Tab 1

Soft computing Practical

Name:
Roll no:
Subject:
Div:

INDEX

NO	Title	Date	Signature
1	Implement the following: A) Design a simple linear neural network model. B) Calculate the output of a neural net using both binary and bipolar sigmoidal function. C) Write a python code to calculate net input and apply the activation function.		
2	Implement the following: A) Generate AND/NOT function using McCulloch-Pitts neural net.		
3	Implement the Following A) Write a program to implement Hebb's rule. B) Write a program to implement the delta rule.		
4	Implement the Following A) Write a program for Back Propagation Algorithm B) Write a program for error Backpropagation algorithm.		
5	Implement the Following A) Write a program for Hopfield Network.		
6	Implement the Following: A) Find ratios using fuzzy logic B) Solve Tipping problem using fuzzy logic		
7	Implementation of Simple genetic algorithm using its following operators: I. Selection II. Mutation II. Crossover		
8	Implement Ant Colony Optimization Technique.		

1) Implement the following:

A) Design a simple linear neural network model.

code:

```
import numpy as np

x = np.array([0.8, 0.6,0.4])
weights = np.array([0.1, 0.3,-0.2])
bias = 0.35

y = np.dot(x, weights) + bias
print("y = ",y)
```

Output

```
y =  0.53
```

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

code:

```
class ActivationF:
  def bnsf(self,x): # binary sigmoid function
    return 1/(1+np.exp(-x))
  def bpsf(self,x): # bipolar sigmoidal function
    return -1+(2/(1+np.exp(-x)))
  def tanh(self,x):
    return (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
def main():
  binary= ActivationF().bnsf(y)
  print("\n The output, after applying the binary sigmoidal function is:")
  print(round(binary,3)) # rounds off the answer by 3 decimal places

  bipolar=ActivationF().bpsf(y)
  print("\n the output, after applying the binary sigmoidal is: ")
  print(round(bipolar,3))

  tanh_output=ActivationF().tanh(y)
  print("\n the output, after applying the tanh function is: ")
  print(round(tanh_output,3))
if __name__ =="__main__":
  main()
```

Output

```
y =  0.53

 The output, after applying the binary sigmoidal function is:
0.629

 the ouppu t, after applying the binary sigmoidal is:
0.259

 the output, after applying the tanh function is:
0.485
```

C) Write a python code to calculate net input and apply the activation function.

code:

2) Implement the following:
A) Generate AND/NOT function using 1 neural net.

Code:

```
# Getting the weights and the threshold value
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 = [0,0,1,1];
x2 = [0,1,0,1];
z = [0,1,1,0] #expected output for eXclusive-OR Function
con = 1;
zin1 = [0,0,0,0]
zin2 = [0,0,0,0]
y1 = [0,0,0,0]
y2 = [0,0,0,0]
yin = [0,0,0,0]
y = [0,0,0,0]
while con:
  for i in range(0,3):
    zin1[i]= x1[i]*w11 + x2[i]*w21
    zin2[i]= x1[i]*w21 + x2[i]*w22
  for i in range(0,3):2
    if zin1[i] >= theta:
      y1[i] = 1;
    else:
      y1[i] = 0;
    if zin2[i] >= theta:
      y2[i] = 1;
    else:
      y2[i] = 0;
```

```
  for i in range(0,3):
    yin[i]= y1[i]*v1 + y2[i]*v2
  print(yin)
  for i in range(0,3):
    if yin[i] >= theta:
      y[i]=1;
    else:
      y[i]=0;

  print("output of net")
  print(y)
  if y==z:
    con = 0
  else:
    print("Net is not learning 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 = "))
    wv1 = int(input("weight v1 = "))
    wv2 = int(input("weight v2 = "))
    theta = int(input("theta = "))
  #endwhile
  print("McCulloh - Pitts Net for XOR Function")
  print("weight of neuron z1")
  print(w11)
  print(w12)
  print("weight of neuron z2")
  print(w21)
  print(w22)
  print(theta)
print("y:",y)
```

Output

```
enter weights:
weight w11 = 1
weight w12 = -1
weight w21 = -1
weight w22 = 1
weight v1 = 1
weight v2 = 1
enter threshold value :
theta = 1
[0, 1, 1, 0]
output of net
[0, 1, 1, 0]
McCulloh - Pitts Net for XOR Function
weight of neuron z1
1
-1
weight of neuron z2
-1
1
1
y: [0, 1, 1, 0]
```

3) Implement the Following
A) Write a program to implement Hebb's rule.

Code:

```
import numpy as np

class HebbRuleNN:
  def __init__(self, input_size, Learning_rate=0.01):
    #initialize weight randomly or to zero
    self.weights = np.zeros(input_size) # Use input_size here
    self.bias=0
    self.learning_rate = Learning_rate # Corrected parameter name

  def train(self, inputs, targets): # Corrected parameter name
    for i in range(len(inputs)):
      input_vector=inputs[i]
      target_output=targets[i] # Corrected parameter name

      delta_weights=self.learning_rate*input_vector*target_output
      self.weights+=delta_weights

      delta_bias=self.learning_rate*target_output # Corrected typo
      self.bias+=delta_bias

  def predict(self,input_vector):
    net_input=np.dot(input_vector,self.weights)+self.bias
    return 1 if net_input>=0 else 0

inputs=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])
targets=np.array([1,-1,-1,1]) # using and gate
targets1=np.array([1,1,1,-1]) # using or gate

network=HebbRuleNN(input_size=2)
network.train(inputs,targets1)

print("Final Weights:",network.weights)
print("Final Bias:",network.bias)
```

Output:

```
Final Weights: [0.02 0.02]
Final Bias: 0.019999999999999997
```

✅ Overall Flow Summary

1. Import library.
2. Define network class → holds weights, bias, train & predict methods.
3. Initialize weights/bias.
4. Define Hebbian learning method to update weights/bias.
5. Provide input patterns and target outputs.
6. Create network object with 2 input neurons.
7. Train network using Hebb rule → adjusts weights and bias.
8. Inspect learned weights/bias → network can now predict OR gate outputs.

B) Write a program to implement the delta rule.

Code:

4) Implement the Following
A) Write a program for Back Propagation Algorithm

Code:

```
import numpy as np

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.weights_input_hidden = np.random.randn(self.input_size,
self.hidden_size)
        self.weights_hidden_output = np.random.randn(self.hidden_size,
self.output_size)

        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.bias_output = np.zeros((1, self.output_size))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    #Defining Feed Forward Network
    def feedforward(self, x):
        self.hidden_activation = np.dot(x, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_activation)
        self.output_activation = np.dot(self.hidden_output,
self.weights_hidden_output) + self.bias_output
        self.predict_output = self.sigmoid(self.output_activation)
        return self.predict_output

    #Defining Backward Network
    def backward(self, X, y, learning_rate):
        output_error = y - self.predict_output
```

```
        output_delta = output_error *
self.sigmoid_derivative(self.predict_output)
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error *
self.sigmoid_derivative(self.hidden_output)
        self.weights_hidden_output += np.dot(self.hidden_output.T,
output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) *
learning_rate
        self.weights_input_hidden += np.dot(X.T, hidden_delta) *
learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) *
learning_rate

    #Training Network
    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            output = self.feedforward(X)
            self.backward(X, y, learning_rate)
            if epoch % 4000 == 0:
                loss = np.mean(np.square(y - output))
                print(f"Epoch {epoch}, Loss {loss}")

# Testing Neural Network
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
nn.train(X, y, epochs=10000, learning_rate=0.1)

output = nn.feedforward(X)
print("predictions after training:")
print(output)
```

Output:

```
Epoch 0, Loss 0.28296172436902217
Epoch 4000, Loss 0.019070304295013777
Epoch 8000, Loss 0.003114981528696914
predictions after training:
[[0.02971497]
 [0.95493644]
 [0.95388609]
 [0.05708703]]
```

B) Write a program for error Backpropagation algorithm

5) Implement the Following
A) Write a program for Hopfield Network.

<u>Code:</u>

```
import numpy as np

nb_patterns = 4   # Number of patterns to learn
pattern_width = 5
pattern_height = 5
max_iterations = 10

# Define Patterns
patterns = np.array([
   [1,-1,-1,-1,1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,-1,-1,1.],   # Letter D
   [-1,-1,-1,-1,-1,1,1,1,-1,1,1,1,1,-1,1,-1,1,1,-1,1,-1,-1,-1,1,1.],    # Letter J
   [1,-1,-1,-1,-1,-1,1,1,1,1,-1,1,1,1,1,-1,1,1,1,1,1,-1,-1,-1,-1.],     # Letter C
   [-1,1,1,1,-1,-1,-1,1,-1,-1,-1,1,-1,1,-1,-1,1,1,1,-1,-1,1,1,1,-1.],], # Letter M
   dtype=float)


# Train the network
W = np.zeros((pattern_width * pattern_height, pattern_width * pattern_height))

for i in range(pattern_width * pattern_height):
    for j in range(pattern_width * pattern_height):
        if i == j or W[i, j] != 0.0:
            continue

        w = 0.0

        for n in range(nb_patterns):
            w += patterns[n, i] * patterns[n, j]

        W[i, j] = w / patterns.shape[0]
        W[j, i] = W[i, j]
```

```
# Test the Network
# Create a corrupted pattern S
S = np.array(
[1,-1,-1,-1,-1,1,1,1,1,1,-1,1,1,1,1,-1,1,1,1,1,1,1,-1,-1,-1.],
   dtype=float)


h = np.zeros((pattern_width * pattern_height))
#Defining Hamming Distance matrix for seeing convergence
hamming_distance = np.zeros((max_iterations,nb_patterns))
for iteration in range(max_iterations):
    for i in range(pattern_width * pattern_height):
        i = np.random.randint(pattern_width * pattern_height)
        h[i] = 0
        for j in range(pattern_width * pattern_height):
            h[i] += W[i, j]*S[j]
        S = np.where(h<0, -1, 1)
    for i in range(nb_patterns):
        hamming_distance[iteration, i] = ((patterns - S)[i]!=0).sum()
print(hamming_distance)
```

Output:

```
[[ 8. 10. 11. 19.]
 [ 6.  8.  9. 21.]
 [ 7.  7. 10. 22.]
 [ 7.  7. 10. 22.]
 [ 7.  7. 10. 22.]
 [ 6.  8.  9. 23.]
 [ 6.  8.  9. 23.]
 [ 6.  8.  9. 23.]
 [ 6.  8.  9. 23.]
 [ 6.  8.  9. 23.]]
```

6) Implement the Following:
A) Find ratios using fuzzy logic

B) Solve Tipping problem using fuzzy logic

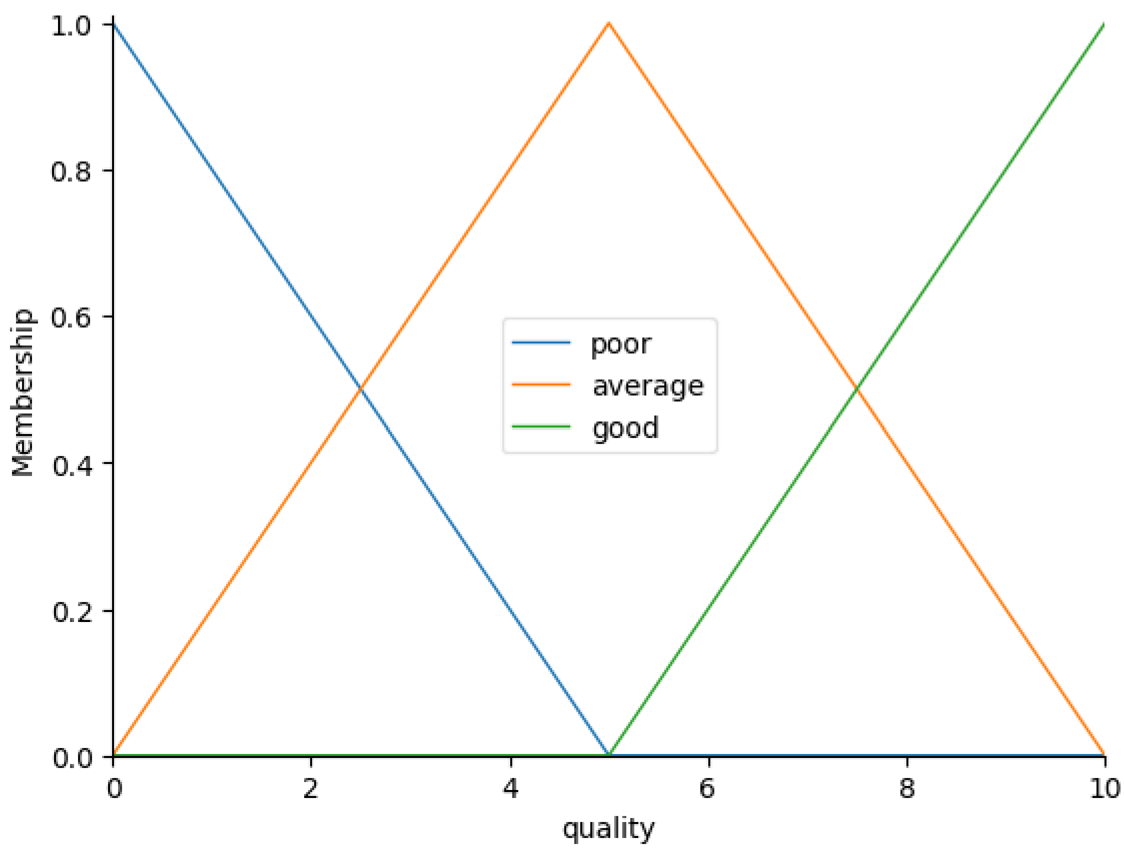
<u>Code:</u>

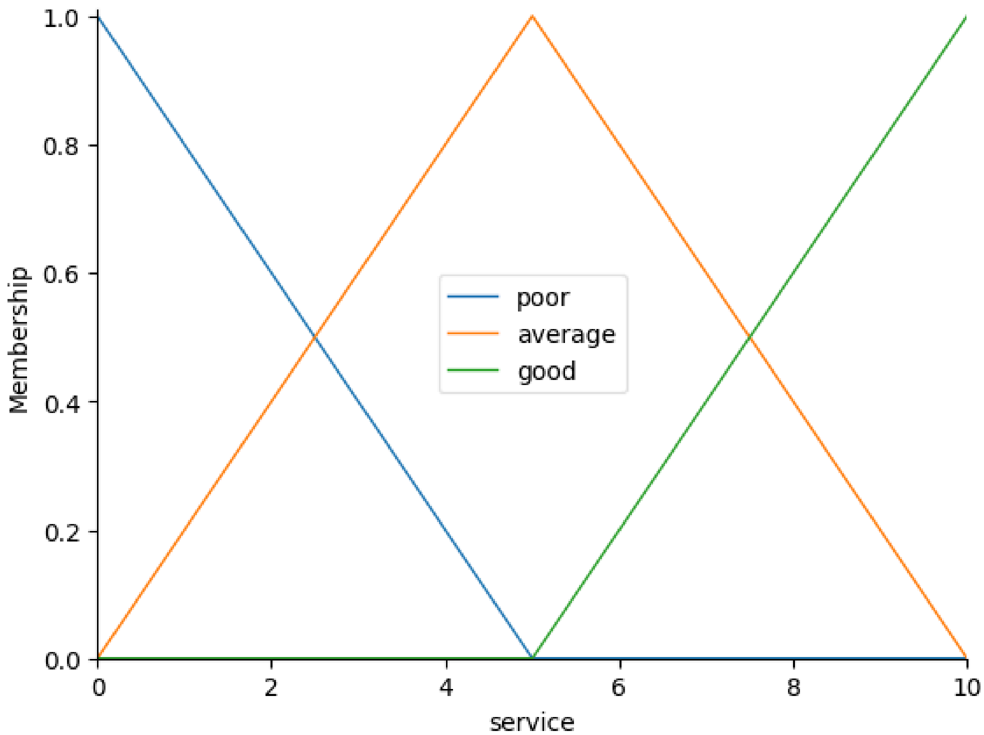
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# 1. Define Linguistic Variables (Inputs and Output)

# Example: Tipping problem - Quality of food, Service, and Tip

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')

# 2. Define Membership Functions

quality ['poor'] = fuzz. trimf (quality. universe, [0, 0, 5])
quality ['average'] = fuzz. trimf (quality. universe, [0, 5, 10])
quality ['good'] = fuzz. trimf (quality.universe, [5, 10, 10])


service['poor'] = fuzz. trimf (service.universe, [0, 0, 5])
service ['average'] = fuzz. trimf (service.universe, [0, 5, 10])
service ['good'] = fuzz. trimf (service. universe, [5, 10, 10])

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])

# 3. Define Fuzzy Rules
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(quality['good'] | service['good'], tip['high'])

# 4. Build the Control System
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

tipping_simulation = ctrl.ControlSystemSimulation (tipping_ctrl)
```

```python
# 5. Provide Inputs and Compute Output
tipping_simulation. input ['quality' ] = 0
tipping_simulation. input ['service'] = 0
tipping_simulation. compute ()

# 6. Display Result
print (f"Predicted Tip: {tipping_simulation. output['tip'] :.2f}%")

# Optional: View the membership functions and the result
quality.view()
service.view()
tip.view(sim=tipping_simulation)
```

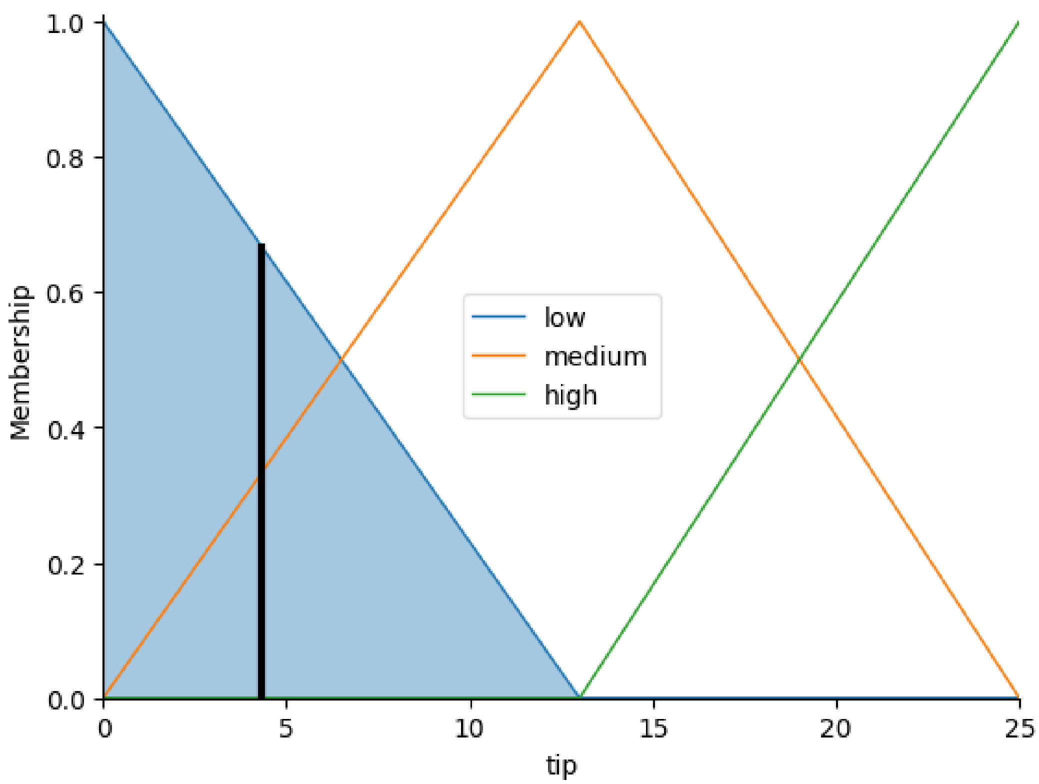
Output:

```
Predicted Tip: 4.33%
```





1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
service
poor
average
good



1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
5
10
15
20
25
tip
low
medium
high

7) Implementation of Simple genetic algorithm using its following operators:
I. Selection
II. Mutation
II. Crossover

Code:

```
from collections import defaultdict
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
import random
import pandas as pd
import matplotlib.pyplot as plt

# Load dataset
df = pd.read_excel("/content/heart.xlsx")
print(df.describe())

# Separate features and target
data_array = df.drop(columns=['target']).values
target_array = df['target'].values
X = data_array
y = target_array
n_features = X.shape[1]

# Standardize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# GA parameters
population_size = 10
n_generations = 10
mutation_rate = 0.1

# Create initial population
def initial_population(size, n_features):
```

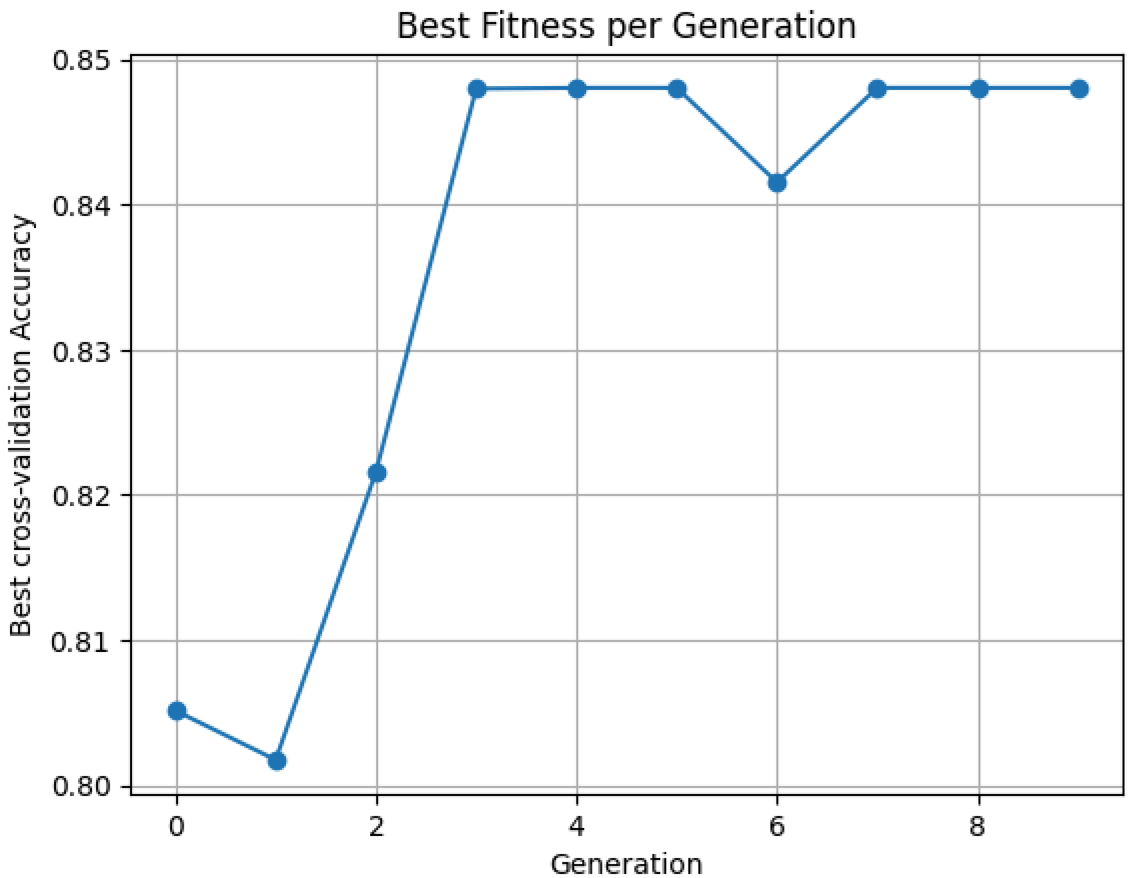
```
    return [np.random.choice([0, 1], size=n_features) for _ in
range(size)]

# Fitness function
def compute_fitness(individual):
    if np.count_nonzero(individual) == 0:
        return 0
    selected_features = X[:, individual == 1]
    model = LogisticRegression(max_iter=500)
    score = cross_val_score(model, selected_features, y, cv=5)
    return score.mean()

# Tournament selection
def select(population, fitnesses, k=3):
    selected = []
    for _ in range(len(population)):
        aspirants = random.sample(list(zip(population, fitnesses)), k)
        selected.append(max(aspirants, key=lambda x: x[1])[0])
    return selected

# Crossover
def crossover(parent1, parent2):
    point = random.randint(1, n_features - 1)
    child1 = np.concatenate((parent1[:point], parent2[point:]))
    child2 = np.concatenate((parent2[:point], parent1[point:]))
    return child1, child2

# Mutation
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]
    return individual

# Run Genetic Algorithm
population = initial_population(population_size, n_features)
print("Initial population:", population)

best_fitness_per_gen = []
```

```
for generation in range(n_generations):
    fitnesses = [compute_fitness(ind) for ind in population]
    best_fitness = max(fitnesses)
    best_fitness_per_gen.append(best_fitness)

    print(f"Generation {generation}, Best Fitness: {best_fitness:.4f}")

    selected = select(population, fitnesses)
    next_population = []
    for i in range(0, population_size, 2):
        p1 = selected[i]
        p2 = selected[min(i + 1, population_size - 1)]
        c1, c2 = crossover(p1, p2)
        next_population.extend([mutate(c1, mutation_rate), mutate(c2,
mutation_rate)])
    population = next_population

# Final evaluation
final_fitnesses = [compute_fitness(ind) for ind in population]
best_index = np.argmax(final_fitnesses)
best_individual = population[best_index]

# Feature selection
feature_names = df.drop(columns=['target']).columns.tolist()
selected_features = np.array(feature_names)[best_individual == 1]

# Plot results
plt.plot(best_fitness_per_gen, marker='o')
plt.xlabel("Generation")
plt.ylabel("Best cross-validation Accuracy")
plt.title("Best Fitness per Generation")
plt.grid(True)
plt.show()

# Best individual info
print(f"Selected feature mask: {best_individual}")
print("Best individual:", best_individual)
print("Selected features:", selected_features.tolist())
print("Total features selected:", np.sum(best_individual))
```

Output:

```
              age         sex          cp    trestbps        chol         fbs  \
count  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000
mean    54.366337    0.683168    0.966997  131.623762  246.264026    0.148515
std      9.082101    0.466011    1.032052   17.538143   51.830751    0.356198
min     29.000000    0.000000    0.000000   94.000000  126.000000    0.000000
25%     47.500000    0.000000    0.000000  120.000000  211.000000    0.000000
50%     55.000000    1.000000    1.000000  130.000000  240.000000    0.000000
75%     61.000000    1.000000    2.000000  140.000000  274.500000    0.000000
max     77.000000    1.000000    3.000000  200.000000  564.000000    1.000000

          restecg     thalach       exang     oldpeak       slope          ca  \
count  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000
mean     0.528053  149.646865    0.326733    1.039604    1.399340    0.729373
std      0.525860   22.905161    0.469794    1.161075    0.616226    1.022606
min      0.000000   71.000000    0.000000    0.000000    0.000000    0.000000
25%      0.000000  133.500000    0.000000    0.000000    1.000000    0.000000
50%      1.000000  153.000000    0.000000    0.800000    1.000000    0.000000
75%      1.000000  166.000000    1.000000    1.600000    2.000000    1.000000
max      2.000000  202.000000    1.000000    6.200000    2.000000    4.000000

             thal      target
count  303.000000  303.000000
mean     2.313531    0.544554
std      0.612277    0.498835
min      0.000000    0.000000
25%      2.000000    0.000000
50%      2.000000    1.000000
75%      3.000000    1.000000
max      3.000000    1.000000
```

```
Initial population: [array([0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0]), array([1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1]), array([1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1]), array([1,
Generation 0, Best Fitness: 0.8051
Generation 1, Best Fitness: 0.8017
Generation 2, Best Fitness: 0.8216
Generation 3, Best Fitness: 0.8480
Generation 4, Best Fitness: 0.8480
Generation 5, Best Fitness: 0.8480
Generation 6, Best Fitness: 0.8415
Generation 7, Best Fitness: 0.8480
Generation 8, Best Fitness: 0.8480
Generation 9, Best Fitness: 0.8480
```



Generation

```
Selected feature mask: [0 1 1 0 1 0 1 1 0 1 0 1 1]
Best individual: [0 1 1 0 1 0 1 1 0 1 0 1 1]
Selected features: ['sex', 'cp', 'chol', 'restecg', 'thalach', 'oldpeak', 'ca', 'thal']
Total features selected: 8
```

8) Implement Ant Colony Optimization Technique