Niyati_Savant_TE_C31_2103156

Experiment 5

AIM - Implement Genetics / Hill Climbing in Python.

Code – Genetics

```
import numpy as np

population_size = 6
gene_length = 4
num_generations = 100
crossover_rate = 0.5
mutation_rate = 0.1
num_parents = int(population_size / 2)

print("Niyati's Code for Genetics algorithm")

def initialize_population(size, gene_length):
    return np.random.randint(0, 31, (size, gene_length))

def calculate_fitness(chromosome):
    objective_value = abs(sum([chromosome[i] * (i+1) for i in range(len(chromosome))]) - 30)
    fitness_value = 1 / (1 + objective_value)  # Modified to match the provided logic
    return fitness_value

def select_parents(population, fitness, num_parents):
    fitness_sum = np.sum(fitness)
    probability = fitness / fitness_sum
    chosen = set()  # Set to keep track of which individuals have been chosen

    parents = np.empty((num_parents, population.shape[1]))
    for parent_num in range(num_parents):
        rand = np.random.rand()
        cumulative_probability = 0.0
        for i in range(len(probability)):
```

```
            if i not in chosen:
                cumulative_probability += probability[i]
                if rand <= cumulative_probability:
                    parents[parent_num, :] = population[i, :]
                    chosen.add(i)  # Mark this individual as chosen
                    break
    return parents



def crossover(parents, offspring_size, crossover_rate):
    offspring = np.empty(offspring_size)
    for k in range(offspring_size[0]):
        if np.random.rand() < crossover_rate:
            parent1_idx = k % parents.shape[0]
            parent2_idx = (k+1) % parents.shape[0]
            crossover_point = np.random.randint(1, offspring_size[1])
            offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
            offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring


def mutate(offspring_crossover, mutation_rate):
    for idx in range(offspring_crossover.shape[0]):
        for gene in range(offspring_crossover.shape[1]):
            if np.random.rand() < mutation_rate:
                offspring_crossover[idx, gene] = np.random.randint(0, 31)
    return offspring_crossover


population = initialize_population(population_size, gene_length)
print("Initial Population:\n", population)


for generation in range(num_generations):
    fitness = np.array([calculate_fitness(individual) for individual in population])
```

```
    print(f"\nGeneration {generation} Fitness:\n", fitness)
    parents = select_parents(population, fitness, num_parents)
    print("Selected Parents:\n", parents)
    offspring_crossover = crossover(parents, (population_size - num_parents, gene_length), crossover_rate)
    print("Crossover Offspring:\n", offspring_crossover)
    offspring_mutation = mutate(offspring_crossover, mutation_rate)
    print("Mutated Offspring:\n", offspring_mutation)
    population[:num_parents, :] = parents
    population[num_parents:, :] = offspring_mutation
    break
def genetic_algorithm(population, population_size, gene_length, num_generations, crossover_rate, mutation_rate, num_parents):


    for generation in range(num_generations):
        fitness = np.array([calculate_fitness(individual) for individual in population])
        parents = select_parents(population, fitness, num_parents)
        offspring_crossover = crossover(parents, (population_size - num_parents, gene_length), crossover_rate)
        offspring_mutation = mutate(offspring_crossover, mutation_rate)


        population[:num_parents, :] = parents
        population[num_parents:, :] = offspring_mutation


    final_fitness = np.array([calculate_fitness(individual) for individual in population])
    best_index = np.argmax(final_fitness)  # Use argmax because we are using inverted fitness values
    best_solution = population[best_index]


    print("\nFinal Best Solution:\n", best_solution)
    print("With Fitness Score:", final_fitness[best_index])



genetic_algorithm(population, population_size, gene_length, num_generations, crossover_rate, mutation_rate,num_parents)
```

Niyati_Savant_TE_C31_2103156

Output –

```
Niyati's Code for Genetics algorithm
Initial Population:
 [[28 14 25 10]
 [16  5 17 15]
 [ 4  9 10 24]
 [29 20 27  1]
 [16 15  3  0]
 [ 7  1  0 23]]

Generation 0 Fitness:
 [0.00704225 0.00925926 0.00840336 0.008      0.03846154 0.01388889]
Selected Parents:
 [[2.90000000e+001 2.00000000e+001 2.70000000e+001 1.00000000e+000]
 [1.60000000e+001 1.50000000e+001 3.00000000e+000 0.00000000e+000]
 [3.56043054e-307 4.45037520e-307 3.11521375e-307 2.78145267e-307]]
Crossover Offspring:
 [[2.90000000e+001 2.00000000e+001 2.70000000e+001 1.00000000e+000]
 [1.60000000e+001 1.50000000e+001 3.00000000e+000 0.00000000e+000]
 [3.56043054e-307 2.00000000e+001 2.70000000e+001 1.00000000e+000]]
Mutated Offspring:
 [[2.90000000e+001 2.00000000e+001 2.70000000e+001 1.00000000e+000]
 [1.60000000e+001 1.50000000e+001 3.00000000e+000 0.00000000e+000]
 [3.56043054e-307 2.00000000e+001 2.70000000e+001 1.00000000e+000]]

Final Best Solution:
 [ 8 11  0  0]
With Fitness Score: 1.0
PS C:\Engineering\3rd Year\Sem VI\PRACTICALS\AI>
```

Niyati_Savant_TE_C31_2103156

Code – Hill Climbing

```
import random


# Objective function to be maximized
def objective_function(x):
    return -x ** 2


# Generate initial solution randomly
def generate_initial_solution():
    return random.randint(-100, 100)


# Generate neighbour solutions
def generate_neighbours(solution):
    neighbours = []
    for delta in [-1, 1]:
        neighbours.append(solution + delta)
    return neighbours


# Get highest quality neighbour of current solution
def get_best_neighbour(neighbours):
    best_neighbour = neighbours[0]
    best_quality = objective_function(best_neighbour)
    for neighbour in neighbours[1:]:
        neighbour_quality = objective_function(neighbour)
        if neighbour_quality > best_quality:
            best_quality = neighbour_quality
            best_neighbour = neighbour
    return best_neighbour


# Hill climbing algorithm
def hill_climbing():
    current_solution = generate_initial_solution()
```

```
    # print("Initial Solution: ", current_solution)
    while True:
        neighbours = generate_neighbours(current_solution)
        best_neighbour = get_best_neighbour(neighbours)
        if objective_function(best_neighbour) <= objective_function(current_solution):
            return current_solution
        current_solution = best_neighbour



best_solution = hill_climbing()
print("Niyati's Code for Hill Climbing")
print("Best solution found:", best_solution)
print("Objective function value:", objective_function(best_solution))
```

Output –

```
PS C:\Engineering\3rd Year\Sem VI\PRACTICALS\AI> python -u "c:\Engineering\3rd Year\Sem
bing.py"
Niyati's Code for Hill Climbing
Best solution found: 0
Objective function value: 0
PS C:\Engineering\3rd Year\Sem VI\PRACTICALS\AI>
```