Programación genética

Alejandro Mendoza Puig[1]

[1] *Posgrado en Ciencia e Ingeniería de la Computación, Universidad Nacional Autónoma de México, Ciudad Universitaria Unidad de Posgrado, Edificio "C" 1er. nivel, Delegación Coyoacán, C.P. 04510, CDMX.*

Fecha de publicación: 21/04/2024

INTRODUCCIÓN

La programación genética es una rama de la computación evolutiva que funciona similar a los algoritmos genéticos, pero en lugar de realizar las operaciones de **cruza, mutación, selección y reemplazo** sobre los individuos de una población los cuales codifican a genotipos que representan una posible solución al problema, con la programación genética se realizan los mismos operadores pero a una función la cual esta buscando optimizar su comportamiento. La población pueden ser vistos como programas de computadora (1)

Los operadores se utilizan para realizar cambios en la estructura de la función, cambiando las operaciones y los valores de las constantes que la componen, con el fin de encontrar una función que se ajuste a los datos de entrenamiento y que sea capaz de generalizar a los datos de prueba.

En esta tarea trabajamos sobre dos problemas, el primero es el de paridad, en el cual dados 3 bits se tiene que generar una respuesta binaria, 1 cuando el número de unos en los 3 bits son pares y 0 de lo contrario. El segundo problema es una regresión simbólica en la cual contamos con 21 valores de X junto con el resultado de f(x).

Para el primer problema estaremos utilizando los siguientes parametros:

- F1: and, or, not
- F2: and, or, not, xor
- T: A, B, C

Para el segundo problema estaremos utilizando los siguientes parametros:

- F: +, -, *, div, cos, sin, log, exp, abs, sqrt, x^y $T : x, R, k's$

Objetivos

- Implementar un algoritmo de programación genética para resolver el problema de paridad.
- Implementar un algoritmo de programación genética para resolver el problema de regresión simbólica.
- Realizar un análisis de los resultados obtenidos.

DESARROLLO

La implementación se realizó en python, utilizando la paquetería de DEAP. Inicialmente intenté hacerlo con arboles y con notación polaca, pero debido a unos bugs (en ambos acercamientos) que no he podido resolver opté por utilizar la paquetería DEAP que recomendaron en clase. Para realizar un algoritmo de programación genética se realizan los siguientes pasos:

- Definición de función de evaluación: Se define una función que pueda recibir a un individuo y lo evalue sobre el set de datos que se tiene de entrenamiento. En el caso de regresión se envia el error cuadrático medio y en el caso de paridad se envia el accuracy (cuantos correctos tuvo).
- Definición de los operadores: Se definen los operadores (nodos) que se utilizarán para resolver los problemas. En **Paridad** fueron (AND, OR, NOT, XOR) y en **Regresión** fueron (+, -, *, division protegida, cos).
- Crear estructura del individuo: Se crea la clase individuo junto con el fitness que tendrá (maximización o minimización).
- Armar toolbox: Se arma la çaja de herramientas"que se utilizará durante las generaciones. Se definen elementos como:
  - Método de selección
  - Método de cruza
  - Método de mutación
  - Max/min de nodos para la población inicial
- Definir la métricas de cada generación: Se definen las métricas que queremos salvar sobre la ejecución de cada generación (promedio, peor/mejor individuo, desviación estandar etc.)

RESULTADOS

Paridad sin XOR

En el caso del problema de paridad sin la función de XOR el modelo logró llegar a individuos los cuales tenian 8/8 aciertos. Aunque no en todas las ejecuciones lo lograban, en múltiples corridas los mejores individuos no excedieron de 7/8 aciertos. El mejor individuo que se encontró fue el siguiente:

and_(or_(and_(var0, or_(var1, not_(var0))), var2), or_(or_(var1, and_(0, and_(var0, var1))), var0))

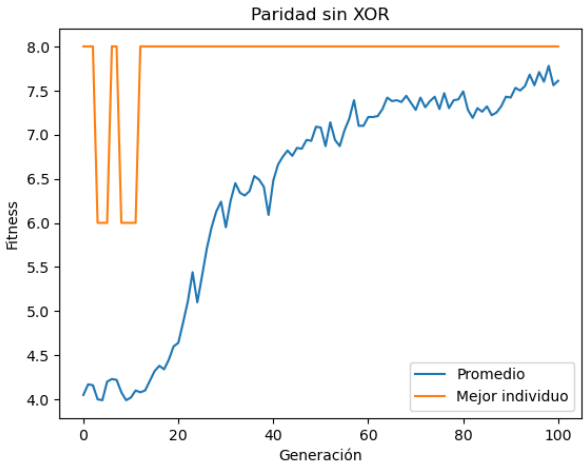

Fig. 1: Resultados de la ejecución del problema de paridad sin XOR

Paridad con XOR

En el caso de paridad utilizando XOR, el modelo fue capaz de encontrar a un individuo que llegara a 8/8 aciertos en un número menor de ejecuciones. Después de correrlo multiples veces algunas soluciones eran notablemente mas largas que otras, aunque lograban solucionar el problema de paridad sin errores. El mejor individuo que se encontró fue el siguiente:

xor(and_(and_(var2, var2), not_(and_(var1, 0))), and_(not_(xor(var0, var1)), not_(and_(var1, 0))))

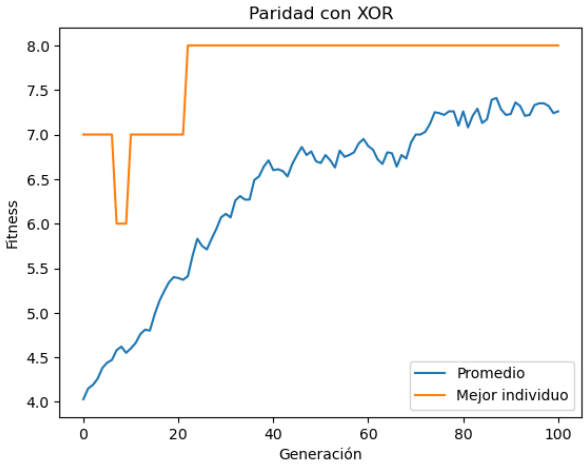

Fig. 2: Resultados de la ejecución del problema de paridad con XOR

Regresión simbólica

Para regresión logistica se utiliza elementos muy similares al problema de paridad, cambiando unicamente los operadores, la función de evaluación y el fitness, en este caso como es un problema de regresión se utiliza error medio cuadrático. El mejor individuo que se encontró fue el siguiente:

sub(sub(sub(add(sub(-1, add(-1, neg(sub(add(cos(x), add(cos(x), 1)), add(cos(sub(sub(protectedDiv(-1, 1, protectedDiv(0, cos(-1))), x), x)), -1))))), sub(sub(protectedDiv(-1, protectedDiv(add(cos(sub(0, cos(-1))), add(sub(sub(protectedDiv(add(cos(-1), add(x, x)), protectedDiv(1, cos(protectedDiv(sub(-1, x), neg(1))))), x), sub(0, x)), -1)), -1)), x), add(cos(-1), sub(neg(sub(add(cos(neg(x)), add(sub(sub(x, x), sub(-1, add(cos(x), add(x, 1)))), 1)), add(cos(1), neg(-1)))), add(add(cos(sub(0, x)), add(sub(protectedDiv(1, cos(protectedDiv(sub(-1, 1), neg(x)))), add(-1, add(protectedDiv(sub(-1, cos(x)), cos(x)), 1))), 0)), add(cos(-1), add(protectedDiv(sub(sub(protectedDiv(-1, protectedDiv(0, cos(1))), x), sub(-1, add(cos(x), add(x, 1)))), cos(protectedDiv(sub(-1, 1), neg(x)))), 1))))))), add(sub(-1, add(cos(cos(protectedDiv(sub(-1, 1), protectedDiv(sub(-1, cos(x)), neg(x))))), add(cos(sub(cos(x), x)), add(cos(sub(0, x)), add(sub(sub(protectedDiv(1, cos(protectedDiv(sub(-1, 1), protectedDiv(sub(-1, cos(x)), neg(x))))), x), sub(-1, add(add(x, add(-1, 1)), add(x, cos(sub(-1, 1)))))), -1))))), sub(neg(sub(add(cos(cos(-1)), add(sub(protectedDiv(add(x, 1), neg(x)), sub(-1, add(add(x, add(protectedDiv(1, cos(x)), cos(protectedDiv(sub(-1, x), neg(1))))), add(x, 1)))), -1)), add(cos(add(protectedDiv(1, cos(protectedDiv(sub(-1, 1), sub(-1, add(cos(-1), cos(x))))), 1)), -1))), 1))), add(sub(x, -1), neg(sub(add(protectedDiv(1, cos(protectedDiv(sub(-1, cos(cos(x))), protectedDiv(sub(-1, cos(x)), neg(x))))), add(sub(sub(protectedDiv(-1, x), x), sub(add(-1, add(protectedDiv(sub(-1, cos(x)), cos(1)), 1)), add(protectedDiv(protectedDiv(1, cos(protectedDiv(sub(-1, 1), neg(x)))), cos(x)), sub(protectedDiv(1, cos(protectedDiv(sub(-1, 1), protectedDiv(sub(-1, cos(x)), neg(x))))), x))), 1)), add(cos(-1), neg(x))))))), mul(sub(add(protectedDiv(sub(protectedDiv(cos(x), cos(protectedDiv(sub(-1, 1), neg(x)))), cos(x)), neg(x)), 1), neg(sub(x, add(cos(-1), neg(-1))))), 0))

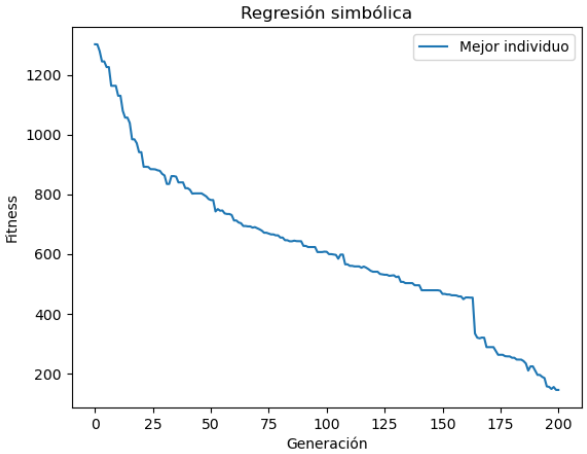

Fig. 3: Resultados de la ejecución del problema de regresión simbólica

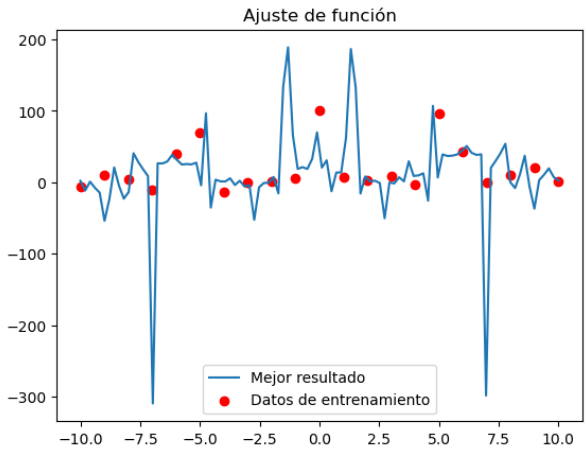

Fig. 4: Ajuste de la regresión simbólica a los puntos de entrenamiento

CONCLUSIONES

Durante el desarrollo de esta tarea tuve algunos problemas con las implementaciones desde cero, fueron problemas de aplicación, no de comprensión de los temas. Pero al utilizar DEAP pude aplicar lo aprendido en clase y en los intentos previos con una paquetería que automatiza algunos de los procesos para realizar el entrenamiento.

En el caso de paridad podemos ver que es un problema el cual es relativamente sencillo para este tipo de algoritmos, tanto usando como no usando XOR, aunque al usar XOR se llego a una solución viable en menos generaciones. Mientras que para el problema de regresión se puede notar que se logra crear una función que es una función sobreajustada (Esto se puede notas ya que para pasar por todos los puntos la función presenta fluctuaciones muy grandes), aunque esto tambien es de esperarse ya que el set de datos fue generado con números aleatorios por lo cual si dejamos correr las poblaciones por muchas generaciones van a seguir haciendo todo para minimizar la función de evaluación.

REFERENCIAS

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, Massachusetts and London, England: MIT Press, 1992.