1.

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
iris=datasets.load_iris()
print("Iris Data set loaded...")
x_train, x_test, y_train, y_test = train_test_split(iris.data,iris.target,test_size=0.1)
#random_state=0
for i in range(len(iris.target_names)):
   print("Label", i , "-",str(iris.target_names[i]))
classifier = KNeighborsClassifier(n_neighbors=2)
classifier.fit(x_train, y_train)
y_pred=classifier.predict(x_test)
print("Results of Classification using K-nn with K=1 ")
for r in range(0,len(x_test)):
   print(" Sample:", str(x_test[r]), " Actual-label:", str(y_test[r])," Predicted-label:", str(y_pred[r]))

   print("Classification Accuracy :" , classifier.score(x_test,y_test))
```

2.

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import adjusted_rand_score
# Load the dataset
data = pd.read_csv("/content/iris1.csv")
X = data.iloc[:, :-1].values # Features
# Preprocess the data
scaler = StandardScaler()
 X_scaled = scaler.fit_transform(X)
# Apply K-means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(X_scaled)
# Apply EM clustering
em = GaussianMixture(n_components=3, random_state=42)
em_labels = em.fit_predict(X_scaled)
# Ground truth labels from the dataset
#true_labels = data["variety"].map({"setosa": 0, "versicolor": 1,
"virginica": 2})
true_labels = data["variety"]
# Evaluate clustering results using Adjusted Rand Index (ARI)
ari_kmeans = adjusted_rand_score(true_labels, kmeans_labels) ari_em =
adjusted_rand_score(true_labels, em_labels) print("K-means ARI:",
ari_kmeans) print("EM ARI:", ari_em)
```

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)
```

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)
```

5.

```
import numpy as np
# Define the fitness function to minimize
def fitness_function(x):
 return x**2 + 4*x + 4
# Genetic Algorithm parameters
population_size = 50
num_generations = 100
mutation_rate = 0.1
# Initialize the population with random solutions
population = np.random.uniform(-10, 10, size=(population_size,))
# Main Genetic Algorithm loop
for generation in range(num_generations):
```

```
 # Evaluate fitness of each individual in the population
 fitness_values = np.array([fitness_function(x) for x in population])
parents = np.random.choice(population, size=population_size, p=fitness_values /
np.sum(fitness_values))
 # Create new population through crossover and mutation
 children = []
 for _ in range(population_size):
 parent1 = np.random.choice(parents)
 parent2 = np.random.choice(parents)
 child = (parent1 + parent2) / 2 # Simple averaging crossover
 if np.random.rand() < mutation_rate:
 child += np.random.normal(scale=0.5)
 children.append(child)
 population = np.array(children)
# Find the best solution
best_solution = population[np.argmin([fitness_function(x) for x in population])]
print("Best Solution:", best_solution)
print("Minimum Value:", fitness_function(best_solution))
```

6.

```
import numpy as np

# Initialize the Q-table
num_states = 5 * 5  # 5x5 grid world
num_actions = 4  # Up, Down, Left, Right
Q_table = np.zeros((num_states, num_actions))

# Define the grid world and obstacles
grid_world = np.zeros((5, 5))
grid_world[1, 1] = -1  # Obstacle at (1, 1)
grid_world[2, 2] = -1  # Obstacle at (2, 2)
grid_world[3, 3] = -1  # Obstacle at (3, 3)
goal = (4, 4)

# Parameters
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 0.1  # Exploration-exploitation trade-off

# Convert grid coordinates to a state index
def state_to_index(state):
    row, col = state
    return row * 5 + col

# Q-learning algorithm
def q_learning(num_episodes):
    for episode in range(num_episodes):
        state = (0, 0)  # Start at the bottom-left corner
        while state != goal:
            # Exploration-exploitation trade-off
```

```
            if np.random.uniform(0, 1) < epsilon:
                action = np.random.choice(num_actions)
            else:
                action = np.argmax(Q_table[state_to_index(state)])

            # Take the chosen action and observe the next state and reward
            next_state = (state[0] + (action == 1) - (action == 0), state[1] + (action == 3) - (action
== 2))

            # Clip the next state to ensure it stays within the bounds of the grid
            next_state = (np.clip(next_state[0], 0, 4), np.clip(next_state[1], 0, 4))

            reward = -1  # Small negative reward for each step

            # Update the Q-value using the Q-learning update rule
            Q_table[state_to_index(state), action] += alpha * (
                reward + gamma * np.max(Q_table[state_to_index(next_state)]) -
Q_table[state_to_index(state), action]
            )

            # Move to the next state
            state = next_state

        print("Q-learning training complete.")

    # Test the learned policy
    def test_policy():
        state = (0, 0)
        path = [state]
        while state != goal:
            action = np.argmax(Q_table[state_to_index(state)])
            state = (state[0] + (action == 1) - (action == 0), state[1] + (action == 3) - (action == 2))

            # Clip the state to ensure it stays within the bounds of the grid
            state = (np.clip(state[0], 0, 4), np.clip(state[1], 0, 4))

            path.append(state)

        print("Optimal path:", path)

    # Run Q-learning
    q_learning(num_episodes=1000)

    # Test the learned policy
    test_policy()
```