# **Exercise Report**

## **Course: 06. Aprendizaje automático (Machine Learning): conceptos, metodología, algoritmos de aprendizaje para analítica descriptiva, predictiva y prescriptiva, y retos en su implementación**

## **Exercise Title:** Homework 1.A - Optimization (basic, obligatory).

## **Submitted by:** Jorge de la Torre Garcia (DNI), Lydia Phoebe Amanda Lilius (DNI), Miguel Galán Cisneros (DNI), Vitor Oliveira de Souza (Z0963220P).

### February 14th, 2023.

# Introduction

This report presents a detailed examination and enhancement of the provided optimization algorithm based on the genetic algorithm approach. The primary objective was to identify the aspects of the algorithm that were causing steadiness problems in its outputs and propose some improvements on its evolutionary operators aiming to achieve a consistently better-performing solution in each generation.

# Algorithm Analysis

The original script implements an evolutionary algorithm to optimize a given objective function, which is defined to maximize the Ackley function, a well-known benchmark in optimization problems. The algorithm employs standard evolutionary operators: selection (roulette wheel selection), crossover (simple averaging), mutation (random Gaussian perturbation), and replacement (entire generation replacement).

# Identified Issues:

Upon analysis, we identified that the algorithm's performance inconsistency across generations was coming from several factors:

1. Replacement Strategy: The main reason that is causing instability in the provided algorithm is the replacement of the entire population in every single step, regardless the performance of each individual, discarding potential good solutions across the generations.
2. Mutation Operators: Applying a small Gaussian noise as mutation did not introduce sufficient exploration, limiting the algorithm's ability to escape local optima.
3. Roulette Wheel Selection: This method could sometimes lead to premature convergence by favoring individuals with marginally better fitness.

# Modifications procedure:

Our approach to enhancing the genetic algorithm involved a series of strategic modifications, each aimed at addressing specific shortcomings identified during our initial analysis. The key modifications implemented are as follows:

1. Selective Replacement: To preserve genetic diversity within the population while ensuring progressive improvement, we adopted a selective replacement strategy. Rather than replacing the entire population each generation, we introduced a method where only the worst-performing individual is replaced, provided that the new candidate exhibits superior fitness. This approach balances the need for diversity with the imperative of continuous population improvement.
2. Crossover and Mutation Operators: The crossover mechanism was maintained due to its effectiveness in combining advantageous traits from parent individuals. However, we adjusted the mutation parameter to ±10% of the boundary difference, aiming to introduce a more dynamic range of genetic variations and enhance the algorithm's ability to explore the search space thoroughly.
3. Enhanced Selection Mechanism: To further encourage exploration and prevent stagnation at local minima, we incorporated a random selection strategy. By introducing randomness in the selection process, we aimed to diversify the genetic material undergoing crossover and mutation, thereby increasing the likelihood of discovering novel and potentially superior solutions.

The cross-validation process involved systematically evaluating the genetic algorithm's performance across a range of parameter settings, specifically focusing on selection methods ('roulette' and 'random') and mutation values (0.025, 0.05, 0.1, 0.2). This exhaustive evaluation, conducted over 200 runs for each parameter combination, provided a comprehensive dataset from which to derive insights into the algorithm's behavior and effectiveness under different configurations.

The culmination of our cross-validation effort is represented graphically, showcasing the average fitness achieved for each parameter combination alongside the standard deviation and the total count of local minima encountered. This visualization offers a succinct overview of the algorithm's performance dynamics, highlighting the impact of our modifications on its optimization capabilities.

Gráfico, Gráfico de caixa estreita

Descrição gerada automaticamente

Best combination

The graph above illustrates a clear trend: as we raise the mutation value, the standard deviation decreases. In fact, at mutation rates of 10% and 20%, the algorithm consistently avoided getting trapped in local minima across all 200 trials. Additionally, it's worth noting that the average fitness scores at mutation rates of 2.5% and 20% surpass those observed at 5% and 10%. This indicates a potential “sweet spot” in mutation rates that optimize the algorithm's performance by balancing exploration and exploitation, thereby enhancing its efficiency in finding optimal solutions.

Observing that a mutation rate of 5% leads to lower average fitness compared to 10%, which sometimes gets stuck in local minima, shows that higher mutation rates aren't always the best way to make fitness better. At the start of the learning process, a higher mutation rate is very helpful in avoiding local minima, highlighting the complex nature of the evolutionary process. This insight calls for a smart, phase-based mutation strategy: starting high to promote broad exploration and then adjusting to focus on fine-tuning solutions.

Building on this understanding and the insights from the graph, which found a "sweet spot" in mutation rates that significantly improve the algorithm's performance, we have adopted a dynamic mutation value strategy. This approach recognizes the different effects of mutation rates on the algorithm's ability to avoid local minima and improve average fitness scores. By changing the mutation value based on how the algorithm is currently doing, our adaptive strategy aims to make the most of the specific benefits of certain mutation rates at different times. This strategy, developed to make the algorithm stronger and better at exploring complex solution spaces, is expected to lead to more optimal and consistently good results.

Gráfico, Gráfico de caixa estreita

Descrição gerada automaticamente

Best combination

Console output:

Average solution for ('roulette', 0.025): -0.18145887314220946 with std: 0.630905702809567 - Total local minima: 12

Average solution for ('roulette', 0.05): -0.10717542527149532 with std: 0.43239277332605214 - Total local minima: 4

Average solution for ('roulette', 0.1): -0.0925001230412599 with std: 0.05075241946002999 - Total local minima: 0

Average solution for ('roulette', 0.2): -0.21506906325728217 with std: 0.14302685604441318 - Total local minima: 0

Average solution for ('roulette', 'auto'): -0.06715868493795363 with std: 0.04385941216404108 - Total local minima: 0

Average solution for ('random', 0.025): -0.27334117665204066 with std: 0.787295139552444 - Total local minima: 17

Average solution for ('random', 0.05): -0.07346991502746535 with std: 0.30761983625630485 - Total local minima: 2

Average solution for ('random', 0.1): -0.10039074604047013 with std: 0.06329651908313769 - Total local minima: 0

Average solution for ('random', 0.2): -0.22500117102411857 with std: 0.16208124718089623 - Total local minima: 0

Average solution for ('random', 'auto'): -0.06532305912542404 with std: 0.03864819139065202 - Total local minima: 0

The introduction of dynamic mutation values has clearly yielded the most favorable outcomes in terms of fitness, standard deviation, and the avoidance of local minima. When employing the 'auto' mutation strategy, both selection methods, 'roulette' and 'random', showcased impressive improvements.

For the 'roulette' selection method, the dynamic mutation approach resulted in an average fitness of -0.0671, coupled with a remarkably low standard deviation of 0.0438, and zero instances of getting trapped in local minima. Similarly, the 'random' selection method under the dynamic mutation setting achieved an average fitness of -0.0653, with an even lower standard deviation of 0.0386, and also successfully avoided any local minima. These results underscore the effectiveness of the dynamic mutation strategy in enhancing the algorithm's performance by not only improving the fitness levels but also ensuring a consistent and reliable exploration of the solution space without falling into local minima traps.

# Results and Discussion:

The modifications implemented have demonstrated considerable improvements in the genetic algorithm's ability to navigate complex optimization landscapes more effectively. By fostering greater diversity within the population and enhancing the algorithm's exploratory mechanisms, we have successfully mitigated issues related not only to performance steadiness but also to premature convergence and local minima entrapment. Our cross-validation and graphical analysis underscore the significance of adaptive strategies in optimizing genetic algorithms for complex problem-solving scenarios.

Gráfico, Gráfico de caixa estreita

Descrição gerada automaticamente

# Conclusion:

The enhancements made to the evolutionary algorithm have demonstrated the importance of carefully designing evolutionary operators to suit the specific requirements of the optimization problem. By addressing the identified issues, we were able to improve the algorithm's efficiency and reliability, as evidenced by the more consistent generation-over-generation performance improvement.

# Appendices:

Modified Python Script

# -\*- coding: utf-8 -\*-

"""

    Module 6: Descriptive and Predictive Modeling

    Exercise 1: Optimization

    Submitted by: Jorge de la Torre Garcia (DNI), Lydia Phoebe Amanda Lilius (DNI), Miguel Galán Cisneros (DNI), Vitor Oliveira de Souza (Z0963220P)

    Date: 14/02/2024

"""

from itertools import product

import random

import math

from matplotlib import pyplot as plt

import numpy as np

# =============================================================================

# Objective function

# =============================================================================

## Calculates the fitness of an individual

def apply\_function(individual):

    x = individual["x"]

    y = individual["y"]

    # Calculate sums of squares of x and y values.

    firstSum = x\*\*2.0 + y\*\*2.0

    # Calculate cosine of 2\*pi\*x and cosine of 2\*pi\*y.

    secondSum = math.cos(2.0 \* math.pi \* x) + math.cos(2.0 \* math.pi \* y)

    # n is the number of values (x and y = 2)

    n = 2

    # Returns a float value calculated by the Ackley function (objective function)

    return -(

        -20.0 \* math.exp(-0.2 \* math.sqrt(firstSum / n))

        - math.exp(secondSum / n)

        + 20

        + math.e

    )

# =============================================================================

# Generate first population

# =============================================================================

## Generates a population of individuals with random x and y values in the given boundaries

def generate\_population(size, x\_boundaries, y\_boundaries):

    lower\_x\_boundary, upper\_x\_boundary = x\_boundaries

    lower\_y\_boundary, upper\_y\_boundary = y\_boundaries

    population = []

    for i in range(size):

        individual = {

            "x": random.uniform(lower\_x\_boundary, upper\_x\_boundary),

            "y": random.uniform(lower\_y\_boundary, upper\_y\_boundary),

        }

        population.append(individual)

    return population

# =============================================================================

# Selection Methods

# =============================================================================

## Select an individual from a popultion based on its fitness

def select\_by\_roulette(sorted\_population, fitness\_sum):

    # Initializes an offset to zero

    offset = 0

    # Initializes normalized fitness sum as fitness sum

    normalized\_fitness\_sum = fitness\_sum

    # If the lowest fitness is negative, we add the absolute value of the

    # lowest fitness multiplied by the number of individuals in the population

    # to the normalized fitness sum

    lowest\_fitness = apply\_function(sorted\_population[0])

    if lowest\_fitness < 0:

        offset = -lowest\_fitness

        normalized\_fitness\_sum += offset \* len(sorted\_population)

    # Gets an random number between 0 and 1

    draw = random.uniform(0, 1)

    # Initializes the probability threashold to zero

    accumulated = 0

    # Test each individual in the sorted polulation (from the lowest to the highest fitness)

    # and then calculate the accumulated probability until it is bigger than the

    # draw value randomly set in the previous step

    for individual in sorted\_population:

        fitness = apply\_function(individual) + offset

        # Increasing probability

        # It never takes the worst individual since its probability is always zero

        probability = fitness / normalized\_fitness\_sum

        accumulated += probability

        if draw <= accumulated:

            return individual

## Select randomly

def select\_random(population):

    return random.choice(population)

# =============================================================================

# Sort Population method

# =============================================================================

## Sorts the population by fitness in ascending order

def sort\_population\_by\_fitness(population):

    return sorted(population, key=apply\_function)

# =============================================================================

# Crossover Method

# =============================================================================

## Create a new individual based on the mean of x and y values of its parents

def crossover(individual\_a, individual\_b):

    xa = individual\_a["x"]

    ya = individual\_a["y"]

    xb = individual\_b["x"]

    yb = individual\_b["y"]

    # Return one dictionary with x and y as the mean of the two individuals sent as parameters

    return {"x": (xa + xb) / 2, "y": (ya + yb) / 2}

# =============================================================================

# Mutation Method

# =============================================================================

def random\_mutation(lower\_boundry, upper\_boundry, value, mutation\_value):

    boundries\_diff = upper\_boundry - lower\_boundry

    # mutation is done by adding a normally distributed random number to the gene's value

    # Keep values in boundry limits

    new\_value = min(

        max(

            value

            + random.uniform(

                -boundries\_diff \* mutation\_value, boundries\_diff \* mutation\_value

            ),

            lower\_boundry,

        ),

        upper\_boundry,

    )

    return new\_value

## Mutate the individual

def mutate(individual, mutation\_value, evolution\_percantage):

    # Sets the boundry values for x and y

    lower\_boundary, upper\_boundary = (-4, 4)

    if mutation\_value == 'auto':

        if evolution\_percantage < 0.3:

            mutation\_value = 0.2

        elif evolution\_percantage < 0.7:

            mutation\_value = 0.1

        else:

            mutation\_value = 0.05

    next\_x = random\_mutation(

        lower\_boundary, upper\_boundary, individual["x"], mutation\_value=mutation\_value

    )

    next\_y = random\_mutation(

        lower\_boundary, upper\_boundary, individual["y"], mutation\_value=mutation\_value

    )

    # Return one dictionary with x and y as the new mutated values (not exceeding the boundries specified)

    return {"x": next\_x, "y": next\_y}

# =============================================================================

# Make Next Generation

# =============================================================================

## Improved version of the function to create th next generation

## In this version we only append the new individuals that have a

## better fitness than the worst individual in the previous generation

def make\_next\_generation(previous\_population, selection\_method, mutation\_value, evolution\_percantage):

    # Sorts population by fitness in ascending order

    # (the higher the fitness, the better the individual)

    sorted\_by\_fitness\_population = sort\_population\_by\_fitness(previous\_population)

    # Gets population size

    population\_size = len(previous\_population)

    # Creates a new individual for each individual in the previous population

    # and replace one individual with the best individual created

    for i in range(population\_size):

        # Gets the sum of all fitnesses

        fitness\_sum = sum(

            apply\_function(individual) for individual in sorted\_by\_fitness\_population

        )

        # Randomly select two individuals from the sorted population

        if selection\_method == "roulette":

            father = select\_by\_roulette(sorted\_by\_fitness\_population, fitness\_sum)

            mother = select\_by\_roulette(sorted\_by\_fitness\_population, fitness\_sum)

        elif selection\_method == "random":

            father = select\_random(sorted\_by\_fitness\_population)

            mother = select\_random(sorted\_by\_fitness\_population)

        # Creates a new individual by crossing the two selected individuals

        individual = crossover(father, mother)

        # Mutates the new individual

        individual = mutate(individual, mutation\_value, evolution\_percantage)

        if apply\_function(individual) > apply\_function(sorted\_by\_fitness\_population[0]):

            # Replace the worst individual in the previous generation with the new individual

            sorted\_by\_fitness\_population[0] = individual

    # Return a list with new individuals (next generation)

    return sorted\_by\_fitness\_population

# =============================================================================

# Genetic Algorithm

# =============================================================================

def genetic\_algorithm(

    selection\_method="random",

    graph=False,

    mutation\_value=0.1,

    verbose=0,

):

    generations = 100

    population = generate\_population(

        size=10, x\_boundaries=(-5, 5), y\_boundaries=(-5, 5)

    )

    i = 1

    bestFitness = []

    while True:

        if verbose > 0:

            print(str(i))

        for individual in population:

            if verbose > 0:

                print(individual, apply\_function(individual))

        if i == generations:

            break

        i += 1

        population = make\_next\_generation(

            previous\_population=population,

            selection\_method=selection\_method,

            mutation\_value=mutation\_value,

            evolution\_percantage= (i / generations)

        )

        best\_individual = sort\_population\_by\_fitness(population)[-1]

        bestFitness.append(apply\_function(best\_individual))

    best\_individual = sort\_population\_by\_fitness(population)[-1]

    if graph:

        fig, ax = plt.subplots(figsize=(10, 6))

        ax.plot(bestFitness)

        ax.set\_xlabel("Generation")

        ax.set\_ylabel("Best Fitness")

        ax.set\_title("Genetic Algorithm Performance History")

        plt.show()

    if verbose > 0:

        print("\nFINAL RESULT")

    if verbose > 0:

        print(best\_individual, apply\_function(best\_individual))

    return best\_individual, apply\_function(best\_individual)

# =============================================================================

# MAIN

# =============================================================================

if \_\_name\_\_ == "\_\_main\_\_":

    cross\_validation = False

    if not cross\_validation:

        genetic\_algorithm(verbose=1, graph=True, mutation\_value='auto')

    else:

        # Define parameter grid

        param\_grid = {

            "selection\_method": ["roulette", "random"],

            "mutation\_value": [0.025, 0.05, 0.1, 0.2, 'auto'],

        }

        # Number of "cross validations" runs each combination of parameters

        n\_cv = 200

        # Variable to store the cross validation results

        results = {}

        # Grid search loop

        for params in product(

            param\_grid["selection\_method"], param\_grid["mutation\_value"]

        ):

            selection\_method, mutation\_value = params

            best\_individuals = []

            best\_fitnesses = []

            # Cross validation loop

            for i in range(n\_cv):

                best\_individual, fitness = genetic\_algorithm(

                    selection\_method=selection\_method,

                    mutation\_value=mutation\_value,

                )

                best\_individuals.append(best\_individual)

                best\_fitnesses.append(fitness)

            # Store results for each parameters combination

            results[params] = {

                "best\_individuals": best\_individuals,

                "best\_fitnesses": best\_fitnesses,

            }

        # Plotting and printing results

        average\_fitnesses = []

        std\_devs = []

        n\_local\_minimas = []

        labels = []

        # Extract data for plotting

        for param, data in results.items():

            average\_fitness = np.mean(data["best\_fitnesses"])

            std\_dev = np.std(data["best\_fitnesses"])

            n\_local\_minima = sum(np.diff(data["best\_fitnesses"]) < -1)

            print(

                f"Average solution for {param}: {average\_fitness} with std: {std\_dev} - Total local minima: {n\_local\_minima}"

            )

            average\_fitnesses.append(average\_fitness)

            std\_devs.append(std\_dev)

            n\_local\_minimas.append(n\_local\_minima)

            labels.append(f"{param[0]}-{param[1]}")

        # Plotting

        fig, ax = plt.subplots(figsize=(10, 6))

        # Create bar plot

        x\_pos = np.arange(len(labels))

        bars = ax.bar(

            x\_pos,

            average\_fitnesses,

            yerr=std\_devs,

            capsize=5,

            alpha=0.75,

            color="skyblue",

        )

        # Annotate bars with the count of local minima

        for bar, n\_minima in zip(bars, n\_local\_minimas):

            yval = bar.get\_height()

            ax.text(

                bar.get\_x() + bar.get\_width() / 2.0,

                yval,

                f"LM: {n\_minima}",

                va="bottom",

            )  # LM: Local Minima

        ax.set\_xlabel("Parameter Combination")

        ax.set\_ylabel("Average Fitness")

        ax.set\_title("Genetic Algorithm Performance")

        ax.set\_xticks(x\_pos)

        ax.set\_xticklabels(labels, rotation=45, ha="right")

        plt.tight\_layout()

        plt.show()