INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Practica 2: Práctica de programación genética

ALUMNO

Díaz Alvarado Daniel Alejandro

Zavaleta Guerrero Joshua Ivan

UNIDAD DE APRENDIZAJE

Algoritmos Bioinspirados

Docente

Uriarte Arcia Abril Valeria

Grupo:

5BM1

Fecha de entrega:

16 de octubre de 2025

Instrucciones:

Práctica de Programación Genética

1. Revisar la documentación de la biblioteca GPLearn: https://gplearn.readthedocs.io/en/stable/index.html

2. Instalar la biblioteca GPLearn.

3. Implementar el ejemplo de regresión simbólica siguiendo la documentación en la sección de ejemplos: https://gplearn.readthedocs.io/en/stable/examples.html

4. Entregar el código de este programa.

5. Revisar la documentación de la biblioteca Deap: https://deap.readthedocs.io/en/master/index.html

6. Instalar la biblioteca Deap.

7. Implementar el ejemplo de regresión simbólica siguiendo la documentación en la sección de ejemplos: https://deap.readthedocs.io/en/master/examples/gp_symbreg.html

8. Modificar la implementación de regresión simbólica realizada en Deap para que se aplique a la función de 2 variables:

$$x^3 * 5y^2 + {}^{x}/_{2}$$

9. Entregar código y reporte de la práctica del programa modificado en DEAP.

PD: aquí es importante entregar el reporte ya que no van a defender la práctica de forma personal y el reporte debe dejar ver que entendieron el uso de la biblioteca con respecto a los conceptos vistos en clase. También es importante no solo realizar modificaciones mínimas para que funcione, deben realizarse ajustes suficientes para que se trate de aproximar la función dada.

Reporte – primera parte (programa de la biblioteca GPLearn)

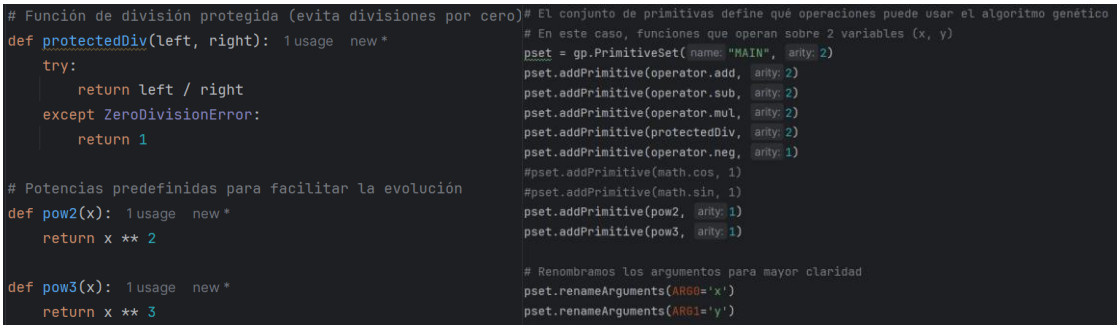
Sobre la ejecución del programa (regresión simbólica con GPLearn), se presentaron algunas complicaciones al momento de querer correr el programa. La razón de esto fue que la versión de GPLearn presentada en el código ya no era compatible con la versión de sklearn, por lo que se tuvieron que hacer unas modificaciones en las versiones que se estaban usando; se tuvo que emplear la versión 1.26.4 de numpy, 1.3.2 de scikit y 0.4.2 de GPLearn.

Reporte – segunda parte (programa de la biblioteca DEAP - regresión simbólica)

La regresión simbólica, como se vio en clase, se dedica a encontrar un programa que coincida con una función matemática dada usando programación genética.

Ahora, la biblioteca DEAP (Distributed Evolutionary Algorithms in Python), es utilizada para la creación de cosas como algoritmos genéticos o programación genética, esta última es la que se utilizará en esta práctica.

En primera instancia, para lograr la regresión simbólica, se utilizaron las operaciones básicas definidas en Python, además, de otras definidas por nosotros, ya sea para ahorrarnos problemas (como del de la división por 0) o para operaciones no definidas por Python (como lo es la potencia al cuadrado y al cubo).



```python
# Función de división protegida (evita divisiones por cero)
def protectedDiv(left, right):  1 usage   new *
    try:
        return left / right
    except ZeroDivisionError:
        return 1

# Potencias predefinidas para facilitar la evolución
def pow2(x):  1 usage   new *
    return x ** 2

def pow3(x):  1 usage   new *
    return x ** 3
```

```python
# El conjunto de primitivas define qué operaciones puede usar el algoritmo genético
# En este caso, funciones que operan sobre 2 variables (x, y)
pset = gp.PrimitiveSet( name: "MAIN",  arity: 2)
pset.addPrimitive(operator.add,  arity: 2)
pset.addPrimitive(operator.sub,  arity: 2)
pset.addPrimitive(operator.mul,  arity: 2)
pset.addPrimitive(protectedDiv,  arity: 2)
pset.addPrimitive(operator.neg,  arity: 1)
#pset.addPrimitive(math.cos, 1)
#pset.addPrimitive(math.sin, 1)
pset.addPrimitive(pow2,  arity: 1)
pset.addPrimitive(pow3,  arity: 1)

# Renombramos los argumentos para mayor claridad
pset.renameArguments(ARG0='x')
pset.renameArguments(ARG1='y')
```

Posteriormente, es importante el definir cosas como los argumentos a utilizar (anteriormente, ya indicamos que la función va a ser de dos variables, y las nombramos ‘x’ y ‘y’), el fitness (decidir si queremos maximizar o minimizar, como en este problema, que se busca minimizar), y empezar a definir los individuos como arboles de primitivas.

```
# FitnessMin → queremos minimizar el error
creator.create( name: "FitnessMin", base.Fitness, weights=(-1.0,))

# Individual → cada individuo es un árbol de primitivas
creator.create( name: "Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)
```

Ahora, para seguir, se introduce la herramienta toolbox, la cual funciona para poder “fabricar” funciones, la siguiente sección de código es necesaria para poder definir las funciones que se encargarán de generar los árboles, crear los individuos y las poblaciones, y convertir los arboles que se obtengan en funciones ejecutables de Python.

```
toolbox = base.Toolbox()

# Cómo generar expresiones (árboles)
toolbox.register( alias: "expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=3)

# Cómo generar individuos y poblaciones
toolbox.register( alias: "individual", tools.initIterate,  *args: creator.Individual, toolbox.expr)
toolbox.register( alias: "population", tools.initRepeat,  *args: list, toolbox.individual)

# Cómo "compilar" un árbol de DEAP a una función de Python
toolbox.register( alias: "compile", gp.compile, pset=pset)
```

Luego, se introduce la función encargada de calcular el fitness de los árboles que se obtengan; esta función logra lo anteriormente mencionado mediante el calculo del error cuadrado, devolviendo finalmente la razón entre el error encontrado sobre el número de puntos tomados (en forma de tupla), ahora, cabe aclarar que esta función necesito de algunos ajustes, puesto que sino se le agrega el try mostrado en la función, existe la posibilidad que algunos valores se vayan al infinito, por lo que, para arreglar eso, en cuanto se identifique un comportamiento así, se penaliza devolviendo un valor exageradamente grande.

```python
def evalSymbReg(individual, points):  1 usage
    func = toolbox.compile(expr=individual)
    sqerrors = []
    for (x, y) in points:
        try:
            val = func(x, y)
            real = 5*x**3*y**2 + x/2
            sqerrors.append((val - real) ** 2)
        except Exception:
            sqerrors.append(1e6)
    return (sum(sqerrors) / len(points),)
```

Con lo anterior, ahora solo queda definir unas últimas opciones para poder ya usar el programa.

Lo primero a definir es la función que se va a encargar de evaluar la función evalSymbReg (anteriormente mostrada) con un conjunto de puntos en específico, en este caso, se opto por un rango de 20 puntos para ‘x’ y ‘y’, lo que nos termina dando un total de 400 puntos.

```python
toolbox.register( alias: "evaluate", evalSymbReg,
                 points=[(x / 10.0, y / 10.0) for x in range(-10, 10) for y in range(-10, 10)])
```

Luego de esto se definen otras funciones que determinan el tipo de selección (torneo en este caso), tipo de cruza, tipo de mutación utilizando el método full.

```python
toolbox.register( alias: "select", tools.selTournament, tournsize=3)
toolbox.register( alias: "mate", gp.cxOnePoint)
toolbox.register( alias: "expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register( alias: "mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)
```

Por último, para obtener a los mejores individuos se crea un objeto HallOfFame que se encargará de guardar los 10 mejores individuos junto con métricas como la media, desviación estándar, mínimo y máximo.

```python
hof = tools.HallOfFame(10)  # Mejores 10 individuos
stats_fit = tools.Statistics(lambda ind: ind.fitness.values)
stats_size = tools.Statistics(len)
mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
mstats.register( name: "avg", np.mean)
mstats.register( name: "std", np.std)
mstats.register( name: "min", np.min)
mstats.register( name: "max", np.max)
```

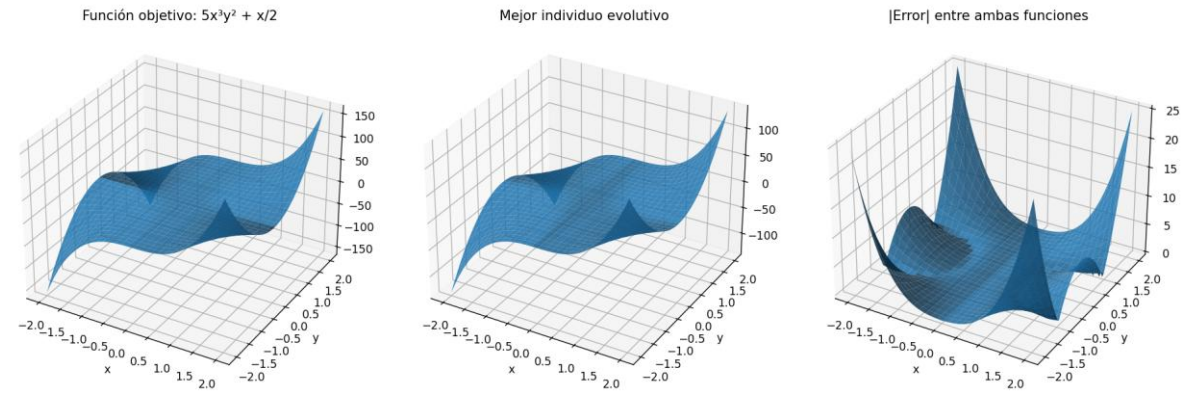
Con lo anterior ya se puede ejecutar el algoritmo de programación genética, estableciendo los parámetros como la semilla, el tamaño de la población, la probabilidad de cruza y mutación, y número de generaciones.

```
random.seed(318)

# Generamos población inicial y ejecutamos la evolución
pop = toolbox.population(n=200)
pop, log = algorithms.eaSimple(pop, toolbox, cxpb: 0.5, mutpb: 0.2, ngen: 40, stats=mstats, halloffame=hof, verbose=True)
```

Por ejemplo, la siguiente captura nos muestra el mejor individuo que encontró para aproximarse a la función objetivo que se le dio al programa (la indicada en las instrucciones arriba):

```
===== MEJOR INDIVIDUO ENCONTRADO =====
🏆Individuo #1 del Hall of Fame
Expresión DEAP: add(mul(pow2(mul(add(y, y), x)), x), pow3(neg(neg(x))))
Expresión simbólica simplificada: x**3*(4*y**2 + 1)
 🔹Similitud numérica con la función real: 99.77%
 🔹Fitness del individuo: 0.027383
```



Como se puede apreciar, se obtiene una función muy parecida. También, como nota, si habilitamos las operaciones del seno y coseno, la función obtenida se acerca más a la original.

Notas

Como se puede apreciar, el programa es capaz de simplificar las expresiones que se generan en los árboles, esto se hizo mediante sympy, sin embargo, la integración se realizo con apoyo de chatgpt, utilizándose su versión 5. Las funciones están incluidas en el programa proporcionado.