**06. Aprendizaje automático (Machine Learning): conceptos, metodología, algoritmos de aprendizaje para analítica descriptiva, predictiva y prescriptiva, y retos en su implementación (2022-24)**

**Tarea 1.A - Optimización (básico, obligatorio)**

Este ejercicio requiere que cada grupo:

1) Entiende el algoritmo de optimización adjunto.

2) Detecta lo que hace que el algoritmo no funcione constantemente mejor a lo largo de generaciones (es decir, la mejor solución de la población no mejora necesariamente a lo largo de generaciones sucesivas).

3) Modificar los diferentes operadores evolutivos (selección, cruce, mutación, reemplazo) para que el algoritmo funcione mejor.

4) Entregar un informe escrito describiendo las modificaciones incluidas, junto con el guión modificado

Recuerda incluir nombres, apellidos y números de DNI de todos los miembros del grupo en el informe y el script de Python.

**RESOLUCION**

Se presentan algunas posibles modificaciones a los operadores evolutivos del algoritmo para mejorar su desempeño:

1. **Selección por torneo:** En lugar de utilizar la selección por ruleta, se puede utilizar la selección por torneo, en la que se eligen al azar dos individuos de la población y se selecciona el mejor de ellos para el cruce.

Este enfoque suele funcionar mejor que la selección por ruleta cuando hay muchos individuos con una aptitud baja.

def select\_by\_tournament(population, tournament\_size):

tournament = random.sample(population, tournament\_size)

individual = max(tournament, key=apply\_function)

return individual

1. Se toma una muestra aleatoria de la población del tamaño especificado en el parámetro "tournament\_size" utilizando la función "random.sample".
2. Se selecciona al individuo con la mejor evaluación (mayor valor devuelto por la función "apply\_function") de la muestra utilizando la función "max" y especificando "apply\_function" como la clave para realizar la comparación.
3. Se devuelve el individuo seleccionado.
4. **Cruce de dos puntos**: El operador de cruce actual es un cruce aritmético simple que toma el promedio de las coordenadas x e y de los padres. Sin embargo, este operador puede generar descendencia que no esté lo suficientemente diversificada. Una forma de mejorar esto es utilizar un operador de cruce más sofisticado, como el cruce de BLX-alpha. Este operador toma en cuenta la distancia entre los padres y genera una descendencia que está en algún lugar entre los padres, pero que se aleja más del padre más alejado.

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

xa = individual\_a["x"]

ya = individual\_a["y"]

xb = individual\_b["x"]

yb = individual\_b["y"]

x\_min = min(xa, xb)

x\_max = max(xa, xb)

y\_min = min(ya, yb)

y\_max = max(ya, yb)

x\_diff = x\_max - x\_min

y\_diff = y\_max - y\_min

new\_x\_min = x\_min - alpha \* x\_diff

new\_x\_max = x\_max + alpha \* x\_diff

new\_y\_min = y\_min - alpha \* y\_diff

new\_y\_max = y\_max + alpha \* y\_diff

new\_x = random.uniform(new\_x\_min, new\_x\_max)

new\_y = random.uniform(new\_y\_min, new\_y\_max)

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

1. Se extraen las coordenadas x e y de los dos individuos utilizando los nombres de clave "x" e "y".
2. Se calculan los valores mínimos y máximos de x e y a partir de los valores extraídos de ambos individuos.
3. Se calculan las diferencias entre los valores máximos y mínimos de x e y.
4. Se calculan los nuevos valores mínimos y máximos de x e y a partir del parámetro de mezcla alpha y las diferencias calculadas en el paso anterior.
5. Se generan dos nuevos valores aleatorios de x e y dentro de los rangos definidos por los nuevos valores mínimos y máximos.
6. Se devuelve un nuevo diccionario con las coordenadas x e y generadas en el paso anterior.
7. **Modificar la mutación**: En el código original, se utiliza un operador de mutación que aplica un desplazamiento gaussiano aleatorio en ambas coordenadas de un individuo. Este operador puede funcionar bien para algunos problemas, pero no parece ser el más adecuado para el problema de optimización en cuestión. Una alternativa puede ser utilizar una mutación más adaptativa, que tenga en cuenta la magnitud de la función de aptitud de cada individuo. Por ejemplo, se podría utilizar una mutación con tamaño adaptativo, de tal forma que los individuos con peores aptitudes tendrían una probabilidad más alta de mutar.

def mutate(individual, population\_size, position, offset=0.01, x\_boundaries=(-5, 5), y\_boundaries=(-5, 5)):

# Dar orden inverso a la posición

actual\_position = population\_size - position-1

actual\_population\_size = population\_size - 1

# Calcula la probabilidad de mutacion en funcion del tamaño de la poblacion

mutation\_prob = actual\_position/actual\_population\_size + offset

print(position, mutation\_prob)

# Ajusta el rango de la mutacion basado en los valores actualizados de la poblacion

x\_range = x\_boundaries[1] - x\_boundaries[0]

y\_range = y\_boundaries[1] - y\_boundaries[0]

x\_mutation\_range = mutation\_prob \* x\_range

y\_mutation\_range = mutation\_prob \* y\_range

# Aplicada mutacion con rango adaptativo

next\_x = individual["x"] + random.gauss(0, x\_mutation\_range)

next\_y = individual["y"] + random.gauss(0, y\_mutation\_range)

# Ajusta los límites de los valores mutados

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

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

next\_x = min(max(next\_x, lower\_x\_boundary), upper\_x\_boundary)

next\_y = min(max(next\_y, lower\_y\_boundary), upper\_y\_boundary)

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

1. Se calcula la posición actual del individuo en la población y se ajusta para que esté en orden inverso, ya que se está asumiendo que la población está ordenada de mayor a menor evaluación.
2. Se calcula la probabilidad de mutación en función de la posición actual y el tamaño de la población, ajustando el valor con un offset opcional.
3. Se calcula el rango de mutación para las coordenadas x e y del individuo utilizando la probabilidad de mutación calculada en el paso anterior y el rango de valores permitidos para cada coordenada.
4. Se aplica una mutación gaussiana a las coordenadas x e y del individuo, utilizando el rango de mutación calculado en el paso anterior.
5. Se ajustan los valores mutados para que estén dentro de los límites permitidos para cada coordenada.
6. **Modificar el reemplazo:** En el código original, se utiliza un esquema de reemplazo generacional, en el que se reemplaza toda la población en cada iteración. Una alternativa podría ser utilizar un esquema de reemplazo por elitismo, en el que se mantiene el mejor individuo de la generación anterior y se reemplazan los demás con la descendencia generada por el operador de cruce y mutación. Este enfoque podría ayudar a preservar la información valiosa de las generaciones anteriores y evitar que se pierda en cada iteración.

def make\_next\_generation(previous\_population, elite\_size=2, tournament\_proportion=.2, alpha=.5):

next\_generation = []

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

population\_size = len(previous\_population)

tournament\_size = int(population\_size\*tournament\_proportion)

for i in range(population\_size - elite\_size):

father = select\_by\_tournament(

sorted\_by\_fitness\_population, tournament\_size)

mother = select\_by\_tournament(

sorted\_by\_fitness\_population, tournament\_size)

individual = crossover(father, mother, alpha)

next\_generation.append(individual)

# Mutaciones

next\_generation = sort\_population\_by\_fitness(next\_generation)

next\_generation\_size = len(next\_generation)

for count, ind in enumerate(next\_generation):

next\_generation[count] = mutate(

next\_generation[count],

next\_generation\_size,

count)

elite = sorted\_by\_fitness\_population[-elite\_size:]

next\_generation.extend(elite)

return next\_generation

1. La población anterior se ordena por aptitud utilizando la función "sort\_population\_by\_fitness".
2. Se define el tamaño de la población y el tamaño del torneo en función de la proporción de la población que se utilizará para el torneo.
3. Se itera sobre la población para crear nuevos individuos utilizando selección por torneo y cruzamiento.
4. Se itera sobre la nueva generación para aplicar mutaciones utilizando la función "mutate". La probabilidad de mutación se calcula en función de la posición del individuo en la población.
5. Se agrega la élite de la población anterior a la nueva generación, con una cantidad definida por el parámetro "elite\_size".
6. Se devuelve la nueva generación ordenada por aptitud.