfuzzy as fuzz from skfuzzy import control as ctrl import numpy as np 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')

quality.automf(3) service.automf(3)

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])

quality['average'].view() service.view() tip.view() rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low']) rule2 = ctrl.Rule(service['average'], tip['medium']) rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()

tipping\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3]) tipping = ctrl.ControlSystemSimulation(tipping\_ctrl)

tipping.input['quality'] = 6.5 tipping.input['service'] = 9.8

tipping.compute() print (tipping.output['tip']) tip.view(sim=tipping) print('Ayush Patel - 53004230035')

# Practical No: 10

**A. Implementation of a Simple Genetic Algorithm.**

**CODE:**

import random

# Number of individuals in each generation

POPULATION\_SIZE = 100

# Valid genes

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP

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

# Target string to be generated TARGET = "UPG College Student " class Individual(object):

'''

Class representing individual in population

''' def \_\_init\_\_(self, chromosome): self.chromosome = chromosome self.fitness = self.cal\_fitness()

@classmethod

def mutated\_genes(self):

''' create random genes for mutation

''' global GENES gene = random.choice(GENES) return gene @classmethod def create\_gnome(self):

''' create chromosome or string of genes

''' global TARGET gnome\_len = len(TARGET) return [self.mutated\_genes() for \_ in range(gnome\_len)] def mate(self, par2):

'''

Perform mating and produce new offspring

'''

# chromosome for offspring child\_chromosome = [] for gp1, gp2 in zip(self.chromosome, par2.chromosome):

# random probability prob = random.random()

# if prob is less than 0.45, insert gene

# from parent 1 if prob < 0.45:

child\_chromosome.append(gp1)

# if prob is between 0.45 and 0.90, insert

# gene from parent 2 elif prob < 0.90:

child\_chromosome.append(gp2)

# otherwise insert random gene(mutate),

# for maintaining diversity else:

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

# create new Individual(offspring) using # generated chromosome for offspring return Individual(child\_chromosome) def cal\_fitness(self):

'''

Calculate fittness score, it is the number of characters in string which differ from target string. ''' global TARGET fitness = 0 for gs, gt in zip(self.chromosome, TARGET):

if gs != gt: fitness+= 1 return fitness # Driver code def main():

global POPULATION\_SIZE #current generation generation = 1 found = False population = []

# create initial population for \_ in range(POPULATION\_SIZE):

gnome = Individual.create\_gnome() population.append(Individual(gnome)) while not found:

# sort the population in increasing order of fitness score population = sorted(population, key = lambda x:x.fitness)

# if the individual having lowest fitness score ie.

# 0 then we know that we have reached to the target

# and break the loop if population[0].fitness <= 0:

found = True break

# Otherwise generate new offsprings for new generation new\_generation = []

# Perform Elitism, that mean 10% of fittest population

# goes to the next generation s = int((10\*POPULATION\_SIZE)/100) new\_generation.extend(population[:s])

# From 50% of fittest population, Individuals

# will mate to produce offspring 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("Generation: {}\tString: {}\tFitness: {}".\ format(generation,

"".join(population[0].chromosome), population[0].fitness)) generation += 1 print("Generation: {}\tString: {}\tFitness: {}".\ format(generation, "".join(population[0].chromosome), population[0].fitness)) if \_\_name\_\_ == '\_\_main\_\_':

main() print('Ayush Patel : 53004230035')

**B. Create two classes: City and Fitness using Genetic Algorithm.**

**CODE:**

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

print('Ayush Patel : 53004230035')