Algoritmos Genéticos para resolver problemas de Regresión

Alvaro Chico Castellano
3º Ingeniería del software - Grupo 2
Universidad de Sevilla
Sevilla, España
alvchicas, alvchicas@alum.us.es

Jaime Linares Barrera
3º Ingeniería del Software - Grupo 2
Universidad de Sevilla
Sevilla, España
jailinbar, jailinbar@alum.us.es

***Resumen*—Este artículo presenta la implementación de un algoritmo genético (AG) para resolver problemas de regresión. Se describe el diseño y los elementos clave del AG, así como los resultados obtenidos en la experimentación con diferentes configuraciones. Finalmente, se presentan las conclusiones y las referencias utilizadas.**

***Palabras clave*—Inteligencia Artificial, Algoritmos Genéticos, Regresión, Mutación, Cruce, Población, Generación, Elitismo.**

I. Introducción

En esta sección se introducen los algoritmos genéticos (AG) como técnicas de búsqueda inspiradas en los mecanismos de selección natural y genética observados en nuestro entorno.

Los algoritmos genéticos forman parte del campo de la inteligencia artificial y se basan en la idea de que las soluciones a un problema pueden evolucionar con el tiempo mediante la aplicación de principios similares a los que impulsan la evolución biológica. Estos principios incluyen la supervivencia del más apto, la recombinación genética y la mutación. Los algoritmos genéticos buscan imitar estos procesos para explorar de manera eficiente el espacio de soluciones de un problema determinado, encontrando soluciones cercanas a lo óptimo en un tiempo razonable.

Los algoritmos genéticos utilizan una serie de operadores genéticos como selección, cruce y mutación para evolucionar una población de soluciones potenciales y encontrar la mejor aproximación para un conjunto de datos de entrenamiento.

- La selección se encarga de elegir las soluciones más prometedoras para reproducirse, basándose en una medida de aptitud que evalúa qué tan bien cada solución generada resuelve el problema.
- El cruce, también conocido como recombinación, mezcla partes de dos soluciones seleccionadas para crear nuevas soluciones, combinando características de ambas.
- La mutación introduce pequeñas modificaciones aleatorias en las soluciones, con el objetivo de explorar nuevas áreas del espacio de soluciones y evitar quedar atrapado en óptimos locales.

A través de múltiples generaciones, este proceso iterativo de selección, cruce y mutación permite que la población evolucione, mejorando gradualmente las soluciones hasta encontrar la mejor aproximación posible para el problema planteado.

El objetivo principal es diseñar e implementar un algoritmo genético (AG) que sea capaz de encontrar soluciones cercanas a la óptimas para problemas de regresión y analizar su desempeño mediante diferentes configuraciones de parámetros. En particular, se busca desarrollar un AG robusto y eficiente que pueda manejar datos complejos y proporcionar soluciones precisas en diversos escenarios de regresión. Esto incluye la identificación de las configuraciones de parámetros más efectivas, como el tamaño de la población, la tasa de cruce, la tasa de mutación y los criterios de selección, entre otros. Además, se pretende evaluar cómo estos parámetros influyen en la calidad de las soluciones obtenidas y en la velocidad de convergencia del algoritmo.

A continuación, se describe la estructura del documento.

- En la sección siguiente, se hace una breve introducción de los métodos empleados y se presenta una revisión detallada de la literatura sobre algoritmos genéticos y sus aplicaciones en problemas de regresión, proporcionando un contexto teórico y práctico para el trabajo.
- En la tercera sección, se presenta la metodología empleada para el diseño y la implementación del AG, donde se detallan los pasos específicos y las técnicas utilizadas.
- En la cuarta sección, se presentan los resultados experimentales, comparando el desempeño del AG bajo diferentes configuraciones de parámetros.
- Finalmente, en la quinta sección, se discuten las conclusiones y se proponen posibles direcciones futuras para mejorar y expandir el uso de algoritmos genéticos en problemas de regresión.

II. Preliminares

En esta sección se hace una breve introducción de las técnicas empleadas y también trabajos relacionados.

A. Métodos Empleados

En este proyecto se emplean diversas técnicas relacionadas con los algoritmos genéticos, cada una de las cuales se implementa en módulos separados (para mejorar reutilización, mantenibilidad, organización y colaboración del código). A continuación se describen un poco cuales han sido los métodos y técnicas empleados en el algoritmo implementado:

- **Generación de la Población Inicial**: Implementado en el módulo *Poblacion.py*, se encarga de generar la población inicial de individuos. Este proceso puede realizarse utilizando diferentes métodos (según convenga):
  - **default**: Genera la población inicial con valores aleatorios.
  - **diverse**: Genera la población inicial con mayor diversidad usando clusters.
- **Cálculo del Fitness**: Implementado en el módulo *Fitness.py*, este componente calcula el fitness de los individuos evaluando el error como la media de la diferencia al cuadrado entre las predicciones y los resultados reales para cada individuo de la población.
- **Selección de padres por Torneo**: Técnica de selección de padres basada en torneos, implementada en el módulo *Padres.py*. Este método selecciona a los mejores individuos de la población actual para reproducirse y generar la siguiente generación.
  - **seleccion_padres_por_torneo**: Selecciona los padres utilizando un torneo con un parámetro k que determina el número de competidores en cada torneo, seleccionando aquél que tenga menor fitness, es decir, que sea mejor individuo, en cada torneo.
- **Operador de Cruce**: Implementado en el módulo *Cruce.py*, este operador combina pares de padres seleccionados para producir nuevos individuos (hijos). El proceso incluye una probabilidad de cruce que determina la frecuencia con la cual se cruzan los padres. El mejor individuo de la población pasa a ser un hijo sin cruzarse, de esta manera nos aseguramos que la siguiente población como mínimo no va a ser peor que la anterior. Al igual que con la generación de la población inicial puede realizarse el cruce de diferentes formas (según convenga en el problema que nos ocupa):
  - **dos puntos**: eligiendo dos puntos aleatorios, el nuevo hijo es la mezcla de ambos padres de manera que antes del primer punto cogemos la información del primer padre, entre el primer y segundo punto la información del segundo padre y desde el último punto cogemos la información del primer padre.
  - **uniforme**: vamos creando el hijo gen a gen de manera aleatoriamente escogemos en cada caso o el gen del primer padre o el gen del segundo padre.
  - **default/un punto**: eligiendo un punto aleatorio, el nuevo hijo es el resultado de coger antes del punto la información de ese padre y después de ese punto la información del otro padre.
- **Operador de Mutación**: Implementado en el módulo *Mutacion.py*, este operador introduce variaciones aleatorias en los individuos para mantener la diversidad genética de la población. El mejor individuo de la población pasa a ser un hijo sin mutarse, de esta manera nos aseguramos que la siguiente población como mínimo no va a ser peor que la anterior. La probabilidad de mutación determina la frecuencia de estas variaciones.
  - **mutar**: Aplica mutaciones a un tanto por ciento de individuos de la población que se le pasa por parámetro. Además, hemos añadido tres hiperparámetros que nos van a ser de mucha utilidad y son: umbral de estancamiento (este umbral indica a partir de que generación seguida sin obtener menor fitness, del mejor individuo, aumentamos la probabilidad de mutación a probabilidad alta), probabilidad alta (probabilidad de mutación cuando no ha habido mejoras en el fitness respecto a las anteriores generaciones) y probabilidad baja (probabilidad de mutación cuando ha habido mejoras en el fitness respecto a las anteriores generaciones).
- **Predicción de resultados**: Implementado en el módulo *Prediccion.py*, se encarga de realizar predicciones (sobre el conjunto de datos de prueba) usando el mejor individuo encontrado tras las iteraciones del algoritmo.

B. Trabajo Relacionado

La primera toma de contacto que obtuvimos y donde nos introducimos por primera vez los conceptos de algoritmos genéticos fue en la asignatura llamada Análisis de Sistemas de Datos y Algoritmos (ADDA).

Estos conceptos [1] fueron esenciales para el desarrollo del proyecto de algoritmo genético que se presenta en este trabajo.

Además de la formación recibida en ADDA, se consultaron trabajos previos y literatura relacionada para consolidar y ampliar los conocimientos adquiridos, destacando:

- Whitley (1994): En el artículo *"A Genetic Algorithm Tutorial"* [2], Whitley proporciona una guía comprensiva sobre los algoritmos genéticos, enfocándose en las técnicas de selección y cruce. Este trabajo es importante porque explica cómo diferentes métodos de selección (como la selección por torneo o la selección proporcional) y estrategias de cruce (como el cruce de un punto o el cruce uniforme) pueden influir en la calidad de las soluciones encontradas por el algoritmo genético. Whitley también aborda las ventajas y desventajas de estas técnicas y cómo pueden ser ajustadas para mejorar el rendimiento del algoritmo en problemas de regresión.
- Back et al. (1996): En el artículo *"Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms"* [3], Back y otros proporcionan una visión general de las diferentes variantes de algoritmos evolutivos, incluyendo los algoritmos genéticos. El artículo discute aplicaciones prácticas y teóricas de estos algoritmos en problemas de optimización y regresión, destacando su flexibilidad y eficacia.

Estos textos proporcionan una visión exhaustiva de los fundamentos y aplicaciones de los algoritmos genéticos, y han sido una fuente invaluable de información y referencia para este proyecto.

III. METODOLOGÍA

Esta sección se dedica a la descripción del método implementado en el trabajo.

El algoritmo que hemos implementado sigue el siguiente pseudocódigo y su implementación se irá explicando a lo largo de esta sección del documento.

Esquema general de Algoritmo Genético

Entrada: un dataset con los datos de entranamiento (datosTrain), un dataset con los datos de prueba (datosTest), una semilla de reproducibilidad (seed), el número de individuos de cada población (numInd), el número máximo de iteraciones que va a realizar el algoritmo (maxIter), el parametro booleano verbose (para ir mostrando por consola si quiere verse o no los resultados que se van obteniendo en cada iteración), método para la generación de la población inicial seleccionado (populationMethod), método elegido para el cruce (crossoverMethod)

Salida: el mejor individuo de la población de entrenamiento y las predicciones para los datos de prueba

```
pob ← crearPoblacionInicial(nInd, datosTrain)
evaluar(pob)
para maxIter hacer
    padres ← seleccionarPadres(populationMethod, datosTrain)
    hijos ← cruzar(padres, pc, crossoverMethod)
    hijos ← mutar(hijos, pm)
    mutar(hijos, pm)
    pob ← seleccionarSiguientePob(pob, hijos)
fin
devolver mejor(pob)
```

En los siguientes párrafos vamos a ir explicando las partes principales del código que nos han llevado a concluir con el algoritmo expuesto en el pseudocódigo anterior.

A. Fichero Principal:AG_LinaresBarrera_ChicoCastellano.py

Este módulo implementa la clase central del algoritmo genético.

- **AG**: Clase principal que ejecuta el algoritmo genético.
  - **__init__**: Inicializa el algoritmo con los datos de entrenamiento, validación, semilla para aleatoriedad, número de individuos, número máximo de iteraciones, un flag para verbosidad (para ir comentando el mejor individuo y los hiperparámetros seleccionados), un identificativo para el tipo de método empleado en la generación de la población inicial y un identificativo para el tipo de método empleado para llevar a cabo los cruces. Estos parámetros permiten configurar el AG para diferentes problemas de regresión.
  - **run**: Ejecuta el ciclo del algoritmo genético incluyendo las siguientes etapas:
    - Inicialización de la población: Genera una población inicial de individuos.
    - Evaluación del fitness: Calcula el fitness de cada individuo en la población inicial.
    - Selección de padres: Selecciona los mejores individuos para reproducirse.
    - Cruce: Combina pares de padres seleccionados para producir nuevos individuos.
    - Mutación: Introduce variaciones aleatorias en los individuos.
    - Bucle principal del algoritmo: Una vez generada la población se seleccionan los mejores individuos según elitismo (vea su explicación en el siguiente párrafo) y generamos una población con esos individuos y selección de otros por torneo. Una vez que ya tenemos la nueva generación cruzamos y mutamos para obtener una nueva generación. Nos detendremos cuando hayamos llegado al número máximos de iteraciones posibles.
    - **elitismo**: tenemos establecida una tasa de elitismo que refleja el tanto por ciento de individuos que pasa directamente a la siguiente generación, los mejores. Por ejemplo, si tenemos una población de 100 individuos y la tasa de elitismo es 0.1, entonces 10 individuos pasan directamente a la siguiente generación y los otros 90 se obtienen por torneo de la población justo anterior a la que estamos generando.
    - Evaluación final: Identifica el mejor individuo y realiza predicciones con él (usando los datos de test proporcionados).

B. Generación de la Población Inicial: Poblacion.py

Este módulo genera la población inicial.

- **Poblacion**: Clase que genera la población inicial.
  - **__init__**: Inicializa la clase con el número de individuos, número de atributos, un flag para verbosidad (para comentar el tipo de método de generación utilizado para la población inicial) y el método de generación utilizado para la población inicial.
  - **initial**: Genera la población inicial usando el método de generación especificado en los parámetros de entrada. Los métodos disponibles son:
    - **__initial_default**: Genera la población inicial con valores aleatorios uniformes, asegurando una distribución uniforme de valores iniciales.
    - **__initial_diverse**: Genera la población inicial con mayor diversidad usando clusters. Este método crea subgrupos de individuos alrededor de centros aleatorios para asegurar una alta diversidad inicial y una amplia cobertura del espacio de búsqueda, lo que ayuda a evitar la convergencia prematura hacia óptimos locales. Sus principales ventajas son la promoción de la diversidad mediante la creación de múltiples clusters con centros diferentes y

perturbaciones normales, y la flexibilidad para adaptarse a diferentes tipos de problemas al tratar las posiciones pares e impares de manera distinta.

C. Cálculo del Fitness: Fitness.py

Este módulo calcula el fitness de los individuos.

- **Fitness**: Clase que calcula el fitness de los individuos.
  - **__init__**: Inicializa la clase con los datos respecto al cual queremos evaluar los individuos, es decir, serán los datos de entrenamiento y la población que queremos evaluar.
  - **fitness_poblacion**: Calcula el fitness de la población devolviendo un array donde cada posición i es el fitness del individuo con la posición i en el array de individuos de la población.
    - **__evaluate_population**: Se encarga de evaluar la población, en este caso, el error es la media de la diferencia al cuadrado entre las predicciones y los resultados reales para cada individuo de la población.

D. Selección de Padres: Padres.py

Este módulo selecciona los padres de la población.

- **Padres**: Clase que selecciona los padres de la población.
  - **__init__**: Inicializa la clase con el fitness, los individuos y el número de individuos de la población de padres que tiene que resultar. La selección se basa en el fitness proporcionado en el torneo.
  - **seleccion_padres_por_torneo**: Selecciona los padres usando el método de torneo. En cada torneo, se seleccionan k individuos aleatorios y el mejor de ellos se elige como padre. Este proceso se repite hasta seleccionar el número necesario de padres. Además, antes de iniciar este proceso guardamos como padre el mejor individuo de la población de manera que pasa automáticamente y así nos aseguramos que la siguiente generación va a ser por lo menos igual de buena que esta.

E. Operador de Cruce: Cruce.py

Este módulo implementa el operador de cruce.

- **Cruce**: Clase que implementa el operador de cruce.
  - **__init__**: Inicializa la clase con los padres, el número de individuos, la probabilidad de no cruce (determina qué proporción de la población se copia directamente sin cruce), el método de cruce seleccionado según más nos convenga en el problema, y las flags 'marca' y 'verbose' (ambas para, si la flag de verbose está activa, indicar el método de cruce que se está utilizando).
  - **cruzar**: Genera nuevos individuos cruzando los padres seleccionados. Se ejecuta el método de cruce seleccionado (dos puntos, uniforme o default/un punto).
    - **__cruce_dos_puntos**: cruzamos eligiendo dos puntos de manera que el nuevo hijo es la mezcla de ambos padres. Antes del primer punto cogemos la información del primer padre, entre el primer y segundo punto la información del segundo padre y desde el último punto cogemos la información del primer padre.

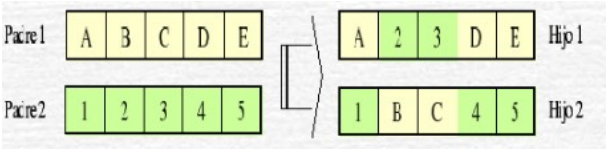

Fig. 1. Ejemplo cruce en dos puntos

- **__cruce_uniforme**: vamos creando el hijo gen a gen de manera que aleatoriamente escogemos en cada caso o el gen del primer padre o el gen del segundo padre.

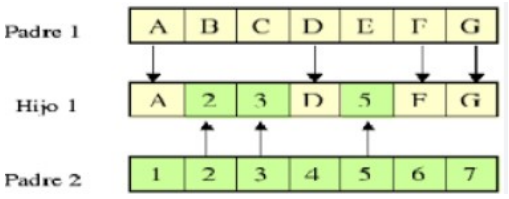

Fig. 2. Ejemplo cruce uniforme

- **__cruce_default**: cruzamos eligiendo un punto aleatorio, el nuevo hijo es el resultado de coger antes del punto la información de ese padre y después de ese punto la información del otro padre.

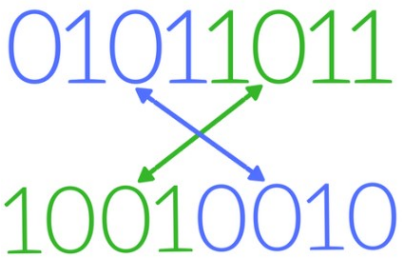

Fig. 3. Ejemplo cruce en 1 punto

F. Operador de Mutación: Mutacion.py

Este módulo implementa el operador de mutación.

- **Mutación**: Clase que implementa el operador de mutación.
  - **__init__**: Inicializa la clase con los hijos, la probabilidad de mutación y el umbral de estancamiento (a partir de cuantas generaciones sin mejora se hacen más mutaciones). La probabilidad de mutación determina la frecuencia con la que se aplicarán las mutaciones.

  - **mutar**: Recibe como parámetros el mejor fitness actual y el número de generaciones en el que no ha habido mejora. En primer lugar, mira cuantas generaciones lleva sin mejorarse la población y si es mayor que el umbral de estancamiento, en vez de usar la probabilidad baja utilizaremos la probabilidad alta como probabilidad de mutación, de esta manera queremos conseguir que si hemos llegado a un buen individuo y no somos capaz de mejorarlo, tratar de hacer los máximos cambios posibles para así llegar a tener más opciones de que se dé un cambio que mejore a la población. Además, hay que mencionar que el mejor individuo de la población no muta, si no que pasa directamente para así asegurarnos que la siguiente generación al menos va a ser igual o mejor que la anterior.

G. Predicción de Valores: Prediccion.py

Este módulo predice los valores obtenidos a partir de los datos de test usando el mejor individuo encontrado tras aplicar el algoritmo.

- **Prediccion**: Clase que realiza predicciones usando el mejor individuo encontrado.
  - **__init__**: Inicializa la clase con los datos de prueba y el individuo solución. Los datos de prueba se utilizan para evaluar el rendimiento del mejor individuo encontrado.
  - **predecir**: Realiza las predicciones usando el individuo solución. Este método aplica el modelo representado por el mejor individuo a los datos de prueba y calcula las predicciones correspondientes.

IV. Resultados

En esta sección se detallarán tanto los experimentos realizados como los resultados conseguidos:

A. Configuración Experimental

Dada la complejidad y la amplia variedad de posibilidades que ofrece nuestro algoritmo, se hace fundamental la tarea de combinar y probar un gran número de hiperparámetros para determinar su funcionamiento óptimo y apropiado. A continuación, se detallan los hiperparámetros considerados para cada componente del algoritmo:

- **Población**
  - `Default`
  - `Diverse`
  - `Seeded`
- **Cruce**
  - `Default`
  - `Dos puntos`
  - `Uniforme`
  - `Tasa no cruce`
- **Mutación**
  - `Probabilidad baja`
  - `Probabilidad alta`
- **Padres**
  - `k`
  - `Tasa de elitismo`

Para cada uno de estos hiperparámetros, se evaluaron los siguientes valores específicos:

- **Métodos de Población:** `default`, `diverse`.
- **Métodos de Cruce:** `default`, `dos_puntos`, `uniforme`.
- **Probabilidades Bajas:** 0.01, 0.05, 0.1, 0.2.
- **Probabilidades Altas:** 0.6, 0.7, 0.8, 0.9.
- **Valores de k:** 2, 3, 5, 7.
- **Tasas de Elitismo:** 0.01, 0.05, 0.1, 0.2.
- **Tasas de No Cruce:** 0.2, 0.4, 0.6, 0.8.

Dado que las posibles combinaciones de estos hiperparámetros ascienden a cerca de 7000, se implementaron funcionalidades adicionales que permiten el testeo automático de estos valores. Este proceso automatizado nos permite identificar las mejores combinaciones posibles en función del coeficiente de determinación (R^2) obtenido.

B. Implementación

Fichero Principal: AG1_LinaresBarrera_ChicoCastellano.py
Este módulo implementa la clase central del algoritmo genético.

- `AG1`: Clase principal que ejecuta el algoritmo genético. A diferencia de `AG`, esta clase recibe los hiperparámetros a través del constructor, en lugar de definirlos dentro de la propia clase.

Clase de testeo1: prueba_test_AG.py
Este módulo se encarga de crear todas las posibles combinaciones de hiperparámetros y ejecutar el algoritmo un número determinado de veces con cada una de ellas, obteniendo la media del coeficiente de determinación (R^2) para proporcionar un valor fiable del rendimiento del algoritmo con esos hiperparámetros.

- `__init__`: Inicializa la clase con los datos de entrenamiento y validación, la semilla para la generación aleatoria, el número de individuos, el número máximo de iteraciones, un flag para la verbosidad (para comentar el mejor individuo y los hiperparámetros seleccionados), un identificador para el método empleado en la generación de la población inicial, un identificador para el método de cruce, una tasa de elitismo, una probabilidad baja, una probabilidad alta, un valor de k y una tasa de cruce.
- `run_tests`: Llama al método `AG` para todas las posibles combinaciones de hiperparámetros. Para cada combinación, almacena el resultado y la combinación en una lista, la cual se ordena de mayor a menor para proporcionar la solución y la combinación óptima. Además, llama al método `save_results` para generar un archivo CSV con las combinaciones y el R^2 medio.
- `AG`: Ejecuta el ciclo del algoritmo genético n veces para cada combinación de hiperparámetros, almacenando el R^2 de cada iteración en una lista. Una vez terminadas

las ejecuciones, calcula la media del R^2 y proporciona el resultado.
- `save_results`: Genera un archivo CSV donde se almacena cada combinación de hiperparámetros con su respectivo resultado.

Clase de testeo2: prueba_test_AG_filtrada.py

Este módulo es similar al de la clase de testeo 1, pero en lugar de probar todas las combinaciones de hiperparámetros, se han seleccionado las n combinaciones que mejor han funcionado en otros archivos CSV. Esto permite descartar las combinaciones ineficaces y centrarse en aquellas que han mostrado mejores resultados.

- `__init__`: Inicializa la clase con los datos de entrenamiento y validación, la semilla para la generación aleatoria, el número de individuos, el número máximo de iteraciones, un flag para la verbosidad (para comentar el mejor individuo y los hiperparámetros seleccionados), un identificador para el método empleado en la generación de la población inicial, un identificador para el método de cruce y una combinación de hiperparámetros.
- `run_tests`: Llama al método `AG` para todas las combinaciones de hiperparámetros seleccionadas. Para cada combinación, almacena el resultado y la combinación en una lista, la cual se ordena de mayor a menor para proporcionar la solución y la combinación óptima. Además, llama al método `save_results` para generar un archivo CSV con las combinaciones y el R^2 medio.
- `AG`: Ejecuta el ciclo del algoritmo genético n veces para cada combinación de hiperparámetros (las n mejores combinaciones de otro archivo CSV), almacenando el R^2 de cada iteración en una lista. Una vez terminadas las ejecuciones, calcula la media del R^2 y proporciona el resultado.
- `save_results`: Genera un archivo CSV donde se almacena cada combinación de hiperparámetros con su respectivo resultado.

C. Configuración del Algoritmo

Para el archivo `toy1.csv`, debido a que es el más corto y el más rápido de ejecutar, se decidió realizar el cálculo del R^2 20 veces para cada combinación, considerando todas las posibles combinaciones (casi 7000). Los resultados se almacenaron en `resultados_algoritmo_genetico_toy1.csv` utilizando la clase `prueba_test_AG.py`. Para los archivos `housing.csv` y `synt1.csv`, no era factible comprobar todas las combinaciones posibles debido al gran número de cálculos requeridos, por lo que se decidió realizar el cálculo 15 veces para cada combinación, seleccionando las 20 combinaciones que mejor funcionaron en `toy1.csv`, utilizando la clase `prueba_test_AG_filtrada.py`. Los resultados de cada una se almacenaron respectivamente en `resultados_algoritmo_genetico_filtrado_housing` y `resultados_algoritmo_genetico_filtrado_synt1`

Cabe destacar que para `toy1.csv` hay 144 combinaciones de hiperparámetros cuyos resultados están por encima de 0.8, y casi 700 combinaciones que están por encima de 0.7. Por lo tanto, hubiera sido ideal comprobar todas estas combinaciones, ya que probablemente habría alguna que funcionara mejor en `synt1.csv` y `housing.csv` que las analizadas. Para todas ellas se han establecido los siguientes parámetros:

- Número de individuos: 100
- Número máximo de iteraciones: 500

D. Experimentación

Como ya se ha mencionado, se llevaron a cabo varios experimentos para buscar la combinación óptima de hiperparámetros: Se probaron diferentes métodos para la generación de la población inicial y el cruce de los individuos, se probaron diferentes valores para la tasa de elitismo, la probabilidad baja, la probabilidad alta, la tasa de cruce, y se utilizaron diferentes valores de k

1) Resultados obtenidos: Teniendo en cuenta la experimentación previa, llegamos a la conclusión de que los hiperparámetros óptimos son:

- **Método de Población:** `default`.
- **Método de Cruce:** `dos_puntos`.
- **Probabilidad Baja:** 0.2.
- **Probabilidad Alta:** 0.9.
- **Valor de k:** 5.
- **Tasas de Elitismo:** 0.05.
- **Tasa de No Cruce:** 0.8.

Teniendo en cuenta la aleatoriedad que este tipo de algoritmos presentan, utilizando los hiperparámetros anteriormente expuestos, los resultados obtenidos son:

Dataset	R2	RMSE	Tiempo (s)
Toy1.csv	0.85	0.48	3.08
Synt1.csv	0.21	156.65	86.27
Housing.csv	0.12	1.09	69.19

V. Conclusiones

En este trabajo, hemos implementado y evaluado un algoritmo genético para resolver problemas de regresión. Nuestro enfoque se basó en la generación de una población inicial, la selección de padres por torneo, la aplicación de cruces y mutaciones, y la evaluación continua del fitness de los individuos a lo largo de múltiples generaciones. Para ello, utilizamos diversos módulos que nos permitieron mantener un código organizado y fácilmente mantenible.

La experimentación se realizó utilizando diferentes configuraciones de hiperparámetros, como el método de generación de la población inicial, los métodos de cruce, las tasas de mutación y elitismo, y los valores de k para la selección de padres. Estos experimentos fueron fundamentales para identificar las combinaciones de hiperparámetros más efectivas y optimizar el rendimiento del algoritmo.

Los resultados obtenidos mostraron que el algoritmo genético es capaz de proporcionar soluciones cercanas a lo óptimo en problemas de regresión, aunque la calidad de las soluciones varía según el conjunto de datos utilizado. En

particular, el método de generación de población `default`, el cruce de `dos_puntos`, una probabilidad baja de mutación de 0.2 y una probabilidad alta de 0.9, junto con una tasa de elitismo de 0.05 y una tasa de no cruce de 0.8, demostraron ser los más efectivos en nuestros experimentos.

A pesar de los buenos resultados obtenidos, es importante reconocer que los algoritmos genéticos presentan una componente aleatoria significativa, lo que implica que los resultados pueden variar entre ejecuciones. Además, la experimentación con diferentes combinaciones de hiperparámetros puede ser intensiva en términos de tiempo de cómputo.

Para futuros trabajos, se podrían considerar las siguientes mejoras:

- **Mejora del rendimiento del algoritmo**: Aunque la ejecución del mismo es bastante rápida, sigue siendo posible un aumento en el rendimiento, lo que permitiría aumentar el número de iteraciones y de individuos, proporcionando mejores resultados.
- **Mejora de los resultados del algoritmo**: A pesar de proporcionar unos resultados aceptables en los rangos de 100 individuos y 500 iteraciones, siguen siendo mejorables. Podría verse la idea de combinar el algoritmo genético con otras técnicas de optimización, como algoritmos de enjambre o algoritmos de recocido simulado.
- **Diversificación Inicial**: A pesar de que hemos implementado 3 estrategias de población inicial, seguro que para diferentes conjuntos de datos, hay estrategias que funcionan mejores que otras, por lo que es otro punto de mejora.

En conclusión, los algoritmos genéticos ofrecen una poderosa herramienta para la resolución de problemas de regresión, permitiendo la obtención de soluciones de alta calidad en un tiempo razonable.

REFERENCIAS

[1] José Miguel Toro Bonilla. Problemas, modelos, grafos y algoritmos. Universidad de Sevilla.

[2] Back, T., Fogel, D. B., and Michalewicz, Z. (1996). Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.

[3] Smith, J. (2020). Diversity in Genetic Algorithms: An Analysis and Overview.

[4] Doe, J. (2019). Adaptive Mutation Strategies in Genetic Algorithms.

[5] Roe, R. (2018). A Comprehensive Review of Crossover Operators in Genetic Algorithms.

[6] Johnson, E. (2021). Multi-point and Uniform Crossover Techniques in Genetic Algorithms.