Informe de la práctica nº6

Computación evolutiva

UNIVERSIDAD DE BURGOS

Por: Adrián Zamora Sánchez

Índice

Introducción al problema

He elegido el problema de regresión para estimar el porcentaje de grasa de un individuo a partir de otras características físicas como en ancho de cadera, altura, peso, etc. El principal motivo por el que he elegido este dataset es porque comprendo mejor el tipo de solución que se pide, es decir generar una función matemática donde los valores que se utilizan como parámetros son estas características físicas de los individuos de los cuales se quiere estimar su porcentaje de grasa.

Los datos antropométricos que se proporcionan son:

- Densidad
- Grasa corporal (el valor a estimar)
- Edad
- Peso
- Altura
- Cuello
- Pecho
- Abdomen
- Cadera
- Muslo
- Rodilla
- Tobillo
- Bíceps
- Antebrazo
- Muñeca

Este famoso dataset, aunque ofrece medidas generales muy útiles para estimar el porcentaje graso de forma general, adolece de un factor clave como es el sexo del individuo al cual se ha medido, puesto que entre hombres y mujeres hay una clara distinción entre la relación de las proporciones corporales y el porcentaje de grasa. Por esto considero que es complicado conseguir un estimador realmente preciso, aunque sí una buena aproximación.

Por lo tanto tendremos bien diferenciados los datos de características, es decir las filas 1 y 3-15 y por otro lado la fila 2, la cual se utiliza para comprobar si las fórmulas generadas son capaces de estimar correctamente el porcentaje de grasa.

Solución planteada

La solución planteada propone generar fórmulas con las cuales estimar la grasa del sujeto utilizando el valor de sus características y una serie de operadores y valores constantes.

El árbol que voy a generar tiene por tanto un valor de aridad de 14, que son el número de características que pueden adoptar sus nodos hoja, sus nombres son definidos en estas líneas para una más sencilla interpretación del resultado:

```
# Árbol con nombre MAIN y aridad 14
pset = gp.PrimitiveSet("MAIN", 14)

# Entradas del árbol para que sean legibles en el resultado
pset.renameArguments(ARG0='Density', ARG1='Age', ARG2='Weight', ARG3='Height',
                     ARG4='Neck', ARG5='Chest', ARG6='Abdomen', ARG7='Hip',
                     ARG8='Thigh', ARG9='Knee', ARG10='Ankle', ARG11='Biceps',
                     ARG12='Forearm', ARG13='Wrist')
```

Los operadores planteados son los siguientes:

```
# Operaciones con dos operandos
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.mul, 2)
pset.addPrimitive(protected_div, 2)  # División protegida definida arriba

# Operaciones de un solo operando
pset.addPrimitive(operator.neg, 1)
pset.addPrimitive(math.cos, 1)
pset.addPrimitive(math.sin, 1)
pset.addPrimitive(protected_sqrt, 1) # Raiz protegida definida arriba

# Constante aleatoria
pset.addEphemeralConstant('rand101', lambda: random.uniform(-2, 2))
```

Tanto la división como la raíz están protegidas frente a posibles usos imprevistos como dividir entre 0 o hacer la raíz cuadrada de un número negativo. Se redefinen estas operaciones como :

$$\text{protected\_div}(x, y, \epsilon) = \begin{cases} 1 & \text{si } |y| < \epsilon, \\ \frac{x}{y} & \text{si } |y| \geq \epsilon \end{cases} \qquad \text{protected\_sqrt}(x) = \begin{cases} 1 & \text{si } x < 0, \\ \sqrt{x} & \text{si } x \geq 0 \end{cases}$$

En código:

```python
def protected_div(left, right, limite=1e-8):
    """Evita casos de x/0 o valores muy cercanos a 0 que den resultados demasiado grandes"""
    if abs(right) < limite:
        return 1 # Valor seguro
    return left / right

def protected_sqrt(x):
    """Raiz protegida contra numeros negativos o incorrectos"""
    try:
        if x < 0:
            return 1 # Valor seguro
        return math.sqrt(x)
    except ValueError:
        return 1  # Valor seguro
```

Los individuos serán evaluados comparando los resultados obtenidos con el resultado esperado, donde el error se mide como la distancia absoluta entre estos dos valores, además se le suma a este error una penalización por complejidad calculada como la suma de cada nodo multiplicada por un valor constante. He experimentado con otras medidas de error como el MSE (error cuadrático medio) y el RMSE (raíz cuadrada del MSE), aunque todos ofrecen resultados muy buenos he elegido el MAE por su simpleza y valores más pequeños y manejables.

```python
def eval_sol(individuo):
    """
    Evalúa un individuo comparando las predicciones realizadas por la expresión que representa
    con los valores reales de los resultados (results) usando el MAE.

    Args:
        individuo: Un árbol de expresión que representa una posible solución.

    Returns:
        Una tupla con el MAE como único elemento, ya que estamos minimizando el error.
    """
    # Compilar el árbol de expresión del individuo
    func = toolbox.compile(expr=individuo)

    # Extraer datos y resultados
    data, results = extract_data()
    data = data.values  # Se convierte el DataFrame a matriz de np para manipulación más eficiente
    results = results.values.flatten()  # Convertir a un array unidimensional

    # Calcular las predicciones y los errores
    errores = []
    for x, real_value in zip(data, results):
        try:
            pred_value = func(*x)
            errores.append(abs(real_value - pred_value))
        except (OverflowError, ValueError):
            errores.append(1e10)  # Penalización alta pero manejable

    # Calcular el MAE
    mae = np.mean(errores)

    # Penalizar ecuaciones largas o complejas
    complexity_penalty = len(individuo) * 0.05

    return (mae + complexity_penalty,)
```

Para el análisis de los resultados se extraen los valores mínimo, máximo y medio del fitness de cada generación:

```
# Ejecutar el algoritmo genético
stats = tools.Statistics(key=lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("min", np.min)
stats.register("max", np.max)
```

Se establece el fitness con el objetivo de minimizar, puesto que queremos reducir el error al estimar. También se establece en la toolbox de DEAP a los individuos como árboles de operaciones.

```
# Se establece un fitness a minimizar
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)

# Se define la toolbox
toolbox = base.Toolbox()
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=2)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("compile", gp.compile, pset=pset)
```

Se establecen los operadores genéticos utilizados y sus limitaciones, como la altura máxima donde se aplican para evitar que los árboles de expresiones sean demasiado complejos. También se establece la función de adaptación:

```
# Operadores genéticos
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", gp.cxOnePoint)

# Mutación y su profundidad
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)

# Limites en altura donde se aplican los operadores genéticos
toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"), max_value=15))
toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"), max_value=15))

# Se define cómo se evaluará cada individuo
toolbox.register("evaluate", eval_sol)
```

Para poder aplicar mejoras como el método de parada y el elitismo, se recorren las generaciones en un bucle, donde al principio de este se guardan los mejores de cada generación (los élites) que representan un 5% de la población total.

```python
# Bucle que ejecuta el algoritmo para cada genración
for gen in range(num_gen):
    # El 5% de la población son élites
    N_elites = round(population_size*0.05)

    # Se aplica elitismo
    elites = tools.selBest(population, N_elites)

    # Se ejecuta una única generación
    population, _ = algorithms.eaSimple(
        population, toolbox,
        cxpb=prob_mate, mutpb=prob_mut,
        ngen=1, verbose=False, stats=stats
    )

    # Sustituir una parte de la población con los mejores individuos
    population[-N_elites:] = elites

    # Guardar estadísticas de la generación actual
    record = stats.compile(population)
    logbook.record(gen=gen, **record)
    print(logbook.stream)

    # Verificar si ha habido convergencia
    if stopMethod(logbook, threshold=5e-8, patience=250):
        print(f"Convergencia alcanzada en la generación {gen}")
        break
```

Para detener el algoritmo cuando no se está mejorando los resultados obtenidos anteriormente se emplea el siguiente método de parada, donde se comprueba si el fitness mínimo de las “patience” generaciones ha cumplido o no con un porcentaje de mejora establecido por “threshold”. De esta forma aseguramos que mientras algún individuo consiga mejorar a los anteriores minimizando el error, el proceso continúa con la explotación de esa generación.

```python
def stopMethod(logbook, threshold, patience):
    """
    Comprueba si el algoritmo ha convergido en función de la mejora porcentual de su mínimo
    - threshold: mejora mínima requerida para seguir buscando
    - patience: número de generaciones que deben cumplir el treshold
    """
    # Se emplea como medida de rendimiento el fitness máximo
    min_fitness = logbook.select("min")

    # Si hay menos de "patience" generaciones, no se puede verificar convergencia
    if len(min_fitness) < patience:
        return False

    # Se comprueba la mejora conseguida
    for i in range(1, patience):
        # Calcula la mejora porcentual
        improvement = (min_fitness[-i] - min_fitness[-i-1]) / min_fitness[-i-1]

        if abs(improvement) >= threshold:
            return False  # Continua el algoritmo

    return True # Termina el algoritmo
```

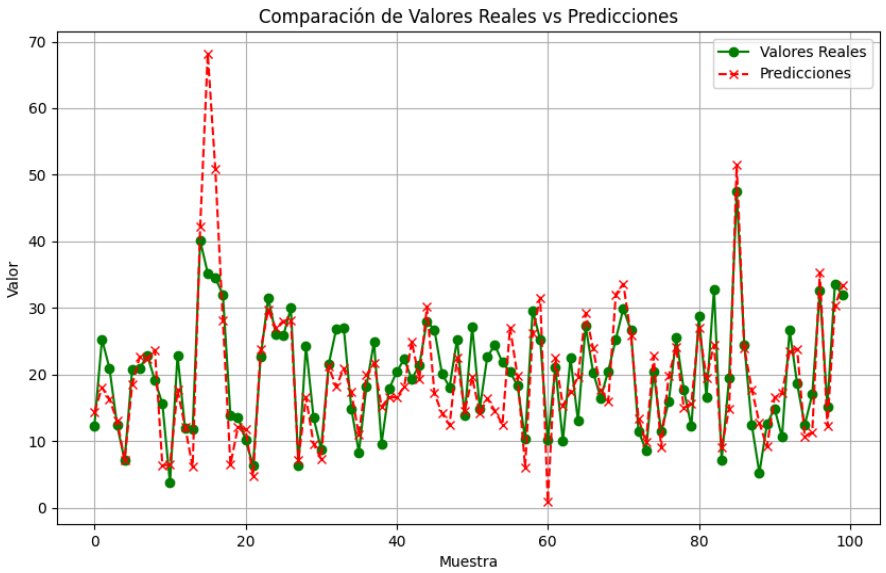
Finalmente los resultados obtenidos se muestran tanto en la consola durante cada generación como al final del programa, donde se imprime por pantalla la expresión que mejor se ha adaptado durante el proceso evolutivo. Además se muestra un gráfico, donde por un lado se separan los fitness mínimos y por otro el fitness promedio y máximos, puestos que estos últimos tienen una gran variabilidad causada por el encadenamiento de algunos nodos como las multiplicaciones que generan valores muy grandes y por tanto aumentan mucho el error:

```python
# Se muestra la expresión del mejor individuo
best = tools.selBest(population, 1)[0]
print("\nLa mejor solución encontrada es:", best)

# Se recuperan los datos desde el log
gen = logbook.select("gen")
avgs = logbook.select("avg")
maxim = logbook.select("max")
minim = logbook.select("min")

# Escalar los datos del logbook
max_fitness = max(maxim)
min_fitness = min(minim)

# Se establecen dos figuras para graficar
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Gráfico 1: Mínimos
ax1.plot(gen, minim, "b-", label="Min Fitness")
ax1.set_xlabel("Generation")
ax1.set_ylabel("Fitness")
ax1.set_title("Fitness mínimo")
ax1.grid(True)
ax1.legend()

# Gráfico 2: Promedios y Máximos
ax2.plot(gen, avgs, "g-", label="Average Fitness")
ax2.plot(gen, maxim, "r-", label="Max Fitness")
ax2.set_xlabel("Generation")
ax2.set_ylabel("Fitness")
ax2.set_title("Fitness promedio y máximo")
ax2.grid(True)
ax2.legend()

# Mostrar ambos gráficos
plt.tight_layout()
plt.show()
```

Para visualizar mejor a los individuos se muestra una gráfica donde se aprecian los valores de las predicciones y los valores reales a estimar de una muestra de por defecto 100 muestras tomadas de forma aleatoria del dataset. El código de la función:

```python
def plot_ind(ind, samples=100):
    """
    Representa las predicciones de un individuo en 50 estimaciones y
    las compara con los valores reales

    Args:
        ind:     individuo a evaluar
        samples: número de muestras a evaluar
    """
    # Compilar el árbol de expresión
    func = toolbox.compile(expr=ind)

    # Extraer datos y resultados
    data, results = extract_data()
    data = data.values
    results = results.values.flatten()

    # Tomar las primeras "samples" muestras
    random_indices = np.random.choice(len(data), 100, replace=False)
    sample_data = data[random_indices]
    sample_results = results[random_indices]

    # Calcular predicciones
    predicciones = []
    for x in sample_data:
        try:
            pred_value = func(*x)
            predicciones.append(pred_value)
        except (OverflowError, ValueError):
            predicciones.append(None)  # Ignora la predicción

    # Muestra la gráfica
    plt.figure(figsize=(10, 6))
    plt.plot(range(samples), sample_results, "g-", label="Valores Reales", marker="o")
    plt.plot(range(samples), predicciones, "r--", label="Predicciones", marker="x")
    plt.title("Comparación de Valores Reales vs Predicciones")
    plt.xlabel("Muestra")
    plt.ylabel("Valor")
    plt.grid(True)
    plt.legend()
    plt.show()
```

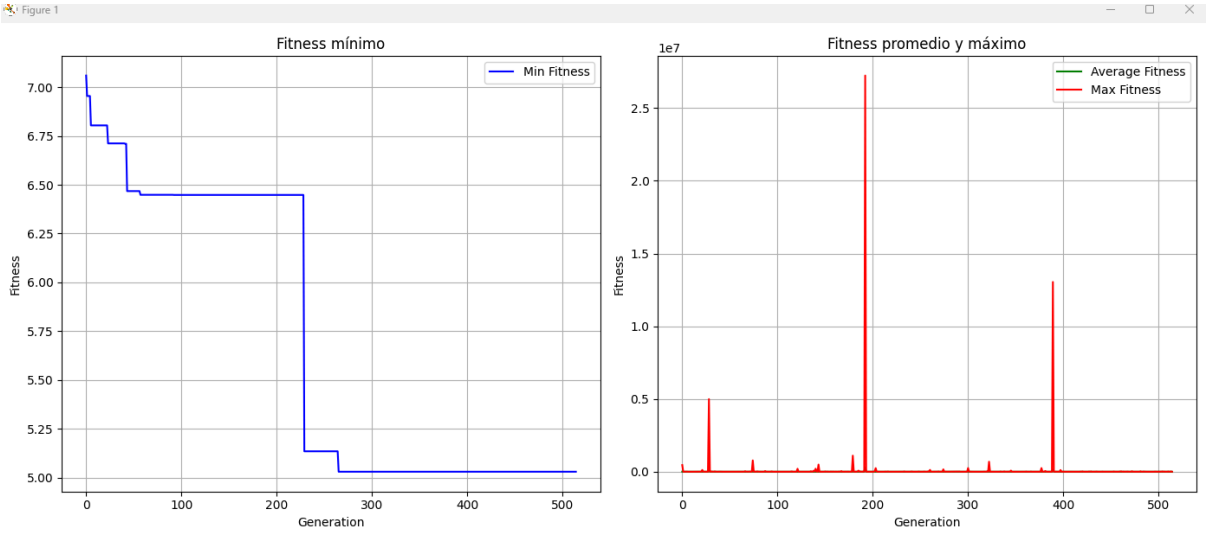
El gráfico generado:



El resultado se puede apreciar en la consola como:

```
509     41.7421 5.02978 1823.69
510     187.512 5.02978 14285
511     38.9204 5.02978 1682.77
512     62.0248 5.02978 3517.08
513     195.89  5.02978 17981.8
514     30.2464 5.02978 1090.3
Convergencia alcanzada en la generación 514

La mejor solución encontrada es: sub(sub(sub(Abdomen, Height), Density), Density)
```

Y en el gráfico:



Pruebas para refinar el algoritmo

A continuación se experimenta con los diferentes parámetros evolutivos de este algoritmo, con el objetivo de determinar cual sería una buena configuración para encontrar la mejor solución posible.

Pruebas de población

Tamaño

- 50
  Generalmente se da una convergencia bastante rápida, puesto que con un tamaño tan pequeño de población no hay suficiente exploración. El rendimiento es excelente.

```
218     146.723 5.56848 2103.25
219     341.352 5.56848 6402.38
220     230.345 5.56848 6304.38
Convergencia alcanzada en la generación 220

La mejor solución encontrada es: sub(Thigh, mul(Density, mul(Density, Neck)))
Tiempo de procesamiento: 8.265847 segundos
```

- 100
  Algo mejor, puesto que se consiguen explorar mejores resultados y por lo tanto se mantiene el proceso evolutivo durante más tiempo, consiguiendo optimizar el resultado. Sin embargo, aún se vería beneficiado el programa de una mayor población.

```
650     90.5432 4.81931 3027.84
651     56.7605 4.81931 2298.63
652     12.4589 4.81931 165.175
Convergencia alcanzada en la generación 652

La mejor solución encontrada es: sub(sub(Abdomen, Thigh), protected_sqrt(Weight))
Tiempo de procesamiento: 25.184502 segundos
```

- 200
  El programa consigue resultados mucho mejores, generalmente converge en menos generaciones que el caso anterior, puesto que explora más soluciones desde el principio. El proceso se vuelve mucho más lento, causado por el significativo aumento de las evaluaciones que se realizan a los individuos.

```
438     38.2963 4.61745 1766.71
439     26.5297 4.61745 1181.58
Convergencia alcanzada en la generación 439

La mejor solución encontrada es: mul(protected_div(Weight, Wrist), protected_div(protected_div(pro
iv(protected_div(protected_div(Abdomen, Density), Density), Density), Density), Density), Neck))
Tiempo de procesamiento: 83.715670 segundos
```

- 400
  Destaca por el gran incremento en el tiempo de ejecución, así como por sus resultados que apenas mejoran respecto del caso anterior.

```
381     76.1938 4.62264 9430.4
382     91.4225 4.62264 18143
383     51.9607 4.62264 2759.72
Convergencia alcanzada en la generación 383

La mejor solución encontrada es: add(add(Abdomen, protected_sqrt(mul(Biceps, Ankle))), neg(Hip))
Tiempo de procesamiento: 158.769456 segundos
```

Generaciones

Puesto que se está empleando un método de parada, definir el número de generaciones es algo irrelevante mientras este número sea lo suficientemente grande para permitir que el algoritmo pare por sí mismo pues haya convergido según el criterio establecido para esto.

Pruebas selección

Tournament (tournsize = 4)

Obtiene resultados muy buenos, gracias a su buena capacidad de explotación mediante los torneos.

```
238     105.59  4.23591 3861.01
239     56.4021 4.23591 2823.07
240     86.7821 4.23591 2823.07
Convergencia alcanzada en la generación 240

La mejor solución encontrada es: mul(sub(Abdomen, neg(sub(Wrist, Height))), cos(Density))
Tiempo de procesamiento: 31.708009 segundos
```

StochasticSampling, Roulette y Random

No son métodos adecuados, al permitir durante su seleccion que los individuos no tan bien adaptados pasen a las siguiente generaciones y por tanto puedan crecer en tamaño hasta el punto de generar errores en tiempo de ejecución.

```
38      3.64308e+293    6.7123  6.19313e+295
39      4.68859e+298    6.7123  7.64439e+300
<string>:1: RuntimeWarning: overflow encountered in scalar multiply
40      3.70394e+302    6.7123  2.00393e+304
<string>:1: RuntimeWarning: overflow encountered in scalar multiply
<string>:1: RuntimeWarning: overflow encountered in scalar multiply
```

Lexicase

Selecciona a los individuos que mejor fitness consiguen, evaluando varios casos individuales de forma aleatoria. Obtiene resultados muy buenos.

```
792     31.4328 4.33955 1932.05
793     39.9004 4.33955 1932.05
794     39.1688 4.33955 1932.05
Convergencia alcanzada en la generación 794

La mejor solución encontrada es: add(protected_div(protected_div(protected_div(protected_div(sub(pr
otected_div(Hip, Density), Height), Density), Density), Density), protected_div(Hip, Abdomen)), 0)
Tiempo de procesamiento: 127.234362 segundos
```

Epsilon Lexicase (epsilon = 0.5)

Similar a lexicase, aunque en algunos casos ha conseguido converger con mayor velocidad.

```
294     22.6223 4.31398 141.642
295     27.0986 4.31398 168.164
296     30.3237 4.31398 1090.74
Convergencia alcanzada en la generación 296

La mejor solución encontrada es: add(sub(protected_div(protected_div(Forearm, Density), Density), I
ip), Abdomen)
Tiempo de procesamiento: 44.849890 segundos
```

Pruebas mutación

MutUniform

Genera una expresión nueva y la sustituye por una del árbol al que se va a mutar. Consigue muy buenos resultados.

```
431     152.048         4.15561 6472.35
432     340.882         4.15561 6472.35
433     282.044         4.15561 12511.2
434     2.06713e+06     4.15561 3.51347e+08
435     338.776         4.15561 12629.1
Convergencia alcanzada en la generación 435

La mejor solución encontrada es: sub(Abdomen, mul(Density, add(Biceps, Neck)))
Tiempo de procesamiento: 64.548136 segundos
```

NodeReplacement

Intercambia sus propios nodos consiguiendo nuevas combinaciones. He obtenido muy malos resultados de este método.

```
196     52.6649 6.8623  724.336
197     56.2563 6.8623  1862.53
198     38.5325 6.8623  609.463
199     39.6133 6.8623  1844.79
Convergencia alcanzada en la generación 199

La mejor solución encontrada es: neg(protected_div(Wrist, -1))
Tiempo de procesamiento: 28.751377 segundos
```

MutEphemeral

Tanto con los modos “all” con el que muta todos los nodos de constantes como “one” que elige y muta solo uno de ellos he conseguido muy malos resultados, probablemente porque las constantes en los nodos no suelen dar buenos resultados.

```
202     44.6196 5.95539 765.444
203     55.1287 5.95539 808.968
204     43.893  5.95539 808.968
Convergencia alcanzada en la generación 204

La mejor solución encontrada es: sub(Knee, mul(mul(mul(Wrist, Density), Density), Density))
Tiempo de procesamiento: 30.121100 segundos
```

MutShrink

Reemplaza una rama completa por alguno de sus argumentos, reduciendo la complejidad del árbol. Posiblemente sus malos resultados se deban a que los árboles son cortos por la restricción en la función de adaptación.

```
196     8.56561 6.44881 19.3008
197     8.86801 6.44881 19.3008
198     8.86801 6.44881 19.3008
199     8.11201 6.44881 19.3008
Convergencia alcanzada en la generación 199

La mejor solución encontrada es: sub(Knee, Wrist)
Tiempo de procesamiento: 28.985102 segundos
```

Pruebas cruce

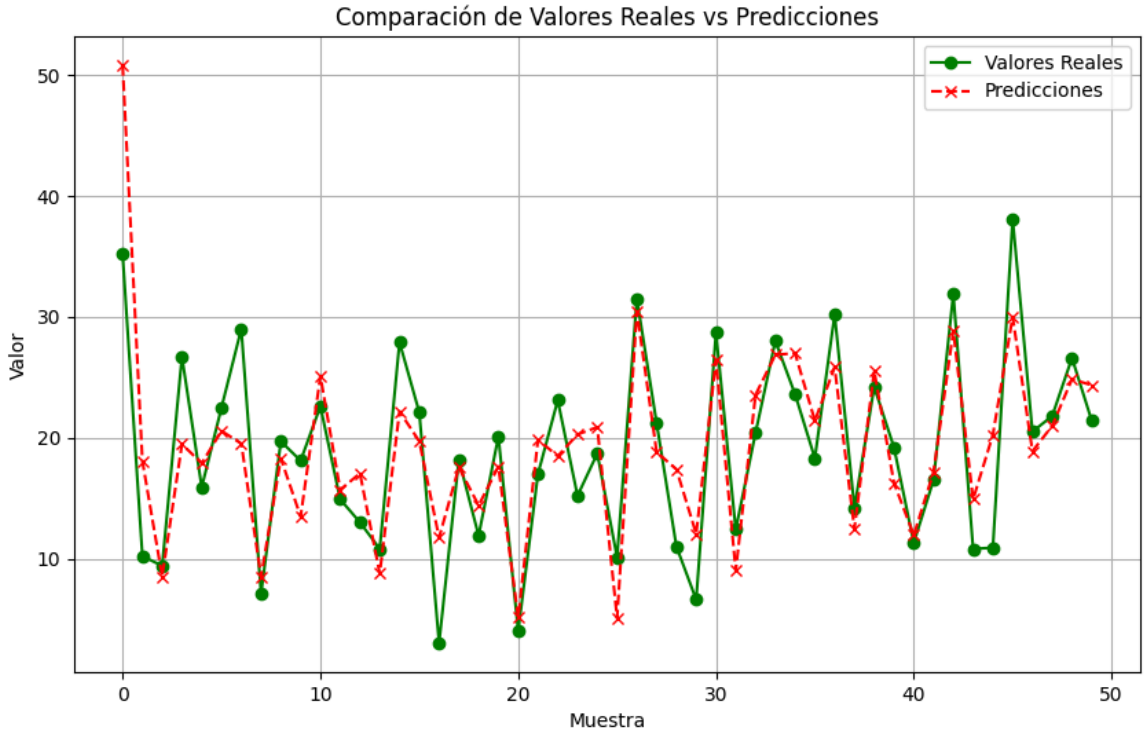
El único operador que he podido utilizar es cxOnePoint, el cual me ha dado buenos resultados al intercambiar ramas a partir de un punto entre dos árboles. Con los otros dos operadores de cruce no he conseguido que funcionen pese a seguir los ejemplos de la documentación de DEAP.

Conclusiones

Para este algoritmo es realmente importante definir con detenimiento los tipos de operaciones que se van a utilizar, puesto que estas operaciones son determinantes sobre los resultados obtenidos, una mala elección o definición de las operaciones puede llevar a errores numéricos como las divisiones por cero o por números muy pequeños que devuelven resultados demasiado grandes.

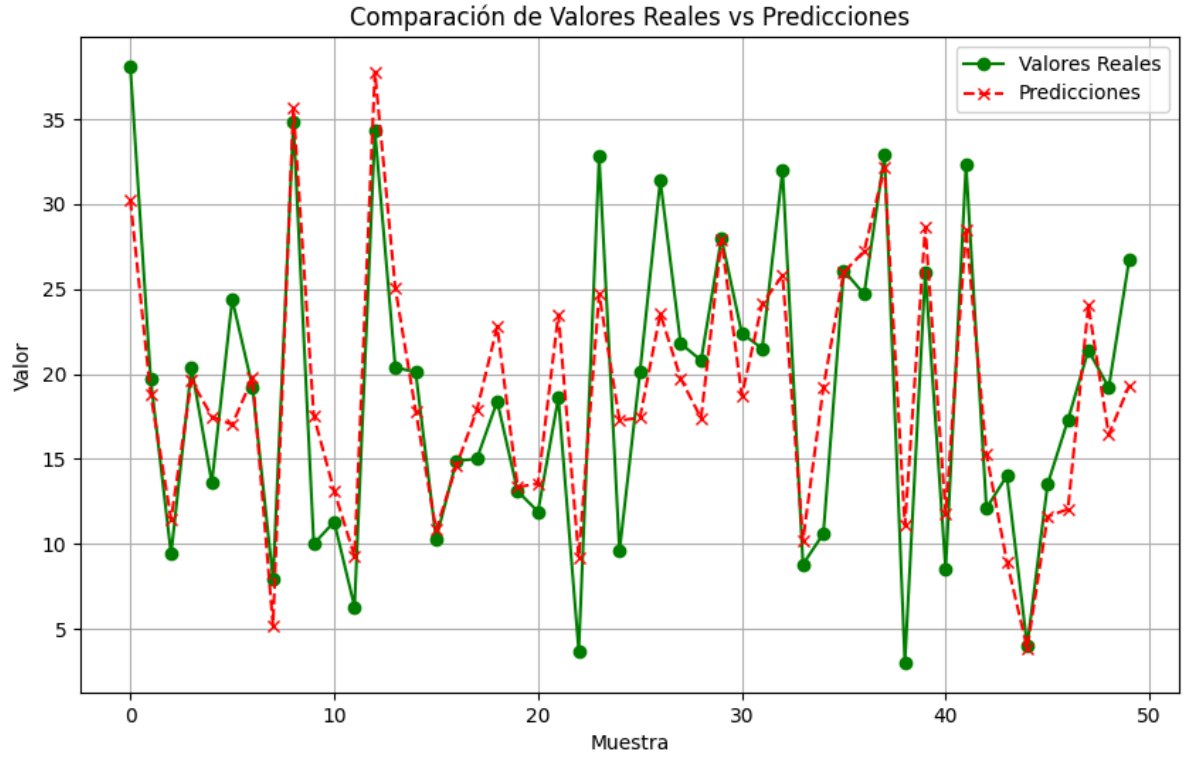
Por otro lado, es importante prestar atención a la complejidad de los árboles de operaciones, puesto que si su complejidad crece demasiado, es probable que aparezcan resultados malos y difíciles de interpretar. Por tanto, conviene establecer un método de penalización dentro de la función de adaptación o plantear este segundo objetivo como un problema de optimización multi objetivo.

Un ejemplo de un buen resultado para estimar podría ser la siguiente función, la cual consigue un bajo error mínimo (4.48) y su complejidad no es excesiva, por lo que sería adecuada para hacer estimaciones generales.




```
971     21.4374 4.48882 164.554
972     46.4632 4.48882 5457.75
Convergencia alcanzada en la generación 972

La mejor solución encontrada es: add(sub(Abdomen, Neck), sub(protected_div(Chest, Biceps), Knee))
Tiempo de procesamiento: 204.431449 segundos
```

Aunque siempre podemos generar árboles más complejos y determinar mediante más pruebas si estos se adaptan bien a otros casos fuera del set de entrenamiento y en algún caso aplicarla a aplicaciones donde la complejidad no sea un problema. Este árbol tan complejo asegura errores mínimos muy bajos (3.46) en el set de entrenamiento:




```
2132    7.54226 3.46187 95.7132
2133    8.80118 3.46187 122.873
Convergencia alcanzada en la generación 2133

La mejor solución encontrada es: sub(sub(sub(sub(sub(Abdomen, neg(protected_div(sub(sub(protected_d
iv(Knee, Density), Height), sub(cos(Biceps), protected_div(Knee, Density))), sub(sub(Abdomen, Neck)
, Neck)))), neg(protected_div(sub(protected_div(protected_div(protected_div(Age, Density), Density)
, Density), sub(sub(sub(Abdomen, Knee), protected_div(Knee, Density)), sub(protected_div(Knee, Dens
ity), sub(Age, Neck)))), sub(Height, sub(Hip, Height))))), neg(protected_div(sub(sub(protected_div(
sub(protected_div(Age, Density), sub(Abdomen, Hip)), Density), sub(sub(Abdomen, Knee), protecte
d_div(Knee, Density)), protected_div(protected_div(Knee, Density), Density))), neg(sub(sub(protecte
d_div(Age, Density), sub(Abdomen, Hip)), sub(sub(Height, Knee), protected_div(Age, Density))))), He
ight))), Knee), Neck)
Tiempo de procesamiento: 568.057697 segundos
```