La Programación Evolutiva (PE) es una subdisciplina de la Computación Evolutiva que se inspira en la evolución natural a nivel de especies, priorizando la adaptación fenotípica. A diferencia de los Algoritmos Genéticos, su énfasis tradicional radica en la mutación como el principal y a veces único operador de variación, para generar nuevas soluciones, relegando o incluso omitiendo el operador de cruce. En la PE, una población de soluciones candidatas, a menudo representadas como vectores de números reales o máquinas de estados finitos, es sometida a un proceso iterativo donde cada solución es evaluada mediante una función de aptitud, y las más aptas son seleccionadas para generar descendencia, principalmente a través de mutaciones que alteran sus características. Este enfoque resulta robusto para la optimización de parámetros en sistemas complejos y la evolución de comportamientos o estrategias.

Ejemplo de Programación Evolutiva para Diagnóstico Médico Basado en Reglas (RBML) con DEAP

Planteamiento del Problema:

Queremos que un sistema aprenda a diagnosticar enfermedades basadas en un conjunto de síntomas y resultados de pruebas. Lo haremos construyendo un conjunto de reglas "SI... ENTONCES...".

- Identidad de una "solución": En este contexto de RBML con PE, una "solución candidata" no es un programa entero (como GP) ni todas las reglas a la vez (como algunos enfoques de AG). En la PE, cada individuo en la población es una única regla de diagnóstico. El *conjunto de reglas* que emerge de la evolución de la población es nuestra solución final.
- Formato de Regla: SI (síntoma1=valor Y síntoma2=valor...) ENTONCES (enfermedad).
- Objetivo: La población de reglas debe ser capaz de diagnosticar correctamente la mayor cantidad de pacientes posible, es decir, maximizar la precisión del conjunto de reglas.

¿Por qué Programación Evolutiva (PE) aquí?

Nos enfocamos en la PE por su énfasis en la mutación fenotípica directa del individuo:

1. Representación directa de la regla: Cada individuo es una regla. Manipularemos sus "características" (antecedente y consecuente) directamente mediante mutación.

2. Mutación como operador primario: La evolución de nuevas reglas y la mejora de las existentes dependerán casi exclusivamente de la mutación, que permitirá:
    - Cambiar los síntomas en el antecedente (hacer la regla más específica o más general).
    - Alterar el valor de un síntoma en el antecedente.
    - Cambiar la enfermedad predicha en el consecuente.
3. Foco en el Fenotipo: No nos preocupamos por una codificación compleja de "genes" de la regla para recombinarse. La regla en sí es la unidad fenotípica que muta.

Explicación Detallada del Planteamiento de PE para RBML:

---

1. Representación del Individuo (Rule):

    - class Rule(list): Cada instancia de Rule representa una única regla "SI... ENTONCES...". Herencia de list facilita que el antecedente (list de tuplas (símbolo, valor)) sea directamente manipulable por algunas herramientas de DEAP si fuera necesario (aunque en este caso la mutación es custom).
    - antecedent: Una lista de tuplas (symptom_id, required_value). symptom_id es el índice numérico del síntoma en SYMPTOMS, y required_value es 0 (ausente) o 1 (presente).
    - consequent: El disease_id (índice numérico de la enfermedad).
    - applies(): Un método para verificar si la regla "dispara" para un paciente dado sus síntomas.

---

2. Inicialización de la Población (init_rule):

    - toolbox.register("individual", init_rule): Se crea una población inicial de reglas aleatorias. Se elige un número aleatorio de condiciones para el antecedente y una enfermedad aleatoria para el consecuente. Esto asegura diversidad inicial.

---

3. Función de Aptitud (evaluate_rule_set):

    - ¡Este es el punto más crucial y distintivo para la PE en RBML! A diferencia de AGs donde cada individuo tiene su propio fitness, aquí

el fitness de *cada regla* en la población (ind.fitness.values) se deriva de la precisión de todo el conjunto de reglas (la población actual).

- Para cada paciente en el TRAINING_DATA:
  - Se simula un sistema de votación: Todas las reglas de la población se "aplican" al paciente. Si una regla dispara, su consecuente recibe un voto.
  - El diagnóstico predicho es la enfermedad con la mayoría de los votos. Se manejan empates (se consideran "Desconocido").
  - Se compara el predicho con el real para calcular la precisión general.
- Este valor de precisión (un solo número) se asigna como fitness a *TODOS* los individuos (reglas) en la población actual. Esto significa que las reglas individualmente no son aptas, sino que lo es su contribución al conjunto.

4. Operador de Mutación (mutRule):
   - toolbox.register("mutate", mutRule): Este operador es el "motor" de la evolución. Refleja el énfasis de la PE en la mutación fenotípica.
   - Toma una Rule y devuelve una Rule mutada.
   - Selecciona aleatoriamente una de varias acciones de mutación:
     - change_consequent: Cambia la enfermedad diagnosticada por la regla.
     - add_condition: Añade un síntoma no presente en el antecedente (haciendo la regla más específica).
     - remove_condition: Elimina un síntoma del antecedente (haciendo la regla más general).
     - change_condition_value: Cambia el valor requerido de un síntoma existente (e.g., de Fiebre=1 a Fiebre=0).
   - No hay operador de cruce (cxpb=0.0 en eaMuPlusLambda): Esto es central para el paradigma de PE aquí. Toda la variación proviene de la mutación.
5. Algoritmo Evolutivo (main con eaMuPlusLambda):

- algorithms.eaMuPlusLambda: Un algoritmo de evolución estándar en PE.

  - mu: Número de individuos (padres) que sobreviven a la siguiente generación (élite de la población combinada de padres e hijos).
  - lambda_: Número de hijos generados por mutación.
  - cxpb=0.0: Probabilidad de cruce del 0%, confirmando el enfoque de PE.
  - mutpb=1.0: Probabilidad de mutación del 100% para cada hijo creado, asegurando que todos los hijos son producto de una mutación.

- Ciclo de Surpervivencia (Mu + Lambda):

---

1. Se genera lambda_ descendientes mutando lambda_ padres aleatorios de la población actual (pop).

2. La población de padres (pop) se combina con la población de descendencia (offspring).

3. De esta población combinada (pop + offspring), se seleccionan los mu individuos más aptos para formar la nueva población de la siguiente generación.

Este ejemplo ilustra cómo aplicar los principios de la Programación Evolutiva para hacer evolucionar un sistema de diagnóstico basado en reglas, donde cada regla es un individuo cuya aptitud se mide por la eficacia del conjunto completo de reglas en la población.

Diagnóstico Médico con Programación Evolutiva: ¡Tu Propio Robot Doctor!

Imagina que queremos enseñar a una máquina a diagnosticar enfermedades. ¿Cómo lo hará? No le daremos un manual gigante con todas las reglas, sino que la dejaremos "aprender" por sí misma, inspirándonos en cómo la naturaleza mejora las especies a lo largo del tiempo: ¡la evolución!

Lo que vamos a construir es un "Robot Doctor" que, en vez de hacer juicios complejos, usará un conjunto de reglas muy simples, como: "SI (tienes fiebre Y tos) ENTONCES (diagnóstico: Gripe)".

Nuestro sistema de inteligencia artificial evolucionará estas reglas para que sean cada vez mejores y más precisas.

La Gran Idea: Cada Regla es un "Individuo", la Población es el "Equipo de Diagnóstico"

Aquí está la clave para entender este programa:

- No buscamos una única súper-regla. Buscamos un conjunto o equipo de reglas que, trabajando juntas, hagan el mejor diagnóstico posible.
- Cada "individuo" en nuestra simulación de evolución no es el diagnóstico final, ¡es una sola regla! Piensa en cada regla como un pequeño "mini-doctor" especializado en algo.
- Nuestra "población" (el montón de individuos que evolucionan) es todo el equipo de mini-doctores.

¿Cómo evolucionan? ¡Por "accidentes felices" (Mutación)!

A diferencia de otros tipos de computación evolutiva (como los Algoritmos Genéticos clásicos donde se "mezclan" genes de dos padres), aquí nos centramos en el concepto de Programación Evolutiva (PE). Esto significa:

- Mutación es la estrella: Las reglas individuales no se mezclan entre sí. En cambio, una regla (el "mini-doctor") se duplica y, al duplicarse, puede sufrir un pequeño "accidente" o "mutación". Tal vez cambia su condición (ahora diagnostica si hay "tos Y dolor de garganta" en vez de solo "tos"), o cambia su diagnóstico final (ahora dice "Sarampión" en vez de "Gripe").
- Los mejores sobreviven: Después de que un montón de reglas mutadas aparecen, evaluamos qué tan bien funciona *todo el equipo de reglas* (la población completa). Los equipos cuyas reglas hacen los mejores diagnósticos son los que tienen más "hijos" (nuevas reglas mutadas para la próxima generación).

¡Manos a la Obra! ¿Qué Hace el Código?

El programa tiene siete partes principales, como un buen informe médico:

1. El "Paciente": Definición del Problema

- SYMPTOMS y DISEASES: Aquí definimos nuestro "vocabulario" médico. Qué síntomas existen (Fiebre, Tos, etc.) y qué enfermedades (Gripe, Resfriado, Sarampión, y un "Desconocido" para cuando no hay un diagnóstico claro). Cada uno tiene un número para que el computador los entienda mejor.
- TRAINING_DATA: Esta es nuestra "historia clínica" de pacientes pasados. Para cada paciente, tenemos una lista de sus síntomas (si los tiene o no) y el diagnóstico real que tuvo. ¡Con esto enseñaremos a nuestras reglas!

2. El "Genoma" de la Regla: class Rule

- Piensa en esto como el "plano" de un mini-doctor.
- antecedent: Es la parte "SI" de la regla. Es una lista de condiciones (ej., [(0, 1), (1, 1)] significaría "Si Síntoma 0 (Fiebre) es 1 (presente) Y Síntoma 1 (Tos) es 1 (presente)").
- consequent: Es la parte "ENTONCES" de la regla. Es un número que representa el ID de la enfermedad que diagnostica.
- applies(self, patient_symptoms_vector): Esta es la habilidad de la regla para "leer" a un paciente. Si el paciente cumple todas las condiciones del antecedent de la regla, entonces la regla "se dispara".
- __str__(self): Esto es solo para que, cuando imprimamos una regla, la veamos bonita y entendible, como "SI (Fiebre=1 AND Tos=1) ENTONCES Gripe".

3. "Preparando el Escenario": Configuración DEAP

DEAP es una librería de Python que nos ayuda a hacer estas simulaciones de evolución.

- creator.create("FitnessMax", ...): Le decimos a DEAP que nuestro objetivo es maximizar la aptitud (queremos que nuestras reglas sean lo más precisas posible).
- creator.create("Individual", Rule, ...): Le decimos a DEAP que cada "individuo" que evoluciona será una Rule (nuestra clase de mini-doctor). ¡DEAP añade automáticamente la capacidad de tener un fitness a cada regla!
- toolbox = base.Toolbox(): Es como una caja de herramientas donde registramos todas las funciones que nuestro proceso evolutivo va a necesitar (cómo crear reglas, cómo mutarlas, cómo evaluarlas).

3.1. "El Nacimiento": Inicialización de Reglas (init_rule)

- toolbox.register("individual", init_rule): Esta función le dice a DEAP cómo crear una nueva regla aleatoria cuando empieza la simulación o cuando necesita "iniciar" nuevas reglas. Simplemente elige algunas condiciones y un diagnóstico al azar.

4. "La Evolución por Accidentes": La Mutación (mutRule)

- toolbox.register("mutate", mutRule): ¡Aquí es donde ocurre la magia de la PE! Esta función toma una regla y, con un poco de azar, la modifica. Piensa que la regla se "duplica", pero la copia podría:
    - change_consequent: Cambiar el diagnóstico que da (de Gripe a Resfriado).
    - add_condition: Añadir una nueva condición a su "SI" (de solo "Fiebre" a "Fiebre Y Tos").
    - remove_condition: Quitar una condición (de "Fiebre Y Tos" a solo "Fiebre").
    - change_condition_value: Cambiar si un síntoma debe estar presente o ausente (de "Tos=1" a "Tos=0").
    - change_condition_symptom: Cambiar el tipo de síntoma que busca (de "Fiebre=1" a "Fatiga=1").
- ¡Importante! No usamos cruce (mezcla) aquí. Toda la variación y novedad viene de estas mutaciones aleatorias.

5. "El Doctor Evaluador": La Función de Aptitud (evaluate_rule_set)

- toolbox.register("evaluate", evaluate_rule_set, ...): Esta es la parte más inteligente del programa. Es la que decide qué tan "bueno" es nuestro "equipo de mini-doctores" (la población actual de reglas).
- Proceso de Votación: Para *cada paciente* en nuestra historia clínica:
    1. Todas las reglas de la población (el equipo de mini-doctores) revisan al paciente.
    2. Las reglas que "se disparan" (cuyas condiciones se cumplen) "votan" por su diagnóstico.
    3. El diagnóstico que recibe más votos es el "diagnóstico del equipo".
    4. Si un paciente no "dispara" ninguna regla, o hay un empate de votos, se diagnostica como "Desconocido".

- Calculando la Precisión: Se cuenta cuántos diagnósticos del equipo fueron correctos. Esta precisión (por ejemplo, 0.8 para 80% de aciertos) se convierte en el fitness de *todas* las reglas de la población. ¡Todas las reglas comparten el mismo fitness, porque son un equipo!

6. "El Ciclo de la Vida": El Algoritmo Evolutivo main()

- Población pop: Empezamos con un montón de reglas creadas al azar.
- Bucle de Generaciones: El corazón de la evolución. Repetimos esto muchas veces (100 generaciones):
    1. "Reproducción por Mutación": Seleccionamos reglas de la población actual al azar (los "padres") y las mutamos (usando mutRule) para crear una nueva "progenie" (los "hijos"). Creamos el doble de hijos que de padres, para tener mucha diversidad.
    2. "Combinar Familias": Juntamos a los "padres" (la población actual) con sus "hijos" (la nueva progenie mutada) en una gran población combinada.
    3. "Evaluación de Rendimiento": Volvemos a llamar a evaluate_rule_set con esta *nueva población combinada*. El resultado (la precisión de este equipo grande) se asigna como el nuevo fitness a *todas* las reglas individuales en esa población combinada.
    4. "Supervivencia del Más Apto": De esta gran población combinada (padres + hijos), seleccionamos solo a las 50 reglas que tienen el fitness más alto (las que contribuyeron al mejor rendimiento del equipo) para que sean la población de la próxima generación.
- Estadísticas: El programa guarda cómo va mejorando el fitness (la precisión) a lo largo de las generaciones y lo imprime.

Resultados y Visualización:

- Al final, el programa te dirá la precisión final que logró el mejor "equipo de mini-doctores" que evolucionó.
- También imprimirá las reglas individuales que conforman ese equipo final, para que puedas ver sus "conocimientos" aprendidos.
- El gráfico mostrará cómo la precisión máxima y promedio del equipo fue mejorando con cada generación. Deberías ver una curva ascendente, mostrando que el sistema está aprendiendo y optimizando sus reglas.

Este programa simula un proceso de evolución donde mini-doctores (reglas) nacen aleatoriamente, sufren "accidentes" (mutaciones), y solo los equipos de mini-doctores que diagnostican mejor sobreviven y se reproducen (generan más reglas mutadas). Con el tiempo, este proceso "ciego" lleva a un conjunto de reglas muy precisas, como si un cerebro colectivo aprendiera a diagnosticar enfermedades.