Assignment No. 1

Aim: To apply the artificial immune pattern recognition to perform a task of structure damage Classification.

Objectives:

The primary objective of applying an artificial immune system (AIS) to the task of structural damage classification is to develop a robust and adaptive algorithm capable of accurately identifying and categorizing various types of structural impairments using pattern recognition techniques inspired by biological immune responses. This project aims to leverage the unique properties of AIS, such as feature extraction, anomaly detection, and continuous learning, to enhance the sensitivity and specificity of structural damage detection in diverse conditions and environments.

Theory:

an Artificial Immune System (AIS) for the task of structural damage classification, we will focus on a simplified scenario where our goal is to classify structures as either 'damaged' or 'not damaged'. Artificial Immune Systems are computational systems inspired by the principles and processes of the biological immune system. They can be particularly effective in pattern recognition tasks, including anomaly detection and classification challenges.

In this example, we will use a simplified AIS algorithm known as a Negative Selection Algorithm (NSA). The NSA is based on the principle that immune cells that react to self-cells are eliminated during the development of T-cells in the thymus. In our context, 'self' will represent the 'not damaged' structures, and 'non-self' will represent the 'damaged' structures.

Step 1: Generating Data

Let's first generate some synthetic data for our task. For simplicity, we will create a dataset with features that might relate to structural integrity such as stress levels, vibration frequencies, and temperature readings. Structures will be labeled as '0' for not damaged and '1' for damaged.

Step 2: Implementing the NSA

We will then implement a basic Negative Selection Algorithm, where we will generate detectors that do not match the normal ('not damaged') data points. These detectors will then be used to identify 'damaged' structures.

Limitations

The simplicity of this model means it may not handle complex structural damage scenarios without modification and additional feature engineering.

Real-world applications would require calibration with real data and potentially more sophisticated immune-inspired algorithms or hybrid approaches.

Parameter tuning (like the radius in NSA) and validation are crucial for achieving high performance in practical scenarios.

Conclusion:

An artificial immune pattern recognition to perform a task of structure damage

Classification proves to be a promising approach

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

Assignment No. 2

Aim: Implement DEAP (Distributed Evolutionary Algorithms) using Python.

Program Code:

Step 1: Installation

First, ensure you have Python installed on your system. You can then install DEAP using pip:

```
pip install deap
```

Step 2: Implementing a Genetic Algorithm

Let's solve a simple problem using a genetic algorithm with DEAP.

Step 2.1: Import Required Libraries

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

Step 2.2: Define the Fitness and Individual

In DEAP, you need to define a fitness strategy and the structure of individuals in the population:

```
# Define the fitness function (minimization)
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
# Create the individual class based on a list of floats and attach the fitness function
creator.create("Individual", list, fitness=creator.FitnessMin)
```

Step 2.3: Define the Toolbox

The toolbox in DEAP is where you register various functions including genetic operators:

```
toolbox = base.Toolbox()
```

```
# Attribute generator: defines how individual genes are generated
toolbox.register("attr_float", random.uniform, -5.12, 5.12)

# Structure initializers: define how individuals and the population are formed
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_float, n=10)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Step 2.4: Register the Genetic Operators

```
# Genetic operators
    toolbox.register("mate", tools.cxBlend, alpha=0.5)
    toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.1)
    toolbox.register("select", tools.selTournament, tournsize=3)
    toolbox.register("evaluate", lambda ind: (sum(x**2 for x in ind),))
```

Step 2.5: Main Evolutionary Algorithm

Set up the main evolutionary process and execute the genetic algorithm:

```
    def main():
        random.seed(64)
        pop = toolbox.population(n=300)
        hof = tools.HallOfFame(1)

        stats = tools.Statistics(lambda ind: ind.fitness.values)
        stats.register("avg", np.mean)
```

```
    stats.register("std", np.std)
    stats.register("min", np.min)
    stats.register("max", np.max)

    # Apply the genetic algorithm
    pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2,
ngen=40,
                                   stats=stats, halloffame=hof, verbose=True)

    return pop, log, hof

if __name__ == "__main__":
    pop, log, hof = main()
    print("Best individual is: ", hof[0])
    print("Best fitness is: ", hof[0].fitness.values)
```

Assignment No. 4

Aim: Design and develop a distributed application to find the coolest/hottest year from the available weather data. Use weather data from the Internet and process it using MapReduce.

Program Code:

Step 1: Setup Environment

Ensure that Hadoop is installed and configured on your cluster. Hadoop's ecosystem includes the HDFS (Hadoop Distributed File System) for storing data and the MapReduce framework for processing the data.

Download Hadoop: Visit Apache Hadoop's official website to download and install it.

Setup Hadoop: Configure the core-site.xml, hdfs-site.xml, and mapred-site.xml as required for your cluster.

Step 2: Obtain Weather Data

Obtain historical weather data, which typically comes in formats such as CSV or JSON. For this example, let's assume we have CSV data that includes at least the year, date, and temperature. Websites like NOAA or similar open data platforms can be a source of such data.

Data Format Example: year, date, average_temperature

Sample Data:

1997,1997-01-01,15.5

1997,1997-01-02,17.0

1998,1998-01-01,14.0

Step 3: Write MapReduce Program

The MapReduce program will process this data to find the hottest and coolest year based on average temperatures.

Mapper: Reads the weather data and outputs year and temperature pairs.

Reducer: Aggregates temperature by year and calculates the average temperature for each year, then determines the hottest and coolest years.

Mapper Code (Java)

```
import java.io.IOException;
```

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;

public class TemperatureMapper extends Mapper<LongWritable, Text, Text, FloatWritable> {

    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String[] line = value.toString().split(",");
        if (line.length > 2) {
            String year = line[0];
            float temp = Float.parseFloat(line[2]);
            context.write(new Text(year), new FloatWritable(temp));
        }
    }
}
```

Reducer Code (Java)

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;

public class TemperatureReducer extends Reducer<Text, FloatWritable, Text, FloatWritable> {

    public void reduce(Text key, Iterable<FloatWritable> values, Context context) throws
IOException, InterruptedException {
        float sumTemp = 0;
        int count = 0;
        for (FloatWritable val : values) {
            sumTemp += val.get();
```

```
            count++;
        }
        float averageTemp = sumTemp / count;
        context.write(key, new FloatWritable(averageTemp));
    }
}
```

Step 4: Driver Program

Set up the Hadoop job, specifying the input and output paths, as well as the mapper and reducer classes.

Driver Code (Java)

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TemperatureDriver {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: TemperatureDriver <input path> <output path>");
            System.exit(-1);
        }

        Job job = Job.getInstance();
        job.setJarByClass(TemperatureDriver.class);
        job.setJobName("Average Temperature");

        FileInputFormat.addInputPath(job, new Path(args[0]));
```

```
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(TemperatureMapper.class);
        job.setReducerClass(TemperatureReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FloatWritable.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Step 5: Run the MapReduce Job

Load the data into HDFS and then submit the MapReduce job.

```
hadoop fs -put local_data_path /user/hadoop/weatherdata
hadoop jar TemperatureDriver.jar TemperatureDriver /user/hadoop/weatherdata /user/hadoop/output
```

Step 6: Analysis and Results

After running, check the output directory for results to identify the hottest and coolest years.

```
hadoop fs -cat /user/hadoop/output/part-r-00000
```

Assignment No. 5

Aim: Implement Ant colony optimization by solving the Traveling salesman problem using python Problem statement- A salesman needs to visit a set of cities exactly once and return to the original city. The task is to find the shortest possible route that the salesman can take to visit all the cities and return to the starting city.

Program Code:

```
import numpy as np


class AntColonyOptimizer:
    def __init__(self, distances, n_ants, n_best, n_iterations, decay, alpha=1, beta=1):
        """
        Ant colony optimizer for solving TSP
        :param distances: 2D list or numpy array with distances between the cities
        :param n_ants: Number of ants to simulate
        :param n_best: Number of best ants who deposit pheromone
        :param n_iterations: Number of iterations
        :param decay: Rate at which pheromone decays. The lower the faster.
        :param alpha: Exponent on pheromone, higher more exploitation
        :param beta: Exponent on distance, higher more exploration
        """
        self.distances = distances
        self.pheromone = np.ones(self.distances.shape) / len(distances)
        self.all_inds = range(len(distances))
        self.n_ants = n_ants
        self.n_best = n_best
        self.n_iterations = n_iterations
```

```
        self.decay = decay
        self.alpha = alpha
        self.beta = beta

    def run(self):
        shortest_path = None
        best_cost = float('inf')
        for _ in range(self.n_iterations):
            all_paths = self.generate_all_paths()
            self.spread_pheromone(all_paths, self.n_best, shortest_path=shortest_path)
            shortest_path, best_cost = self.find_best_path(all_paths, best_cost,
shortest_path)
        return shortest_path, best_cost

    def spread_pheromone(self, all_paths, n_best, shortest_path):
        sorted_paths = sorted(all_paths, key=lambda x: x[1])
        for path, cost in sorted_paths[:n_best]:
            for move in path:
                self.pheromone[move] += 1.0 / self.distances[move]

    def generate_all_paths(self):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.generate_path(0)
            all_paths.append((path, self.path_cost(path)))
        return all_paths
```

```
    def generate_path(self, start):
        path = []
        visited = set()
        visited.add(start)
        prev = start
        for _ in range(len(self.distances) - 1):
            move = self.select_next_city(self.pheromone[prev], self.distances[prev],
visited)
            path.append((prev, move))
            prev = move
            visited.add(move)
        path.append((prev, start)) # return to start
        return path

    def select_next_city(self, pheromone, dist, visited):
        pheromone = np.copy(pheromone)
        pheromone[list(visited)] = 0
        row = pheromone ** self.alpha * ((1.0 / dist) ** self.beta)
        norm_row = row / row.sum()
        move = np_choice(self.all_inds, 1, p=norm_row)[0]
        return move

    def path_cost(self, path):
        return sum([self.distances[i][j] for i, j in path])
```

```
    def find_best_path(self, all_paths, best_cost, shortest_path):
        for path, cost in all_paths:
            if cost < best_cost:
                best_cost = cost
                shortest_path = path
        return shortest_path, best_cost

def np_choice(a, size, replace=True, p=None):
    return np.array(np.random.choice(a, size=size, replace=replace, p=p))

# Example usage:
distances = np.array([
    [0, 2, 9, 10],
    [1, 0, 6, 4],
    [15, 7, 0, 8],
    [6, 3, 12, 0]
])
aco = AntColonyOptimizer(distances, n_ants=10, n_best=5, n_iterations=100,
decay=0.1, alpha=1, beta=2)
path, cost = aco.run()
print('Shortest path:', path)
print('Cost of the path:', cost)
```

Key Components of This Implementation:

Initialization: Sets up pheromone levels and parameters for the algorithm.

Run Algorithm: For a fixed number of iterations, generate solutions and update pheromones based on the quality of solutions.

Generate Paths: Each ant generates a path based on pheromone levels and heuristic information (here inverse of distance).

Update Pheromones: After all ants have completed their paths, update pheromones by reducing them (decay) and increasing based on the quality of the best solutions.

Selection of Next City: Probabilistic selection based on pheromone levels and distance to the next city, adjusted by alpha and beta parameters.