**Program -1**

import numpy as np

import pandas as pd

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn import metrics

from sklearn.datasets import load\_iris

names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

data = load\_iris()

X = data.data

y = data.target

Xtrain, Xtest, ytrain, ytest = train\_test\_split(X, y, *test\_size*=0.10)

classifier = KNeighborsClassifier(*n\_neighbors*=5)

classifier.fit(Xtrain, ytrain)

ypred = classifier.predict(Xtest)

i = 0

print ("\n-------------------------------------------------------------------------")

print ('%-25s %-25s %-25s' % ('Original Label', 'Predicted Label', 'Correct/Wrong'))

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

for label in ytest:

    print ('%-25s %-25s' % (label, ypred[i]), *end*="")

    if (label == ypred[i]):

        print (' %-25s' % ('Correct'))

    else:

        print (' %-25s' % ('Wrong'))

    i = i + 1

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

print("\nConfusion Matrix:\n",metrics.confusion\_matrix(ytest, ypred))

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

print("\nClassification Report:\n",metrics.classification\_report(ytest, ypred))

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

print('Accuracy of the classifer is %0.2f' % metrics.accuracy\_score(ytest,ypred))

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

**Program - 2:**

from sklearn.cluster import KMeans

from sklearn.mixture import GaussianMixture

import sklearn.metrics as metrics

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

names = ['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width', 'Class']

dataset = load\_iris()

X = dataset.data

y = dataset.target

plt.figure(*figsize*=(14,7))

colormap=np.array(['red','lime','black'])

# REAL PLOT

plt.subplot(1,3,1)

plt.title('Real')

plt.scatter(X[:,2],X[:,3],*c*=colormap[y])

# K-PLOT

model=KMeans(*n\_clusters*=3, *random\_state*=0).fit(X)

plt.subplot(1,3,2)

plt.title('KMeans')

plt.scatter(X[:,2],X[:,3],*c*=colormap[model.labels\_])

print('The accuracy score of K-Mean: ',metrics.accuracy\_score(y, model.labels\_))

print('The Confusion matrixof K-Mean:\n',metrics.confusion\_matrix(y, model.labels\_))

# GMM PLOT

gmm=GaussianMixture(*n\_components*=3, *random\_state*=0).fit(X)

y\_cluster\_gmm=gmm.predict(X)

plt.subplot(1,3,3)

plt.title('GMM Classification')

plt.scatter(X[:,2],X[:,3],*c*=colormap[y\_cluster\_gmm])

plt.show()

print('The accuracy score of EM: ',metrics.accuracy\_score(y, y\_cluster\_gmm))

print('The Confusion matrix of EM:\n ',metrics.confusion\_matrix(y, y\_cluster\_gmm))

**Program - 3:**

import numpy as np

import matplotlib.pyplot as plt

*def* local\_regression(*x0*, *X*, *Y*, *tau*):

*x0* = [1, *x0*]

*X* = [[1, i] for i in *X*]

*X* = np.asarray(*X*)

    xw = (*X*.T) \* np.exp(np.sum((*X* - *x0*) \*\* 2, *axis*=1) / (-2 \* *tau*))

    beta = np.linalg.pinv(xw @ *X*) @ xw @ *Y* @ *x0*

    return beta

*def* draw(*tau*):

    prediction = [local\_regression(x0, X, Y, *tau*) for x0 in domain]

    plt.plot(X, Y, 'o', *color*='black')

    plt.plot(domain, prediction, *color*='red')

    plt.show()

X = np.linspace(-3, 3, *num*=1000)

domain = X

Y = np.log(np.abs(X \*\* 2 - 1) + .5)

draw(10)

draw(0.1)

draw(0.01)

draw(0.001)

**Program - 4:**

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) # maximum of X array longitudinally

y = y/100

#Sigmoid Function

*def* sigmoid (*x*):

    return 1/(1 + np.exp(-*x*))

#Derivative of Sigmoid Function

*def* derivatives\_sigmoid(*x*):

    return *x* \* (1 - *x*)

#Variable initialization

epoch=5000                #Setting training iterations

lr=0.1                    #Setting learning rate

inputlayer\_neurons = 2    #number of features in data set

hiddenlayer\_neurons = 3   #number of hidden layers neurons

output\_neurons = 1        #number of neurons at output layer

#weight and bias initialization

wh=np.random.uniform(*size*=(inputlayer\_neurons,hiddenlayer\_neurons))

bh=np.random.uniform(*size*=(1,hiddenlayer\_neurons))

wout=np.random.uniform(*size*=(hiddenlayer\_neurons,output\_neurons))

bout=np.random.uniform(*size*=(1,output\_neurons))

#draws a random range of numbers uniformly of dim x\*y

for i in range(epoch):

#Forward Propogation

    hinp1=np.dot(X,wh)

    hinp=hinp1 + bh

    hlayer\_act = sigmoid(hinp)

    outinp1=np.dot(hlayer\_act,wout)

    outinp= outinp1+ bout

    output = sigmoid(outinp)

#Backpropagation

    EO = y-output

    outgrad = derivatives\_sigmoid(output)

    d\_output = EO\* outgrad

    EH = d\_output.dot(wout.T)

#how much hidden layer wts contributed to error

    hiddengrad = derivatives\_sigmoid(hlayer\_act)

    d\_hiddenlayer = EH \* hiddengrad

# dotproduct of nextlayererror and currentlayerop

    wout += hlayer\_act.T.dot(d\_output) \*lr

    wh += X.T.dot(d\_hiddenlayer) \*lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)

**Program - 5:**

import random

import string

# Define the target string

target\_string = "HELLO, GENETIC ALGORITHM!"

target\_length = len(target\_string)

# Define other parameters

population\_size = 100

mutation\_rate = 0.01

# Generate an initial population of random strings

*def* generate\_random\_string(*length*):

    return ''.join(random.choice(string.ascii\_uppercase + ' ,') for \_ in range(*length*))

*def* initial\_population():

    return [generate\_random\_string(target\_length) for \_ in range(population\_size)]

# Calculate the fitness of each individual in the population

*def* calculate\_fitness(*individual*):

    return sum(1 for i in range(target\_length) if *individual*[i] == target\_string[i])

# Select parents based on their fitness

*def* select\_parents(*population*):

    return random.choices(*population*, *weights*=[calculate\_fitness(individual) for individual in *population*], *k*=2)

# Crossover two parents to produce a child

*def* crossover(*parent1*, *parent2*):

    crossover\_point = random.randint(1, target\_length - 1)

    child = *parent1*[:crossover\_point] + *parent2*[crossover\_point:]

    return child

# Mutate an individual with a given probability

*def* mutate(*individual*):

    return ''.join(random.choice(string.ascii\_uppercase + ' ,') if random.random() < mutation\_rate else char for char in *individual*)

# Main genetic algorithm loop

population = initial\_population()

generation = 1

while True:

    population = sorted(population, *key*=calculate\_fitness, *reverse*=True)

    best\_individual = population[0]

    print(*f*"Generation {generation}: {best\_individual} (Fitness: {calculate\_fitness(best\_individual)})")

    if best\_individual == target\_string:

        break

    new\_population = [best\_individual]

    for \_ in range(population\_size - 1):

        parent1, parent2 = select\_parents(population)

        child = crossover(parent1, parent2)

        child = mutate(child)

        new\_population.append(child)

    population = new\_population

    generation += 1

**Program - 6:**

import numpy as np

grid\_size = 5

goal\_state = (grid\_size-1, grid\_size-1)

obstacles = [(2, 2), (3, 1)]

learning\_rate = 0.8

discount\_factor = 0.95

num\_episodes = 1000

q\_table = np.zeros((grid\_size, grid\_size, 4))

*def* take\_action(*state*, *epsilon*):

    if np.random.rand() < *epsilon*:

        return np.random.choice([0, 1, 2, 3])

    else:

        return np.argmax(q\_table[*state*])

*def* update\_q\_value(*state*, *action*, *reward*, *next\_state*):

    old\_q\_value = q\_table[*state*][*action*]

    next\_max\_q = np.max(q\_table[*next\_state*])

    new\_q\_value = old\_q\_value + learning\_rate \* (*reward* + discount\_factor \* next\_max\_q - old\_q\_value)

    q\_table[*state*][*action*] = new\_q\_value

*def* get\_next\_state(*state*, *action*):

    row, col = *state*

    next\_row = row + (*action* == 2) - (*action* == 0)

    next\_col = col + (*action* == 1) - (*action* == 3)

    if 0 <= next\_row < grid\_size and 0 <= next\_col < grid\_size:

        if (next\_row, next\_col) not in obstacles:

            return (next\_row, next\_col)

    return *state*

*def* get\_reward(*state*):

    if *state* == goal\_state:

        return 10

    elif *state* in obstacles:

        return -1

    else:

        return 0

*def* is\_goal\_state(*state*):

    return *state* == goal\_state

*def* is\_obstacle\_state(*state*):

    return *state* in obstacles

for episode in range(num\_episodes):

    state = (0, 0)

    done = False

    epsilon = 1.0 / (episode + 1)

    while not done:

        action = take\_action(state, epsilon)

        next\_state = get\_next\_state(state, action)

        reward = get\_reward(next\_state)

        update\_q\_value(state, action, reward, next\_state)

        state = next\_state

        done = is\_goal\_state(state) or is\_obstacle\_state(state)

print("Final Q-table:")

print(q\_table)