Design a distributed application using RPC for remote computation where client submits an integer value to the server and server calculates factorial and returns the result to the client program.

1st file:Factserver.py

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
class FactorialServer:
    def calculate_factorial(self, n):
        if n < 0:
            raise ValueError("Input must be a non-negative integer.")
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)
# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()
    # Register the FactorialServer class
    server.register_instance(FactorialServer())
    print("FactorialServer is ready to accept requests.")
    # Run the server's main loop
    server.serve_forever()
```

2nd file: Factclient.py

```
import xmlrpc.client
# Create an XML-RPC client
```

```python
with xmlrpc.client.ServerProxy("http://localhost:8000/RPC2") as proxy:
    try:
        # Replace 5 with the desired integer value
        input_value = 5
        result = proxy.calculate_factorial(input_value)
        print(f"Factorial of {input_value} is: {result}")
    except Exception as e:
        print(f"Error: {e}")
```

OUTPUT:

1. Open Command Prompt:

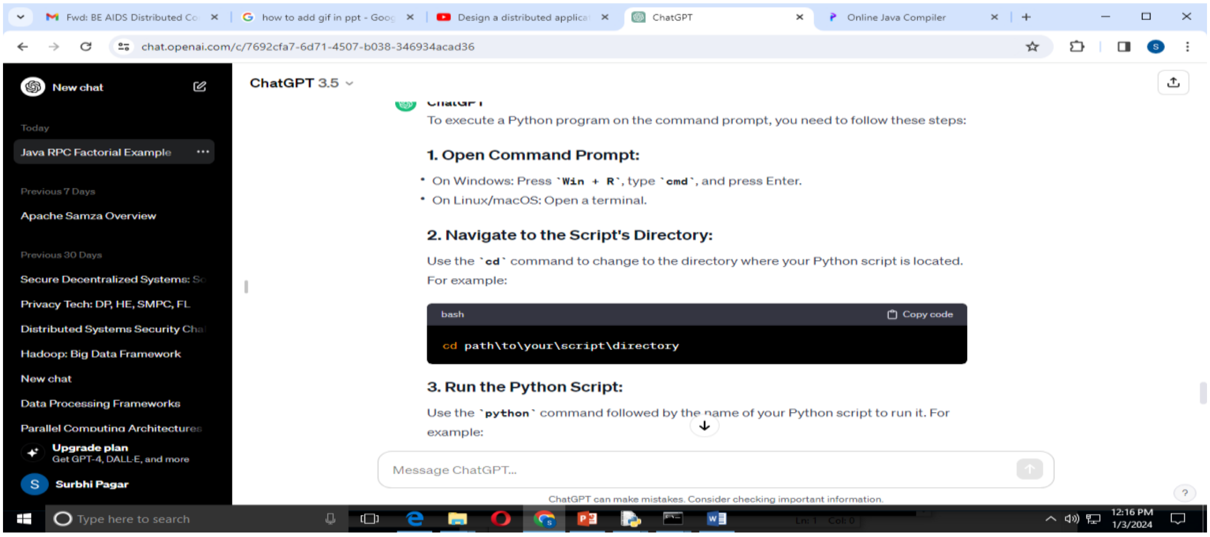
On Windows: Press Win + R, type cmd, and press Enter.

On Linux/macOS: Open a terminal.

2. Navigate to the Script's Directory:

Use the **cd command** to change to the directory where your Python script is located.

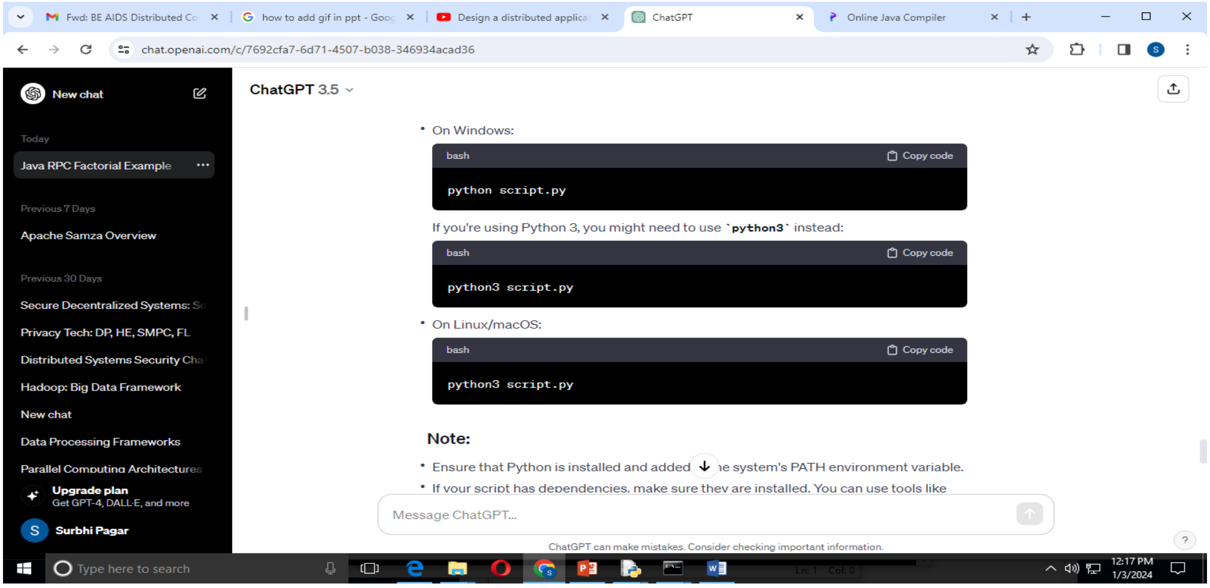
For example:



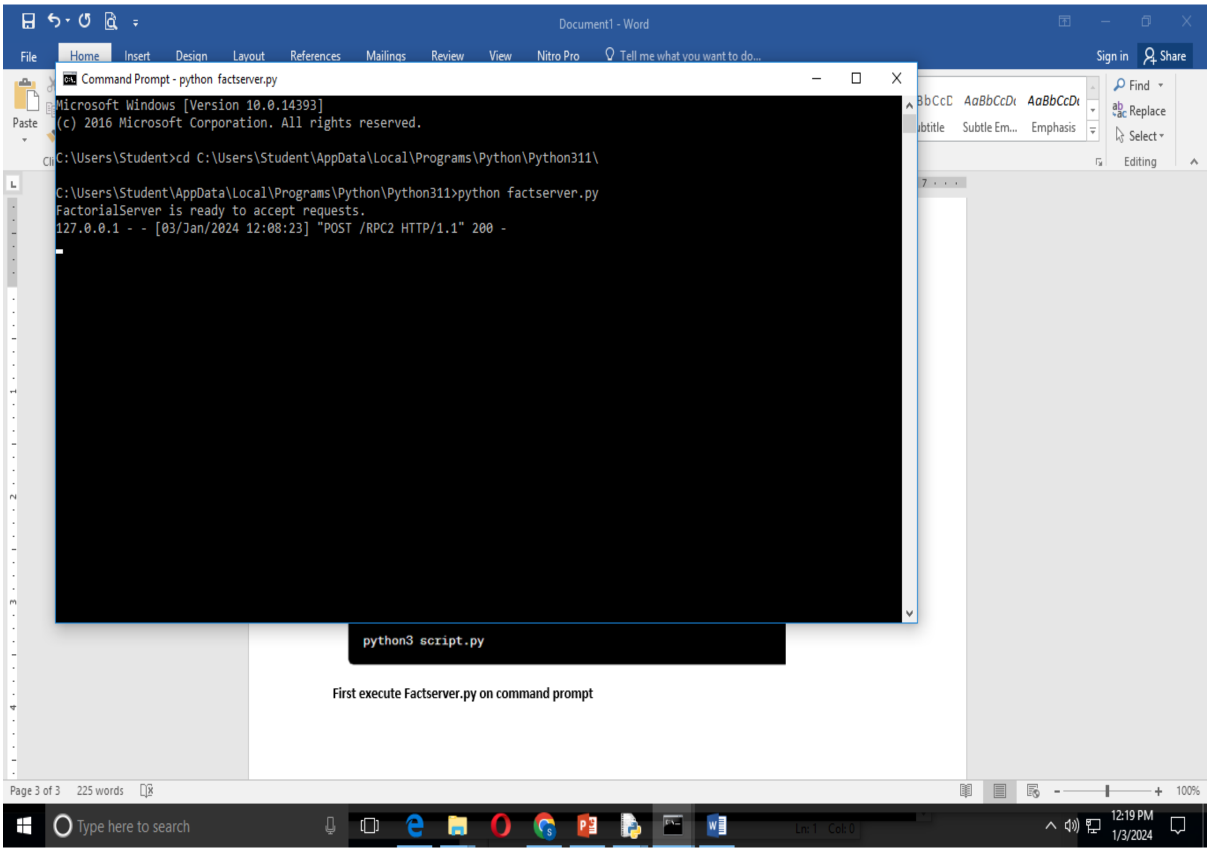
3. Run the Python Script:

Use the python command followed by the name of your Python script to run it.

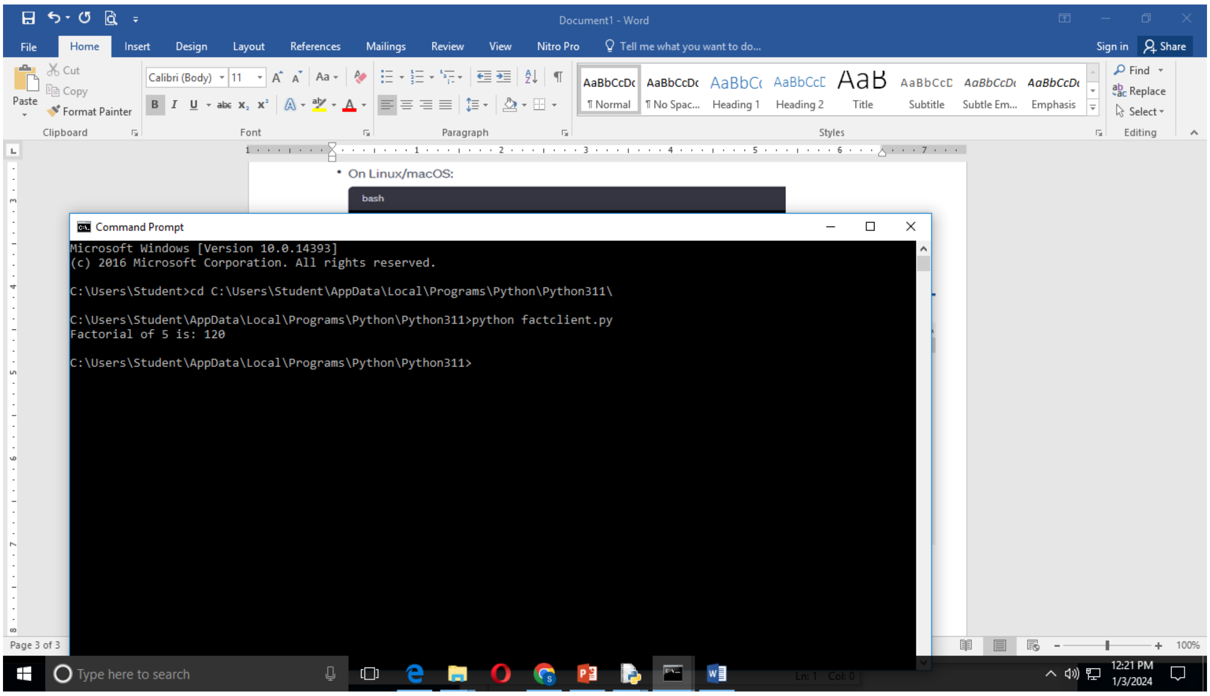
For example:



First execute factserver.py on command prompt



Then open another Command Prompt window and execute factclient.py



Document1 - Word
On Linux/macOS:
bash
Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\Users\Student>cd C:\Users\Student\AppData\Local\Programs\Python\Python311\
C:\Users\Student\AppData\Local\Programs\Python\Python311>python factclient.py
Factorial of 5 is: 120
C:\Users\Student\AppData\Local\Programs\Python\Python311>
Page 3 of 3
Type here to search
12:21 PM
1/3/2024

Design a distributed application using RMI for remote computation where client submits two strings to the server and server returns the concatenation of the given strings.

 # server.py

```
import Pyro4


@Pyro4.expose
class StringConcatenationServer:
    def concatenate_strings(self, str1, str2):
        result = str1 + str2
        return result


def main():
    daemon = Pyro4.Daemon()  # Create a Pyro daemon
    ns = Pyro4.locateNS()  # Locate the Pyro nameserver


    server = StringConcatenationServer()
    uri = daemon.register(server)
    ns.register("string.concatenation", uri)


    print("Server URI:", uri)


    with open("server_uri.txt", "w") as f:
        f.write(str(uri))


    daemon.requestLoop()


if __name__ == "__main__":
    main()
```
Client Implementation:

 # client.py

```
import Pyro4
```

```
def main():
    with open("server_uri.txt", "r") as f:
        uri = f.read()

    server = Pyro4.Proxy(uri)

    str1 = input("Enter the first string: ")
    str2 = input("Enter the second string: ")

    result = server.concatenate_strings(str1, str2)
    print("Concatenated Result:", result)

if __name__ == "__main__":
    main()
```

OUTPUT

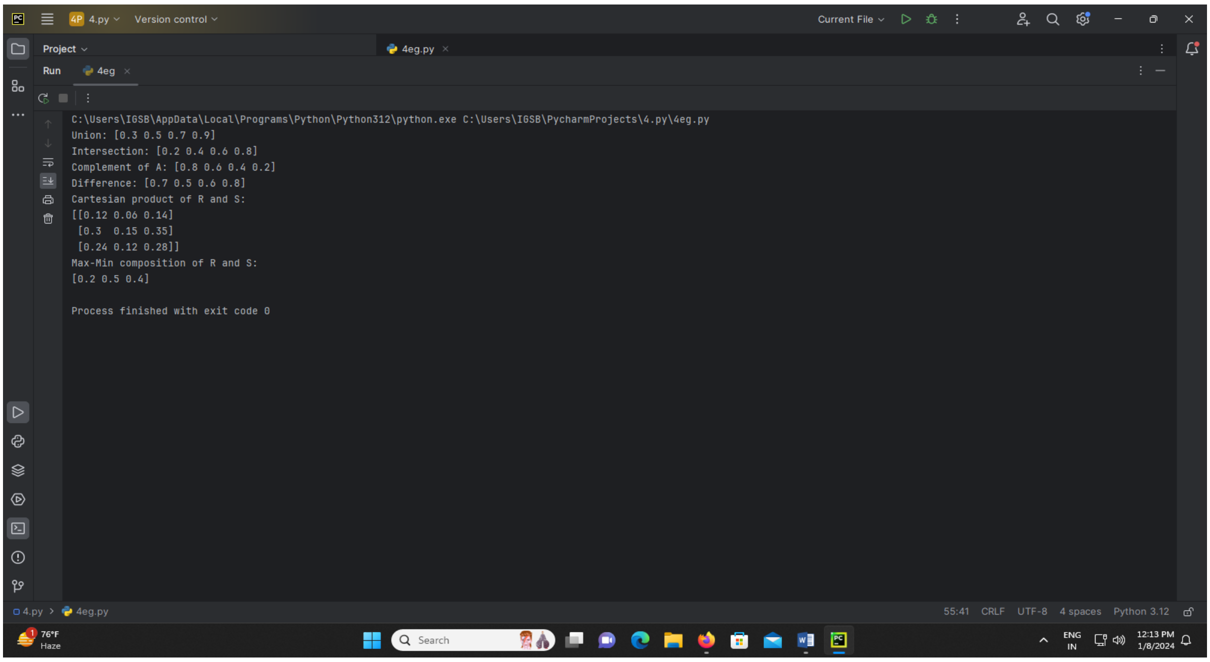
```
Enter the first string: Hello
Enter the second string: World
Concatenated Result: HelloWorld
```

Implement Union, Intersection, Complement and Difference operations on fuzzy sets. Also create fuzzy relations by Cartesian product of any two fuzzy sets and perform max-min composition on any two fuzzy relations.

CODE:

```
import numpy as np

# Function to perform Union operation on fuzzy sets
def fuzzy_union(A, B):
    return np.maximum(A, B)

# Function to perform Intersection operation on fuzzy sets
def fuzzy_intersection(A, B):
    return np.minimum(A, B)

# Function to perform Complement operation on a fuzzy set
def fuzzy_complement(A):
    return 1 - A

# Function to perform Difference operation on fuzzy sets
def fuzzy_difference(A, B):
    return np.maximum(A, 1 - B)

# Function to create fuzzy relation by Cartesian product of two fuzzy sets
def cartesian_product(A, B):
    return np.outer(A, B)

# Function to perform Max-Min composition on two fuzzy relations
def max_min_composition(R, S):
    return np.max(np.minimum.outer(R, S), axis=1)

# Example usage
```

```
A = np.array([0.2, 0.4, 0.6, 0.8])  # Fuzzy set A
B = np.array([0.3, 0.5, 0.7, 0.9])  # Fuzzy set B

# Operations on fuzzy sets
union_result = fuzzy_union(A, B)
intersection_result = fuzzy_intersection(A, B)
complement_A = fuzzy_complement(A)
difference_result = fuzzy_difference(A, B)

print("Union:", union_result)
print("Intersection:", intersection_result)
print("Complement of A:", complement_A)
print("Difference:", difference_result)

# Fuzzy relations
R = np.array([0.2, 0.5, 0.4])  # Fuzzy relation R
S = np.array([0.6, 0.3, 0.7])  # Fuzzy relation S

# Cartesian product of fuzzy relations
cartesian_result = cartesian_product(R, S)

# Max-Min composition of fuzzy relations
composition_result = max_min_composition(R, S)

print("Cartesian product of R and S:")
print(cartesian_result)

print("Max-Min composition of R and S:")
print(composition_result)
```



USE PYCHARM IDE

INSTALL NUMPY PACKAGE

Write code to simulate requests coming from clients and distribute them among the servers using the load balancing algorithms.

```
import random


class LoadBalancer:
   def __init__(self, servers):
      self.servers = servers
      self.server_index_rr = 0


   def round_robin(self):
      server = self.servers[self.server_index_rr]
      self.server_index_rr = (self.server_index_rr + 1) % len(self.servers)
      return server


   def random_selection(self):
      return random.choice(self.servers)


def simulate_client_requests(load_balancer, num_requests):
   for i in range(num_requests):
      # Simulating client request
      print(f"Request {i+1}: ", end="")


      # Using Round Robin algorithm for load balancing
      server_rr = load_balancer.round_robin()
      print(f"Round Robin - Server {server_rr}")


      # Using Random algorithm for load balancing
      server_random = load_balancer.random_selection()
      print(f"Random - Server {server_random}")


      print()
```

```python
if __name__ == "__main__":
    # List of servers
    servers = ["Server A", "Server B", "Server C"]

    # Create a LoadBalancer instance
    load_balancer = LoadBalancer(servers)

    # Simulate 10 client requests
    simulate_client_requests(load_balancer, 10)
```

// OUPUT Online Execution on PyCharm IDE Shows Us

```
Request 10: Round Robin - Server Server A Random - Server Server B

** Process exited - Return Code: 0 **

Press Enter to exit terminal
```

Optimization of genetic algorithm parameter in hybrid genetic algorithm-neural network modelling: Application to spray drying of coconut milk.

CODE: (imp: pip install deap)

```
import random
from deap import base, creator, tools, algorithms

# Define evaluation function (this is a mock function, replace this with your actual evaluation function)
def evaluate(individual):
    # Here 'individual' represents the parameters for the neural network
    # You'll need to replace this with your actual evaluation function that trains the neural network and evaluates its performance
    # Return a fitness value (here, a random number is used as an example)
    return random.random(),

# Define genetic algorithm parameters
POPULATION_SIZE = 10
GENERATIONS = 5

# Create types for fitness and individuals in the genetic algorithm
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

# Initialize toolbox
toolbox = base.Toolbox()

# Define attributes and individuals
toolbox.register("attr_neurons", random.randint, 1, 100)  # Example: number of neurons
toolbox.register("attr_layers", random.randint, 1, 5)  # Example: number of layers
```

```
toolbox.register("individual", tools.initCycle, creator.Individual, (toolbox.attr_neurons,
toolbox.attr_layers), n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Genetic operators
toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=1, up=100, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Create initial population
population = toolbox.population(n=POPULATION_SIZE)

# Run the genetic algorithm
for gen in range(GENERATIONS):
  offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)

  fitnesses = toolbox.map(toolbox.evaluate, offspring)
  for ind, fit in zip(offspring, fitnesses):
    ind.fitness.values = fit

  population = toolbox.select(offspring, k=len(population))

# Get the best individual from the final population
best_individual = tools.selBest(population, k=1)[0]
best_params = best_individual

# Print the best parameters found
print("Best Parameters:", best_params)
```

```
Best Parameters: [53, 3]
```

```
C:\ProgramData\anaconda3\Lib\site-packages\deap\creator.py:185: RuntimeWarning: A class named 'FitnessMin' has already bee
n created and it will be overwritten. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
C:\ProgramData\anaconda3\Lib\site-packages\deap\creator.py:185: RuntimeWarning: A class named 'Individual' has already bee
n created and it will be overwritten. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
```

USE JUPYTER NOTEBOOK IDE

Title : Implementation of Clonal selection algorithm using Python

** CODE ***

```
import random
import numpy as np


# Define objective function (Sphere function for minimization)
def objective_function(x):
  return sum(i**2 for i in x)  # Sphere function (minimization)


# Clonal Selection Algorithm
def clonal_selection_algorithm(dimensions, num_candidates, num_clones, mutation_rate,
max_iterations):
  # Initialize population with random solutions
  population = [np.random.uniform(-5, 5, dimensions) for _ in range(num_candidates)]


  for iteration in range(max_iterations):
     # Evaluate fitness of each candidate solution
     fitness = [objective_function(candidate) for candidate in population]


     # Sort candidates by fitness (minimization problem)
     sorted_indices = np.argsort(fitness)
     population = [population[i] for i in sorted_indices]
     fitness = [fitness[i] for i in sorted_indices]


     # Select top candidates for cloning (proportional to fitness)
     selected_candidates = population[:num_clones]


     # Clone candidates (proportional to their ranking)
     clones = []
     for i, candidate in enumerate(selected_candidates):
        num_clones_for_candidate = int(num_clones * (1 - i / len(selected_candidates)))  #
More clones for better candidates
        clones.extend([candidate.copy() for _ in range(num_clones_for_candidate)])


     # Mutate clones
     for i in range(len(clones)):
        for j in range(dimensions):
```

```
            if random.random() < mutation_rate:
                clones[i][j] += np.random.normal(0, 0.2)  # Gaussian mutation

        # Evaluate fitness of clones
        clone_fitness = [objective_function(clone) for clone in clones]

        # Select the best candidates among original and clones (elitism)
        combined_population = population + clones
        combined_fitness = fitness + clone_fitness

        # Sort combined population by fitness
        sorted_indices = np.argsort(combined_fitness)
        population = [combined_population[i] for i in sorted_indices[:num_candidates]]

        # Output best solution in this iteration
        print(f"Iteration {iteration + 1}: Best solution - {population[0]}, Fitness -
{combined_fitness[sorted_indices[0]]}")

    # Return best solution found
    best_solution = population[0]
    best_fitness = objective_function(best_solution)
    return best_solution, best_fitness

# Example usage
dimensions = 3
num_candidates = 20
num_clones = 10
mutation_rate = 0.1
max_iterations = 50
best_solution, best_fitness = clonal_selection_algorithm(dimensions, num_candidates,
num_clones, mutation_rate, max_iterations)
print("\nFinal Best Solution:", best_solution)
print("Final Best Fitness:", best_fitness)
```

************************************** OUTPUT **

/home/aids/PycharmProjects/DC_prac5/.venv/bin/python
/home/aids/PycharmProjects/DC_prac5/Clonal_Selection.py
Iteration 1: Best solution - [1.49222394 -1.16876067 -0.23274204], Fitness - 3.6469026548879593
Iteration 2: Best solution - [1.19807954 -1.16876067 -0.23274204], Fitness - 2.8555649509873584
Iteration 3: Best solution - [1.12856552 -0.98906853 -0.00294783], Fitness - 2.2519253632274894
Iteration 4: Best solution - [1.12856552 -0.98906853 -0.00294783], Fitness - 2.2519253632274894
Iteration 5: Best solution - [0.76323712 -0.98906853 -0.00294783], Fitness - 1.5607961357894975
Iteration 6: Best solution - [0.76323712 -0.83763667 -0.00294783], Fitness - 1.2841747753224635
Iteration 7: Best solution - [0.76323712 -0.63458323 -0.00294783], Fitness - 0.9852354670697464
Iteration 8: Best solution - [0.62487892 -0.63458323 -0.00294783], Fitness - 0.7931782281466417
Iteration 9: Best solution - [0.03476966 -0.63458323 -0.00294783], Fitness - 0.40391349865819093
…….
Iteration 49: Best solution - [0.00017947 -0.0013291 -0.00294783], Fitness - 1.0488403290383097e-05
Iteration 50: Best solution - [0.00017947 -0.0013291 -0.00294783], Fitness - 1.0488403290383097e-05

Final Best Solution: [0.00017947 -0.0013291 -0.00294783]
Final Best Fitness: 1.0488403290383097e-05

Program Code:

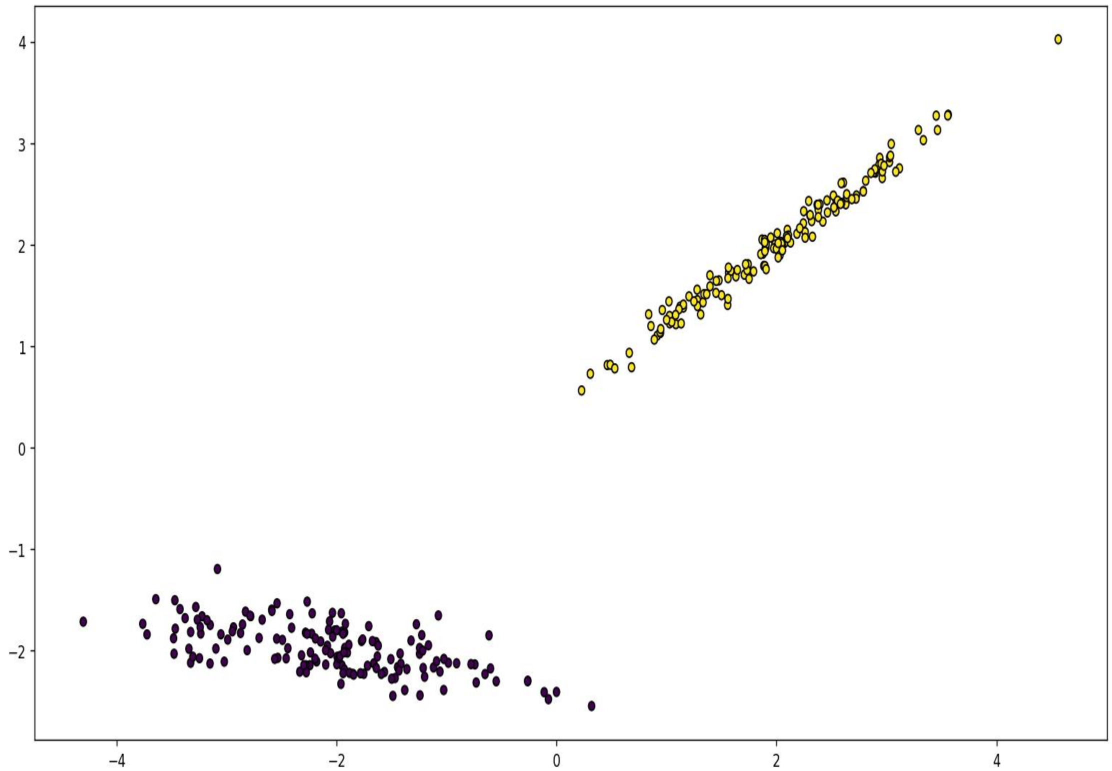
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification

from sklearn.model_selection import train_test_split


# Step 1: Generate synthetic data

X, y = make_classification(n_samples=300, n_features=2, n_informative=2,
n_redundant=0, n_clusters_per_class=1, weights=[0.5], flip_y=0, class_sep=2)

# Visualizing the data

plt.scatter(X[:, 0], X[:, 1], marker='o', c=y, s=25, edgecolor='k')

plt.show()


# Splitting data into train (self) and test (new samples, possibly non-self)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Step 2: NSA Implementation
class NegativeSelectionClassifier:
    def __init__(self, radius=0.5):
        self.radius = radius
        self.detectors = []

    def fit(self, X_train, y_train):
        # Generate detectors
        self.detectors = []
        for point in X_train[y_train == 0]:  # Consider only 'not damaged' (self) data
            new_detectors = True
            for other_point in X_train[y_train == 0]:
                if np.linalg.norm(point - other_point) < self.radius:
                    new_detectors = False
                    break
            if new_detectors:
                self.detectors.append(point)

    def predict(self, X_test):
        predictions = []
        for test_point in X_test:
            nonself = False
            for detector in self.detectors:
```

```
            if np.linalg.norm(test_point - detector) < self.radius:
                nonself = True
                break
        predictions.append(1 if nonself else 0)
    return predictions

# Train the NSA
nsa = NegativeSelectionClassifier(radius=0.5)
nsa.fit(X_train, y_train)

# Test the NSA
y_pred = nsa.predict(X_test)

# Visualizing the results
plt.scatter(X_test[:, 0], X_test[:, 1], marker='o', c=y_pred, s=25, edgecolor='k')
plt.title("Test results")
plt.show()

# Print accuracy (naively calculated)
accuracy = sum(y_pred == y_test) / len(y_test)
print("Accuracy:", accuracy)
```

Output:

Accuracy: 0.5444444444444444


4
3
2
1
0
−1
−2
−4
−2
0
2
4

Code:

```
import random
from deap import base, creator, tools, algorithms

# Define the evaluation function (minimize a simple mathematical
function)
def eval_func(individual):
    # Example evaluation function (minimize a quadratic function)
    return sum(x ** 2 for x in individual),

# DEAP setup
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Define attributes and individuals
toolbox.register("attr_float", random.uniform, -5.0, 5.0)  # Example:
Float values between -5 and 5
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_float, n=3)  # Example: 3-dimensional individual
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

# Evaluation function and genetic operators
toolbox.register("evaluate", eval_func)
toolbox.register("mate", tools.cxBlend, alpha=0.5)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1,
indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Create population
population = toolbox.population(n=50)

# Genetic Algorithm parameters
generations = 20

# Run the algorithm
for gen in range(generations):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5,
mutpb=0.1)

    fits = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
```

```
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

# Get the best individual after generations
best_ind = tools.selBest(population, k=1)[0]
best_fitness = best_ind.fitness.values[0]

print("Best individual:", best_ind)
print("Best fitness:", best_fitness)
```

Output:

```
PS D:\BE SEM VIII> python -u "d:\BE SEM VIII\CL_III_Code\DEAP.py"
Best individual: [-0.011174776506688588, -0.006348837481336l935, -0.03303542448457376 4]
Best fitness: 0.0012565226382148342
PS D:\BE SEM VIII>
```

Code:

```
import csv
from functools import reduce
from collections import defaultdict

# Define mapper function to emit (year, temperature) pairs
def mapper(row):
    year = row["Date/Time"].split("-")[0]  # Extract year from
"Date/Time" column
    temperature = float(row["Temp_C"])  # Convert temperature to
float
    return (year, temperature)

# Define reducer function to calculate sum and count of temperatures
for each year
def reducer(accumulated, current):
    accumulated[current[0]][0] += current[1]
    accumulated[current[0]][1] += 1
    return accumulated

# Read the weather dataset
weather_data = []
with open("weather_data.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        weather_data.append(row)

# Map phase
mapped_data = map(mapper, weather_data)

# Reduce phase
reduced_data = reduce(reducer, mapped_data, defaultdict(lambda: [0,
0]))

# Calculate average temperature for each year
avg_temp_per_year = {year: total_temp / count for year, (total_temp,
count) in reduced_data.items()}

# Find coolest and hottest year
coolest_year = min(avg_temp_per_year.items(), key=lambda x: x[1])
hottest_year = max(avg_temp_per_year.items(), key=lambda x: x[1])

print("Coolest Year:", coolest_year[0], "Average Temperature:",
coolest_year[1])
```

```
print("Hottest Year:", hottest_year[0], "Average Temperature:",
hottest_year[1])
```
Output:

```
Coolest Year: 1/15/2012 8:00 Average Temperature: -23.3
Hottest Year: 6/21/2012 15:00 Average Temperature: 33.0
```

Code:

```
import numpy as np
import random

# Define the distance matrix (distances between cities)
# Replace this with your distance matrix or generate one based on
your problem
# Example distance matrix (replace this with your actual data)
distance_matrix = np.array([
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
])

# Parameters for Ant Colony Optimization
num_ants = 10
num_iterations = 50
evaporation_rate = 0.5
pheromone_constant = 1.0
heuristic_constant = 1.0

# Initialize pheromone matrix and visibility matrix
num_cities = len(distance_matrix)
pheromone = np.ones((num_cities, num_cities))  # Pheromone matrix
visibility = 1 / distance_matrix  # Visibility matrix (inverse of
distance)

# ACO algorithm
for iteration in range(num_iterations):
    ant_routes = []
    for ant in range(num_ants):
        current_city = random.randint(0, num_cities - 1)
        visited_cities = [current_city]
        route = [current_city]

        while len(visited_cities) < num_cities:
            probabilities = []
            for city in range(num_cities):
                if city not in visited_cities:
                    pheromone_value = pheromone[current_city][city]
                    visibility_value = visibility[current_city][city]
                    probability = (pheromone_value **
pheromone_constant) * (visibility_value ** heuristic_constant)
                    probabilities.append((city, probability))
```

```
            probabilities = sorted(probabilities, key=lambda x: x[1],
reverse=True)
            selected_city = probabilities[0][0]
            route.append(selected_city)
            visited_cities.append(selected_city)
            current_city = selected_city

        ant_routes.append(route)

    # Update pheromone levels
    delta_pheromone = np.zeros((num_cities, num_cities))

    for ant, route in enumerate(ant_routes):
        for i in range(len(route) - 1):
            city_a = route[i]
            city_b = route[i + 1]
            delta_pheromone[city_a][city_b] += 1 /
distance_matrix[city_a][city_b]
            delta_pheromone[city_b][city_a] += 1 /
distance_matrix[city_a][city_b]

    pheromone = (1 - evaporation_rate) * pheromone + delta_pheromone

# Find the best route
best_route_index =
np.argmax([sum(distance_matrix[cities[i]][cities[(i + 1) %
num_cities]] for i in range(num_cities)) for cities in ant_routes])
best_route = ant_routes[best_route_index]
shortest_distance = sum(distance_matrix[best_route[i]][best_route[(i
+ 1) % num_cities]] for i in range(num_cities))

print("Best route:", best_route)
print("Shortest distance:", shortest_distance)
```

Output:

```
PS D:\BE SEM VIII> python -u "d:\BE SEM VIII\CL_III_Code\TSP.py"
d:\BE SEM VIII\CL_III_Code\TSP.py:24: RuntimeWarning: divide by zero encountered in divide
  visibility = 1 / distance_matrix  # Visibility matrix (inverse of distance)
Best route: [0, 1, 3, 2]
Shortest distance: 80
PS D:\BE SEM VIII>
```