Genetic:  
# what is mutation:

# we need mutation for when we got the same answer on every cycle or itration

# therefore we do mutation to change the answer using mutation

# what is Genetic Algorithm:

# Genetic algorithms are optimization techniques inspired by biological evolution, where solutions to a problem evolve over generations through selection, crossover, and mutation to find the best solution.

import random

best = -10000

populations = [[random.randint(0, 1) for x in range(6)] for i in range(4)]

print(type(populations))

parents = []

new\_populations = []

print(populations)

def fitness\_score():

global populations, best

fit\_value = []

fit\_score = []

for i in range(4):

chromosome\_value = 0

for j in range(5, 0, -1):

chromosome\_value += populations[i][j] \* (2 \*\* (5 - j))

chromosome\_value = -1 \* chromosome\_value if populations[i][0] == 1 else chromosome\_value

print(chromosome\_value)

fit\_value.append(-(chromosome\_value \*\* 2) + 5)

print(fit\_value)

fit\_value, populations = zip(\*sorted(zip(fit\_value, populations), reverse=True))

best = fit\_value[0]

def selectparent():

global parents

parents = populations[0:2]

print(type(parents))

print(parents)

def crossover():

global parents

cross\_point = random.randint(0, 5)

parents = parents + [tuple(parents[0][0:cross\_point + 1] + parents[1][cross\_point + 1:6])]

parents = parents + [tuple(parents[1][0:cross\_point + 1] + parents[0][cross\_point + 1:6])]

def mutation():

global populations, parents

mute = random.randint(0, 49)

if mute == 20:

x = random.randint(0, 3)

y = random.randint(0, 5)

parents[x][y] = 1 - parents[x][y]

populations = parents

print(populations)

def inverse\_mutation():

global populations, parents

mute = random.randint(0, 49)

if mute == 20:

x = random.randint(0, 3)

start = random.randint(0, 4)

end = random.randint(start, 5)

for y in range(start, end + 1):

parents[x][y] = 1 - parents[x][y]

populations = parents

print(populations)

for i in range(1000):

fitness\_score()

selectparent()

crossover()

mutation()

print("Best Score:")

print(best)

print("sequence..")

print(populations[0])

Kmean:

from sklearn.cluster import KMeans

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from matplotlib import pyplot as plt

%matplotlib inline

df = pd.read\_csv("income.csv")

df.head()

#

plt.scatter(df.Age,df['Income($)'])

plt.xlabel('Age')

plt.ylabel('Income($)')

km = KMeans(n\_clusters=3)

y\_predicted= km.fit\_predict(df[['Age', 'Income($)']])

y\_predicted

df['cluster']=y\_predicted

df.head()

km.cluster\_centers\_

df1=df[df.cluster==0]

df2=df[df.cluster==1]

df3=df[df.cluster==2]

plt.scatter(df1.Age, df1 ['Income($)'], color='green')

plt.scatter(df2.Age, df2['Income($)'], color='red')

plt.scatter(df3.Age, df3['Income($)'], color='black')

plt.scatter(km.cluster\_centers\_[:,0], km.cluster\_centers\_[:,1], color='purple', marker='\*', label='centroid')

plt.xlabel('Age')

plt.ylabel('Income ($)')

plt.legend()

#

scaler = MinMaxScaler()

scaler.fit(df[['Income($)']])

df['Income($)'] = scaler.transform(df [['Income($)']])

scaler.fit(df[['Age']])

df['Age'] = scaler.transform(df[['Age']])

df.head()

#

plt.scatter(df.Age, df['Income($)'])

km = KMeans (n\_clusters=3)

y\_predicted = km.fit\_predict(df[['Age', 'Income($)']])

y\_predicted

df['cluster']=y\_predicted

df.head()

km.cluster\_centers\_

df1 = df[df.cluster==0]

df2 = df[df.cluster==1]

df3 = df[df.cluster==2]

plt.scatter(df1.Age, df1['Income($)'], color='green')

plt.scatter(df2.Age, df2['Income($)'], color='red')

plt.scatter(df3.Age, df3['Income($)'], color='black')

plt.scatter(km.cluster\_centers\_[:,0], km.cluster\_centers\_[:,1], color='purple', marker='\*', label='centroid')

plt.legend()

#

#Elbow Plot

sse = []

k\_rng = range(1,10)

for k in k\_rng:

km = KMeans(n\_clusters=k)

km.fit(df[['Age','Income($)']])

sse.append(km.inertia\_)

plt.xlabel('K')

plt.ylabel('Sum of squared error')

plt.plot(k\_rng,sse)

MLP:

import numpy as np

import matplotlib.pyplot as plt

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

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

no\_x = 2

no\_y = 1

no\_h = 2

tot = x.shape[1]

lr = 0.1

np.random.seed(2) # Define random seed for consistent results

w1 = np.random.rand(no\_h,no\_x)

w2 = np.random.rand(no\_y,no\_h)

losses = []

def sigmoid(z):

z = 1 / (1 + np.exp(-z))

return z

# Forward propagation

def frwd\_prop(w1, w2, x):

z1 = np.dot(w1, x)

a1 = sigmoid(z1)

z2 = np.dot(w2, a1)

a2 = sigmoid(z2)

return z1, a1, z2, a2

# Backward propagation

def back\_prop(tot, w1, w2, z1, a1, z2, a2, y):

dz2 = a2 - y

dw2 = np.dot(dz2, a1.T) / tot

dz1 = np.dot(w2.T, dz2) \* a1 \* (1 - a1)

dw1 = np.dot(dz1, x.T) / tot

dw1 = np.reshape(dw1, w1.shape)

dw2 = np.reshape(dw2, w2.shape)

return dz2, dw2, dz1, dw1

# Calculating Loss value and updating weights

epochs = 20000

for i in range(epochs):

z1, a1, z2, a2 = frwd\_prop(w1, w2, x)

loss = -(1/tot) \* np.sum(y \* np.log(a2) + (1 - y) \* np.log(1 - a2))

losses.append(loss)

dz2, dw2, dz1, dw1 = back\_prop(tot, w1, w2, z1, a1, z2, a2, y)

w2 = w2 - lr \* dw2

w1 = w1 - lr \* dw1

# Plotting losses to see how the network is performing

plt.plot(losses)

plt.xlabel("Epochs")

plt.ylabel("Loss value")

plt.show()

# Defining Prediction method

def predict(w1, w2, input):

z1,z1,z2,a2 = frwd\_prop(w1,w2,mlp\_test)

a2 = np.squeeze(z2)

if a2 >= 0.5:

print("For input", [i[0] for i in input], "output is 1")

else:

print("For input", [i[0] for i in input], "output is 0")

# Calling Network Model

mlp\_test = np.array([[1], [0]])

predict(w1, w2, mlp\_test)

mlp\_test = np.array([[0], [0]])

predict(w1, w2, mlp\_test)

mlp\_test = np.array([[0], [1]])

predict(w1, w2, mlp\_test)

mlp\_test = np.array([[1], [1]])

predict(w1, w2, mlp\_test)