Conceptos Clave de AIS en este Contexto:

1. Detección de Anomalías ("Propio" vs. "No Propio"):
    - "Propio" (Self): Comportamiento, tráfico o patrones de sistema que son considerados normales y benignos.
    - "No Propio" (Non-Self): Comportamiento malicioso, ataques o desviaciones significativas de lo normal, que deben ser detectadas como anomalías.
    - El objetivo del sistema es aprender a distinguir estas dos categorías.
2. Aprendizaje de "Anticuerpos" (Detectores/Reglas):
    - Cada "anticuerpo" es una regla (o un conjunto de condiciones) que define un patrón de "no propio" (o, alternativamente, de "propio").
    - El GA se encargará de evolucionar una población de estas reglas, buscando las más efectivas.
    - Umbral de Afinidad: En nuestro contexto de reglas, la "afinidad" se traduce en qué tan bien una regla se ajusta o "cubre" una instancia de tráfico. Si las condiciones de una regla se cumplen para un paquete o conexión, se dice que la regla tiene una "alta afinidad" con ese evento, lo que podría llevar a una alerta.

Ejemplo: Detección de Intrusiones con GA y RBML (DEAP)

Vamos a simular un escenario donde queremos detectar paquetes de red anómalos o intrusiones basándonos en algunas características simplificadas (tamaño del paquete, duración de la conexión, número de fallos de login).

1. Definición del Problema y Datos:

- Características del Tráfico:
    - packet_size (tamaño en bytes)
    - connection_duration (duración en segundos)
    - failed_login_attempts (número de intentos fallidos de login)
- Clases:
    - 0: Tráfico normal ("Propio")
    - 1: Tráfico anómalo/intrusión ("No Propio")

2. Representación del Individuo (Regla/Conjunto de reglas):

Cada "individuo" en nuestro algoritmo genético será un *conjunto de reglas*. Cada regla será del tipo:
IF (feature OPERATOR value) THEN ANOMALY

Por ejemplo: IF (packet_size > 1500) THEN ANOMALY
O: IF (failed_login_attempts > 3 AND connection_duration < 5) THEN ANOMALY (Para simplicidad, usaremos reglas simples con un solo operador por ahora).

Un individuo será una lista de tuplas, donde cada tupla representa una regla: (feature_index, operator_type, threshold_value).

- feature_index: 0 para packet_size, 1 para connection_duration, 2 para failed_login_attempts.
- operator_type: 0 para >, 1 para <.
- threshold_value: El valor numérico contra el cual se compara la característica.

3. Función de Fitness:

La función de fitness evaluará qué tan bien un conjunto de reglas (un individuo) detecta anomalías mientras minimiza los falsos positivos. Utilizaremos el F1-Score para la clase "anomalía" (1), ya que es un buen balance entre precisión y recall, especialmente útil en datasets desbalanceados (donde las anomalías son raras).

4. Operadores Genéticos (DEAP):

- Selección: selTournament (selección por torneo).
- Cruce (Crossover): cxTwoPoint (intercambia segmentos de reglas entre dos individuos).
- Mutación: mutRules (mutación personalizada que altera aleatoriamente el índice de la característica, el operador o el umbral de una regla elegida).

Explicación de los Cambios y Relación con los Conceptos de AIS:

1. Programación Evolutiva (EP):
    - Eliminación del Cruce: Hemos comentado la línea toolbox.register("mate", tools.cxTwoPoint) y, lo más importante, hemos cambiado algorithms.eaSimple por algorithms.eaMuPlusLambda.

- eaMuPlusLambda es un algoritmo de EP que genera la nueva población casi exclusivamente a través de la mutación. Se selecciona un número mu de padres, se generan lambda_ hijos por mutación (estableciendo cxpb=0.0 y mu_prob=1.0 para que la mutación sea predominante), y luego se seleccionan los mu mejores individuos de la combinación de padres e hijos.
- La mutación (mutate_rule_set) se convierte en el mecanismo principal para explorar nuevas combinaciones de reglas y ajustar las existentes, emulando cómo un sistema inmune biológico genera diversidad mediante mutaciones somáticas para crear nuevos anticuerpos.

2. Datos de Ejemplo y Detección de Anomalías Más Clara:
   - He modificado generate_data para que las anomalías sean mucho más distintas de los datos normales. Por ejemplo:
     - Los ataques de packet_size grande tienen valores claramente fuera del rango normal.
     - Los ataques de failed_login_attempts alto tienen un valor de 0 para lo normal y valores positivos para la anomalía.
   - Este ajuste facilita que el algoritmo de Programación Evolutiva encuentre reglas que capturen estas diferencias, lo que se traduce en un F1-Score inicial y final más alto, haciendo el aprendizaje más perceptible.
3. Simplificación del Código y "Afinidad":
   - Comentarios Detallados: Se han añadido más comentarios explicativos, especialmente en las secciones clave como la generación de datos, la estructura de las reglas y la función de evaluación.
   - Nomenclatura: Nombres de variables y funciones son más descriptivos.
   - Concepto de "Afinidad" Simplificado: He enfatizado que la comprobación de la condición de una regla (feature_value > threshold_value o feature_value < threshold_value) es la implementación del "umbral de afinidad". Si un dato cumple la condición de una regla, significa que tiene suficiente "afinidad" con el patrón de "no propio" que detecta esa regla, y por lo tanto, se clasifica como anomalía.

- Eliminación de la Matriz de Confusión Gráfica: Se ha omitido la visualización seaborn.heatmap para cumplir con el requisito de simplificación visual. Se mantiene el classification_report y ejemplos específicos que son más directos de leer.

4. Flujo Sencillo de RBML + EP para Detección de Intrusiones:
    - Generación de Datos: Simula tráfico normal y tráfico de ataque.
    - Representación de Reglas: Cada "individuo" del GA es un conjunto de reglas (un "sistema inmune").
    - Mutación: Las reglas se modifican aleatoriamente (nuevos "anticuerpos" o ajustes a los existentes).
    - Evaluación: Se mide qué tan bien un conjunto de reglas identifica anomalías (F1-Score como "fitness").
    - Selección: Los conjuntos de reglas más efectivos sobreviven y "se reproducen" (propagando sus características mutadas exitosas).
    - Simulación de Detección: El mejor conjunto de reglas encontrado se aplica para clasificar nuevos eventos, mostrando su capacidad para diferenciar lo "propio" de lo "no propio".

Este ejemplo modificado ilustra de manera más directa cómo la Programación Evolutiva puede ser utilizada para evolucionar sistemas de detección basados en reglas, con una clara analogía a la forma en que los Sistemas Inmunológicos Artificiales operan para identificar anomalías.

El problema principal es que el F1-Score que estás obteniendo es cero (o muy cercano a cero) para la mayoría de los individuos, especialmente en las primeras generaciones. Cuando todos los individuos tienen un fitness de 0, el algoritmo genético no tiene un "gradiente" para seguir: no hay individuos mejores que otros para seleccionar y evolucionar, por lo que las líneas en la gráfica se quedan planas.

Esto suele ocurrir por varias razones interconectadas:

1. Reglas aleatorias iniciales no detectan nada: Con 5 reglas generadas completamente al azar, es muy probable que ninguna de ellas capture efectivamente un patrón de anomalía, resultando en que el conjunto de reglas no clasifique ninguna instancia como anómala. Si predictions solo contiene 0s (no se predice ninguna anomalía), y hay anomalías reales en labels, el F1-score es 0.

2. Tamaño de la población y número de generaciones: Si la población es pequeña o el número de generaciones es insuficiente, el algoritmo puede no tener tiempo o diversidad para "tropezar" con reglas útiles.
3. Probabilidades de mutación: Aunque la programación evolutiva se basa principalmente en la mutación, probabilidades de mutación mal ajustadas (demasiado altas o demasiado bajas) pueden impedir una evolución efectiva. Si son muy altas, las reglas buenas se destruyen antes de propagarse. Si son muy bajas, el espacio de búsqueda se explora muy lentamente.

Vamos a ajustar los parámetros para fomentar una evolución más clara:

- Aumentar N_RULES_PER_INDIVIDUAL: Más reglas por individuo aumentan las posibilidades de que, al menos algunas, sean "útiles" en la población inicial aleatoria.
- Aumentar POP_SIZE: Una población más grande asegura una mayor diversidad en cada generación.
- Aumentar NGEN: Más generaciones dan al algoritmo más tiempo para evolucionar reglas efectivas.
- Ajustar rule_mut_prob y param_mut_prob: Vamos a hacer que los cambios no sean tan drásticos en cada mutación, para permitir que las reglas buenas se refinen en lugar de ser completamente reemplazadas.

Cambios Clave y Justificación:

1. Parámetros Aumentados:
    - N_RULES_PER_INDIVIDUAL = 10 (antes 5): Permite que cada "individuo" tenga un conjunto de detectores más variado desde el inicio, aumentando la probabilidad de que al menos uno sea relevante.
    - POP_SIZE = 200 (antes 100): Una población más grande aumenta la probabilidad de generar una regla "suertuda" al principio y mantiene una mayor diversidad a lo largo de la evolución.
    - NGEN = 100 (antes 70): Más generaciones permiten que el algoritmo explore el espacio de soluciones por más tiempo y refine las reglas.
    - n_samples_anomaly_type1=70, n_samples_anomaly_type2=70 (antes 50, 50): Unas pocas anomalías más en el dataset de entrenamiento aseguran que haya

suficientes ejemplos positivos para que el F1-Score no sea 0 si una regla detecta solo algunas.

2. Probabilidades de Mutación Ajustadas:
    - rule_mut_prob=0.3 (antes 0.4)
    - param_mut_prob=0.2 (antes 0.4)
    - Cuando las probabilidades de mutación son muy altas y el fitness inicial es 0, el algoritmo puede comportarse de manera caótica, destruyendo cualquier pequeña mejora antes de que pueda ser explotada. Al reducir ligeramente estas probabilidades, las mutaciones son menos drásticas, permitiendo que las reglas incipientes "buenas" se refinen y se propaguen, en lugar de ser completamente aleatorizadas de nuevo.
3. Visualización del Gráfico:
    - plt.ylim([-0.05, 1.05]): Se ha añadido esta línea para forzar el rango del eje Y de la gráfica de fitness a ir de -0.05 a 1.05. Esto garantiza que el F1-Score (que siempre está entre 0 y 1) se visualice correctamente, incluso si los valores iniciales son muy bajos. Lo negativo es solo por estética de un rango ligeramente más amplio de lo estrictamente 0 a 1.
    - zero_division=0 en f1_score: Esto suprime el RuntimeWarning que a veces aparece cuando no hay predicciones positivas (o verdaderos positivos) para la clase pos_label. Al establecerlo en 0, te aseguras de que el F1-score sea 0 en esos casos sin un warning, lo cual es el comportamiento correcto cuando el detector no encuentra nada y no hay un verdadero positivo en las predicciones.

Con estos ajustes, es muy probable que observes una evolución clara en la gráfica, viendo cómo el F1-Score promedio y máximo aumentan a medida que el algoritmo evoluciona mejores conjuntos de reglas para detectar las anomalías.