Inteligencia Artificial

Introducción

Alejandro Medina Reyes

Índice general

Parte uno

1. Prólogo y agradecimientos

1.1 Prólogo

De manera informal se puede definir a la inteligencia artificial como la simulación de comportamientos inteligentes por parte de un sistema informático.

He decidido escribir este libro debido a que considero que es una rama de la computación muy interesante, además estos últimos años se ha incrementado el interés por la inteligencia artificial, ésto debido principalmente a dos factores, la gran cantidad de información disponible que combinada con técnicas de deep learning nos permite hacer predicciones o generalizaciones muy exactas y es el avance que hemos tenido tecnológicamente, siendo más preciso el desarrollo de potentes unidades de procesamiento gráfico, esto último es relevante ya que son capaces de trabajar eficazmente con operaciones de matrices que son ampliamente utilizadas por las redes neuronales.

A pesar de ser un campo con muchos años de investigación pienso que no hay mejor momento para descubrir todo el potencial que ésta área nos ofrece y explorar cómo puede repercutir en el mundo que nos rodea.

Este libro busca explorar diversos temas del campo de la inteligencia artificial con el fin de introducir diferentes paradigmas con los cuales se trabaja, muchos temas no son descritos a profundidad por lo cual a lo largo del libro trataré de colocar links a algunos recursos que puedan ser de utilidad para investigar más acerca de un tema en específico. Es importante tomar en cuenta que este libro debe ser tomando solo como una introducción a la inteligencia artificial.

1.2 Agradecimientos

Primero quiero agradecer a mis padres ya que me han apoyado en todo mi desarrollo personal y académico, mis logros son un reflejo del gran ejemplo que me han dado y de lo mucho que han hecho por mi. Igualmente es importante reconocer el apoyo de mi hermana en las distintas actividades que he desempeñado y en los proyectos que me he propuesto.

También quiero agradecer a mis maestros que me han enseñado mucho a lo largo de mi carrera

de Ingeniería en Sistemas Computacionales, agradezco principalmente a aquellos que me han motivado a crecer y a perseguir mis sueños además de brindarme los conocimientos y herramientas del curso correspondiente. Entre mis maestros quiero darle las gracias especialmente a tres de ellos ya que sin su apoyo no creo que me hubiera decidido a hacer este libro <TODO– pedirles autorización para ponerlos aquí>.

Finalmente quiero darle gracias a mi esposa que me ha motivado a desarrollarme como persona y que siempre ha estado a mi lado dándome el impulso necesario para llevar a cabo este tipo de proyectos.

II Parte dos: Fundamentos

2. Historia de la inteligencia artificial

2.1 Ideas sobre inteligencia artificial

El deseo del ser humano por entender la inteligencia y ser capaces de replicarla se remonta a la antigüedad, un ejemplo es la existencia de Talos en la mitología griega, un gigante de bronce que protegía a la Creta minoica de posibles invasores, la versión más dominante de su origen dice que Talos era un autómata creado por Hefesto (Dios del fuego y la forja).

Ejemplos de autómatas antropomorfos son el robot de Leonardo da Vinci con la apariencia externa de una armadura, capaz de mover piernas y brazos. Otro ejemplo es el autómata creado por Pierre Jacques-Droz en 1774, su creación podía escribir una carta compuesta por 50 caracteres determinados por el usuario.

2.2 Orígenes de la inteligencia artificial

Si bien muchos consideran que el trabajo de Alan Turing “Computing Machinery and Intelligence” [39] dio inicio al campo de la inteligencia artificial me gustaría iniciar hablando del trabajo realizado por Warren McCulloch and Walter Pitts. En 1943 McCulloch, un neurocientífico, y Pitts, un lógico, publicaron .A logical calculus of the ideas immanent in nervous activity", aquí describieron un modelo de neurona que sentó las bases de lo que serían en un futuro las redes neuronales.

En el artículo publicado en 1950 Turing habló acerca del Test que lleva su nombre, mediante el cual en lugar de determinar si una máquina está “pensando”, tratamos de averiguar si es capaz de ganar en “el juego de la imitación” (Haciendo pensar a un ser humano que está hablando con otro humano), este trabajo fue de gran relevancia para el campo de la IA.

2.3 Nacimiento de la inteligencia artificial como ciencia

La conferencia de Dartmouth (“Dartmouth Summer Research Project on Artificial Intelligence”), realizada en 1956 y que duró 2 meses, es considerada como el evento que dio como resultado el nacimiento de la Inteligencia artificial como ciencia.

En 1955 fue John McCarthy quien decidió organizar un grupo para estudiar la siguiente conjetura: cada aspecto del aprendizaje o característica de la inteligencia puede en principio ser descrito de manera tan precisa que una máquina pueda simularlo. A continuación muestro la propuesta de la conferencia [19]:

"We propose that a 2-month, 10-man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer. "

Es aquí donde de forma oficial se introduce el término Inteligencia Artificial. (Si bien es un término que ya se había utilizado fue aquí donde se popularizó y se convirtió en el término dominante).

2.4 La edad de oro 1956-1974

A la conferencia de Dartmouth le siguió un gran entusiasmo en el campo de la inteligencia artificial, incluso se estimaba que en 20 años se tendrían máquinas completamente inteligentes, a pesar de que era demasiado optimista si hubo avances durante este periodo de tiempo.

Entre los trabajos más relevantes se encuentran algoritmos que usan el paradigma "Reasoning as search", en los cuales se aproximaba a la solución de problemas paso a paso como si de un laberinto se tratará, retrocediendo al llegar callejón sin salida, Allen Newell y Herbert A. Simon trataron de capturar una versión general de este algoritmo. También se dieron avances en el reconocimiento del lenguaje general, ELIZA desarrollado en el MIT permitía conversar mediante frases preprogramadas. A finales de 1960 Marvin Minsky y Seymour Papert propusieron que el estudio de la inteligencia artificial debía dirigirse a solucionar problemas en situaciones simples un enfoque denominado como micro-mundos, Minsky y Papert desarrollaron un robot que era capaz de apilar cubos. También se desarrolló el Perceptrón un tipo de red neuronal artificial que veremos más adelante en este libro.

2.5 El primer invierno de la inteligencia artificial 1974-1980

Debido a las altas expectativas desarrolladas previamente hubo una gran decepción al ver que las promesas de la IA no se cumplían. Hubo un gran recorte en los presupuestos de investigación y el campo de las redes neuronales fue mayormente ignorado debido a las fuertes críticas de Minsky sobre las limitaciones del perceptrón.

2.6 El boom de la inteligencia artificial 1980–1987

La llegada de sistemas expertos (capaces de tomar decisiones como si fuese un experto humano mediante el uso de reglas lógicas) permitieron un nuevo boom en la IA gracias a su utilidad en las empresas.

Otro punto importante es el resurgimiento de las redes neuronales, gracias a las redes de Hopfield y el desarrollo del algoritmo de retropropagación.

2.7 El segundo invierno de la inteligencia artificial 1987-1993

Durante este tiempo se dio otra reducción en las inversiones hacia el campo de la IA, sin embargo hubo avances principalmente en el campo de la robótica. Yo considero que una de las razones por las cuales tuvo lugar este segundo invierno de la IA son las desventajas o limitaciones de los sistemas expertos:

- Existen tareas demasiado complejas, la necesidad de diseñar estas reglas de manera manual es una limitante.
- Existe conocimiento en constante cambio, muchos sistemas expertos requieren que las reglas sean actualizadas manualmente lo cual puede llegar a ser un problema.
- Estos sistemas suelen contener conocimiento de un área específica pero carecen de sentido común.

2.8 Siglo XXI

Como mencione en el prólogo gracias a el aumento de potencia computacional y el acceso a grandes cantidades de información la IA ha visto grandes avances, entre los avances más notorios se encuentran las investigaciones y algoritmos de machine learning y deep learning.

3. Fundamentos de la IA

3.1 Definición

Existen diferentes definiciones del término inteligencia artificial, de acuerdo a las ciencias de la computación la inteligencia artificial es el estudio de agentes inteligentes [26].

Un agente inteligente es capaz de percibir su entorno y actuar sobre él tratando de optimizar algún objetivo, un agente es inteligente debido a que presenta características como el razonamiento o el aprendizaje.

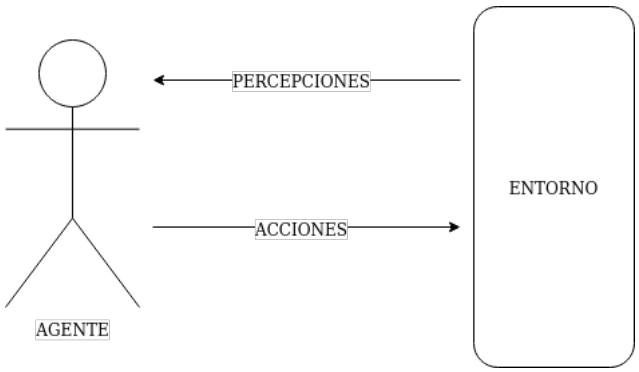


Figura 3.1: Representación de un agente

Otra definición determina que la inteligencia artificial es el desarrollo de sistemas capaces de interpretar correctamente información externa, aprender de esta información para cumplir con ciertos objetivos o tareas adaptándose de manera flexible [12].

Dependiendo de la definición diversos tipos de desarrollos pueden considerarse inteligentes o no, por ello yo prefiero una definición más abierta como la siguiente: la inteligencia artificial es la simulación de comportamientos inteligentes por parte de un sistema informático.

3.2 Ciencias relacionadas con la IA

La inteligencia artificial se relaciona con diversas ciencias que sirven de base para el desarrollo de esta ciencia, algunos ejemplos son los siguientes:

- Filosofía
- Lógica/matemática
- Ciencias computacionales
- Psicología
- Biología
- Neurociencia

Es evidente como estas ciencias han ayudado al desarrollo de la inteligencia artificial, gracias a la teoría de la evolución se lograron generar algoritmos evolutivos, el estudio del cerebro nos dio ideas sobre la manera de tratar problemas como el reconocimiento de imágenes, la lógica nos permitió modelar la manera en la cual razonamos y gracias a la matemática somos capaces de formalizar los modelos para poder implementarlos en los sistemas desarrollados.

3.3 Paradigmas de la inteligencia artificial

De acuerdo a Palma y Marín [18] existen 4 paradigmas principales:

1. Simbólico o representacional: El conocimiento se representa por medio de descripciones declarativas y en lenguaje natural, éstos son los hechos, otro conjunto de conocimientos son las reglas de inferencia que describen las relaciones sobre los hechos, al aplicar dichas reglas sobre un conjunto de conceptos de entrada se razona y se obtiene una inferencia. Un ejemplo de este tipo de desarrollos son los sistemas expertos, este paradigma fue dominante desde 1956 a 1986.
2. Situado o reactivo: Toda conducta es resultado de una percepción, por lo cual éstas se tienen una conexión directa, condicionada o secuencial.
3. Conexionista: Este paradigma corresponde a las redes neuronales artificiales (RNA) se definen modelos con entradas y salidas en los cuales se ajustan parámetros de la red mediante diferentes algoritmos de aprendizaje.
4. Híbrido: Para resolver diversos problemas existe la necesidad de integrar soluciones que corresponden a distintos paradigmas, por ello los sistemas híbridos son de gran utilidad para problemas reales.

Una ventaja del paradigma simbólico es que es fácil interpretar lo que sucede dentro del sistema, en cambio en sistemas conexionistas suele ser complicado determinar las razones detrás de una decisión, sin embargo, actualmente se han desarrollado técnicas como Grad-CAM [29] que permiten reducir esta incertidumbre en el campo de clasificación de imágenes.

III Parte tres: Técnicas clásicas

4. Busquedas inteligentes

4.1 Introducción al capítulo

Antes de empezar a hablar directamente sobre estos temas relacionados principalmente con los grafos quiero hacer mención de un fenómeno descrito por John McCarthy “As soon as it works, no one calls it AI any more”; ésta frase me parece particularmente interesante debido a que la delimitación de los temas que comprenden la IA como ciencia no están perfectamente definidos, habrá autores que consideren estos temas como una rama de las estructuras de datos y no como parte de la IA.

En los inicios de la inteligencia artificial era sorprendente cuando una máquina lograba algo remotamente inteligente y el asombro llevaba a generar altas expectativas del alcance de esta ciencia. Hoy en día quizá no se vea como algo tan sorprendente pero debido a que algunos de estos algoritmos nacieron siendo parte de la IA he decidido dedicarle una pequeña sección.

Como último comentario respecto a esto quiero mencionar a Deep Blue, primera máquina capaz de vencer a un campeón mundial de ajedrez, esta computadora utiliza un algoritmo que en realidad no es tan inteligente como en principio aparenta, más adelante en este capítulo veremos a más detalle parte de su funcionamiento.

No profundizaré en este tema, sin embargo a continuación proporcionaré un link a el libro inteligencia artificial: introducción y tareas de búsqueda de Roberto J. de la Fuente López. (`http://www.aconute.es/iartificial/documentos/ia_intro_busqueda.pdf`)

4.2 ¿Qué es una búsqueda inteligente?

Una búsqueda inteligente es aquel algoritmo que nos permita recorrer una estructura de datos de manera eficiente para obtener una solución potencialmente óptima.

4.3 La IA que venció al campeón del mundo

Son famosos los juegos entre Garry Kasparov y Deep blue, antes de ver el funcionamiento de esta computadora veremos un poco de la historia de estos encuentros.

Creo que es poco mencionado el hecho de que Kasparov ganó la primera partida en 1996, se dieron seis juegos de los cuales 2 fueron ganados por Kasparov y uno por Deep Blue, el otro terminó en empate. (Link de la victoria de Kasparov sobre Deep Blue: `http://hemeroteca.abc.es/nav/Navigate.exe/hemeroteca/madrid/abc/1996/02/19/084.html`)

En la partida de 1997 Deep Blue derrotó a Kasparov, este último ganó un solo juego y Deep Blue ganó 2, los otros 3 quedaron en empate. Algo interesante es el hecho de que Kasparov acusó de hacer trampa a IBM [10] después del segundo juego ya que mostraba signos de inteligencia o creatividad, IBM negó esto y dijo que solo se había dado intervención humana entre los juegos (lo cual estaba permitido en las reglas acordadas).

4.4 ¿Cómo funcionaba Deep Blue?

El algoritmo detrás de Deep Blue no es tan inteligente como hace parecer, incluso uno de sus programadores (Joe Hoane) menciona en una entrevista que no es un proyecto de inteligencia artificial cuando se le preguntó cuánto de su trabajo era dedicado específicamente a la inteligencia artificial en emular el pensamiento humano [27].

Las principales características de Deep Blue eran las siguientes [5]:

- Libro de jugadas iniciales: Esto le permitía a la computadora realizar buenos movimientos iniciales
- Hardware especializado: Deep Blue contaba con chips especializados que permitían evaluar tableros de ajedrez con una gran rapidez, la función de evaluación contaba con más de 8000 características y cada chip tenía una velocidad de búsqueda de 2 a 2.5 posiciones por segundo.
- Paralelización de búsqueda: Deep Blue era un sistema con alta paralelización contando con más de 500 procesadores disponibles para realizar el árbol de búsqueda.

El algoritmo de búsqueda que utilizó Deep Blue está basado en el algoritmo alpha-beta que se detallará más adelante en este capítulo.

Si se quiere profundizar a detalle en el funcionamiento del sistema de esta computadora recomiendo leer el siguiente articulo (`https://doi.org/10.1016/S0004-3702(01)00129-1`).

4.5 El algoritmo Minimax

El algoritmo de minimax nos permite elegir el mejor movimiento en un juego con adversario considerando que éste último siempre escogerá el peor movimiento para nosotros (el mejor para él).

En el juego existen dos jugadores:

1. Maximizador (MAX): trata de obtener la puntuación más alta.
2. Minimizador (MIN): trata de obtener la puntuación más baja.

Algoritmo de minimax con movimientos alternativos:

1. Generación del árbol de juego. Se generarán todos los nodos hasta llegar a un estado terminal (o a alguna condición determinada).
2. Uso de función de evaluación sobre los nodos terminales.

3. Calcular el valor de los nodos superiores a partir del valor de los inferiores, dependiendo de si el nivel corresponde a MAX o MIN se escogerá el valor más alto o más bajo.
4. Elegir la jugada valorando los valores que han llegado al nivel superior. Para ilustrar el funcionamiento de este algoritmo a continuación mostraré en imágenes los diferentes pasos con el ejemplo del juego de gato (Si X gana el estado vale 1, Si O pierde el estado vale -1, Si se empata el valor es 0):

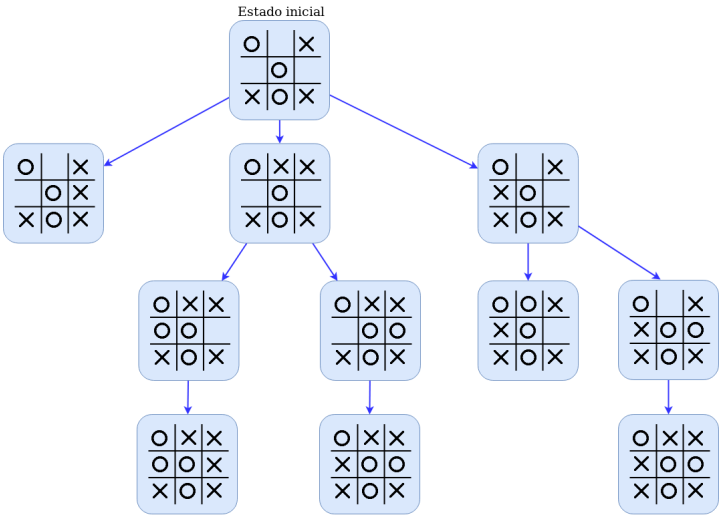


Figura 4.1: Generación de estados

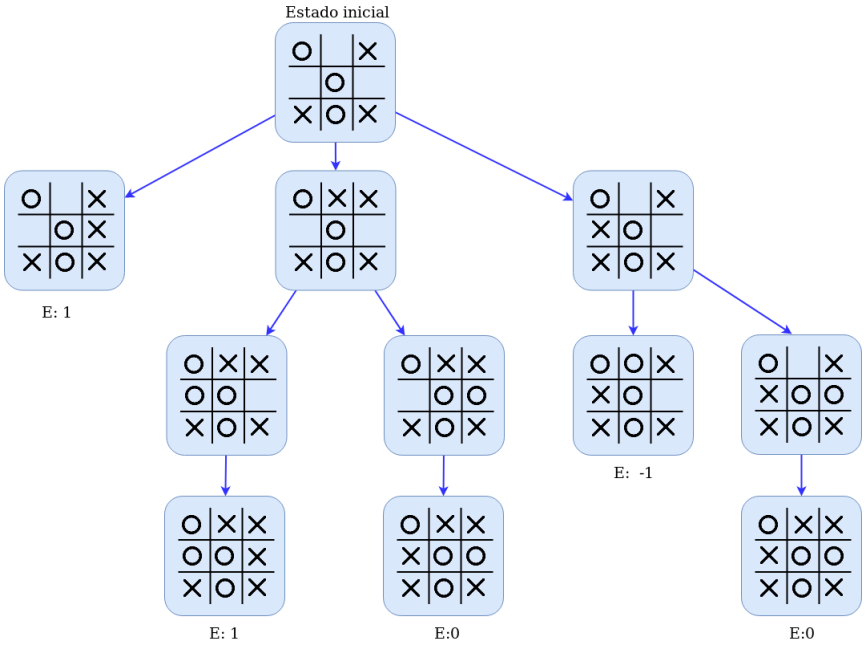


Figura 4.2: Evaluación de estados finales

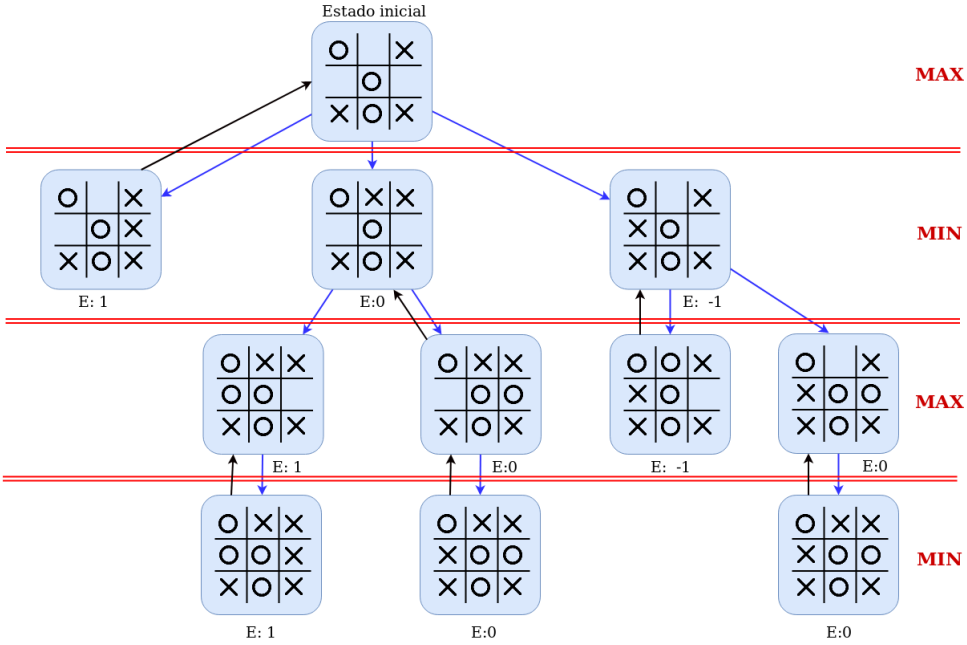


Figura 4.3: Cálculo de los valores en los nodos superiores

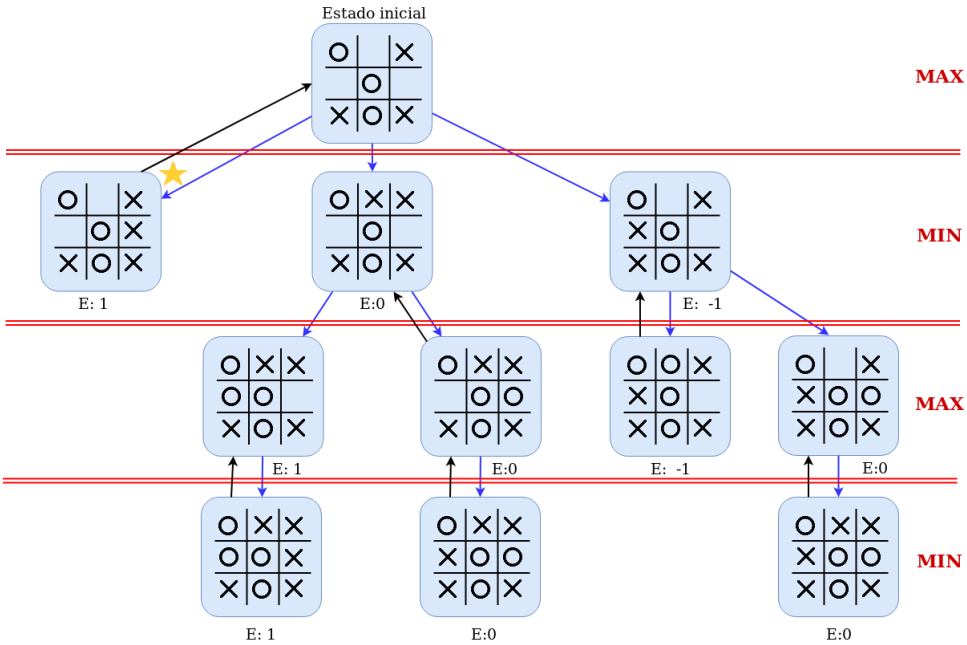


Figura 4.4: Elección de estado o jugada

El algoritmo MINIMAX es un procedimiento recursivo, a continuación se presenta el pseudocódigo correspondiente, se recomienda al lector analizar como el siguiente pseudocódigo hace lo mismo que se describió con anterioridad.

MINIMAX(nodo N)

SI el nodo N es terminal:

    MINIMAX(N) <-- Evaluar nodo N

SI NO

    Generar nodos sucesores N_1, N_2, N_3, ... N_n

    Evaluar los nodos generados MINIMAX(N_1), MINIMAX(N_2), ... MINIMAX(N_n)

    SI N es un nodo MAX

        MINIMAX(N) <-- MAX (N_1, N_2, N_3, ... N_n)

    SI N es un nodo MIN

        MINIMAX(N) <-- MIN (N_1, N_2, N_3, ... N_n)

Figura 4.5: Pseudocódigo del algoritmo minimax

Es importante notar que dependiendo del "juego" sobre el cuál se esté aplicando este algoritmo varía la función de evaluación y la función sucesor, encargada de generar los nuevos estados.

En el pseudocódigo descrito con anterioridad se generan todos los estados finales posibles, en el juego de gato esto no es un gran problema ya que el número de estados posibles es relativamente pequeño (alrededor de 362,800), sin embargo en otros juegos como el ajedrez este número es mucho más alto por lo cual se puede verificar a qué nivel de profundidad pertenece el nodo y si se ha llegado a ese límite establecido previamente evaluar el nodo aunque no sea un estado final, esto implica además el tener que generar funciones de evaluación más complejas ya que en ese punto no se puede saber con certeza si el jugador ganará o perderá.

Otra nota importante es que en este pseudocódigo es en los nodos donde se determina si es el turno del maximizador o minimizador, esto también podría hacerse implementando un parámetro en la función y cambiando su estado cada que ésta fuese llamada.

Ejercicio de programación:

Yo recomiendo realizar el siguiente ejercicio para fortalecer los conocimientos adquiridos: En cualquier lenguaje de programación programar una inteligencia artificial capaz de jugar gato utilizando el algoritmo Minimax.

(Ejemplo: `https://github.com/amr205/TicTacToe-AI---Minimax`)

4.6 El algoritmo Alpha-beta pruning

El algoritmo Alpha-beta pruning tiene el objetivo de realizar la misma tarea que el algoritmo Minimax sin embargo poda las ramas que no necesitan ser revisadas, sigue regresando el mismo resultado que el algoritmo minimax pero reduce el nivel de nodos que visita.

En este caso incluiré primero el pseudocódigo y posteriormente procederé a explicar el funcionamiento de este algoritmo.

```
Alpha-beta(Nodo N, Número alpha, Número beta)
SI el nodo N es terminal:
    MINIMAX(N, alpha, beta)  <-- Evaluar nodo N
SI NO
    Generar nodos sucesores N1, N2, N3, .... Nn
    SI N es un nodo MAX
        nodoMayor = -infinito
        PARA cada nodo sucesor Ni
            actual = Minimax(Ni, alpha, beta)
            alpha  = MAX(alpha, actual)
            nodoMayor = MAX(nodoMayor, actual)
            SI beta <= alpha
                BREAK
        MINIMAX(N, alpha, beta)  <--  nodoMayor
    SI N es un nodo MIN
        nodoMenor = infinito
        PARA cada nodo sucesor Ni
            actual = Minimax(Ni, alpha, beta)
            beta  = MIN(alpha, actual)
            nodoMenor = MIN(nodoMenor, actual)
            SI beta <= alpha
                BREAK
        MINIMAX(N, alpha, beta)  <--  nodoMenor
```

Figura 4.6: Pseudocódigo del algoritmo alpha-beta pruning

Se puede observar que el funcionamiento es muy similar al algoritmo Minimax pero se utilizan dos variables, alpha y beta. Se puede observar que alpha guardaría el mejor estado posible que el maximizador tiene, y beta el mejor estado posible para el minimizador, por la manera en la que se visitan los nodos cuando beta es menor o igual no tiene mucho sentido continuar revisando la rama

ya que el jugador contrario ya tiene una mejor opción en un nivel superior.

Un ejemplo de cómo funciona puede ser muy útil para entender el funcionamiento de este algoritmo, en lo personal considero que el siguiente video de Sebastian Lague muestra un ejemplo muy completo y descrito paso por paso (`https://youtu.be/l-hh51ncgDI?t=546`).

Ejercicio de programación:

Yo recomiendo realizar el siguiente ejercicio para fortalecer los conocimientos adquiridos:

En cualquier lenguaje de programación programar una inteligencia artificial capaz de jugar ajedrez utilizando el algoritmo Alpha-beta pruning.

(Ejemplo: `https://github.com/amr205/Chess-AI-using-Alpha-Beta-pruning`)

5. Algoritmos evolutivos

5.1 Introducción al capítulo

Estos algoritmos inspirados en la evolución natural son útiles debido a que nos permiten tratar problemas que con técnicas de búsqueda no informada requerirían de un tiempo de proceso demasiado grande y con técnicas de búsqueda informada corren el riesgo de llegar a un óptimo local.

Una ventaja de los algoritmos evolutivos es que se requiere solo una pequeña cantidad de conocimiento específico sobre el problema que se está tratando, en concreto la función de evaluación (Fitness function) que debe ser optimizada en el proceso [37], más adelante en este capítulo serán más evidentes las razones que llevan a esta afirmación.

5.2 Orígenes

Desde la década de los 50s los científicos han estudiado este tipo de algoritmos, en 1954 Nils Barricelli creó el primer algoritmo genético que imitaba la reproducción y mutación natural, su objetivo no era resolver problemas de optimización, sino crear vida artificial. Durante los siguientes años científicos como Alexander Fraser usaron su trabajo, Fraser quería simular la evolución debido a que observarla de manera directa en nuestro mundo requeriría de millones de años.

John Holland es considerado una de las personas más importantes en el campo de los algoritmos genéticos, ya que él introdujo el uso de una población para evaluarla y posteriormente usar procesos como el crossover, recombination, etc. En 1975 publicó su libro que sería la base teórica de muchos trabajos posteriores.

En 1988 John Koza, patentó su idea de usar algoritmos evolutivos para la generación de programas, continuó su trabajo con múltiples publicaciones relacionadas con la programación genética, por lo cual su trabajo es de mucha importancia en esta área de los algoritmos evolutivos.

En 1986 Holland sentó las bases de los sistemas clasificadores (LCS), estos algoritmos tenían el objetivo de solucionar la tarea de clasificación y utilizan elementos de aprendizaje y algoritmos

genéticos, Stewart Wilson continuó el desarrollo de nuevos sistemas clasificadores como el "Zeroth-level" usando métodos más modernos de aprendizaje reforzado.

5.3 Definición

Los algoritmos evolutivos tienen su base en la selección natural, una definición que yo considero apropiada es la siguiente: Los algoritmos evolutivos mediante la heurística son capaces de resolver tareas de optimización imitando aspectos de la evolución natural, suelen trabajar en poblaciones completas de posibles soluciones para una determinada tarea (Streichert, 2002).

NOTA: Diferencia entre un algoritmo evolutivo y un algoritmo genético

Un algoritmo genético es una subclase de los algoritmos evolutivos. Todo algoritmo evolutivo está basado en las leyes de la evolución natural, un algoritmo genético tiene sus bases en el uso de poblaciones, cruzamiento o recombinación (crossover) y mutación. En cambio otros tipos de algoritmos evolutivos se basan principalmente en la mutación.

5.4 Clasificación

Existen distintos tipos de algoritmos evolutivos, en este capítulo se revisarán aquellos más populares, sin embargo también hay sistemas y algoritmos que presentan un comportamiento híbrido, estos algoritmos no serán cubiertos hasta que se hayan visto los temas necesarios para poder abordarlos de manera completa, un ejemplo de esto último es el algoritmo NEAT (NeuroEvolution of Augmenting Topologies) que combina los algoritmos genéticos con el paradigma conexionista.

A continuación se presentan los tipos de algoritmos evolutivos más comunes:

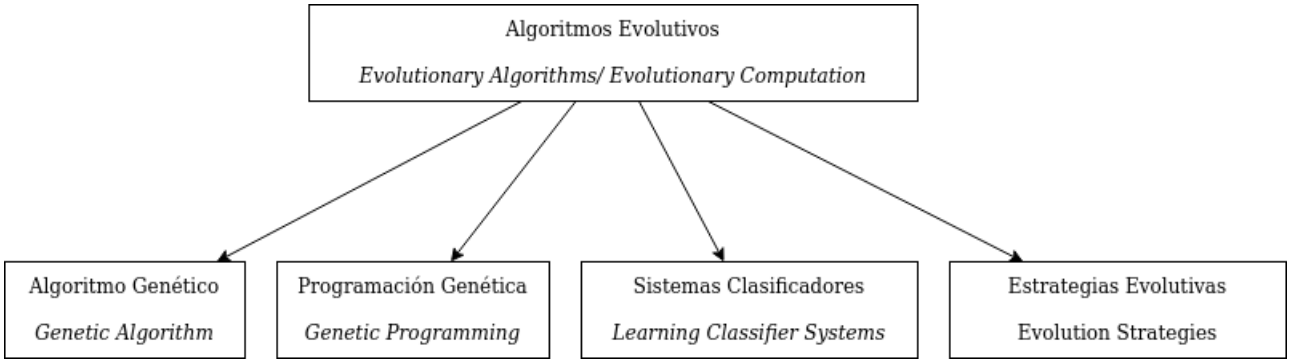


Figura 5.1: Clasificación de los algoritmos evolutivos más comunes

5.5 Algoritmos genéticos

Los componentes principales de los algoritmos genéticos son los siguientes:

- Una función de evaluación a optimizar (fitness function)
- Una población de cromosomas
- Un operador de selección
- Un operador de cruzamiento
- Un operador de mutación

Antes de describir éstas partes, veamos el funcionamiento básico del algoritmo.

1. Se genera una población inicial

2. Se evalúa la población (si algún elemento supera algún nivel barrera se da por terminado el algoritmo)
3. Se seleccionan los mejores individuos de la población y se guardan en un grupo(en inglés a este grupo se le llama mating pool)
4. Se seleccionan pares del grupo generado y se aplica el operador de cruzamiento, también se aplica el operador de mutación de acuerdo a una tasa de mutación determinada por el desarrollador, al terminar la generación de la población se vuelve al paso 2.

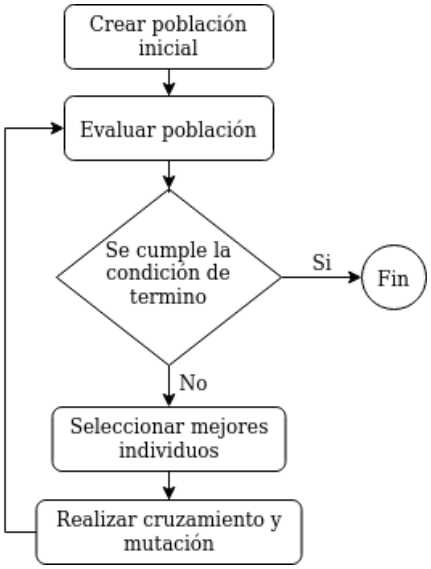


Figura 5.2: Diagrama de flujo de un algoritmo genético

A continuación se presenta un pseudocódigo de la implementación de un algoritmo genético simple, se recomienda leerlo y regresar a esta figura cuando se realice el ejercicio de programación propuesto.

```
BEGIN /* Algoritmo Genetico Simple */
    Generar una poblacion inicial.
    Computar la funcion de evaluacion de cada individuo.
    WHILE NOT Terminado DO
    BEGIN /* Producir nueva generacion */
        FOR Tamaño poblacion/2 DO
        BEGIN /*Ciclo Reproductivo */
            Seleccionar dos individuos de la anterior generacion,
            para el cruce (probabilidad de seleccion proporcional
            a la funcion de evaluacion del individuo).
            Cruzar con cierta probabilidad los dos
            individuos obteniendo dos descendientes.
            Mutar los dos descendientes con cierta probabilidad.
            Computar la funcion de evaluacion de los dos
            descendientes mutados.
            Insertar los dos descendientes mutados en la nueva generacion.
        END
        IF  la poblacion ha convergido THEN
            Terminado := TRUE
    END
END
```

Figura 5.3: Pseudocódigo del Algoritmo Genético Simple, Figura tomada de Algoritmos Genéticos. 3 de Febrero 2020, de Universidad del País Vasco Sitio web: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf

5.5.1 Población

Estos algoritmos trabajan sobre una población de cromosomas, el término cromosoma hace referencia a un valor o conjunto de valores que representan a una posible solución o individuo. A cada uno de estos valores dentro del cromosoma se les llama gen.

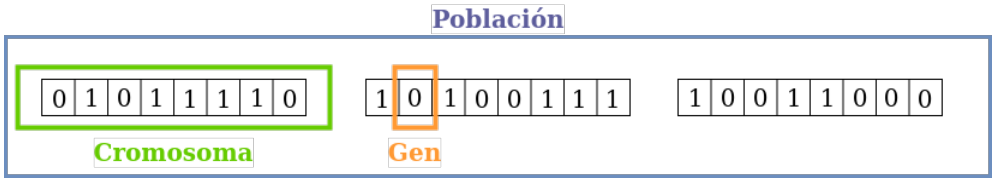


Figura 5.4: Población, cromosoma y gen

chromosome = [p1,p2,...,pNpar]

NOTA: Diferencia entre genotipo y fenotipo

Al momento de hablar sobre la representación de los individuos de la población se suelen utilizar los términos genotipo y fenotipo, estos términos fueron creados por Wilhelm Johannsen en 1911, el genotipo es la información hereditaria completa de un organismo y el fenotipo son las propiedades observadas. En el campo de los algoritmos genéticos, el genotipo es una representación de bajo nivel (con menos abstracción) que el fenotipo.

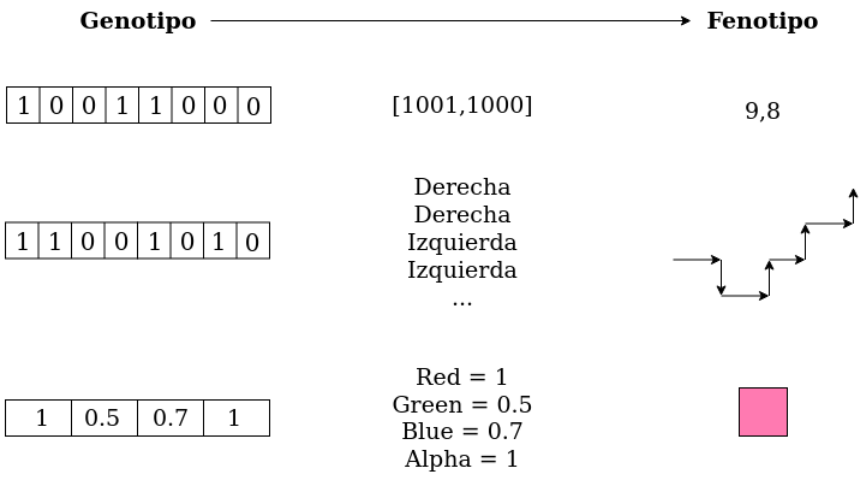


Figura 5.5: Diferencia entre genotipo y fenotipo

Es importante hacer notar que los genes no tienen que ser de tipo binario, un carácter o un elemento de alguna estructura (como un árbol) puede ser un gen por sí mismo.

Generalmente la población inicial es creada con valores al azar, un parámetro importante que el desarrollador debe determinar es el tamaño de la población ya que si el número es muy reducido no se tendrá suficiente variación y es posible que los individuos nunca sobrepasen un óptimo local, también se debe evitar poblaciones muy grandes para evitar la redundancia y para reducir el tiempo necesario para llegar a una solución adecuada.

5.5.2 Evaluación

Para realizar la evaluación de los individuos se requiere tener conocimiento detallado sobre el problema que se está abordando, en algunas ocasiones se requiere realizar una simulación para encontrar el valor de aptitud (Fitness value), es común mantener el máximo valor en 1 y el menor en 0, para favorecer a los individuos con mejor aptitud se puede elevar a alguna potencia.

No siempre se trata de satisfacer un solo objetivo por lo cual a veces se tendrán distintas funciones enfocadas a evaluar los diferentes objetivos y estos deberán ser integrados en un solo valor.

Otra situación importante son los problemas con restricciones, para esto voy a proponer un ejemplo. Si quisiéramos diseñar un algoritmo genético capaz de resolver un sudoku, es probable que definiéramos el problema de la siguiente manera:

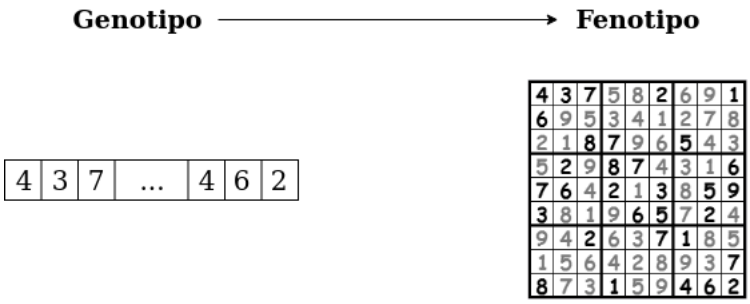


Figura 5.6: Genotipo y fenotipo en un problema de resolución de sudoku

Como se puede observar en la imagen anterior cada gen es un número del 1 al 9 que representa el valor que tendría en la casilla de la cuadrícula. Sin embargo se presentan ciertas restricciones:

1. No se pueden repetir números en un mismo renglón.
2. No se pueden repetir números en una misma columna.
3. No se pueden repetir números dentro de la misma subcuadrícula.
4. Se deben respetar los valores asignados a las casillas dados al momento de plantear el problema.

Para solucionar problemas con restricciones se pueden tomar diferentes medidas, las más comunes son reparación y penalización, la reparación evita que las restricciones sean violadas y la penalización disminuye el valor de aptitud de un individuo, más adelante se revisará la medida de reparación; en este subtema de evaluación se describe el proceso de penalización.

En este problema específico se podría tener una función como la siguiente:

$$F(I) = \frac{(243-x-y-z)}{243}$$

Siendo x el número de casillas repetidas en los renglones, y el número de casillas repetidas en las columnas y z el número de casillas repetidas en las subcuadrículas. En este problema la función depende altamente de las restricciones, pero supongamos que hacemos un algoritmo genético cuya función sea conducir en el menor tiempo posible con la restricción de no chocar ningún obstáculo, entonces podríamos definir una función que considerará el tiempo pero disminuyera la aptitud según el número de obstáculos golpeados.

$$F(I) = \frac{(300-T-40*O)}{100}$$

Siendo T el tiempo que se tardó el individuo en recorrer la pista y O el número de obstáculos golpeados.

5.5.3 Selección

El proceso para la generación de una nueva población involucra el seleccionar padres para realizar el cruzamiento y la mutación, existen diversos métodos utilizados para realizar la selección de los padres, en este libro se explorarán las siguientes opciones [11]:

1. Selección por ruleta (Roulette Wheel Selection)
2. Muestreo universal estocástico (Stochastic Universal Sampling)
3. Selección por rango lineal (Linear Rank Selection)
4. Selección por rango exponencial(Exponential Rank Selection)
5. Selección por torneo (Tournament Selection)
6. Selección por truncamiento (Truncation Selection)

Selección por ruleta

En este método la probabilidad de un individuo para ser elegido como padre es directamente proporcional a su valor de aptitud.

$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$

Donde n es el tamaño de la población y f(i) es la aptitud del individuo i

Una manera de implementar este método es el siguiente:

- Calcular el valor de S ($S = \sum_{j=1}^{n} f(j)$)
- Inicializar en 0 las variables: $p_{acumulada}$ y j
- Generar un número al azar α entre los valores 0 y S
- Mientras $p_{acumulada} < \alpha$ y $j < n$:
  - $p_{acumulada} = p_{acumulada} + f(j)$
  - $j = j + 1$
- Fin del ciclo
- Seleccionar individuo j

Una desventaja de este método es el riesgo que existe donde el algoritmo genético termina de manera prematura en un óptimo local, esto debido a la presencia de un individuo con una aptitud considerablemente superior a la del resto de la población.

Muestreo universal estocástico

Este método, desarrollado por Baker en 1987, es una variación del anterior y pretende eliminar el riesgo de convergencia prematura en un óptimo local. Consiste en generar un número aleatorio α entre 0 y P (siendo P el promedio de la aptitud de los individuos) y posteriormente elegir a n individuos espaciados de manera uniforme (el valor de espaciado β suele ser el promedio de la aptitud pero no es una regla).

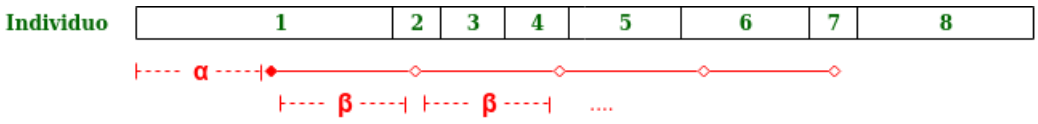


Figura 5.7: Elección de 5 individuos mediante muestreo universal estocástico

Existen diferentes implementaciones de este algoritmo, yo recomiendo utilizar el siguiente proceso para seleccionar m individuos.

- Calcular el valor de P ($P = \frac{1}{n}\sum_{i=1}^{n} f(i)$)
- Inicializar en 0 las variables: $p_{acumulada}$, j y s
- Generar un número al azar α entre los valores 0 y P
- Mientras $s < m$ y $j < n$:
  - $p_{acumulada} = p_{acumulada} + f(j)$
  - $j = j + 1$
  - Si $p_{acumulada} < \alpha + s * \beta$
    - Añadir elemento j al conjunto C
    - $s = s + 1$
- Fin del ciclo
- Devolver conjunto C

Selección por rango lineal

Este método pretende evitar la convergencia del algoritmo genético en un óptimo local, es importante hacer notar que una desventaja es que disminuye la diferencia que hay entre los mejores y peores individuos por lo que puede aumentar el tiempo de convergencia, además de aumentar el tiempo de proceso necesario durante la generación de los rangos.

De manera intuitiva se puede decir que se le da un rango de 1 al peor individuo y al mejor un rango n, siendo n el tamaño de la población.

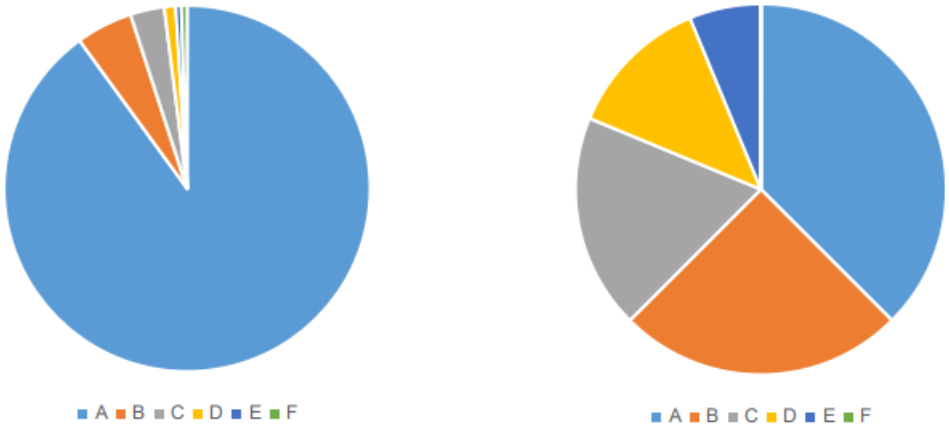


Figura 5.8: Ejemplo del cambio de probabilidad mediante el uso de selección por rango lineal (Derecha)

La fórmula que se usa para determinar la nueva aptitud en este método es de tipo lineal, un ejemplo sería el siguiente:

$$f(pos) = \alpha + \frac{pos}{n}$$

Mientras mayor sea el valor de α menor será la diferencia entre las probabilidades de los individuos, en la Figura 5.9 se usó 0 como valor de α.

Otra fórmula que suele utilizarse es la siguiente:

$$f(pos) = 2 - SP + \left(2 * (SP - 1) * \frac{pos-1}{n-1}\right)$$

SP corresponde al término en inglés Selective Pressure (presión selectiva) y $2 \geq SP \geq 1$. A mayor presión selectiva más probabilidad de ser elegidos tienen los mejores individuos.

Para implementar este método se sugieren los siguientes pasos:

1. Ordenar los individuos de acuerdo a su aptitud
2. Calcular la nueva aptitud de acuerdo a una fórmula lineal
3. Implementar selección por ruleta

Selección por rango exponencial

Este método pretende aumentar la presión selectiva, se proponen diversas fórmulas, en este libro se sugiere la siguiente:

$f(pos) = exp(\frac{pos}{c})$

$c = \frac{n*2*(n-1)}{6*(n-1)+n}$

Existen distintos tipos de fórmulas que se pueden aplicar, es importante tratar de evitar fórmulas muy complejas que impacten altamente el tiempo de procesamiento, en la siguiente figura se observa una comparación que utiliza la fórmula propuesta aquí con anterioridad.

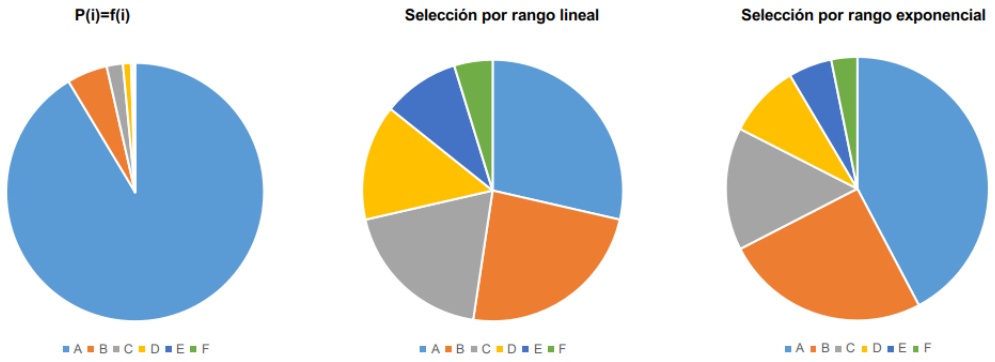


Figura 5.9: Comparación de probabilidad de selección entre tres métodos distintos

Para implementar este método se sugieren los siguientes pasos:

1. Ordenar los individuos de acuerdo a su aptitud
2. Calcular la nueva aptitud de acuerdo a una fórmula exponencial
3. Implementar selección por ruleta

Selección por torneo

Este método consiste en obtener k individuos y posteriormente seleccionar el individuo con mayor aptitud, este proceso se repite n veces para obtener todos los padres.

La forma más fácil de implementar esta técnica es con $k = 2$ se generan dos números aleatorios α y β de 0 al tamaño de la población y se selecciona el elemento α o β con mayor aptitud.

Selección por truncamiento

Este método es bastante simple y no es muy utilizado en la práctica, su mayor caso de aplicación es en poblaciones de gran tamaño.

Consiste en ordenar los individuos de acuerdo a su aptitud y seleccionar una porción de los mismos para realizar la reproducción o cruzamiento entre ellos.

En el siguiente estudio Khalid Jeba [11] analiza los métodos aquí descritos para estudiar el número de generaciones necesarias para llegar a la convergencia, así como el óptimo obtenido, también se propone un nuevo método que busca obtener un mejor óptimo sin sacrificar mucho tiempo para llegar a la convergencia. (`https://www.researchgate.net/publication/259009318_Parent_Selection_Operators_for_Genetic_Algorithms`)

Otros métodos que me gustaría mencionar es la selección uniforme determinista que selecciona todos los elementos de la población para el cruzamiento, y la selección uniforme estocástica que selecciona elementos al azar de la población.

5.5.4 Cruzamiento

Los operadores de cruce nos ayudan a generar la siguiente población a evaluar, consisten en generar 1 o más hijos a partir de dos individuos padre, el uso de algoritmos de cruzamiento lleva a un mejor desempeño en comparación a solo utilizar mutación, esto es más evidente cuando se tienen poblaciones grandes [36]. Existen diversas maneras de realizar el cruzamiento, en este libro solo se revisarán algunos de los más comunes, más específicamente se revisarán los siguientes [13]: Cruce de 1 punto, Cruce de k puntos, Cruce uniforme y Cruce por promedio.

Cruce de 1 punto

Este es uno de los operadores de cruce más simples, dados dos individuos padres se elige un punto de cruce pi al azar, posteriormente se crean dos descendientes combinando los dos padres por el punto de cruce.

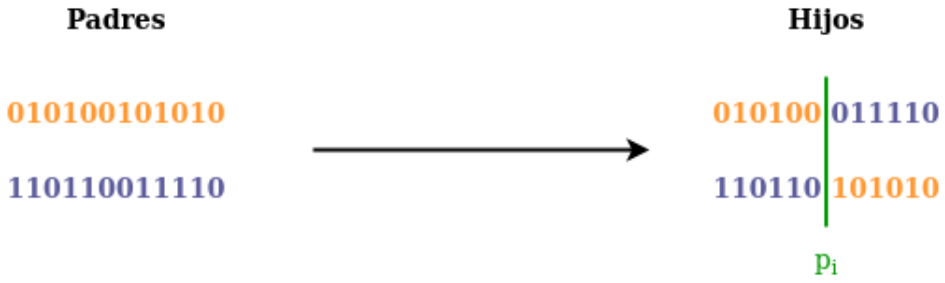


Figura 5.10: Ejemplo de cruce de 1 punto, en este caso el punto de cruce se encuentra entre el sexto y el séptimo gen

Cruzamiento de k puntos

Este operador es muy similar al anterior, la diferencia consiste en el número de puntos de cruce, se eligen k puntos de cruce para generar los descendientes.

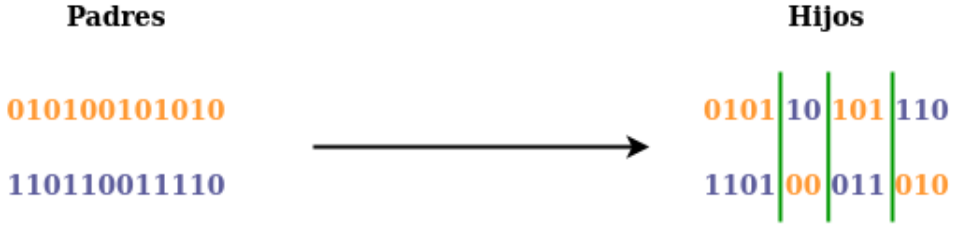


Figura 5.11: Ejemplo de cruce de k-puntos, en este caso se usan 3 puntos de cruce

Cruce uniforme

Este método consiste en combinar los genes de ambos padres, para cada gen se genera un número aleatorio que determina si el primer descendiente tomará el valor del gen del primer padre o del segundo.

El pseudocódigo es similar al siguiente, dados dos padres a, b y dos descendientes x, y:

- Para cada gen
  - Sea h un número aleatorio entre 0 y 1
  - Si h >0.5
    - El valor del gen para x es igual al valor del gen en a
    - El valor del gen para y es igual al valor del gen en b
  - Sino
    - El valor del gen para x es igual al valor del gen en b
    - El valor del gen para y es igual al valor del gen en a

Cruce por promedio

Este operador se utiliza cuando se tienen genes de tipo entero o real, dados dos padres se genera un solo descendiente, el valor de cada gen es el promedio del valor de los genes de los padres.

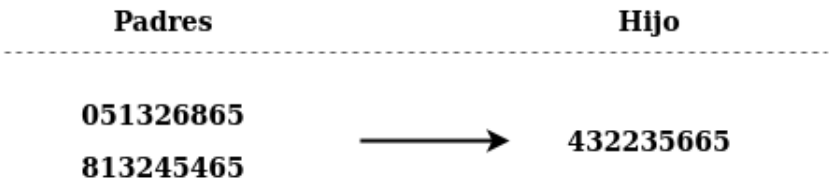


Figura 5.12: Ejemplo del uso del cruce por promedio

En el siguiente enlace pueden encontrar más técnicas de cruzamiento, algunas técnicas como el cruzamiento promediado se ajustan muy bien a cierto tipo de problemas por lo cuál puede valer la pena leer el siguiente artículo para observar si existe algún algoritmo de cruzamiento que se adapte a nuestro problema (`http://ictactjournals.in/paper/IJSC_V6_I1_paper_4_pp_1083_1092.pdf`).

5.5.5 Mutación

La mutación permite que la población mantenga diversidad y mediante la generación de nuevos individuos no presentes en la población actual evita que el algoritmo converja en un valor prematuro.

Este operador se aplica sobre un cromosoma, dada una tasa de mutación (En inglés mutation rate) p_m se genera un número aleatorio y si el número es menor a p_m se realiza una o más modificaciones en los genes del individuo. En los algoritmos genéticos tradicionales (también llamados canónicos) el valor de p_m es fijo, y solo se aplica un operador, sin embargo existen investigaciones que demuestran que es posible utilizar varios operadores con una tasa de mutación dinámica para cada operador, esto permite descubrir cuáles operadores son más útiles sin tener que realizar múltiples pruebas [40]

Bit Flip Mutation

Este operador se aplica para genes con valor binario, se selecciona uno o más genes y se invierte su valor.

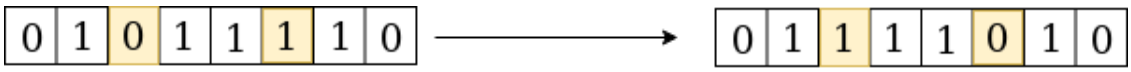


Figura 5.13: Aplicación del operador de mutación “Bit Flip”

Random Resetting

Se selecciona uno o más genes y se le asigna al azar uno de los valores permitidos para el gen.

Figura 5.14: Aplicación del operador de mutación “Random Resetting”

Swap Mutation

Consiste en seleccionar uno o más pares de genes e intercambiar su valor.

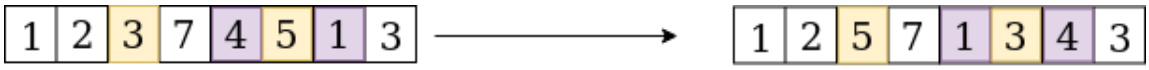


Figura 5.15: Aplicación del operador de mutación “Swap mutation”

Scramble Mutation

Consiste en subconjunto de genes y ordenarlos de manera aleatoria para insertarlos nuevamente.

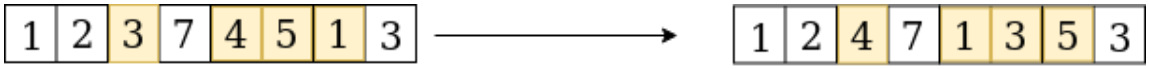


Figura 5.16: Aplicación del operador de mutación “Scramble mutation”

Inverse Mutation

Consiste en subconjunto de genes e invertir su orden para insertarlos nuevamente.

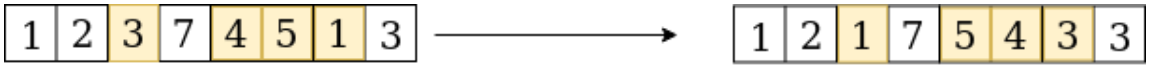


Figura 5.17: Aplicación del operador de mutación “Inverse mutation”

5.5.6 Resolver problemas con restricciones

Anteriormente en la sección correspondiente a la evaluación en los algoritmos genéticos se exploró el problema del sudoku como una situación donde existen restricciones, las tres maneras más comunes de implementar algoritmos con este tipo de problemas son las siguientes:

1. Uso de Funciones de penalización que reducen severamente el valor de aptitud de los individuos que no satisfacen las restricciones.
2. Uso de Funciones de reparación que toman una solución y la modifican para que cumpla con todas las restricciones.
3. No permitir que se genere ningún individuo que no cumpla con las restricciones.

5.5.7 Elitismo en algoritmos genéticos

El elitismo en los algoritmos genéticos consiste en asegurar que un porcentaje de los mejores individuos pasen a la siguiente generación de la población, se recomienda mantener este porcentaje por debajo del 10 % para asegurar la diversidad de la población. Usualmente estos individuos pasan a la siguiente generación sin ninguna mutación, posteriormente se realiza el proceso de crossover y mutación de manera habitual para completar la nueva población.

¿Porqué utilizar elitismo?

Usar elitismo puede tener un alto impacto en el rendimiento de nuestro algoritmo (Aumentando la velocidad de convergencia) [28] debido a que no se tienen que re-descubrir soluciones que ya han probado tener una alta aptitud. De esta manera se asegura también que la aptitud del mejor individuo nunca va a reducirse a través del paso de las generaciones de la población.

¿Porqué NO utilizar elitismo?

Usar elitismo puede hacer al algoritmo converger en un óptimo local de manera prematura, esto depende también del porcentaje de la población que se use para aplicar el elitismo, mientras mayor sea el porcentaje mayor riesgo se corre de limitar el espacio de búsqueda de nuestro algoritmo.

Implementación

La manera más simple de implementar elitismo es ordenando los elementos de la población de acuerdo a su aptitud para posteriormente copiar el porcentaje de cromosomas determinados con anterioridad por el desarrollador, es importante no sobrescribir los valores de estos elementos por lo que solo se deben generar la cantidad de individuos restantes de la población.

5.5.8 Aplicaciones de los algoritmos genéticos

Los algoritmos genéticos resultan extremadamente para resolver problemas de parametrización, en problemas con múltiples óptimos locales las soluciones basadas en gradientes no suelen resolver el problema por lo cual los algoritmos genéticos son una buena alternativa.

- Machine learning: Se pueden utilizar los algoritmos genéticos para crear sistemas que aprendan reglas de producción o sistemas clasificadores (Wang, Bayer)
- Multimodal optimization: Los algoritmos genéticos nos pueden ayudar a encontrar múltiples soluciones en contraste a solo una.
- Problemas de ingeniería: Si se posee el conocimiento suficiente para crear una buena función de evaluación se pueden resolver problemas de diversas áreas de la ingeniería.

5.5.9 Construcción de un algoritmo genético

A continuación se presenta una tabla que resume los puntos que el desarrollador debe determinar o tomar en cuenta cuando construye un algoritmo genético.

Cuadro 5.1: Parámetros y consideraciones en la construcción de un algoritmo genético

Parámetros de la población	Tamaño de la población
Representación de la población	Método de selección y parámetros del método seleccionado. Ej. Si se selecciona Selección por rango lineal se debe determinar el valor de Selective Pressure
Cruzamiento	Método de cruzamiento
Mutación	Implementación de la mutación sobre nuestra población y tasa de mutación
Elitismo	Determinar si se usará o no y el porcentaje de la población que pasaría a la siguiente generación mediante elitismo.
Terminación	Determinar la condición de finalización

Ejercicio de programación:

Ejercicio para fortalecer los conocimientos adquiridos:

En cualquier lenguaje de programación hacer un programa capaz de realizar alguna de las siguientes tareas:

- Resolver un tablero de sudoku con algunas celdas llenadas previamente.
- Resolver el problema del viajero. (`https://es.wikipedia.org/wiki/Problema_del_viajante`)
- Adaptar la posición y rotación de líneas en un espacio tridimensional para representar una imagen. (`https://youtu.be/iV-hah6xs2A`)

A continuación se presenta un repositorio donde aplicó un algoritmo genético para solucionar tableros de sudoku:

`https://github.com/amr205/SudokuSolver---Genethic-Algorithm`

5.6 Programación genética

En este libro se revisarán las bases de la programación genética para culminar con un proyecto que demuestre que se entienden estos conceptos y se poseen las habilidades de programación necesarias para poner en práctica lo aprendido. Si tú como lector quieres explorar más a profundidad este tema recomiendo leer el siguiente libro:

(`http://www.lulu.com/shop/riccardo-poli-and-william-b-langdon-and-nicholas-freitag-mcphee/a-field-guide-to-genetic-programming/ebook/product-17447670.html`).

¿Qué es la programación genética?

La programación genética es un tipo de algoritmo evolutivo en el cual se determina "que debe hacerse" y se generan programas computacionales para resolver dicho problema.

El funcionamiento general de la programación genética es muy similar a los algoritmos genéticos ya que también tiene su base en la selección natural de la teoría de la evolución de Darwin. De hecho son tan similares que la Figura 5.2 presentada para los algoritmos genéticos describe igualmente el flujo de un programa que implementa programación genética.

Diferencia entre un algoritmos genéticos y la programación genética

La respuesta más simple a esta pregunta son los individuos de la población, en la programación genética cada individuo es un programa computacional. Como se verá a continuación esto impacta en la representación de nuestros individuos, generación de la población y en los métodos de cruzamiento y mutación.

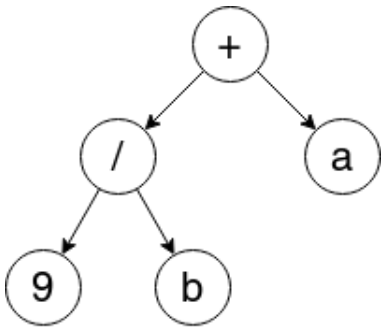
5.6.1 Tipos de programación genética

De acuerdo a la representación de los programas existen diferentes tipos de programación genética, en este libro se abordará el tipo basado en árboles debido a que es uno de los tipos más comunes.

Programación genética basada en árboles

En este tipo de PG el programa se representa mediante árboles, este tipo de representaciones son bastante comunes y suelen usarse para resolver problemas de un dominio específico. Este tipo de representación suele trabajar muy bien con lenguajes funcionales, una de las primeras implementaciones fue en el lenguaje LISP debido a que la estructura del lenguaje se presta para utilizar este tipo de algoritmos.

John Koza [14] menciona que esta representación es más natural que una basada en cadenas de caracteres de longitud fijo (o representaciones de tipo cromosoma) debido a que permite a los programas de la población tener variedad en su tamaño y longitud.



(9 / b) + a

Figura 5.18: Representación de un programa mediante un árbol.

Programación genética lineal

Este tipo de representación se asemeja mucho más a los lenguajes imperativos ya que los programas son representados como una serie de instrucciones.

Es importante notar la diferencia entre este tipo de algoritmos y los algoritmos genéticos, una de las diferencias claves consiste en el hecho de que los programas generados pueden tener diferente tamaño (cantidad de instrucciones). A pesar de su estructura lineal este tipo de programación genética es capaz de generar soluciones para problemas de alta complejidad [33]. En la siguiente figura se puede observar una comparación entre la representación lineal y la representación lineal.

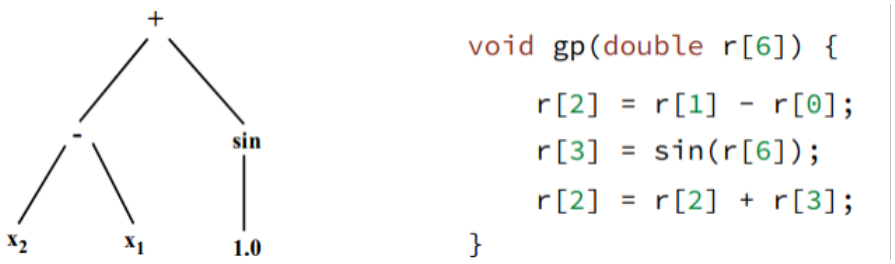


Figura 5.19: Comparación entre la representación basada en árboles (izquierda) y la representación lineal, el array proporcionado como argumento para el algoritmo de programación genética lineal es el siguiente r=x1,x2,1.0,1.0,0.0,1.0, Figura tomada de Jed Simson. (2017). Open-Source Linear Genetic Programming. : Faculty of Computing and Mathematical Sciences University of Waikato, Waikato, New Zealand.[33]

Otros tipos de programación genética

Existen otros tipos de representaciones como la evolución gramatical que utiliza la estructura gramatical del lenguaje para generar programas, así como otros tipos de representaciones, sin embargo estos tipos de PG se encuentran fuera del alcance de este libro.

5.6.2 Representación de los individuos

NOTA: Como se mencionó anteriormente este libro está centrado en la programación genética basada en árboles, por lo cual todos los subtemas posteriores del tema de programación genética usan solamente esta representación.

Los programas que se generan no suelen ser programas completos como los que nosotros como desarrolladores desarrollamos, en cambio suelen ser de dominios más específicos. Como se mencionó anteriormente los programas suelen representarse como árboles sintácticos, en formas más avanzadas de la programación genética, el programa consta de diferentes subrutinas unidas entre sí, en este libro solo se tratará la forma básica que consiste de un solo árbol sintáctico [33].

Para formar el árbol sintáctico se utilizan un conjunto de funciones y un conjunto de elementos terminales, los nodos internos utilizan elementos del conjunto de funciones y las hojas del árbol toman valores del conjunto de elementos terminales. Estos conjuntos deben contener los elementos suficientes para poder generar soluciones apropiadas para nuestro problema planteado.

Conjunto terminal

Este conjunto puede contener los siguientes elementos [33]:

- Constantes: Estas suelen ser generadas de manera aleatoria durante la creación del árbol o creadas durante el proceso de mutación. El símbolo $\Re$ representa una constante aleatoria efímera (En inglés llamada ephemeral random constant), esta constante representa un conjunto de constantes fijas. Ej $\Re = \{x | x \text{ es un entero y } 0 \leq x \leq 10\}$
- Funciones sin argumentos: Este tipo de funciones pueden regresar un valor distinto cada vez que se usan, un ejemplo sería una función que genere un número aleatorio.

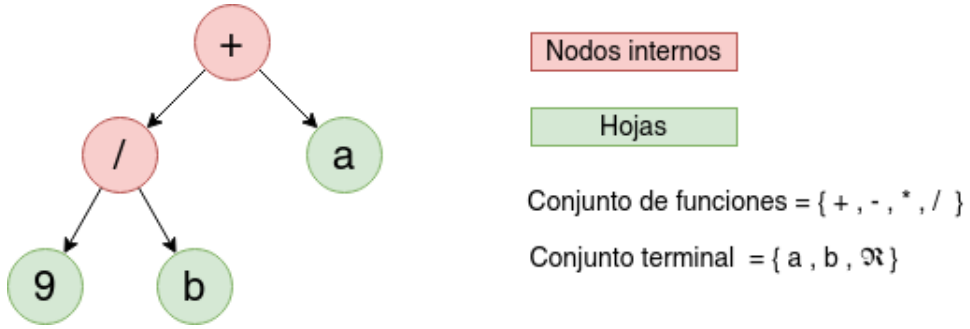


Figura 5.20: Árbol sintáctico de la operación (9 / b)+a.

- Valores de entrada externos: Son los argumentos del algoritmo de PG, suelen ser representados con el nombre de alguna variable, en la Figura 5.20 a y b serían un ejemplo de este tipo de elementos.

Conjunto de funciones

Las funciones estarán determinadas por el tipo de problema que se necesita resolver, en el caso de la Figura 5.20 las funciones son de carácter aritmético. Para que nuestro algoritmo de PG funcione de manera correcta se tienen que cumplir dos propiedades: consistencia de tipo y seguridad en la evaluación.

Consistencia de tipos

Para cumplir con la consistencia de tipo todas nuestras funciones tienen que utilizar argumentos del mismo tipo y devolver valores del mismo tipo, esto puede limitar el tipo de funciones que podemos incluir en nuestro conjunto, sin embargo algunas funciones pueden ser re-interpretadas para que trabajen con el mismo tipo que las demás. Ejemplo: Las función IF-THEN trabaja con dos argumentos, uno booleano y uno numérico, si la quisiéramos usar con operadores aritméticos podríamos reestructurarla para que tome tres argumentos numéricos y si el primer argumento sea mayor al segundo devuelva el valor del tercero, así se habría conservado la consistencia de tipos.

Si no se cumple la consistencia de tipos se tendrían que implementar medidas en la fase de cruzamiento y mutación que asegurarán que los árboles generados siguieran siendo válidos.

Seguridad en la evaluación

Básicamente se debe evitar que el programa produzca errores al ejecutarse, un ejemplo claro es el de la división sobre cero. Se pueden tomar distintas acciones para tratar este tipo de situaciones, la primera es reducir altamente la aptitud de los programas que produzcan errores, la segunda es utilizar funciones adaptadas para responder con algún valor ante estas situaciones, la función de división protegida, denotada comúnmente con el símbolo % suele devolver un valor de 1 ante una división sobre 0.

5.6.3 Generación de la población inicial

Existen diferentes maneras en las cuales se puede generar la población inicial, tener programas duplicados en nuestra población es un gasto de recursos computacionales por lo que se sugiere evitar que se generen, sin embargo es recomendable pero no necesario [14]. A continuación se describen dos técnicas básicas (y comunes) para generar una población inicial.

Full

El desarrollador define una profundidad de los programas, se genera un nodo raíz a partir del conjunto de funciones y se va formando el árbol a partir de estos elementos hasta que se llega a la profundidad definida, en ese momento se seleccionan elementos del conjunto terminal.

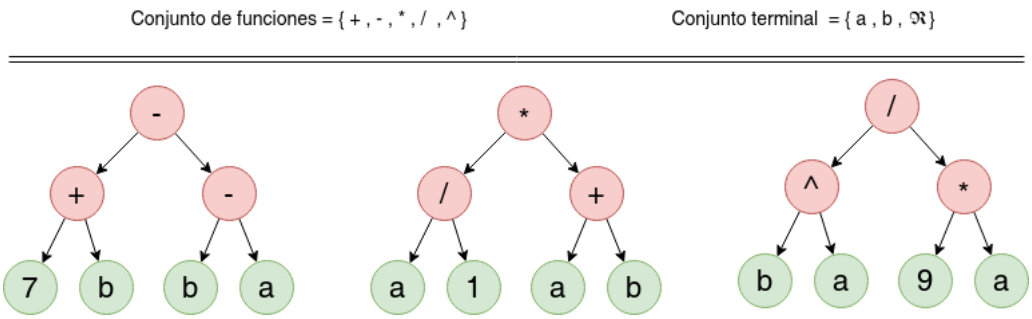


Figura 5.21: Ejemplo de tres individuos de un población generada con el método full con una profundidad de 3.

Grow

El desarrollador define la profundidad máxima de los programas, cuando se generan los nodos del árbol estos se generan a partir de la combinación del conjunto de funciones y el conjunto terminal, si se llega a la profundidad máxima solo se seleccionan elementos del conjunto terminal. Esto permite generar árboles con distinto tamaño.

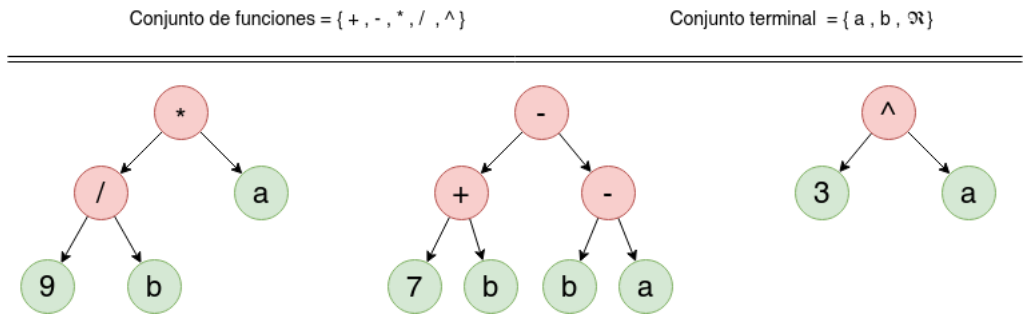


Figura 5.22: Ejemplo de tres individuos de un población generada con el método grow con una profundidad de 3.

Koza [14] sugiere generar la mitad de la población usando el método full y la otra mitad usando el método grow, al uso de esta combinación se le conoce como “ramped half and half”.

5.6.4 Evaluación de los individuos

La evaluación de la aptitud de los individuos depende ampliamente del tipo de problema que se trata de resolver, para evaluar un individuo se tiene que ejecutar el programa, en problemas donde tenemos un conjunto de entradas con su correspondiente salida, se podría calcular el resultado con base en la diferencia entre el resultado proporcionado por el programa y el resultado esperado.

$\frac{1}{m}\sum_{i=0}^{m}(y_i - x_i)^2$

Donde m es el tamaño del conjunto que contiene las entradas con la salida correspondiente, y_i es la salida esperada y x_i es la salida obtenida.

Como se mencionó anteriormente se deben penalizar los programas que generen errores en su ejecución. Muchos de las aplicaciones prácticas requieren de funciones de evaluación multiobjetivo, en este caso se podría analizar la diferencia entre el resultado esperado y el obtenido, el tiempo de ejecución, y los recursos de memoria utilizados, de esta manera se podría obtener una solución que encontrará un buen balance entre estos tres aspectos.

Para correr el programa se puede construir la sentencia para ejecutar en algún lenguaje como por ejemplo LISP, otra opción es evaluar nuestro programa dentro del mismo lenguaje haciendo uso de una función de evaluación, a continuación se presenta un algoritmo que realiza esta última tarea, se recomienda al lector interiorizar y reflexionar sobre el algoritmo que se presenta a continuación y como usa la recursividad para obtener el valor del programa.

procedure: eval(expr)

```
1:  if expr is a list then
2:     proc = expr(1) {Non-terminal: extract root}
3:     if proc is a function then
4:        value = proc( eval(expr(2)), eval(expr(3)), ... ) {Function: evaluate
          arguments}
5:     else
6:        value = proc( expr(2), expr(3), ...) {Macro: don't evaluate argu-
          ments}
7:     end if
8:  else
9:     if expr is a variable or expr is a constant then
10:       value = expr {Terminal variable or constant: just read the value}
11:    else
12:       value = expr() {Terminal 0-arity function: execute}
13:    end if
14: end if
15: return value
```

Notes: expr is an expression in prefix notation, expr(1) represents the primitive at the root of the expression, expr(2) represents the first argument of that primitive, expr(3) represents the second argument, etc.

Figura 5.23: Algoritmo que evalúa un programa representado mediante un árbol sintáctico, Figura tomada de Jed Simson. (2017). Open-Source Linear Genetic Programming. : Faculty of Computing and Mathematical Sciences University of Waikato, Waikato, New Zealand.[33]

5.6.5 Selección

Debido a que los programas generados ya poseen un valor de aptitud se pueden utilizar los métodos descritos en la sección de selección para algoritmos genéticos 5.5.3 de este libro, así como cualquier otro método estándar de selección en algoritmos evolutivos.

En este libro se sugiere analizar el comportamiento del algoritmo de programación genética con el método de selección elegido para determinar si es conveniente utilizar algún método con mayor o menor presión selectiva (A mayor presión selectiva mayor probabilidad hay de que los mejores individuos sean seleccionados como padres). Como punto de inicio se puede utilizar el método de selección por torneo debido a su fácil implementación.

5.6.6 El rol de los operadores de cruzamiento y mutación

Han existido discusiones de acuerdo a la importancia de los operadores de cruzamiento y mutación en los algoritmos evolutivos, un punto de vista tradicional nos indica que la mutación nos permite mantener diversidad en nuestra población y explorar el espacio de soluciones de nuestro problema, en cambio el cruzamiento nos permite ir mejorando la aptitud promedio de nuestra población y generar mejores individuos para llegar a la convergencia de nuestro algoritmo. La pregunta no es ¿Cruzamiento ó mutación?, lo ideal es usar ambas y encontrar el balance haciendo uso de los parámetros que como desarrolladores podemos modificar.

Si queremos analizar el comportamiento del uso de cruzamiento o mutación por separado se puede observar [17] que en general el desempeño de algoritmos de programación genética que usan solo cruzamiento superan a aquellos que solo usan mutación y esta diferencia se hace más marcada al incrementar el tamaño de la población (Esto tiene sentido que ya que sin la mutación se requiere de una población grande para poseer suficiente diversidad para la convergencia exitosa del algoritmo).

Debido a que la representación de los individuos es muy distinta a la representación usada en los algoritmos genéticos los métodos de cruzamiento y mutación difieren a los presentados anteriormente.

5.6.7 Cruzamiento

A continuación se presenta uno de los métodos más comunes para realizar el cruzamiento en un algoritmo genético.

Subtree crossover (Cruzamiento de un punto)

Dados dos programas padres A y B se selecciona un punto de cruzamiento (un nodo) en cada padre P_a y P_b , para crear un nuevo programa se toma una copia del programa A y se reemplaza el subárbol cuya raíz es el nodo P_a por el subárbol del padre B cuya raíz sea el punto P_b. Esta técnica puede usarse para crear uno o dos hijos, el otro hijo tendría como base al padre B y se reemplazaría el subárbol cuya raíz es el nodo P_b por el subárbol del padre A.

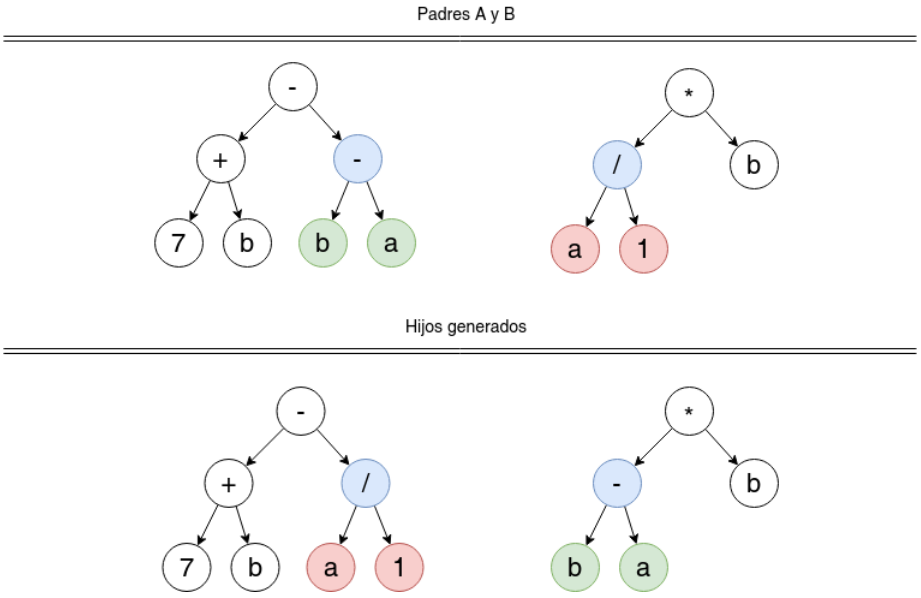


Figura 5.24: Representación de la operación de cruzamiento, en la imagen se encuentran marcados con color azul los puntos de cruzamiento.

Este es uno de los métodos más simples y existen versiones distintas del mismo, por ejemplo size-fair crossover es una variante que asegura que ambos subárboles utilizados para el cruzamiento tengan el mismo tamaño. Debido al alcance de este libro solo se mencionará este método de cruzamiento, si se desean conocer otras maneras de aplicar el operador de cruzamiento se recomienda leer el capítulo 5.3 del libro "A Field Guide to Genetic Programing" [25].

5.6.8 Mutación

A continuación se presentan tres de los métodos más comunes para realizar la operación de mutación, de igual manera si se desean conocer más métodos se recomienda leer el capítulo 5.2 del libro "A Field Guide to Genetic Programing" [25].

Point mutation (Node replacement mutation)

Este método es muy similar al método Bit Flip mutation utilizado en los algoritmos genéticos, se selecciona un nodo en el árbol y se le cambia el valor, si este es un nodo de tipo terminal se cambia por otro nodo del mismo tipo, si el nodo es un nodo interno se cambia por otro nodo del conjunto de funciones que tenga el mismo número de argumentos.

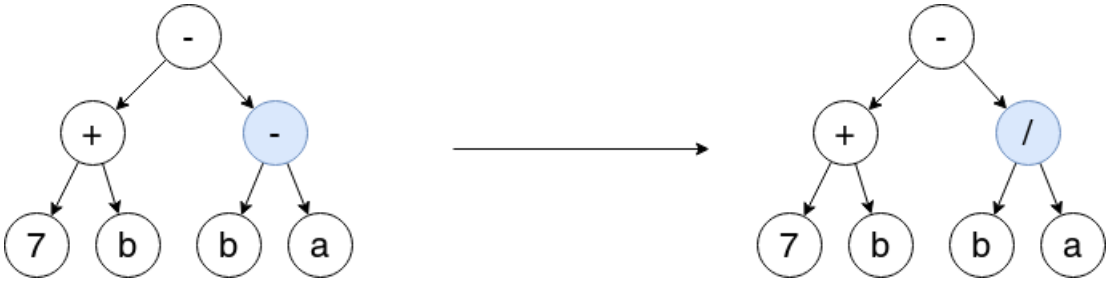


Figura 5.25: Resultado del operador de mutación "point mutation"

Subtree mutation

Se selecciona de manera aleatoria un subárbol dentro del individuo y este se reemplaza por un subárbol generado de manera aleatoria. La forma más básica de este método no limita la profundidad del nuevo subárbol, sin embargo existen variantes que restringen la profundidad del nuevo subárbol a ser del mismo tamaño o a ser como máximo 15 % (o algún otro porcentaje) más profundo que el subárbol original.

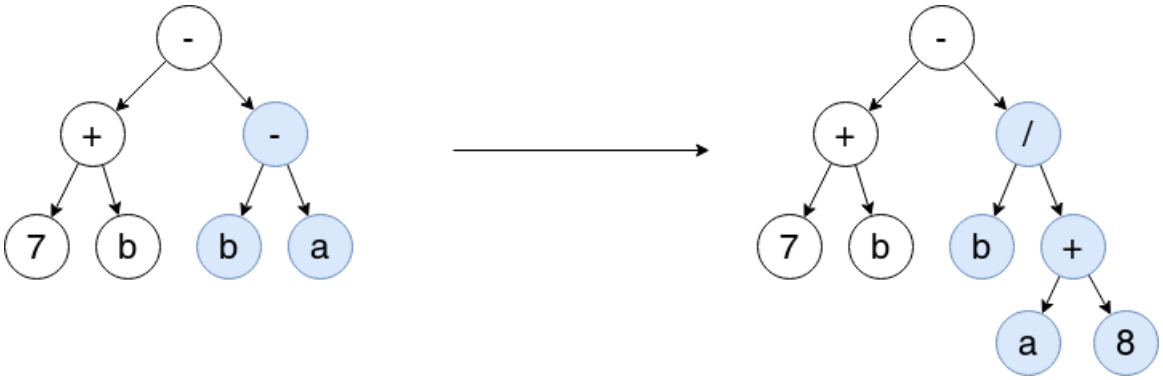


Figura 5.26: Resultado del operador de mutación "subtree mutation"

Shrink mutation

Se selecciona de manera aleatoria un subárbol dentro del individuo y este se reemplaza por un nodo terminal. El objetivo de este método de mutación es el de reducir el tamaño del programa.

5.6.9 Construcción de un algoritmo de programación genética

A continuación se presenta una tabla que resume los puntos que el desarrollador debe determinar o tomar en cuenta cuando construye un algoritmo de programación genética.

Cuadro 5.2: Parámetros y consideraciones en la construcción de un algoritmo de programación genética

Parámetros de la población	Tamaño de la población
Representación de la población	Determinar el conjunto de funciones y el conjunto terminal (En el conjunto terminal hay que determinar los valores que puede tomar la constante aleatoria efímera $\Re$)
Selección	Método de selección y parámetros del método seleccionado. Ej. Si se selecciona Selección por rango lineal se debe determinar el valor de Selective Pressure
Cruzamiento	Método de cruzamiento.
Mutación	Implementación de la mutación sobre nuestra población y tasa de mutación
Terminación	Determinar la condición de finalización

Ejercicio de programación:

En cualquier lenguaje de programación hacer un programa capaz de obtener una formula a partir de los datos de entrada y salida.

(`https://github.com/amr205/FunctionDiscoverer---Genetic-Programming`)

5.7 Sistemas clasificadores (Learning classifier system)

Los sistemas clasificadores buscan aprender un conjunto de reglas que se almacenan y se aplican para realizar la tarea de clasificación. Este tipo de algoritmo de clasificación utiliza las bases de los algoritmos evolutivos para el aprendizaje de las reglas (Por su funcionamiento también se considera un algoritmo de machine learning de aprendizaje supervisado o reforzado, si se quiere conocer a que se refiere esto se recomienda leer los primeros temas del capítulo Machine Learning).

¿En qué consiste la tarea de clasificación?

Dado un nuevo ejemplo de un elemento de un dominio específico ser capaz de etiquetarlo de manera correcta (asignarle un ejemplo). Estos elementos o instancias comparten una estructura que contiene una serie finita de atributos. Las etiquetas pueden ser de carácter discreto o continuo.

La tarea de clasificación puede servirnos para un espectro amplio de problemas, incluso nos puede servir para determinar qué acción realizar ante una situación determinada, el elemento del dominio sería la situación actual del entorno y la etiqueta la acción a realizar.

5.7.1 Funcionamiento básico de las reglas en un LCS

Antes de analizar los tipos de LCS más comunes y su funcionamiento se describirá qué son las reglas y cómo determinan la clasificación de un ejemplo del dominio.

Para poder modelar el dominio se utilizan reglas, cada regla es parte de ese modelo. Cada regla está compuesta de una condición y una acción, la condición nos indica el valor que deberían tomar uno o más atributos, si esta condición se cumple la acción nos dice la clase a la cual corresponde el ejemplo. Supongamos que se tiene un problema donde cada ejemplo del dominio contiene 7 atributos que pueden tomar el valor de 0 o 1.

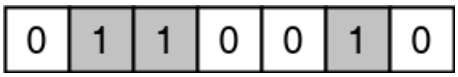


Figura 5.27: Representación de una instancia que contiene 7 atributos donde cada uno de ellos puede tomar el valor de 0 o 1.

Como se mencionó anteriormente la condición de una regla contiene los valores esperados en uno o más atributos (se suelen preferir reglas con menos atributos ya que son más generales, más adelante en este capítulo se verá cómo se favorecen este tipo de reglas), estas reglas suelen ser descritas como una sentencia condicional, a continuación se presentan reglas aplicadas sobre la instancia anterior para demostrar de manera visual su comportamiento.

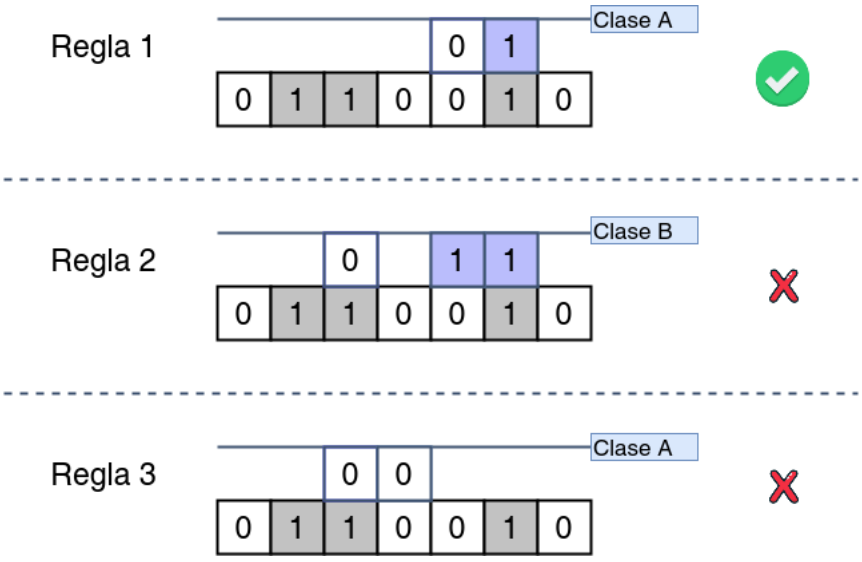


Figura 5.28: Reglas diferentes siendo aplicadas a una misma instancia.

En la figura 5.28 se puede observar que la única regla cuya condición se cumple es la regla 1 por ende la acción de esta regla indica que pertenece a la clase A. Si hubiera varias reglas que

coincidieran la predicción estaría basada en la clase con mayor número de reglas cuya condición se cumpliera. Ejemplo: Si para una instancia 5 reglas coinciden, 3 de ellas reglas con acción clase A y 2 con acción clase B, la clase que a predecir sería la clase A.

Esta explicación de cómo se utilizan las reglas para clasificar se retomará más adelante en el capítulo, en este momento se espera que el lector entienda su funcionamiento básico para observar la utilidad del algoritmo.

5.7.2 Tipos de LCS

Los tipos más comunes de LCS que utilizan algoritmos evolutivos son LCS estilo Pittsburgh [35] y el estilo Michigan [9], estos dos estilos fueron contemporáneos. Actualmente LCS estilo Pittsburgh siguen siendo utilizados, sin embargo el estilo Michigan de LCS se ha convertido en el estándar [32].

El estilo Pittsburgh usa como único elemento de adaptación un algoritmo genético, cada individuo de la población es un conjunto de reglas que se usan para la clasificación, es decir cada individuo es una solución completa al problema de clasificación.

El estilo Michigan utiliza elementos de aprendizaje reforzado en conjunto con un algoritmo genético cuyos individuos son reglas, este algoritmo se utiliza para descubrir nuevas reglas y es altamente elitista. En este libro nos centraremos en el estilo Michigan debido a que actualmente es el estándar de los LCS que usan algoritmos evolutivos, además en este libro ya se cubrió el tema de algoritmos genéticos y se espera que el lector de este libro sea capaz de implementar un LCS estilo Pittsburgh haciendo uso de los conocimientos adquiridos en la sección 5.5 (Algoritmos genéticos).

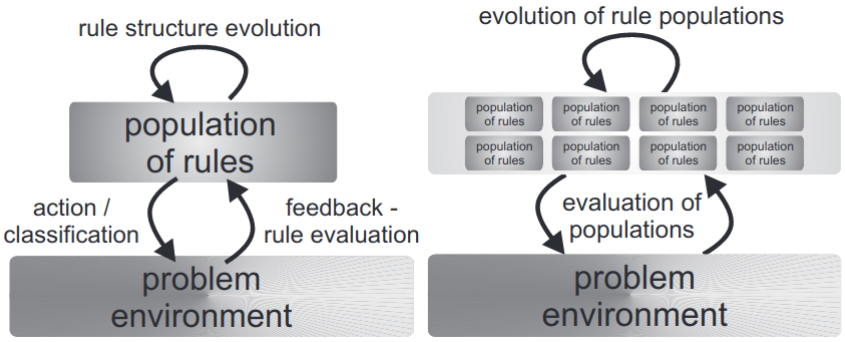


Figura 5.29: Comparación entre el uso de algoritmos genéticos en los dos estilos (Pittsburgh-Derecha, Michigan-Izquierda), En el estilo Michigan un set de reglas evoluciona, en cambio en el estilo Pittsburgh poblaciones formadas por conjuntos de reglas compiten de una manera tradicional (basada fuertemente en el funcionamiento de los algoritmos genéticos). Figura tomada de Bacardit, J., Bernadó-Mansilla, E., and Butz, M.V. (2007). Learning Classifier Systems: Looking Back and Glimpsing Ahead. IWLCS. [2]

5.7.3 Mecanismos principales en un LCS

Este libro se centrará en el estilo Michigan, a partir de este punto se sobreentiende que el estilo que se está describiendo es este. Antes de presentar los componentes principales que contiene un LCS se describirán los mecanismos principales de un LCS con la finalidad de que sea más fácil identificar la finalidad de cada componente.

Discovery o "descubrimiento"

Este componente se refiere a la creación o el descubrimiento de nuevas reglas que no se encuentren actualmente en nuestra población, idealmente estas reglas serán mejores para solucionar el problema de clasificación. La forma más común de realizar este mecanismo es mediante un algoritmo genético [41]. El funcionamiento de este algoritmo genético es el descrito en la sección 5.5(Algoritmos genéticos) de este libro.

Learning o aprendizaje

El aprendizaje, en el contexto de la inteligencia artificial puede ser descrito como la mejora en el desempeño de una tarea en un ambiente a través de la adquisición de conocimiento, resultado de la experiencia en dicho ambiente [15].

Como se verá posteriormente a mayor detalle en este capítulo cada regla que se encuentra dentro de la población tiene un conjunto de parámetros asociados, estos parámetros son actualizados en cada iteración mediante el mecanismo de aprendizaje.

El tipo de aprendizaje usado comúnmente en un LCS es aprendizaje reforzado, en este el agente interactúa con el ambiente y recibe una recompensa o penalización si se desempeña en este (el ambiente) de manera correcta o incorrecta respectivamente (asignación de créditos, "credit assignment"). Otro tipo de aprendizaje que puede usarse en un LCS es el aprendizaje supervisado, aquí durante el proceso de aprendizaje cada instancia es acompañada por la etiqueta de la clase a la cual pertenece, aquí los parámetros de las reglas son actualizados dependiendo de si la regla pudo clasificar de manera correcta o no la instancia.

Dentro del estilo Michigan existen diferentes implementaciones, estas implementaciones determinan el estilo de aprendizaje utilizado y los parámetros asociados a las reglas.

5.7.4 Componentes y procesos de un LCS con aprendizaje reforzado

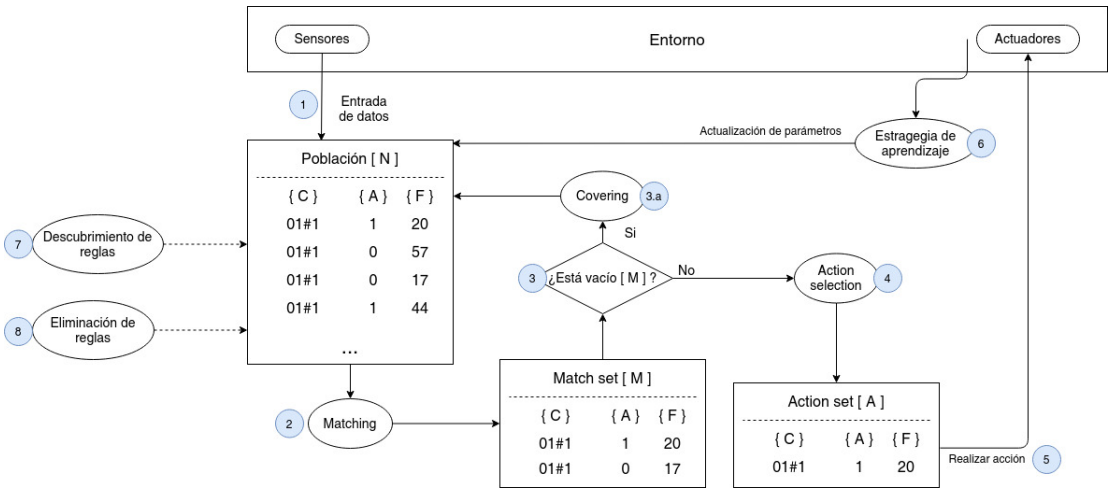


Figura 5.30: Representación del proceso y los componentes que forman parte de un LCS con aprendizaje reforzado, en la figura cada elemento de la población de reglas contiene tres elementos: C el clasificador, A la acción correspondiente, F la aptitud (fitness).

En la figura 5.33 he tratado de representar el proceso que sigue un LCS con aprendizaje supervisado, a continuación describiré los diferentes elementos presentes en la figura.

Environment o entorno

el elemento con el cual nuestro agente interactúa, el agente contiene sensores que nos permiten obtener información y actuadores que nos permiten modificar el entorno. Para entenderlo propongo el siguiente ejemplo: Tenemos un programa que juega baseball, mediante los sensores recibe la velocidad de la pelota,su distancia, y el ángulo de la misma, y mediante los actuadores puede determinar la velocidad y el ángulo con el cual el debe mover el bate.

Población de reglas

Este set contiene las reglas (clasificadores) que nos ayudan a realizar la tarea de clasificación.

Como se ve en la figura 5.28, el cuerpo de la regla o condición { C } no contiene un modelo, sino una parte del mismo, mediante un conjunto de reglas se puede modelar el problema.

Los caracteres que forman el cuerpo de la regla no están limitados a 0 y 1, dependiendo de cómo se forma el genotipo de las instancias del dominio del problema se formará de igual manera la regla. En la literatura se suele utilizar # para denotar el "wildcard", este elemento de la regla no se considerará para el proceso de matching, si representamos las tres reglas presentes en la figura 5.28 usando # como wildcard se verían de la siguiente manera: #|#|#|#|0|1|#, #|#|0|#|1|1|#, #|#|0|0|#|#|#.

Además del cuerpo de la regla, en LCS con aprendizaje reforzado, cada regla tiene asociada la acción { A } que se realizará en el entorno, esta acción puede estar compuesta de un solo valor o un conjunto de valores, estos valores no están limitados al tipo binario y la forma está determinada por los valores que esperan los actuadores de nuestro agente. Por poner un ejemplo supongamos que nuestro agente debe manejar un dron y espera dos valores, un primer valor entero que especifique hacia que dirección (0-Adelante, 1-Atrás, 2-Izquierda, 3-Derecha) y un segundo valor flotante que detalle la velocidad con la que debe moverse. Además de estos dos elementos cada regla tiene asociado un valor de aptitud { F } y otros valores (que dependen de la implementación específica del LCS) llamados parámetros que nos sirven para realizar el proceso de aprendizaje y descubrimiento de reglas.

Matching

Es el proceso mediante el cual se seleccionan las reglas cuya condición satisface a la instancia en la iteración actual del proceso de aprendizaje. La Figura 5.28 muestra este proceso siendo aplicado sobre una instancia. Aquellas reglas cuya condición sea satisfecha pasan al conjunto de elementos llamados "match set" [M].

Covering

Si el match set [M] se encuentra vacío se realiza el siguiente proceso, se seleccionan un subconjunto de las características de nuestra instancia actual, el resto de elementos se llenan con wildcards para formar el cuerpo de una nueva regla, se le asigna una acción al azar y se inicializan con el promedio de los valores de la población. El número de wildcards está determinada por un valor $p_{\#}$ determinado por el programador.

Comúnmente los LCS inician con un conjunto de reglas [N] vacío (No en todas las implementaciones), por lo cual este proceso nos ayuda también a inicializar las reglas del LCS.

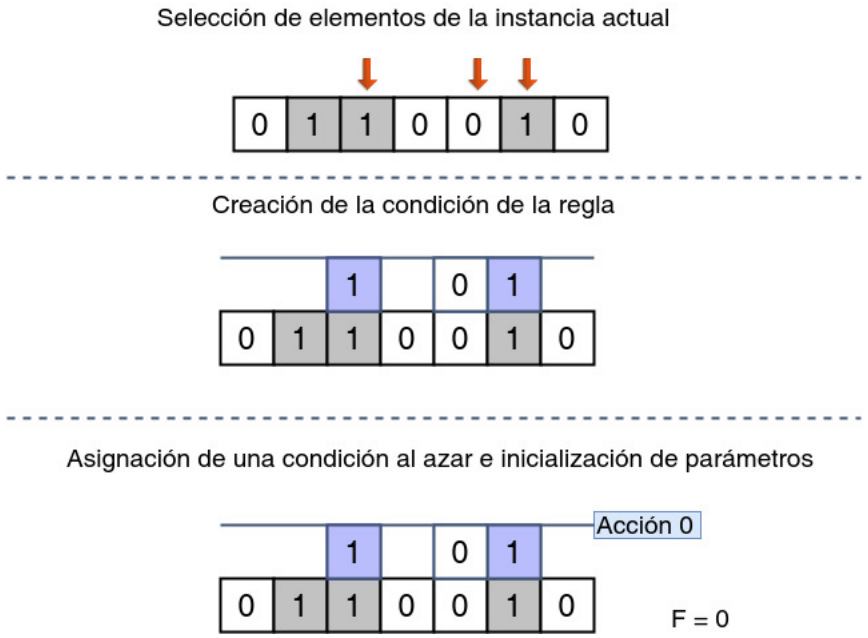


Figura 5.31: Representación visual del proceso de covering.

Action selection

Durante este proceso se determina la acción que se realizará en el entorno y se forma el action set [A], la forma de realizar este proceso depende altamente de la implementación.

Estrategia de aprendizaje

Tras realizar la acción en nuestro entorno se nos devuelve un valor de recompensa P, usando este valor se modificarán los parámetros de las reglas, los parámetros y la manera en la que se modifican dependen de la implementación.

Descubrimiento de reglas

Como se mencionó anteriormente la manera más común de realizar esta parte del proceso en LCS estilo Michigan es mediante el uso de un algoritmo genético, este algoritmo genético suele ser altamente elitista por lo cuál solo una o dos reglas nuevas se añaden en cada iteración del proceso de aprendizaje. Los detalles del algoritmo como el tipo de cruzamiento, mutación y selección son dependientes de la implementación.

Eliminación de reglas

En los LCS se trata de mantener un número de reglas constante, en esta parte del proceso se eliminan las reglas con menor valor de aptitud hasta que el número de reglas sea igual o menor al límite establecido.

Algunas implementaciones de LCS pueden contener algunos componentes extras como el proceso de subsumption que se encarga de eliminar reglas específicas cuyo valor de aptitud sea menor o igual que el de una regla más generalizada.

5.7.5 ZCS (LCS con aprendizaje reforzado)

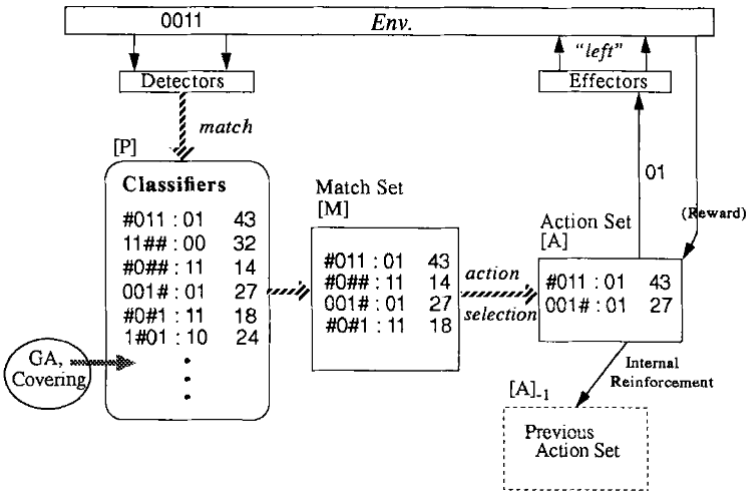


Figura 5.32: Representación del sistema clasificador ZCS. Figura tomada de Wilson, Stewart. (1994). ZCS: A zeroth level classifier system. Evolutionary Computation. 2. 10.1162/evco.1994.2.1.1. [42]

El sistema clasificador "Zeroth-level" System Classifier (ZCS) fue introducido por Wilson [42], este LCS toma como base los trabajos de Holland y utiliza aprendizaje reforzado como estrategia de aprendizaje. Como se puede observar en la figura 5.32 la arquitectura del sistema clasificador ZCS es muy similar a la arquitectura presentada en la Figura 5.30.

Funcionamiento del ZCS

El ZCS comienza con una población de reglas [P] generada de manera aleatoria, el valor inicial de aptitud que toman las reglas es el valor S_0. En diversas publicaciones el valor de aptitud es llamado fitness o strength.A continuación se describe una iteración en el proceso de aprendizaje en un ZCS [4]:

1. Se obtiene la información de entrada del entorno
2. Se realiza el proceso de matching para obtener el match set [M]. **(matching)**
3. Si [M] se encuentra vacío se activa el mecanismo de covering (el único parámetro en ZCS que se requiere para cada regla es el valor de aptitud) añadiendo una nueva regla a la población, eliminando la regla con menor aptitud en la población de reglas y regresando al paso 2. **(covering)**.
4. Usando el método de ruleta (descrito en la sección 5.5.3) se selecciona una regla R. Se copia el valor del action set [A] al conjunto [A-1], Se vacía el action set [A] y cada regla que contenga la misma acción que la regla R se añade al action set [A]. **(action selection)**
5. Se actualiza el valor de aptitud de cada regla en [M] que no se encuentre en [A] de acuerdo a la siguiente fórmula. (empieza la estrategia de aprendizaje)

$$fitness_j = fitness_j - fitness_j * \tau \tag{5.1}$$

τ es un parámetro (no un parámetro de una regla sino del LCS) determinado por el programador o usuario, el dominio de τ es (0,1).

6. Se calcula la siguiente variable para cada regla que se encuentra en [A]

$$value_j = fitness_j * \beta \tag{5.2}$$

β es un segundo parámetro cuyo valor tiene el mismo dominio que τ. Posteriormente se actualiza el valor de aptitud de las reglas en [A] de acuerdo a la siguiente fórmula.

$$fitness_j = fitness_j - value_j \tag{5.3}$$

Se guarda temporalmente el valor "Bucket" B definido de la siguiente manera:

$$B = \sum_{j=1}^{b} value_j \tag{5.4}$$

7. Se ejecuta la acción en el entorno y se obtiene un valor de recompensa reward. Usando este valor se actualiza el valor de aptitud de las reglas en [A] de la siguiente manera:

$$fitness_j = fitness_j + \beta * \frac{reward}{|A|} \tag{5.5}$$

En la fórmula anterior | A | es la cardinalidad del conjunto [A]

8. Por último para terminar la estrategia de aprendizaje se actualiza el valor del conjunto [A-1]:

$$fitness_j = fitness_j + \gamma * \frac{B}{|A-1|} \tag{5.6}$$

Donde γ es un parametro del LCS cuyo valor esta entre 0 y 1.
(termina la estrategia de aprendizaje)

9. El siguiente paso es el proceso de descubrimiento de reglas, Wilson [42] no describe los detalles del algoritmo genético, sin embargo Cádrik y Mach [4], mencionan que se usa un algoritmo genético con selección por ruleta, cruzamiento de un punto y para la mutación cada carácter del genotipo tiene una probabilidad pm de mutar tomando uno de los tres valores posibles 0,1 y # (siendo # un wildcard). Este algoritmo es altamente elitista, por ende solo se generan dos reglas nuevas que se añaden al conjunto [P] **(descubrimiento de reglas)**.
10. Para mantener constante el número de reglas en la población [P] se eliminan dos reglas de la población, preferiblemente aquellas con un valor bajo de aptitud. **(eliminación de reglas)**

5.7.6 Componentes y procesos de un LCS con aprendizaje supervisado

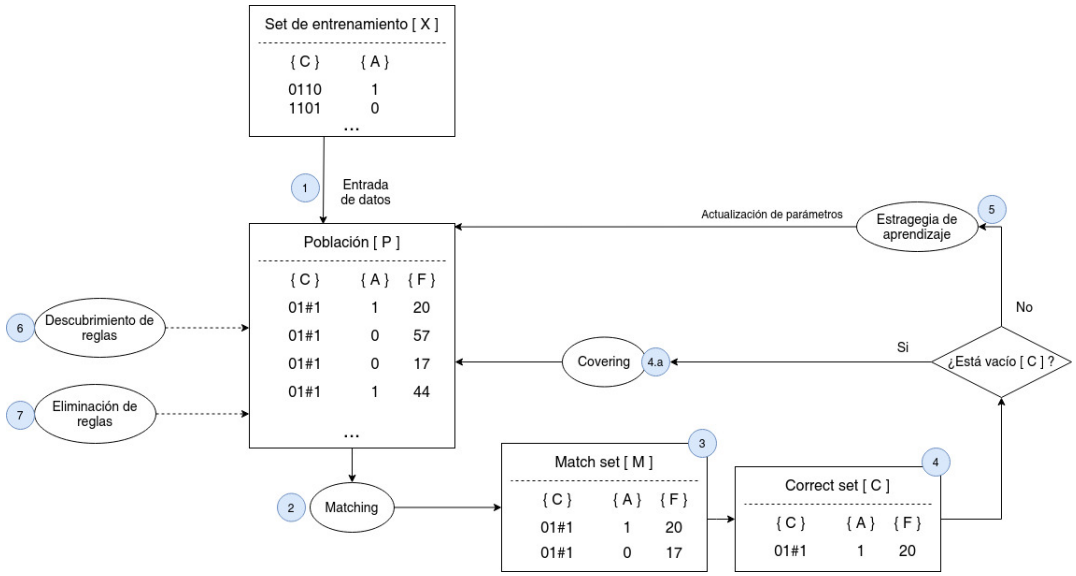


Figura 5.33: Representación del proceso y los componentes que forman parte de un LCS con aprendizaje supervisado, en la figura cada elemento de la población de reglas contiene tres elementos: C el clasificador, A la acción correspondiente, F la aptitud (fitness).

Podemos observar que el LCS con aprendizaje supervisado no difiere tanto en comparación a un LCS con aprendizaje reforzado, la principal diferencia radica en la ausencia del entorno, aquí para el proceso de aprendizaje se requiere de un set de entrenamiento que contiene ejemplos de nuestro dominio con su etiqueta correspondiente.

Para evitar ser repetitivo no mencionaré los componentes que también se encuentran en una arquitectura LCS con aprendizaje reforzado y me centraré en las diferencias.

En lugar de un action set tenemos un correct set [C] este set contiene todos los elementos del match set [M] que contengan la misma etiqueta (acción) que la instancia de la iteración actual.

Para el proceso de aprendizaje ya no es necesario interactuar con el entorno, una vez creados el match set [M] y correct set [C] se puede proceder a la actualización de parámetros, los parámetros y la manera en la que se actualizan es dependiente de la implementación.

Otra diferencia es el momento en el que el mecanismo de covering se activa, en los LCS con aprendizaje reforzado se utilizaban cuando el match set [M] estaba vacío, sin embargo en este tipo de LCS se activa el mecanismo cuando el correct set [C] no tiene ningún elemento, además de esto la clase que toma la regla creada por el covering debe tener la misma clase que la instancia de la iteración actual.

5.7.7 UCS (LCS con aprendizaje supervisado)

El clasificador UCS deriva del XCS (un clasificador basado en la exactitud o “accuracy” en inglés, con aprendizaje reforzado), a diferencia del clasificador XCS adapta sus componentes para utilizar aprendizaje supervisado.

Parámetros de las reglas o clasificadores en UCS

Estos parámetros tendrán que ser actualizados durante el proceso de aprendizaje: exactitud o “accuracy” { acc }, aptitud o “fitness” { F }, tamaño del correct set o “correct set size” { cs },

"numerosity" { num } y experiencia o "experience" { exp }.

La aptitud y exactitud de la regla nos sirve para determinar la calidad de la misma, el tamaño del correct set es el tamaño promedio del correct set [C] en los cuales ha participado, el valor de "numerosity" indica cuantas veces se encuentra presente la misma regla en la población [P] y la experiencia nos indica cuantas veces esta regla ha pertenecido a algún match set [M]. [24].

A continuación se describe una iteración en el proceso de aprendizaje en un clasificador UCS:

1. Se obtiene una instancia xi del set de entrenamiento [X] junto con su clase correspondiente.
2. Se realiza el proceso de matching para obtener el match set [M]. **(matching)**
3. Se obtienen todas las reglas del match set [M] que posean la misma clase o etiqueta de la instancia actual x_i para formar el correct set [C].
4. Si [C] se encuentra vacío se activa el mecanismo de covering, los parámetros son inicializados de la siguiente manera: exp = 1, num = 1, cs = 1, acc = 1 and F = 1. Después de añadir la nueva regla se elimina la regla con menor valor de aptitud. **(covering)**.
5. Ahora se van a actualizar los valores de los parámetros de las reglas: Se aumenta el valor de experiencia { exp } de todas las reglas en el match set [M], y se actualiza el valor de exactitud de las reglas de acuerdo a si estas pertenecen o no al correct set [C]:

$$acc = \frac{clasificacionses_{correctas}}{exp} \tag{5.7}$$

Para poder lograr actualizar los parámetros yo propongo el aplicar las siguientes fórmulas en el orden mostrado, primero empezamos aplicando las siguientes fórmulas sobre todos los elementos del match set [M]:

$$exp_j = exp_j + 1 \tag{5.8}$$

$$acc_j = acc_j * \frac{exp_j - 1}{exp_j} \tag{5.9}$$

Ahora actualizamos los siguientes valores de las reglas presentes en el correct set [C]:

$$acc_j = acc_j + \frac{1}{exp_j} \tag{5.10}$$

$$css_j = \frac{css_j * ((acc_j * exp_j) - 1) + |C|}{exp_j} \tag{5.11}$$

Por último actualizamos el valor de aptitud de todos los elementos en el match set [M]

$$F = (acc_j)^v \tag{5.12}$$

Donde v en la fórmula anterior suele tener el valor v = 10. **(estrategia de aprendizaje)**

6. El siguiente paso es el proceso de descubrimiento de reglas, se puede utilizar un algoritmo genético con selección por ruleta, cruzamiento de un punto y para la mutación cada carácter del genotipo tiene una probabilidad pm de mutar tomando uno de los tres valores posibles

0,1 y # (siendo # un wildcard). Este algoritmo es altamente elitista, por ende solo se generan dos reglas nuevas que se añaden al conjunto [P] (descubrimiento de reglas). Los iniciales que toman los parámetros son los siguientes: exp = 0, num = 1, cs = (csp1 + csp2)/2 (Donde p1 y p2 son los padres), acc = 1 y F=1. **(descubrimiento de reglas)**

7. Para mantener constante el número de reglas en la población [P] se eliminan dos reglas de la población, preferiblemente aquellas con un valor bajo de aptitud. **(eliminación de reglas)**

5.7.8 Conclusión de los LCS

Actualmente gracias a técnicas de Deep Learning se han logrado resolver tareas de clasificación muy complejas, por lo tanto el uso de los LCS ha disminuido, sin embargo los LCS se han aplicado exitosamente para resolver fenómenos biológicos. Es interesante observar como John Holland incursionó en el campo de los algoritmos genéticos y luego usando ideas propias del machine learning desarrolló un algoritmo híbrido que utiliza algoritmos genéticos como un componente del sistema. En mi opinión una de las ventajas de estos sistemas es la facilidad con la cuál puede ser interpretada una regla, sin embargo mientras más grande es el tamaño de la población se vuelve más confuso el interpretar el sistema como un conjunto.

En este tema solo se vieron dos implementaciones específicas de los LCS y existen muchas otras que podrían adaptarse a algún problema que se quiera resolver, por estás razones colocaré aquí el link del trabajo de Urbanowicz [41]. (`https://www.researchgate.net/publication/26850330_Learning_Classifier_Systems_A_Complete_Introduction_Review_and_Roadmap`)

5.7.9 Panorama actual de los algoritmos evolutivos

La investigación y el desarrollo de los algoritmos evolutivos lleva más de 50 años por lo cual hoy en día existen algoritmos robustos que permiten solucionar diversos problemas. Han demostrado que pueden dar soluciones a problemas difíciles, sin embargo a pesar de tener resultados en el ámbito académico se puede observar que en las industrias ha tenido menor éxito. Las nuevas tendencias apuntan al uso de algoritmos híbridos que combinen diversos paradigmas para solucionar problemas más complejos [34].

En lo personal considero que los algoritmos evolutivos si tienen aplicaciones importantes como su uso en la parametrización de dispositivos y programas, es importante observar las ventajas que nos provee cada tipo de algoritmo para poder identificar aquellos problemas que puedan ser resueltos utilizando un determinado tipo de programa. (La posibilidad de encontrar explorar un espacio de soluciones y encontrar una buena solución haciendo uso de una función de aptitud me parece una de las grandes ventajas ya que existen problemas que no pueden utilizar un algoritmo de optimización basado en el uso de gradientes).

6. Inteligencia Artificial Simbólica

6.1 Introducción al capítulo

A continuación se presentan tres de las implicaciones principales de los sistemas de inteligencia artificial simbólica [6]:

- Un modelo que represente un sistema inteligente puede ser definido de manera explícita.
- El conocimiento de estos modelos puede ser representado de manera simbólica (Estos símbolos suelen ser de alto nivel por lo cual pueden ser interpretados por humanos, algunos ejemplos son el uso de grafos, fórmulas lógicas, fórmulas matemáticas, etc)
- Las operaciones mentales y cognitivas pueden ser descritas de manera formal utilizando las estructuras que corresponden al conocimiento descrito en nuestros modelos.

En los modelos de inteligencia artificial se asume que muchos aspectos de la inteligencia pueden ser simulados mediante la manipulación de símbolos.

6.2 Ventajas y desventajas del paradigma simbólico

Entre las ventajas de estos sistemas se encuentran las siguientes [7]:

- **Interpretabilidad:** Debido al uso de símbolos (generalmente de alto nivel, como nodos, predicados, etc) y la naturaleza de las operaciones realizadas con ellos (transición entre estados válidos, operaciones de inferencia, etc) es fácil entender el funcionamiento de los sistemas y la manera en la cual llegan a los resultados obtenidos.
- **Generalización:** Las representaciones simbólicas de alto nivel pueden permitir generalización.
- **Eficiencia de los datos:** Este paradigma a diferencia de otros no requiere una gran cantidad de datos (un ejemplo de lo contrario son muchos algoritmos del paradigma conexionista), el uso de los símbolos también permite que estos puedan ser reutilizados en otros escenarios.

Dentro de las principales desventajas es el hecho de que el conocimiento no suele ser aprendido sino diseñado de manera “manual”, otro punto a considerar es el hecho de que hay conocimiento

demasiado complejo como para ser plasmado de esta manera. Por lo anteriormente mencionado no hubo mucho avance con este paradigma en el reconocimiento de imágenes y el procesamiento del lenguaje natural.

6.3 Orígenes

Uno de los primeros programas basados en el uso de reglas lógicas fue "Logic Theorist" creado por Allen Newell, Herbert A. Simon y Cliff Shaw en 1956, este programa eventualmente demostró 38 de los primeros 52 teoremas del trabajo Principia Mathematica.

El trabajo realizado por Newell, Simon y Shaw precedió a la conferencia de Dartmouth, a pesar de que ellos ya habían realizado un programa que utilizaba uno de los paradigmas más importantes de la inteligencia artificial simbólica (Simulación cognitiva) parece que nadie salvo ellos se dieron cuenta de la importancia de su trabajo a largo plazo [20].

La idea de usar la lógica como representación de la información en un programa se le atribuye a John McCarthy por su propuesta del "advice taker" en 1958. El programa propuesto por John McCarthy era un programa hipotético, J. Alan Robinson en 1963 desarrolló una manera de implementar deducción en una computadora mediante el algoritmo de resolución y unificación, sin embargo las implementaciones de estos algoritmos resultaban en programas que tardaban demasiadas iteraciones en dar resultados.

En 1970 se obtuvieron mejores resultados al reducir la lógica al uso de cláusulas de Horn, de esta manera se desarrollaron mejores algoritmos de deducción y se creó el lenguaje de programación Prolog (Un lenguaje de programación declarativo basado en la programación lógica).

6.4 Clasificación

Existen diversos acercamientos a la inteligencia artificial simbólica que cumplen con las características presentadas al inicio de este capítulo, a continuación se presenta una pequeña descripción de cada uno de ellos [6]:

- **Simulación cognitiva:** Se basa en simular habilidades cognitivas del ser humano (resolución de problemas, razonamiento, aprendizaje) mediante la definición de algoritmos que implementen la heurística. Por lo tanto, en el diseño de estos algoritmos se trata de descubrir conceptos y reglas que nos permiten resolver problemas. Anteriormente se mencionó que el trabajo de Newell, Simon y Shaw (Logic Theorist) incorporaba este acercamiento de manera exitosa.
- **Acercamiento basado en lógica:** John McCarthy fue el precursor de este tipo de sistemas ya que aseguraba que un sistema inteligente debería de estar basado en sistemas formales de razonamiento lógica en lugar de "simuladores" de procesos mentales basados en algoritmos heurísticos. De esta manera el conocimiento podía ser representado mediante reglas lógicas y un programa universal (un programa dedicado a resolver problemas mediante la inferencia) se encargaría de encontrar la solución.
- **Representación de conocimiento basado en reglas:** Newell y Simon continuaron su investigación en modelos cognitivos y en 1972 propusieron sistemas basados en la memoria a corto y largo plazo. La memoria a largo plazo (production memory) es representada por reglas simples (si .... entonces ...) y la memoria a corto plazo (working memory) contiene la información del entorno sobre el cual opera, continuamente monitorea los datos de la memoria a corto plazo para determinar si alguna regla de la production memory se cumple, en caso de que la condición de la regla sea satisfecha se ejecuta la acción de la regla que puede ser una conclusión, la adición de una regla a la memoria a largo plazo, una acción

que permita interactuar con el entorno, etc. Este acercamiento no tuvo tanto éxito, uno de sus programas más relevantes fue la creación de un sistema experto basado en reglas, estos sistemas expertos son una subclase de los sistemas expertos que se pueden producir usando el acercamiento basado en la lógica.

- **Representación estructurada del conocimiento:** Este acercamiento se basa en la manipulación de estructuras que contienen conocimiento, estas estructuras pueden ser redes semánticas, marcos (frames), etc. Mediante el uso de estas estructuras se busca desarrollar programas capaces de resolver problemas específicos.

De estos acercamientos yo considero que el más exitoso fue el de la programación lógica (o acercamiento basado en lógica), gracias a este acercamiento se desarrollaron los sistemas expertos que dieron lugar al boom de la inteligencia artificial (1980–1987), sin embargo es importante tener en cuenta las limitaciones, ventajas y desventajas de este tipo de sistemas.

Este capítulo se centrará en el acercamiento basado en lógica debido no solamente a ser el de mayor éxito, sino también por su base formal.

6.5 Simulación cognitiva

Como se mencionó anteriormente la simulación cognitiva se basa en simular habilidades cognitivas del ser humano mediante la definición de algoritmos que implementen la heurística. Vamos a profundizar un poco en su funcionamiento y luego exploraremos de manera simple el funcionamiento del programa “logic theorist”.

En la resolución de problemas partimos de un estado inicial que representa la situación actual del problema, mediante funciones de transición podemos pasar a otros estados (tomando el ejemplo del ajedrez estas funciones de transición serían todos los movimientos válidos) si estos estados son nuestra solución se les llama “goal states” en caso de que no lo sea son estados intermedios. Un espacio de estados está constituido por estos estados (inicial, intermedio y objetivo), se puede representar mediante un grafo, donde los nodos son los estados y los ejes son las transiciones entre los mismos.

En la simulación cognitiva podemos usar estos conceptos para generar soluciones, partiendo de un estado inicial se aplican funciones de transición para ampliar nuestro espacio de estados y buscar una nueva solución, aquí podemos observar que el espacio de estados crecería de manera exponencial en la búsqueda de soluciones por lo cual estos algoritmos suelen implementar heurística para cortar algunas de las ramas y reducir el espacio de búsqueda.

Ahora vamos a observar como el programa “logic theorist” utiliza estos conceptos para probar teoremas matemáticos:

Parte de un conjunto de teoremas, que es nuestro estado inicial. Utiliza diferentes funciones de transición para expandir el espacio de estados, estas funciones pueden ser las siguientes:

- Método de reemplazo
- Método de separación (modus ponendo ponens)
- Método de encadenamiento

Estos métodos se aplican sobre teoremas que se quieren comprobar, utilizando los teoremas en nuestro estado actual, si alguno de estos métodos logra comprobar la veracidad de un teorema, este se añade a los teoremas comprobados ampliando así nuestro espacio de estados.

Logic theorist utiliza la heurística en el funcionamiento de estos métodos tomando los axiomas comprobados que sean más “prometedores” en la comprobación del teorema.

6.6 Programación lógica

Para explorar este acercamiento del paradigma simbólico vamos a partir de la pregunta ¿Qué es la razón?, la razón la capacidad de la mente humana de establecer relaciones entre ideas o conceptos y obtener conclusiones o formar juicios, este acercamiento busca simular o imitar esta facultad de la mente por medio del razonamiento lógico, dado un conjunto de juicios que mantienen relaciones lógicas entre sí (premisas) se puede deducir o inferir un nuevo juicio al que denominamos conclusión.

"La ciencia que estudia que tipos de esquemas de inferencia aseguran la validez de las conclusiones es la lógica" [22].

Mediante la lógica podemos representar el conocimiento y utilizarlo de tal manera que si las premisas son verdaderas, la conclusión también lo será.

6.6.1 La lógica formal

La lógica formal o lógica matemática estudia los principios y métodos que se emplean para diferenciar el razonamiento correcto del incorrecto. Analicemos el siguiente razonamiento:

Si estoy corriendo entonces voy más lento
Estoy corriendo
-Por lo tanto: voy más lento

Este razonamiento podría parecer incorrecto debido a que sabemos que si corremos vamos más rápido, sin embargo la lógica formal estudia la lógica del razonamiento, si le asignamos letras a las distintas proposiciones podemos damos cuenta de que en realidad este razonamiento es correcto.

p: estoy corriendo
q: voy más lento

$p \rightarrow q$
p

---

q

Podemos decir que esta inferencia posee validez formal

6.6.2 Clasificación de la lógica

La lógica tiene diversas subdivisiones, cada una con su propia semántica y sintaxis, algo importante es observar cómo cada una de las divisiones de la lógica expande la lógica del orden anterior, en consecuencia es más expresiva y por ende requiere de añadir más recursos o eliminar restricciones sobre el uso de los ya existentes [22].

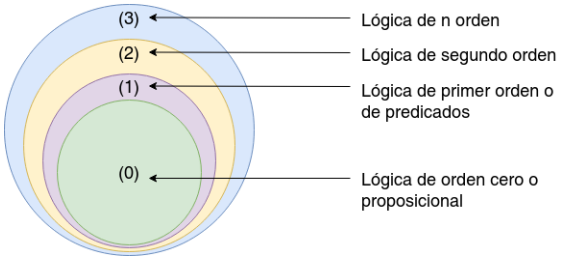


Figura 6.1: Representación de las diversas divisiones en la lógica.

También se puede dividir la lógica en lógica clásica y no clásica, en la lógica clásica las fórmulas lógicas solo pueden tener dos valores (verdadero o falso), por eso se le llama cálculo bivalente, en la lógica no clásica las fórmulas pueden tomar más valores, un ejemplo es la lógica trivalente que además de los valores verdadero y falso contempla un tercer valor que no representa que el valor se desconoce o es incierto [22].

Existen otro tipo de lógicas que contemplan elementos como el tiempo, en el cual el valor de verdad depende del momento actual o en el que dio lugar.

Nota: Diferencia entre lógica de predicados y cálculo de predicados

Es común encontrar estos términos y preguntarse la diferencia entre los mismos, estos términos suelen usarse de manera intercambiable y representan lo mismo, cada cálculo o división de la lógica debe componerse de los siguientes elementos [22]:

- La **semántica** de la lógica, el significado de cada uno de los elementos que la compone.
- Una **sintaxis** que nos permita formar combinaciones correctas de los elementos primitivos . Mediante la definición de un conjunto de **reglas de formación** podemos definir como es una **fórmula bien** formada (fbf, otro término con el cual es probable encontrarse), de esta manera se puede determinar si la combinación de los elementos es correcta o no.
- Un conjunto de **reglas de transformación** de carácter algorítmico que nos permitan ir de una fbf a otra, estas transformaciones deben asegurar la validez formal de las fórmulas lógicas.

En este tema veremos cómo podemos utilizar la lógica de primer orden para implementar un sistema experto, este programa emula las capacidades de tomar decisiones del ser humano, debido a que como se mencionó aquí los cálculos de la lógica se construyen sobre la lógica del orden anterior primero revisaremos la lógica de orden cero.

6.6.3 Lógica de orden cero o proposicional

La lógica proposicional es la más simple de los tipos de lógica y como su nombre indica está basada en sentencias o proposiciones, una **sentencia o proposición** es una oración capaz de tener un valor de verdad (verdadero o falso).

Mediante las proposiciones podemos representar información, ejemplo:

p: estoy corriendo
q: voy más rápido

Estas dos proposiciones contienen un solo elemento por lo cual son conocidas como **proposiciones atómicas o proposiciones simples** (También se les conoce como proposiciones primitivas).

Mediante el uso de conectores operadores lógicos podemos formar **proposiciones compuestas**, que nos muestran las relaciones entre las proposiciones, ejemplo (si llueve, entonces voy más rápido):

$p \rightarrow q$

Operadores lógicos

A continuación se describirán los diferentes operadores lógicos y mediante el uso de tablas de verdad [1] se mostrará su comportamiento (1 - Verdadero, 0 - Falso):

[1] Una tabla de verdad, o tabla de valores de verdades, es una tabla que muestra el valor de verdad de una proposición compuesta, para cada combinación de verdad que se pueda asignar

Negación ¬

Cuando la variable es verdadera al negarla se convierte en falsa, y si es falsa, al negarla se hace verdadera.

A	¬A
1	0
0	1

Disyunción v

Es falsa cuando ambas proposiciones son falsas.

A	B	A v B
1	1	1
1	0	1
0	1	1
0	0	0

Conjunción Λ

Solo es verdadera cuando ambas proposiciones son verdaderas.

A	B	A Λ B
1	1	1
1	0	0
0	1	0
0	0	0

Condicional →

Solo es falsa cuando la primera proposición es verdadera y la segunda falsa.

A	B	A → B
1	1	1
1	0	0
0	1	1
0	0	1

Bicondicional ↔

Solo es verdadera cuando ambas proposiciones tienen el mismo valor.

A	B	A ↔ B
1	1	1
1	0	0
0	1	0
0	0	1

Utilizando estas conectivas podemos representar conocimiento más complejo

Si es sábado o es domingo entonces saco a pasear a mi perro y desayuno hotcakes.
a: es sábado
b: es domingo
c: saco a pasear a mi perro
d: desayuno hotcakes

(a v b) → (c Λ d)

Aquí quiero hacer notar el uso de la condicional, cuando no es sábado o domingo puede ser que saque a pasear a mi perro y desayune hotcakes, pero siempre que sea sábado o domingo voy a sacar a mi perro a pasear y desayunar hotcakes.

Ahora vamos a dar las reglas de formación para crear fórmulas bien formadas usando la notación de Backus-Naur (BNF):

<Fórmula >::= ProposiciónAtómica
| ¬ <Fórmula >
| <Fórmula >Λ <Fórmula >
| <Fórmula >v <Fórmula >
| <Fórmula >→ <Fórmula >
| <Fórmula >↔ <Fórmula >
| (<Fórmula >)

Tipos de fórmulas bien formadas en lógica proposicional

Tautología: es una fbf que siempre es verdadera para cualquier interpretación (para cualquier combinación de valores de verdad que tomen sus proposiciones atómicas).

Un ejemplo simple de tautología es la siguiente: A v (¬A)

A	(¬ A)	A v (¬ A)
1	0	1
0	1	1

Contradicción: es una fbf que siempre es falsa para cualquier interpretación (para cualquier combinación de valores de verdad que tomen sus proposiciones atómicas).

Un ejemplo simple de contradicción es la siguiente: A Λ (¬A)

A	(¬ A)	A Λ (¬A)
1	0	0
0	1	0

Contingencia: es aquella proposición que puede ser verdadera o falsa dependiendo de los valores de las proposiciones que la integran.

Otro concepto importante son las equivalencias lógicas, las equivalencias lógicas se dan cuando dos proposiciones p y q son equivalentes en la lógica.

Es decir: p ↔ q

Reglas de inferencia

Estas reglas de transformación nos permiten deducir nuevas proposiciones a partir de premisas. Considero importante estas reglas de transformación como parte de la lógica proposicional, sin embargo para los temas posteriores no es imperativo que se entiendan a detalle estas reglas, de cualquier manera recomiendo revisar algunas para entender como podemos deducir nuevas proposiciones a partir de conocimiento previo.

Regla de inferencia	Tautología	Nombre
P ——— P v Q	P → (P v Q)	adición
P Λ Q ——— P	(P Λ Q) → P	simplificación
P P → Q ———- Q	[P Λ (P → Q)] → Q	modus ponens
Q P → Q ——— P	[Q Λ (P → Q)] → P	modus tollens
P v Q P ——— Q	[(P v Q) Λ P] → Q	silogismo disyuntivo
P → Q Q → R ——— P → R	[(P → Q) Λ (Q → R)] → [P → R]	silogismo hipotético
(P → Q) Λ (R → S) P v R ——— Q v S	[(P → Q) Λ (R → S) Λ (P v R)] → [Q v S]	dilema constructivo
(P → Q) Λ (R → S) Q v S ———- P v R	[(P → Q) Λ (R → S) Λ (Q v S)] → [P v R]	dilema destructivo

Reglas de reemplazo

Este tipo de reglas de transformación nos permiten transformar las fórmulas en otras fórmulas con equivalencia lógica, otra diferencia es la bidireccionalidad de las mismas y que pueden ser aplicadas en porciones de la fórmula, si desean conocer más de este tema considero que en el siguiente link hay información útil: `https://www.iep.utm.edu/prop-log/`

Nombre	Regla
Doble negación	$\neg \neg a = a$
Conmutatividad	$a \land b = b \land a$ $a \lor b = b \lor a$
Asociatividad	$(a \land b) \land c = a \land (b \land c)$ $(a \lor b) \lor c = a \lor (b \lor c)$
Tautología	$a \land a = a$ $a \lor a = a$
Ley de morgan	$\neg(a \land b) = \neg a \lor \neg b$ $\neg(a \lor b) = \neg a \land \neg b$
Implicación material	$a \rightarrow b = \neg a \lor b$
Distribución	$a \land (b \lor c) = (a \land b) \lor (a \land c)$ $a \lor (b \land c) = (a \lor b) \land (a \lor c)$
Exportación	$a \rightarrow (b \rightarrow c) = (a \land b) \rightarrow c$
Transposición	$a \rightarrow b = \neg b \rightarrow \neg a$

Limitaciones de la lógica proposicional

Como se mencionó anteriormente la lógica de orden cero es el tipo más básico de lógica lo que implica que no posee un lenguaje lo suficientemente expresivo para representar una diversa cantidad de conocimientos del mundo real, pongamos a continuación un ejemplo simple, ¿Cómo representamos (usando la lógica proposicional) las siguientes oraciones?

Todos los hombres son mortales
Sócrates es un hombre
Entonces: Sócrates es mortal

Podemos observar que en la lógica proposicional carecemos de los recursos lingüísticos necesarios para representar este conocimiento.

6.6.4 Lógica de primer orden o de predicados

La lógica proposicional mediante oraciones declarativas representa hechos, la lógica de primer orden o lógica de predicados aumenta la expresividad permitiendo representar los objetos y sus relaciones [8].

En la figura 6.2 se puede observar un modelo en el que se incluye la noción de relaciones, funciones y objetos. A continuación, se describen los nuevos recursos de los que disponemos en la lógica de primer orden.

- **Objetos:** Nos permiten expresarnos acerca de los diferentes elementos en un determinado dominio. Al conjunto de todos los objetos utilizados se les conoce como **Dominio** o **Universo de discurso** [8].

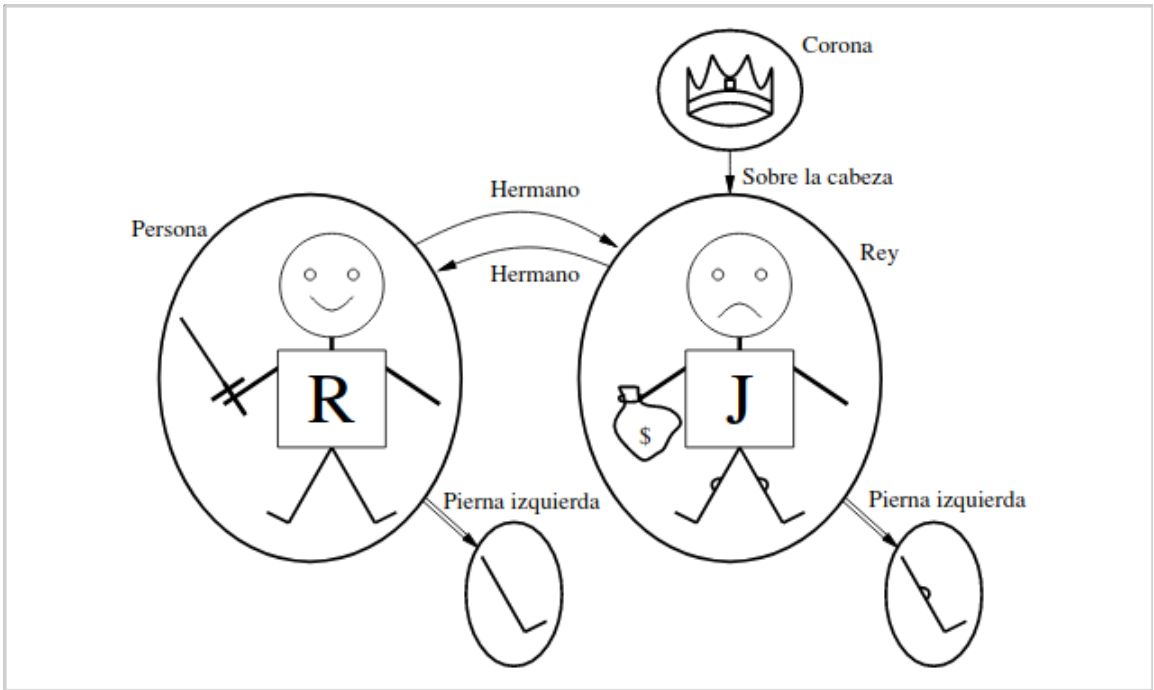


Figura 6.2: Modelo que contiene cinco objetos, dos relaciones binarias, tres relaciones unita-rias (indicadas mediante etiquetas sobre los objetos), y una función unitaria: pierna izquierda. Figura tomada de Russel Stuart; Norvig Peter. (2006). Inteligencia Artificial. Un enfoque moderno. Pearson Prentice Hall. Madrid, España.[31]

- **Función:** Una función es un tipo especial de relación que mapea un conjunto de objetos de entrada con un único objeto de salida. Al conjunto de todas las funciones se le conoce como **Base funcional** [8].
  Las funciones nos sirven para evitar la necesidad de declarar muchos objetos, supongamos que tuviéramos como Dominio los estados de un país, si quisiéramos formar predicados que tuvieran como objetos sus capitales tendríamos dos opciones crear un nuevo objeto para cada estado o crear una función llamada por ejemplo: “capital”.
- **Predicado:** El predicado indica relaciones entre objetos, al conjunto de todos los predicados se le conoce como **Base relacional** [8].
  Supongamos que tenemos dos objetos, Juan y Pablo, si quisieramos indicar que Juan es hijo de Pablo podríamos hacerlo mediante el siguiente predicado: esHijo(Juan,Pablo)
- **Variables:** Son un elemento importante para la lógica de primer orden y suelen ser representadas mediante cualquier secuencia de caracteres que inicie con mayúscula, éstas variables representan a objetos del universo de discurso [8].
  Un ejemplo de un predicado que usa variables sería el siguiente: esHijo(X,Y), más adelante veremos cómo pueden usarse para expresar conocimiento.
- **Cuantificadores:** El cuantificador universal $\forall$ que nos permite expresar relaciones acerca de todos los objetos en el dominio [8].

  $\forall$ X podría leerse como “Para todo objeto X”.

  El cuantificador existencial $\exists$ nos permite expresar la existencia de un objeto en el dominio [8]. Por ejemplo podríamos expresar la siguiente oración: “Existe un objeto X que es azul y es grande”

$\exists X$ esAzul(X) Λ esGrande(X)

Se pueden utilizar múltiples cuantificadores en una fórmula pero es importante tomar en cuenta que el orden de lo mismo si importa, por ejemplo $\forall X \exists Y$ es interpretado como para todo X existe un elemento Y, en cambio $\exists X \forall Y$ es interpretado como existe algún elemento X para el cual todos los elementos Y.

- **Términos:** Son todos los elementos que nos permitan denotar objetos y están formados por funciones, variables y constantes. Un ejemplo sería el siguiente calif (hermano(alex) , sma) es un término que denota la calificación obtenida por el hermano de Álex en el curso de Sistemas Multi-Agentes [8].

Ahora veremos la sintaxis de la lógica de primer orden para posteriormente ver algunos ejemplos de fbf.

$$
\begin{array}{rcl}
\textit{Sentencia} & \rightarrow & \textit{SentenciaAtómica} \\
 & | & (\textit{Sentencia Conectiva Sentencia}) \\
 & | & \textit{Cuantificador Variable}\ldots \textit{Sentencia} \\
 & | & \neg \textit{Sentencia} \\
 & & \\
\textit{SentenciaAtómica} & \rightarrow & \textit{Predicado}(\textit{Término}\ldots) \mid \textit{Término} = \textit{Término} \\
 & & \\
\textit{Término} & \rightarrow & \textit{Función}(\textit{Término}\ldots) \\
 & | & \textit{Constante} \\
 & | & \textit{Variable} \\
 & & \\
\textit{Conectiva} & \rightarrow & \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
\textit{Cuantificador} & \rightarrow & \forall \mid \exists \\
\textit{Constante} & \rightarrow & A \mid X_1 \mid \textit{Juan} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicado} & \rightarrow & \textit{AntesDe} \mid \textit{TieneColor} \mid \textit{EstáLLoviendo} \mid \cdots \\
\textit{Función} & \rightarrow & \textit{Madre} \mid \textit{PiernaIzquierda} \mid \cdots
\end{array}
$$

Figura 6.3: Sintaxis de la lógica de primer orden en BNF. Figura tomada de Russel Stuart; Norvig Peter. (2006). Inteligencia Artificial. Un enfoque moderno. Pearson Prentice Hall. Madrid, España. [31]

Ahora veamos un ejemplo:

Dado como dominio las personas, representar el siguiente conocimiento:

Todos los maestros son responsables
Algunos maestros son doctores
Todos los miembros del grupo son maestros o alumnos
Juan es maestro
Juan le da clase a Pedro
Algunos maestros le dan clase a todos los alumnos

Solución:

D = personas
$\forall$X maestro(X) $\rightarrow$ responsable(X)
$\exists$X (maestro(X) Λ doctor(X))
$\forall$X miembro(X) $\rightarrow$ (maestro(X) v alumno(X))
maestro(Juan)
daClase(Juan,Pedro)
$\exists$X $\forall$Y maestro(X) Λ alumno(Y) Λ daClase(X,Y)

Sustitución en la lógica de primer orden

La sustitución es un conjunto finito de pares de la forma { v1$\rightarrow$t1, v2$\rightarrow$t2, ... vn$\rightarrow$tn }, (también es utilizada la notación { v1/t1, v2/t2, ... vn/tn }, cuando se aplica una sustitución a una expresión se obtiene una nueva, reemplazando en la expresión original cada aparición de la variable vi por el término ti ($1 \leq i \leq n$) [23].

Ejemplo:

α = esPequeño(x) Λ juegaMucho(x) $\rightarrow$ esNiño(x)

Sust(x/Juan, α) = esPequeño(Juan) Λ juegaMucho(Juan) $\rightarrow$ esNiño(Juan)

Sust(x/hijo(Pedro), α) = esPequeño(hijo(Pedro)) Λ juegaMucho(hijo(Pedro)) $\rightarrow$ esNiño(hijo(Pedro))

Inferencia en la lógica de primer orden

Reglas de inferencia para cuantificadores y proposicionalización

Si eliminamos los cuantificadores de nuestros predicados podemos utilizar las reglas de inferencia de la lógica proposicional [31].

Regla de especificación universal: Podemos inferir cualquier sentencia obtenida por sustitución de la variable por un término base (un término sin variables) [31].

$$\frac{\forall v\ \alpha}{Sust(\{v/g, \alpha\})} \tag{6.1}$$

Ejemplo:

α = esPequeño(x) Λ juegaMucho(x) $\rightarrow$ esNiño(x)

Sust(x/Juan, α) = esPequeño(Juan) Λ juegaMucho(Juan) $\rightarrow$ esNiño(Juan)

Regla de especificación existencial: Esta regla es más complicada ya que requiere que el símbolo de constante k no aparezca en ninguna otra parte de la base de conocimientos[2] [31].

$$\frac{\exists v\ \alpha}{Sust(\{v/k, \alpha\})} \tag{6.2}$$

Para reducir a la inferencia proposicional partimos de dos ideas [31]:

- Toda sentencia que haga uso del cuantificador existencial se puede sustituir por el conjunto de todas las especificaciones posibles.
- Mediante el uso de la regla de especificación universal, se pueden aplicar las sustituciones de todos los términos base posibles para obtener una base de conocimiento proposicional, a esta técnica se le conoce como proposicionalización.

Al hacer uso de estas ideas podemos aplicar posteriormente las reglas de inferencia vistas en el tema anterior.

Modus Ponens Generalizado

El Modus Ponens Generalizado es una versión del Modus Ponens que puede ser usada directamente en la lógica de predicados sin la necesidad de aplicar la proposicionalización.

La ventaja clave de este tipo de reglas "elevadas" es que solo realizan aquellas sustituciones que se necesitan para realizar la inferencia [31]. A continuación se muestra el proceso de inferencia:

$$\frac{p_1^{'}, p_1^{'}, ..., p_n^{'}, (p_1 \Lambda p_2 \Lambda ... \Lambda p_n \rightarrow q)}{Sust(\theta, q)} \tag{6.3}$$

Se tienen premisas de implicación p_i y sentencias atómicas en nuestra base de conocimientos $p_i^{'}$, para aplicar esta regla de inferencia se tiene que encontrar una sustitución θ que permita que al aplicarse en las premisas de implicación y las sentencias de la base de conocimientos las haga idénticas.

A continuación se presenta un ejemplo:

$\forall$x esPequeño(x) Λ juegaMucho(x) $\rightarrow$ esNiño(x)
esPequeño(Juan)
juegaMucho(Juan)

p_1 = esPequeño(x) p_2 = juegaMucho(x) q = esNiño(x)
$p_1^{'}$ = esPequeño(Juan) $p_2^{'}$ = juegaMucho(Juan)

Para lograr que las sentencias atómicas sean iguales a las premisas de implicación se aplica la sustitución θ = { x/Juan }, por lo tanto

Sust(θ,q) = esNiño(Juan)

[2]La base de conocimiento es el lugar donde se almacena el conocimiento del experto a manera de hechos y reglas, más adelante cuando se vea el tema de sistemas expertos se verá el rol de este elemento.

Vamos a explorar un segundo ejemplo que contenga una sentencia atómica con una variable, para esto supondremos que todos los individuos de nuestro dominio juegan mucho:

$\forall$x esPequeño(x) Λ juegaMucho(x) $\rightarrow$ esNiño(x)
esPequeño(Juan)
$\forall$y juegaMucho(y)

p_1 = esPequeño(x) p_2 = juegaMucho(x) q = esNiño(x)
$p_1^{'}$ = esPequeño(Juan) $p_2^{'}$ = juegaMucho(y)

Para lograr que las sentencias atómicas sean iguales a las premisas de implicación se aplica la sustitución θ = { x/Juan }, por lo tanto

Sust(θ,q) = esNiño(Juan)

Unificación

Hasta este momento no nos hemos preguntado cómo encontrar el valor de la substitución que permita que expresiones lógicas distintas se hagan lógicas, a este proceso se le conoce como unificación [31].

Existen ocasiones donde se puede obtener más de un unificador, por ejemplo:

Unificar(madre(María,x), madre(y,z)) = { y/María, z/Juan, x/Juan }
Unificar(madre(María,x), madre(y,z)) = { y/María, x/z }

Si se realizan las sustituciones se puede observar que ambas unificaciones son válidas, generalmente se va a buscar obtener el **unificador más general (UMG)**, cada par de expresiones lógicas tiene su umg que es aquel que aplica menos restricciones sobre las variables. En el ejemplo anterior el umg sería: { y/María, x/z }.

Existe un algoritmo para obtener el umg, sin embargo en este libro no se detallará su funcionamiento debido a que en las herramientas modernas no se suele tener la necesidad de programar el algoritmo de unificación.

Si se tienen dudas o se quiere profundizar en la lógica proposicional o de predicados recomiendo leer el libro de Russel y Norvig [31].

6.6.5 Cláusulas de Horn

Las cláusulas de Horn son fórmulas lógicas con una estructura particular, son una disyunción de literales con a lo sumo (como máximo) un literal positivo.

$\neg p_1 \vee \neg p_2 \vee ... \neg p_k \vee$ q

Una cláusula de Horn también puede ser representada de la siguiente forma (para lograr esta representación se hace uso de reglas de transformación, se hace uso de la ley de morgan y la regla de implicación material):

$(p_1 \Lambda p_2 \Lambda ... \Lambda p_k) \rightarrow$ q

A la literal no negada (q) se le conoce como la cabeza de la cláusula y al resto de literales se les conoce como el cuerpo.

Importancia de las cláusulas de Horn

Como veremos a lo largo de este tema, las cláusulas de Horn juegan un rol muy importante en la programación lógica, mediante el uso de este tipo de fórmulas lógicas se pueden utilizar mecanismos de inferencia eficientes.

Desventajas de las cláusulas de Horn

Si bien es cierto que limitan la expresividad de la lógica de primer orden generalmente son lo suficientemente expresivas para representar una vasta cantidad de conocimiento.

La inferencia con cláusulas de Horn es indecidible [3]. Sin embargo es semidecidible, cuando la fórmula es un teorema a veces se puede demostrar su validez, sin embargo cuando la fórmula no es un teorema siempre se puede demostrar su inconsistencia.

Tipos de cláusulas de Horn

A continuación se desglosan las tres posibles formas que puede tener una fórmula lógica y que cumplan con la estructura de una cláusula de Horn. Las cláusulas determinadas son aquellas que tienen un literal positivo, estas a su vez se dividen en hechos y reglas, aquellas cláusulas que no tienen ningún literal positivo se llaman objetivos determinados.

Nombre	**Estructura**
Hecho	q
Regla	$\neg p_1$ v $\neg p_2$ v ... $\neg p_k$ v q
Objetivo	$\neg p_1$ v $\neg p_2$ v ... $\neg p_k$

Inferencia mediante cláusulas de Horn

Encadenamiento hacia adelante

Si tenemos nuestra base de conocimiento descrita mediante cláusulas de horn se puede realizar un algoritmo de encadenamiento hacia adelante muy sencillo que consiste en encontrar todas las reglas cuyo cuerpo pueda ser satisfecho con los hechos actuales y agregar a la base de hechos la cabeza de las reglas. Se repetirá el proceso hasta que no se puedan generar hechos nuevos o se haya comprobado algún objetivo. Hay que evitar añadir "renombramientos" a nuestra base de hechos, un renombramiento se da cuando sustituimos una variable por otra, lo cual da por resultado un predicado que tiene exactamente el mismo significado, ejemplo:

Come(x, hamburguesa) significa lo mismo que Come(y, hamburguesa)

En la figura 6.4 se muestra el pseudocódigo del algoritmo de encadenamiento hacia adelante.

Este algoritmo de encadenamiento es ineficiente ya que cada iteración trataría de agregar hechos ya conocidos, para mejorar la eficiencia se podría usar un encadenamiento hacia adelante incremental que utilizará solo aquellas reglas cuyo cuerpo contenga algún conjuntor p_i que se unifique con un hecho obtenido en la iteración anterior.

[3] La indecidibilidad es la propiedad de un sistema de conducir siempre a una respuesta verdadera o falsa, por lo cual no siempre puede demostrar o refutar una sentencia.

función PREGUNTA-EHD-LPO(BC, α) **devuelve** una sustitución o *falso*
  entradas: BC, la base de conocimiento, un conjunto de cláusulas positivas de primer orden
    α, la petición, una sentencia atómica
  variables locales: *nuevas*, las nuevas sentencias inferidas en cada iteración

repetir hasta *nuevo* está vacío
  nuevo $\leftarrow$ { }
  para cada sentencia r **en** BC **hacer**
    $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ ESTANDARIZAR-VAR(r)
    para cada θ tal que SUST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUST($p_1' \wedge \ldots \wedge p_n'$)
      para algún $p_1' \ldots p_n'$ en BC
      $q' \leftarrow$ SUST(θ, q)
      si q' no es el renombramiento de una sentencia de BC o *nuevo* **entonces hacer**
      añadir q' a *nuevo*
      $\phi \leftarrow$ UNIFICA(q', α)
      si ϕ no es *fallo* **entonces devolver** ϕ
  añadir *nuevo* a BC
  devolver *falso*

Figura 6.4: Algoritmo de encadenamiento hacia adelante simple. Figura tomada de Russel Stuart; Norvig Peter. (2006). Inteligencia Artificial. Un enfoque moderno. Pearson Prentice Hall. Madrid, España. [31]

Ejercicio de programación:

Realizar un programa capaz de realizar encadenamiento hacia adelante usando cualquier lenguaje.Este ejercicio puede ayudar a comprender los conceptos aprendidos en este capítulo.

En mi caso usé (junto con mis compañeros de equipo en ese trabajo) el lenguaje de programación Java, añadiré el link al repositorio por si alguien quiere revisar el código del algoritmo.

(`https://github.com/amr205/LogicProgramming---Forward-chaining`)

Encadenamiento hacia atrás

Con el encadenamiento hacia adelante vamos "descubriendo" nuevos hechos y paramos cuando no se pueden inferir nuevos hechos o se ha cumplido una meta. En el encadenamiento hacia atrás partimos de la meta que queremos demostrar y tratamos de determinar su validez.

Un algoritmo simple de encadenamiento hacia atrás sería el siguiente.

- Se tiene un objetivo

- Se recorren las reglas y se selecciona la regla que permita unificar la cabeza de la regla con el objetivo, de esta manera se obtiene el umg y se sustituye el cuerpo de la regla con el umg, todos los predicados dentro de la regla se vuelven sub-objetivos y para cada uno de ellos se debe realizar este proceso.
- Si no se encontró ninguna regla se recorren los hechos, si el objetivo es igual a algún hecho ese objetivo se considera como verdadero, en caso contrario se considera falso.

Veamos el siguiente ejemplo:

En nuestra base de conocimientos tenemos los siguientes hechos y reglas.

1. haceCroac(Fritz)
2. comeMoscas(Fritz)
3. haceCroac(x) Λ comeMoscas(x)→esRana(x)
4. canta(x)ΛtieneAlas(x)→esCanario(x)
5. esRana(x)→esVerde(x)
6. esCanario(x)→esAmarillo(x)

El objetivo es determinar si: esVerde(Fritz)

A continuación se muestra un diagrama de cómo se realiza el encadenamiento hacia atrás para demostrar que el objetivo es verdadero.

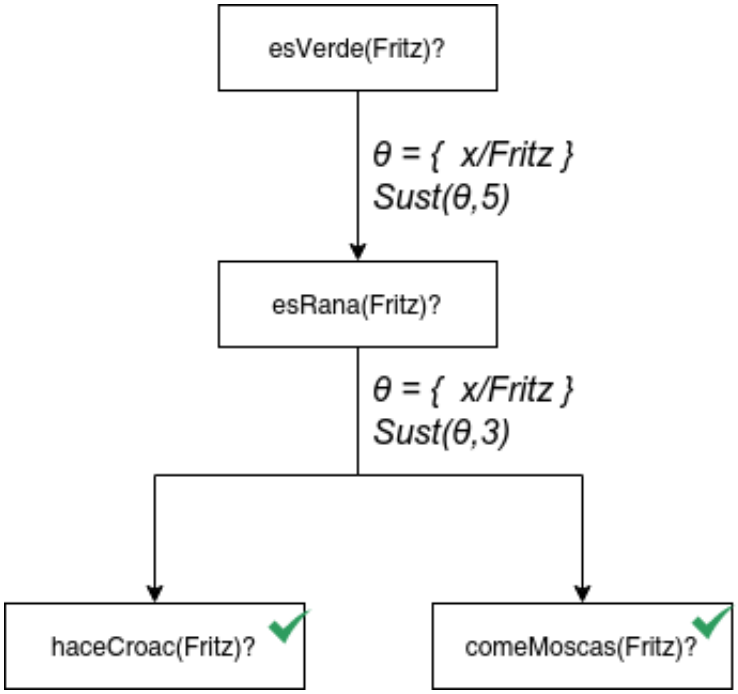


Figura 6.5: Proceso de encadenamiento hacia atrás del objetivo esVerde(Fritz).

Mecanismo de resolución SLD

Para realizar un encadenamiento hacia atrás podemos utilizar el mecanismo de resolución SLD (Selective Linear Definite clause resolution). Para explicar cómo funciona este mecanismo primero hay que entender la función de selección φ, esta función dado un objetivo Q devuelve uno y solo uno de los átomos de Q.

Ejemplo:

Q = $A_1 \Lambda A_2 \Lambda A_j \Lambda ... \Lambda A_n$
φ(Q) = A_j

Funcionamiento del mecanismo de resolución SLD

En este caso la función de selección devolverá el primer átomo del objetivo. Dado un objetivo Q de la forma $A_1 \Lambda A_2 \Lambda A_j \Lambda ... \Lambda A_n$. se buscará la primera sentencia C del programa $B_1 \Lambda ... \Lambda B_m \rightarrow$ H tal que φ(Q) = A y H sean unificables por un umg θ. Entonces la nueva pregunta $Q^{'}$ (el resolvente) será igual a la sustitución Sust(($B_1 \Lambda ... \Lambda B_m \Lambda A_2 \Lambda ... \Lambda A_n$), θ). El proceso continuará de igual forma hasta obtener la pregunta vacía (lo que corresponde a un éxito, se obtiene una pregunta vacía si existe un umg θ tal que Sust(Q,θ)=Sust(C,θ)), o una pregunta para la cual no exista resolvente con ninguna sentencia del programa (lo que corresponderá a un fracaso) [23].

Vamos a explorar este proceso separándolo en pasos:

1. Se tiene un objetivo Q de la forma $A_1 \Lambda A_2 \Lambda A_j \Lambda ... \Lambda A_n$
2. Se selecciona un átomo A del objetivo Q, A = φ(Q)
3. Se busca la primera sentencia C de la forma $B_1 \Lambda ... \Lambda B_m \rightarrow$ H tal que A y H sean unificables por un umg θ.
4. Se obtiene el nuevo resolvente (también llamado derivación ya que $Q^{'}$ deriva de Q), $Q^{'}$ = Sust(($B_1 \Lambda ... \Lambda B_m \Lambda A_2 \Lambda ... \Lambda A_n$), θ) En este paso cuando la sentencia C es un hecho solo se tendría la cabeza H de la cláusula por lo cual $Q^{'}$ sería igual a $Q^{'}$ = Sust(($A_2 \Lambda ... \Lambda A_n$), θ)

Este proceso se repite siendo $Q^{'}$ el nuevo objetivo hasta que se cumpla una de las siguientes condiciones:

- En el paso 3 no se encontró ninguna sentencia que permita generar un nuevo resolvente, por lo cual la resolución fracaso.
- Tras el paso 4 $Q^{'}$ es igual a una pregunta vacía, lo que corresponde a un éxito.

6.6.6 Sistemas expertos

Los sistemas expertos son sistemas basados en conocimiento diseñados para realizar tareas que normalmente requerirán un experto. Estos sistemas son usados para resolver problemas de dominio específico y se comportan como un sistema asesor en la toma de decisiones [38].

El desarrollo de los sistemas expertos empezó alrededor del año 1965 con un proyecto de Edward Feigenbaum, a quien se le atribuye el título de padre de los sistemas expertos), el desarrollo de estos sistemas continuó durante las siguientes décadas, donde fueron utilizados tanto en la industria como académicamente. Su auge fue durante el boom de la inteligencia artificial (1980–1987), en la actualidad la popularidad de los sistemas expertos ha decaído y si bien existen sistemas expertos implementados en diversas aplicaciones la opinión popular parece indicar que fallaron en cumplir las expectativas esperadas [16].

Componentes de los sistemas expertos

A continuación se describen los principales componentes de un sistema experto [38]:

- Interfaz de usuario: Es el mecanismo mediante el cual el usuario final va a interactuar con el sistema experto.
- Base de conocimiento: Contiene todo el conocimiento adquirido del experto a manera de reglas y hechos.
- Motor de inferencia: Es el mecanismo que se encarga de realizar la inferencia manipulando las reglas y hechos de la base de conocimiento.

- Subsistema de explicación o justificación: Este módulo o subsistema es el encargado de explicar el razonamiento mediante el cual el sistema experto llegó a una conclusión.
- Subsistema de adquisición de conocimiento: Este módulo o subsistema permite al usuario añadir nuevo conocimiento sin la necesidad de que un ingeniero o personal especializado se vea involucrado en este proceso. Algunos sistemas expertos cuentan con un módulo de aprendizaje que les permite adaptarse de acuerdo a la información que reciben.

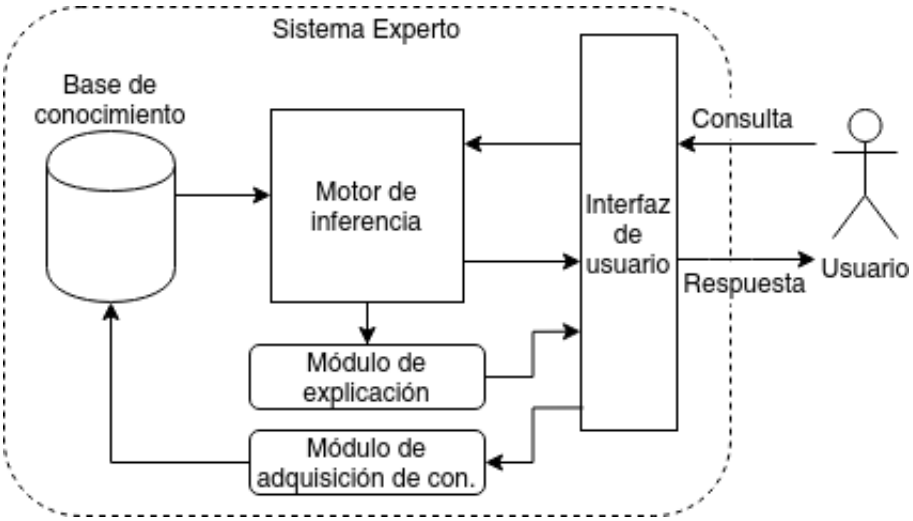


Figura 6.6: Representación gráfica de los componentes de un sistema experto.

Desarrollo de un sistema experto

Para el desarrollo de un sistema experto se tiene que realizar la extracción del conocimiento para posteriormente desarrollar el sistema. A continuación se presentan algunos pasos sugeridos que se pueden seguir durante el desarrollo:

1. Identificación del problema: Es muy importante saber qué problema pretende resolver nuestro sistema experto y los objetivos que van a utilizar nuestros usuarios finales. También en esta fase el desarrollador debe realizar una investigación sobre el dominio del problema con el fin de tratar de adquirir conocimiento general sobre el problema o situación a resolver. Posteriormente se debe obtener información de expertos mediante técnicas como la entrevista (es importante la investigación previa para realizar preguntas puntuales).
2. Conceptualización del conocimiento: En esta fase se debe representar el conocimiento mediante técnicas como los marcos, redes semánticas, etc. Existen libros sobre la ingeniería del conocimiento que repasan este tipo de técnicas. En esta sección se recomienda mantener la comunicación con el experto para que nos indique si nuestras representaciones son adecuadas y coinciden con su conocimiento.
3. Formalización del conocimiento: El conocimiento se va a representar mediante un modelo formal, es importante tomar en cuenta como va a funcionar nuestro modelo de inferencia, lo más probable es que tengamos que representar nuestro conocimiento mediante el uso de cláusulas de Horn.
4. Implementación del sistema experto: En esta fase se desarrollará el sistema experto y se realizará la programación necesaria, se pueden usar herramientas existentes para facilitar el desarrollo.
5. Validación y mantenimiento del sistema: Se debe validar que las respuestas del sistema experto y de los expertos coincidan de manera consistente.

Uso del lenguaje de programación prolog

PROLOG es un lenguaje declarativo de programación lógica diseñado para representar y utilizar el conocimiento que se tiene sobre un determinado dominio.

¿Qué es un lenguaje declarativo?

Existen dos estilos principales en los lenguajes de programación, imperativos y declarativos. Los programas en los lenguajes imperativos constan de instrucciones que nos permiten modificar el estado de un programa mediante la modificación de variables, básicamente se describe el proceso de cómo hacer algo; ejemplos de estos lenguajes son Java, C++, PHP. En los programas de los lenguajes declarativos se describe qué se quiere hacer pero no se especifica el cómo; PROLOG es un ejemplo de este tipo de lenguajes.

Sintaxis de la base de conocimientos

En un programa de prolog se pueden guardar reglas y hechos, estas reglas y hechos deben ser cláusulas de Horn, a continuación se describe la sintaxis: Los objetos tienen que iniciar con minúscula y las variables con mayúscula.

Sintaxis de una regla en prolog

Cabeza :- Cuerpo.

Las cláusulas del cuerpo pueden estás separadas por “,” que representa una conjunción o “;” que representa una disyunción.

Ejemplo:

haceCroac(x)ΛcomeMoscas(x)→esRana(x)

Esta regla se representaría de la siguiente manera en prolog:

esRana(X) :- haceCroac(X),comeMoscas(X).

Sintaxis de un hecho en prolog

Cabeza.

Ejemplo:

haceCroac(Fritz)

Este hecho se representaría de la siguiente manera en prolog:

haceCroac(fritz).

Objetivos en prolog

Una vez tenemos descrita nuestra base de conocimientos podemos realizar “preguntas” mediante los objetivos. Para preguntar ¿es fritz verde? construiríamos el objetivo en prolog “esVerde(fritz).”, también podemos hacer uso de variables para preguntar ¿Quién es verde? construyendo el objetivo de la siguiente manera “esVerde(X).”.

Inferencia en prolog

Prolog utiliza como método de inferencia el mecanismo de resolución SLD descrito anteriormente en este libro.

Ejercicio de programación:

Realizar el desarrollo de un sistema experto del tema que el lector prefiera y usando cualquier lenguaje de programación. Yo recomiendo utilizar un lenguaje de programación como prolog para no tener que realizar la programación del motor de inferencia.

(`https://github.com/amr205/animals-prolog`)

IV Parte cuatro: Machine Learning

7. Introducción al capítulo

7.1 Definición

El machine learning o aprendizaje automático es una disciplina del campo de la inteligencia artificial que pretende generar sistemas capaces de aprender a través de la “experiencia”. El problema de aprendizaje puede ser descrito de la siguiente manera: Un programa se dice que aprende a través de la experiencia E con respecto a un tipo de tareas T y una medición de su desempeño P, si su desempeño en las tareas T, medidas por P, mejora a través de la experiencia E [21].

Esta definición puede ser muy formal sin embargo describe un problema general sobre el cual se puede aplicar machine learning. Dicho de forma más simple en el aprendizaje automático no se desarrolla de manera explicita la lógica que nos permite ir de unos datos de entrada a un resultado, en cambio se proporcionan datos de entrada junto con los resultados que esperamos aprender y los usamos para entrenar un algoritmo de aprendizaje, este algoritmo genera un "programa"que nos permite predecir el resultado a partir de nuevos datos de entrada.

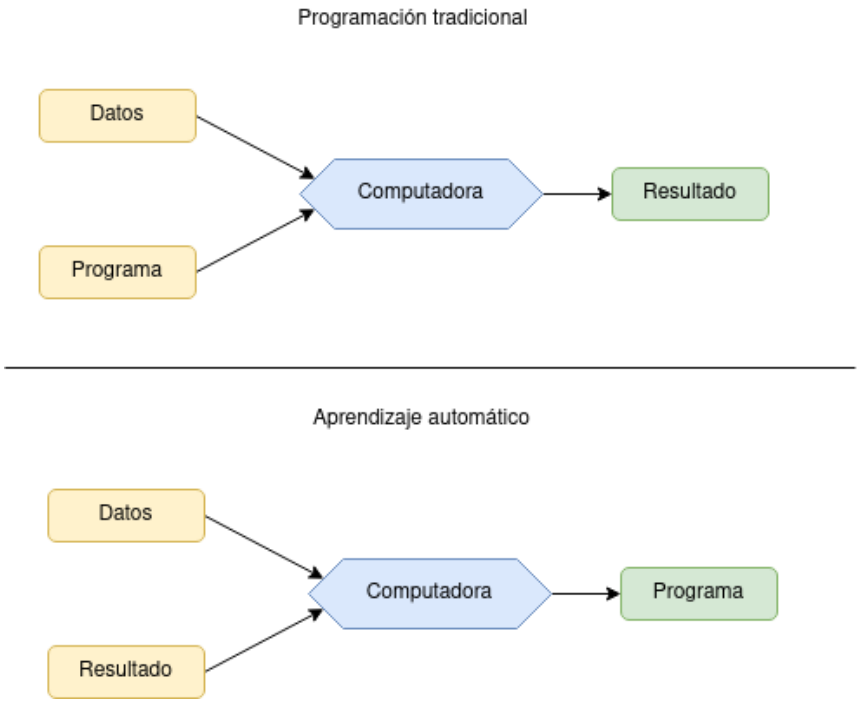


Figura 7.1: Comparación entre la programación tradicional y el aprendizaje automático.

7.2 Clasificación

Existen tres tipos de algoritmos principales dependiendo del tipo de aprendizaje que realizan, a continuación se describen de manera breve:

- **Aprendizaje supervisado:** Se aprende a través de ejemplos que se encuentran previamente clasificados o emparejados con un valor, un ejemplo puede ser un clasificador de imágenes que debe aprender a distinguir entre perros y gatos que para el proceso de aprendizaje requerirá de un conjunto de imágenes asociadas a la clase correcta.
- **Aprendizaje no supervisado:** Este tipo de algoritmos aprenden a través de ejemplos sin clasificar y tienen que aprender patrones que les permitan organizarlos de alguna manera, un ejemplo podría el siguiente: una tienda de ropa necesita elegir las medidas que tendrán sus prendas de ropa para las tallas chica, mediana y grande, con el objetivo de lograr esto se recopilan las medidas de la gente que vive cerca de la tienda y se utiliza un algoritmo de machine learning que le indica cuales serían las medidas adecuadas (más adelante en este libro veremos que clase de algoritmo podría ser útil para esta tarea).
- **Aprendizaje semi-supervisado:** En este tipo de algoritmo nuestro set de entrenamiento contiene ejemplos clasificados y no clasificados, la mayoría de los ejemplos no suelen estar clasificados, aunque usar ejemplos sin clasificar pueda parecer contra-intuitivo estos añaden información sobre el problema lo que puede resultar en un mejor modelo.
- **Aprendizaje reforzado:** Este tipo de algoritmos aprenden a través de la experiencia, interactúan con su entorno y reciben recompensas que les indican si las acciones realizadas han sido correctas o no.

7.3 Problemas que resuelve

Los algoritmos de machine learning nos permiten resolver principalmente problemas de clasificación,regresión, asociación y agrupamiento, a continuación se describe su significado [3]:

- **Clasificación:** este problema consiste en asignar una etiqueta a un ejemplo sin clasificar, uno de los ejemplos más famosos es la detección de spam.
- **Regresión:** este problema consiste en predecir un valor dado un ejemplo sin etiquetar, por ejemplo otorgar las características de una casa y predecir su costo.
- **Agrupación (Clustering):** Consiste en agrupar los datos de tal forma que los elementos de estos grupos sean similares, esto puede ser útil por ejemplo para segmentar nuestros clientes según su forma de comprar.
- **Asociación:** Consiste en relacionar instancias de nuestro set de entrenamiento con características similares, por ejemplo encontrar la noticia más similar a la que acaba de leer el usuario.

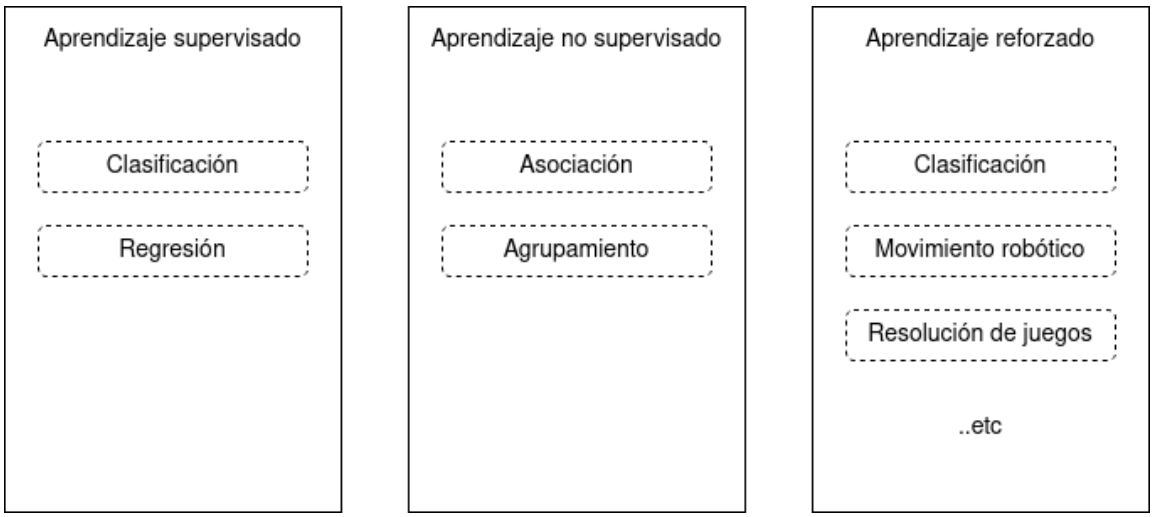


Figura 7.2: Clasificación del aprendizaje automático y los tipos de problemas o tareas que pueden solucionar.

¿En este libro ya utilizamos machine learning?

Si, en el tema Sistemas clasificadores (Learning classifier system) realizamos algoritmos de machine learning y precisamente vimos como uno de ellos pertenece a la categoría de aprendizaje supervisado y el otro a la categoría de aprendizaje reforzado.

7.4 Importancia del machine learning

Este tipo de algoritmos nos permiten resolver tareas que de otra manera serían prácticamente imposibles, por poner un ejemplo simple, sin el uso de este tipo de algoritmos cómo sería posible crear un algoritmo al cual le dieras una imagen y la clasificara como un perro o un gato.

Hoy en día el machine learning ha sido aplicado en una variedad de ámbitos de manera exitosa, a continuación se listan algunos de ellos:

- Procesamiento del lenguaje natural
- Computer vision (Procesamiento de imágenes)
- Detección de amenazas terroristas
- Evaluación de clientes
- Sistemas de recomendación
- Sistemas de detección de anomalías
- Reconocimiento de genes en secuencias de ADN
- Etc.

El paradigma simbólico visto en el capítulo anterior tiene el problema de que el aprendizaje no se adquiere directamente del entorno o a través de ejemplos, por lo cual hay dominios muy complejos de modelar o al ser vaciado el conocimiento se corre el riesgo de perder detalles importantes del modelo, el machine learning hoy en día presenta una gran oportunidad debido a que la cantidad de datos e información generada en estos últimos años hacen posible el uso de estos algoritmos para la resolución de problemas complejos.

8. Aprendizaje supervisado

8.1 Introducción al capítulo

En este capítulo se revisarán diversos algoritmos de machine learning que utilizan aprendizaje supervisado, antes de revisar distintos algoritmos que realizan la tarea de clasificación o regresión vamos a observar algunas generalidades presentes en este tipo de aprendizaje.

El proceso suele iniciar con la recolección de los datos. Estos datos van a conformar nuestro conjunto de datos (dataset), consisten de una serie de pares de datos (entrada, salida). Los valores de entrada pueden ser cualquier cosa, correos, imágenes, etc. Los valores de salida pueden ser etiquetas que correspondan a una clase (perros, gatos, etc), valores numéricos (precio, probabilidad, etc), o secuencia de datos [3].

Para poder procesar los datos la entrada suele estar compuesta de un vector de atributos (feature vector ó attribute vector), por ejemplo si cada elemento de nuestro conjunto de datos esta compuesto de color, anchura y altura, nuestro vector de atributos debería representar adecuadamente esto, algunos algoritmos requieren que nuestros vectores estén compuestos de datos númericos, por lo cual para el ejemplo anterior se podría descomponer el atributo de colos en tres atributos que serían el valor de la intensidad del color en el canal rojo, azul y verde. El vector de atributos de una instancia con color azul, anchura de 3 metros y altura de 15 sería el siguiente: [0,0,1,3,15], al número de atributos de nuestro vector se le conoce como dimensionalidad, en este ejemplo simple se tiene una dimensionalidad de 6.

Es deber del analista de datos determinar cómo convertir una entidad del mundo real a un vector de atributos y como transformar o extraer las características relevantes del problema. Un ejemplo más complejo sería el convertir un correo electrónico, esta tarea la dejaré para más adelante ya que será uno de los ejercicios a realizar en temas posteriores.

Nuestro conjunto de datos suele estar dividido en otros subconjuntos, al inicio de este capítulo usaremos solamente el set de entrenamiento pero más adelante veremos cómo utilizar los otros, a continuación se describen los principales subconjuntos:

- **Set de entrenamiento (training set):** Estos elementos que ya poseen clasificación son aquellos que nos servirán para entrenar a nuestro algoritmo.
- **Set de prueba (test set):** Con estos elementos se probará el desempeño de nuestro algoritmo, usando instancias que nuestro algoritmo no observó durante el entrenamiento.
- **Set de validación (validation set):** Este conjunto se utiliza para ajustar parámetros del algoritmo para obtener un mejor resultado, dependiendo del algoritmo existen diferentes hiperparametros [1] a optimizar y también existen diferentes técnicas para usar este conjunto.

Usando nuestro conjunto de datos, aplicaremos un **algoritmo de aprendizaje**, estos algoritmos suelen estar compuestos de lo siguiente [3]:

- **Una función de pérdida (loss function):** Nos permite medir la diferencia entre el resultado obtenido por el algoritmo de aprendizaje y el valor de salida real de una sola instancia.
- **Un criterio de optimización (optimization criteria):** El criterio mediante el cual se va medir la efectividad del modelo, este criterio está basado en la función de pérdida, un ejemplo podría ser **una función de coste o error (cost function)**, esta función mide el error del algoritmo de aprendizaje a través de todas las instancias, suele ser el promedio del valor obtenido al aplicar la función de pérdida sobre cada instancia.
- **Una rutina de optimización (optimization routine):** Este es el proceso mediante el cual se va a manejar la información para encontrar una solución al criterio de optimización.

El resultado de aplicar el algoritmo de aprendizaje sobre un conjunto de datos es un **modelo**, este modelo nos permitirá realizar predicciones sobre los nuevos elementos no asociados a una información de salida.

En la especialización de machine learning realizado por la Universidad de Washington en la plataforma de Coursera se presenta una imagen similar a la siguiente que muestra los elementos descritos anteriormente.

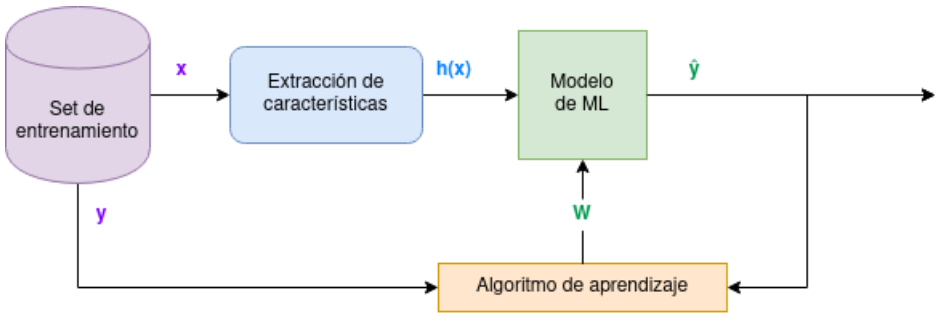


Figura 8.1: Diagrama representando un problema de aprendizaje supervisado, donde se tiene un set de entrenamiento con datos de entrada X y datos de salida Y, se entrena un modelo con parámetros W para generar predicciones $\hat{Y}$

Se invita al lector a identificar los elementos aquí descritos en el tema UCS (LCS con aprendizaje supervisado) de este libro.

Antes de revisar algunos de los diferentes algoritmos de aprendizaje vamos a identificar que es el overfitting, algunas técnicas para resolver este problema y una rutina de optimización que nos permitirá aprender los parametros W de nuestros modelos. Es por esto que se motiva al lector entender claramente las partes y procesos que forman parte del aprendizaje supervisado presente en la figura 8.1.

[1] Un hiperparámetro es una propiedad del algoritmo de aprendizaje, estos valores no son aprendidos a partir del proceso de aprendizaje y tienen que ser determinados por el desarrollador.

8.2 Overfitting y underfitting

Cuando se entrenan diferentes modelos para desempeñar las tareas de clasificación o regresión es probable que nos encontremos con problemas de overfitting o underfitting.

Overfitting

El problema de overfitting se presenta cuando el modelo se ha ajustado demasiado bien a los datos con los cuales fue entrenado (Aprendiendo incluso el ruido presente en nuestros datos de entrenamiento), por lo cual su capacidad de generalización [2] es bastante mala.

Ruido vs Señal

La señal es la función o patrón "verdadera"que pretendemos extraer de nuestros datos. El ruido presente en los datos se puede dar por errores durante la medición, aleatoriedad presente en los datos o valores atípicos (Outliers) [3].

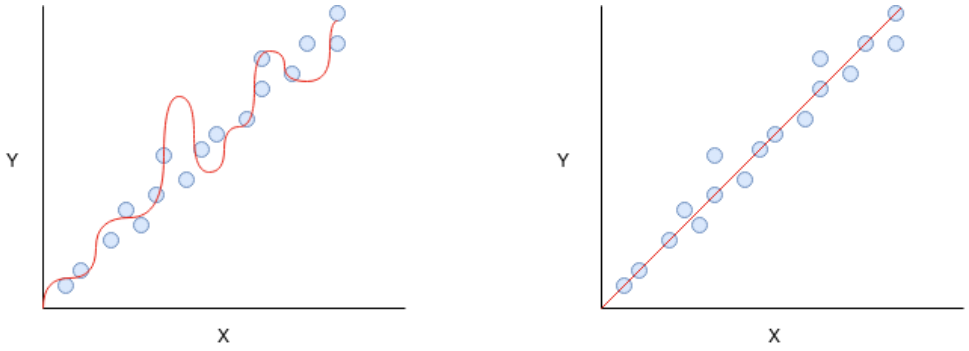


Figura 8.2: Dos modelos con los mismos datos de entrada. (Izquierda) Modelo con overfitting, (Derecha) Modelo ajustado correctamente.

Underfitting

Este es el problema contrario al overfitting, se da cuando nuestro modelo no es lo suficientemente expresivo para representar la relación entre los datos de entrada X y los datos de salida Y. En este caso nuestro modelo no se desempeña bien ni siquiera en nuestro set de entrenamiento.

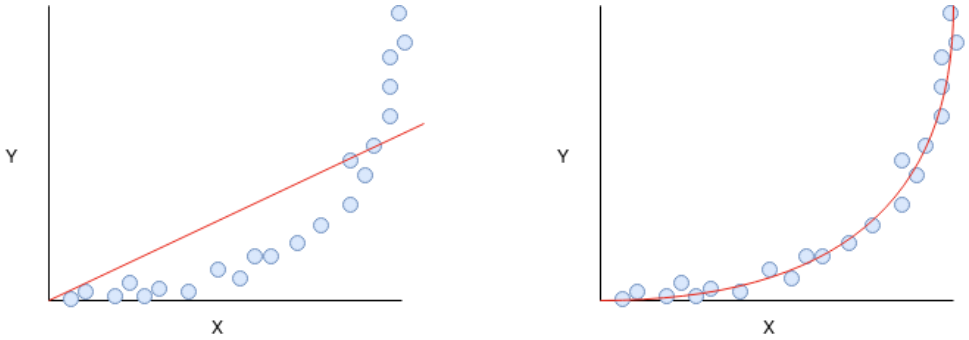


Figura 8.3: Dos modelos con los mismos datos de entrada. (Izquierda) Modelo con underfitting, (Derecha) Modelo ajustado correctamente.

[2] La generalización es la capacidad de desempeñarse exitosamente las tareas de clasificación o regresión en datos que el modelo no ha observado.

[3] Los valores atípicos (Outliers) son aquellos valores que difieren significativamente del resto de los datos o que no pertenecen al dataset.

¿Cómo detectar el overfitting o el underfitting?

Posteriormente en la parte TODO de este libro se tratan formas más avanzadas de detectar el overfitting o underfitting, por ahora basta con evaluar nuestros modelos (tema que se explorará más adelante en este capítulo) en el set de entrenamiento y el set de prueba.

De manera general si nuestro desempeño en el set de entrenamiento es mucho mejor que nuestro desempeño en el set de prueba nuestro modelo tiene una alta probabilidad de sufrir overfitting.

En cambio si nuestro desempeño en el set de entrenamiento y el set de prueba es bajo es probable que nuestro modelo sufra de underfitting.

¿Cómo combatir el overfitting?

A continuación se describen de manera general algunas maneras de prevenir el overfitting en nuestro modelo, algunos de estos temas serán tratados con más profundidad más adelante en el libro:

- **Recolectar más datos:** Usar más datos para el entrenamiento suele ayudar para mejorar la capacidad de generalización de nuestro modelo, sin embargo esto suele ser bastante costoso por lo cual se recomienda primero probar con otras maneras de combatir el overfitting.
- **Remover características o atributos en nuestros datos:** Si nuestro modelo tiene acceso a datos que no son relevantes para el problema es más fácil que nuestro modelo presente overfitting. Se puede hacer manualmente o utilizar algoritmos diseñados precisamente para realizar esta tarea.
  Por ejemplo supongamos que tenemos que predecir que fruta estamos analizando en base a una serie de características (color, altura, anchura, peso y hora del análisis), en este caso la última característica no es relevante, si la persona encargada siempre analizará las manzanas en la noche, un modelo con overfitting "pensaría"que una fruta analizada en la mañana no puede ser una manzana.
- **Early stopping:** Cuando se entrene el modelo con algoritmos de aprendizaje iterativos es posible medir el desempeño en cada iteración, por ende podría detenerse el proceso de aprendizaje cuando nuestro modelo deje de generalizar exitosamente y empiece a presentar overfitting.
  Esta técnica no suele recomendarse debido a que durante el entrenamiento se suele buscar minimizar o maximizar una función, si se usa early stopping esto deja de ser verdadero por lo cual es recomendable utilizar otras técnicas.
- **Regularización:** Estas técnicas fuerzan a nuestro modelo a ser más simple reduciendo el problema de overfitting en modelos complejos. La regularización es una de las maneras más recomendadas para combatir este problema.

Los modelos más complejos suelen tener predisposición al overfitting, y los modelos más simples al underfitting.

Algunos modelos y algoritmos de aprendizaje tienen sus propios parámetros, mediante el uso de un set de validación se pueden ajustar estos parámetros para evitar el overfitting o el underfitting.

8.3 Técnicas de regularización

8.3.1 Regularización L2 (Ridge penalisation)

Los modelos con overfitting suelen tener valores muy altos en sus parámetros, por lo cual al penalizar valores muy altos en los mismos se puede regularizar el modelo.

La fórmula correspondiente a una función de coste J con regularización L2 es la siguiente:

$$J(X,Y,\theta) = cf(X,Y,\theta) + \lambda \sum_{j=1}^{n} \theta_j{}^2 \tag{8.1}$$

Donde:

X son los atributos de nuestro set de entrenamiento
Y son las etiquetas o variable a predecir de nuestro set de entrenamiento
θ son los atributos del modelo
$cf(X,Y,\theta)$ es la función de coste o error sin regularizar de nuestro modelo
λ es un parámetro que controla el nivel de regularización aplicado al modelo

La regularización de tipo L2 tiende a disminuir el valor de los parámetros del modelo sin llegar a fijar algunos en 0, en la figura 8.4 se presenta un modelo de regresión polinomial de grado 9, aplicando distintos valores en λ se puede apreciar el efecto resultante.

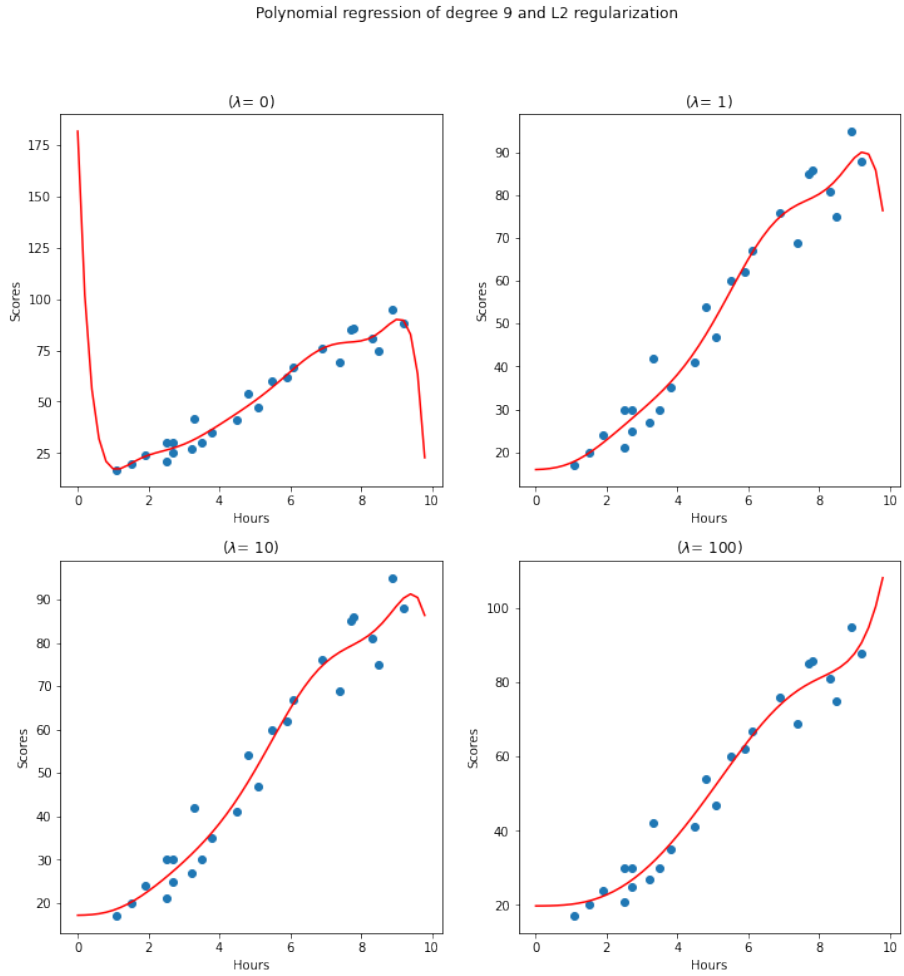


Figura 8.4: Ejemplo de una regresión polinomial de grado 9 con regularización L2, en cada figura se puede observar el valor de λ.

Cuando se tienen muchas características, la regularización l2 tiende a funcionar de una manera aproximada a lo que se muestra en la figura.

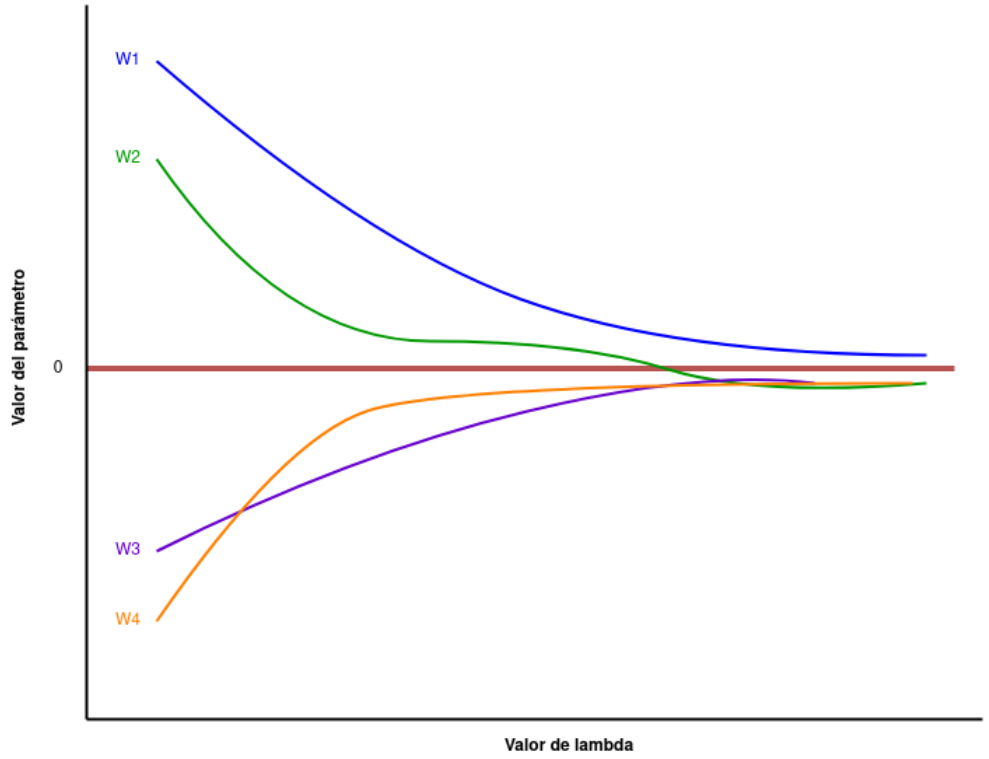


Figura 8.5: Forma general en el cambio de los pesos de un modelo de acuerdo al cambio del valor de λ en regularización l2.

8.3.2 Regularización L1 (Lasso penalisation)

La fórmula correspondiente a una función de coste J con regularización L1 es la siguiente:

$$J(X,Y,\theta) = cf(X,Y,\theta) + \lambda \sum_{j=1}^{n} |\theta_j| \tag{8.2}$$

Donde:

X son los atributos de nuestro set de entrenamiento
Y son las etiquetas o variable a predecir de nuestro set de entrenamiento
θ son los atributos del modelo
$cf(X,Y,\theta)$ es la función de coste o error sin regularizar de nuestro modelo
λ es un parámetro que controla el nivel de regularización aplicado al modelo

La principal diferencia con regularización l2 es la manera en la cual disminuyen los pesos, ya que como se muestra en la figura 8.6 al aumentar el valor de λ se van fijando en 0, por esta razón este tipo de regularización es usada para el proceso de selección de características, descartando aquellas que sean menos relevantes.

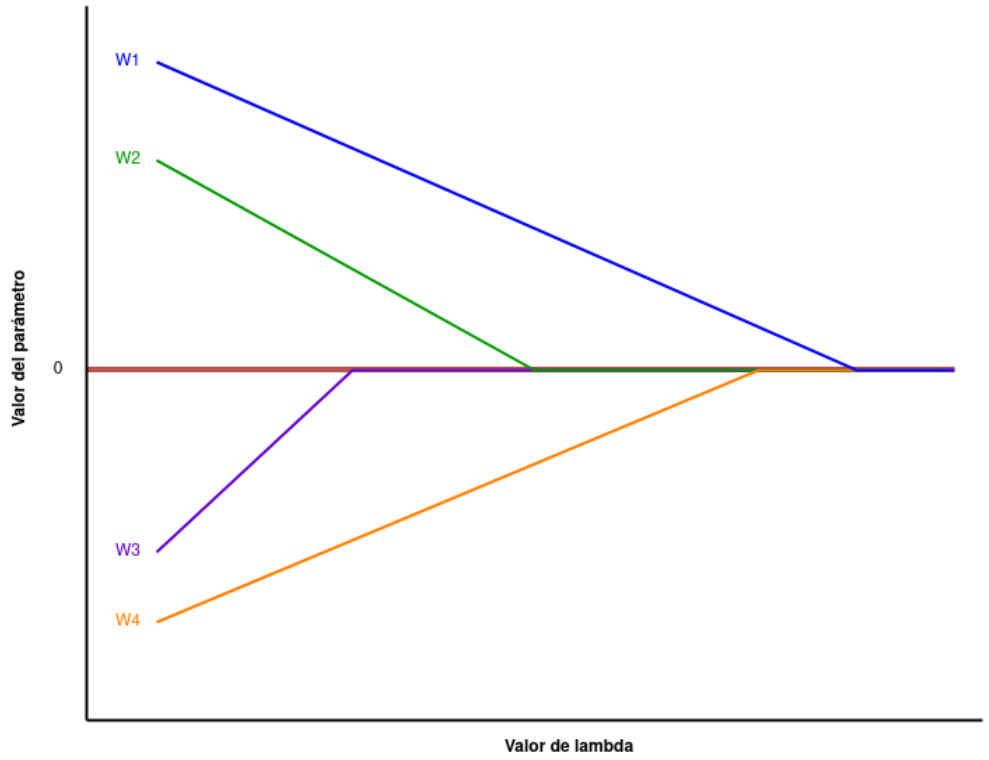


Figura 8.6: Forma general en el cambio de los pesos de un modelo de acuerdo al cambio del valor de λ en regularización l1.

8.4 Descenso del gradiente

En esta sección examinaremos el funcionamiento básico de la rutina de optimización llamada descenso del gradiente, esta rutina tiene diversas variantes y algunas implementaciones más complejas que pretenden mejorarla sin embargo en este capitulo nos centraremos en la más simple y dejaremos los demás temas para la siguiente parte de este libro.

¿Porqué este tema se encuentra en esta sección del libro?

Al inicio puede parecer un poco contra intuitivo el revisar este tema antes de ver algún algoritmo de aprendizaje de regresión o clasificación, sin embargo esta rutina es utilizada por diversos algoritmos y es ampliamente usada actualmente, por lo cuál considero importante que el lector entienda su funcionamiento de manera general para posteriormente ver su implementación específica en los temas posteriores. Otra razón para colocar esta información aquí es la posibilidad de encontrarlo fácilmente en el índice por si se requiere repasar el funcionamiento de esta rutina de optimización.

¿Qué es el descenso del gradiente?

El descenso del gradiente es una manera de minimizar una función objetivo (referido como criterio de optimización en la introducción del capítulo) J(θ) donde los parámetros del modelo $\theta \in \mathbb{R}^d$ son ajustados de manera iterativa en la dirección contraria al gradiente de la función objetivo respecto a los parámetros $\nabla_\theta J(\theta)$ [30].

Esta definición formal es bastante adecuada una vez se entienden los conceptos que se están utilizando, es por ello que vamos a revisar el funcionamiento de esta rutina de optimización de la manera más simple posible.

Función objetivo $J(\theta)$

La función objetivo como fue descrito en la introducción del capítulo mide la efectividad de nuestro modelo usando generalmente una función de coste o error, mientras menor sea el valor quiere decir que el desempeño de nuestro algoritmo de aprendizaje es mejor. Por intuición sabemos que hay una combinación de valores en nuestros parámetros θ que minimizan esta función objetivo.

Para simplificar el problema y para poder mostrar de manera gráfica esta situación en la figura 8.7 se puede observar que existe un valor para θ_1 que minimiza la función $J(\theta)$.

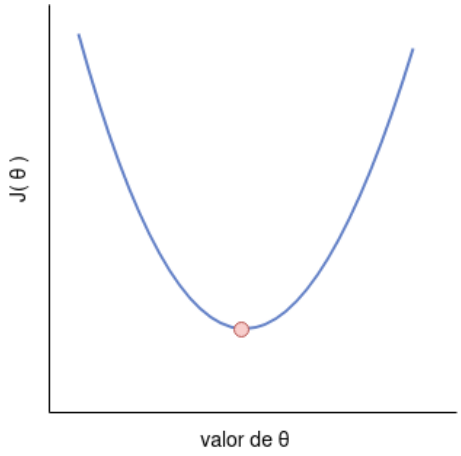


Figura 8.7: Representación visual de la función $J(\theta)$, en esta figura el valor mínimo de $J(\theta)$ se encuentra señalado con un punto rojo

Gradiente $\nabla_\theta J(\theta)$

Cuando nosotros empezamos el entrenamiento iniciamos con valores aleatorios de θ por lo cual nuestro objetivo es lograr encontrar los valores θ que optimizen la función $J(\theta)$, entonces nuestra pregunta es ¿Cómo pasar de nuestro valor inicial al valor óptimo?.

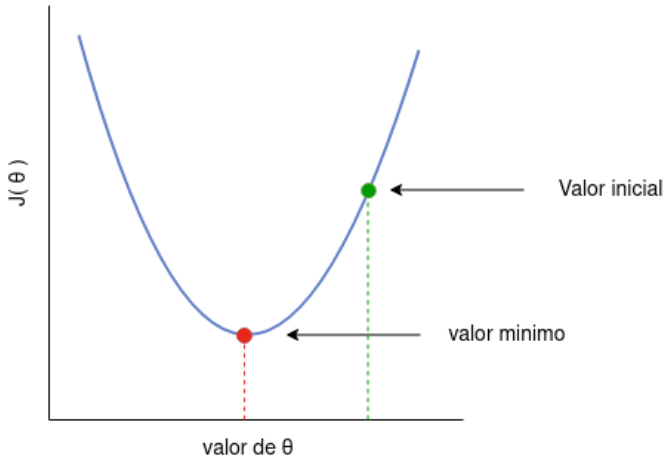


Figura 8.8: Representación visual de la función $J(\theta)$, en esta figura el valor mínimo de $J(\theta)$ se encuentra señalado con un punto rojo y el valor inicial se encuentra marcado con el punto verde

En el caso presentado en la figura 8.8 solo tenemos un parámetro θ por lo cual mediante el uso de una derivada podemos obtener la pendiente e ir en la dirección contraria para irnos acercando al valor de θ que minimize $J(\theta)$.

Recordemos que la derivada de la función en un punto marcado es equivalente a la pendiente de la recta tangente.

De esta manera al calcular la derivada de $J(\theta)$ respecto a θ podemos obtener la dirección hacia la cual debemos llevar nuestro valor θ para minimizar la función.

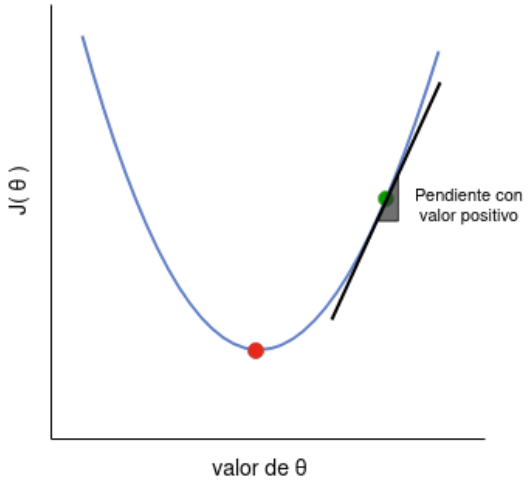


Figura 8.9: Representación visual de la función $J(\theta)$, en esta figura el valor mínimo de $J(\theta)$ se encuentra señalado con un punto rojo y el valor inicial se encuentra marcado con el punto verde y la recta tangente a la función $J(\theta)$ se encuentra dibujada con color negro.

¿Qué hacemos si tenemos más de un parámetro?

La respuesta es relativamente sencilla, el objetivo es obtener el gradiente, el gradiente es una generalización multivariable de la derivada. Mientras que una derivada se puede definir solo en funciones de una sola variable, para funciones de varias variables, el gradiente toma su lugar. El gradiente es una función de valor vectorial, a diferencia de una derivada, que es una función de valor escalar. De esta manera al obtener el gradiente sabremos en que dirección actualizar cada uno de nuestros parámetros.

El cálculo del gradiente dependerá del algoritmo de aprendizaje que se este utilizando por lo cual en este momento no veremos ningún ejemplo en específico.

Descenso del gradiente

Ahora que sabemos que es el gradiente y porqué queremos obtenerlo, el descenso del gradiente tiene mucho más sentido.

Esta rutina de optimización en su forma más básica (Descenso del gradiente por lotes o Batch Gradient Descent) consiste en calcular el gradiente de la función objetivo $J(\theta)$ respecto a los parámetros θ a través de todo el set de entrenamiento para de manera iterativa ajustar los parámetros del algoritmo de aprendizaje de acuerdo a la siguiente fórmula:

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta) \tag{8.3}$$

Donde:

α es un parámetro de la rutina de optimización llamado tasa de aprendizaje (learning rate) que controla que tan grande es la actualización de los parámetros.

Es importante no usar valores muy grandes de α ya que podríamos no llegar a ningún mínimo local, sin embargo valores muy bajos de α harán que tardemos mucho en llegar al mínimo. Más adelante dentro del tema de machine learning se verán algunas técnicas para ajustar los parámetros de nuestros algoritmos de aprendizaje. Para finalizar este tema me gustaría mostrar la siguiente imagen sacada del curso de Machine Learning en Coursera impartido por Andrew Ng.

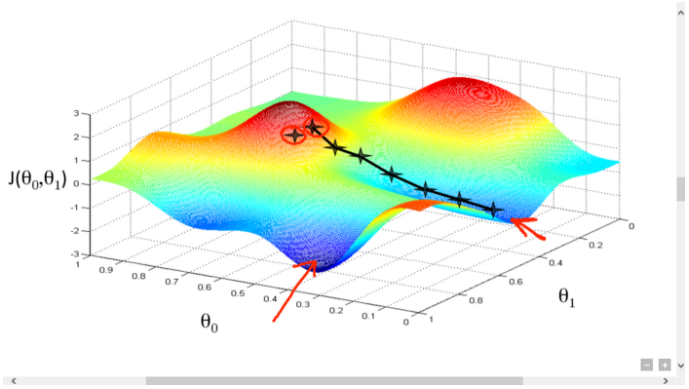


Figura 8.10: Descenso del gradiente con dos parámetros θ_0 y θ_1

8.5 Regresión

8.5.1 Regresión lineal

Descripción del modelo

La regresión lineal es un modelo matemático usado para aproximar la relación de dependencia entre una variable dependiente Y y las variables independientes X. Este modelo puede ser expresado como:

$$\hat{Y} = \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + \beta_0 \tag{8.4}$$

Donde:

$\hat{Y}$ es la predicción que produce el modelo.
$x_1, x_2, ..., x_n$ son las variables independientes.
$\beta_0, \beta_1, ..., \beta_n$ son los parámetros de nuestro modelo

Si consideramos que solo tenemos una variable independiente x_1 nuestro modelo tendría la siguiente forma:

$$\hat{Y} = \beta_1 x_1 + \beta_0 \tag{8.5}$$

Este modelo simple es la ecuación de la recta, es decir $\hat{Y} = \beta_1 x_1 + \beta_0$ es lo mismo que $y = mx + b$. Si se tienen 2 variables independientes se utilizaría un plano para la regresión, siempre se trata de modelar un hiperplano [4], cuando solo se tiene una variable independiente al modelo se le conoce como regresión lineal simple, en caso de tener dos o más variables se le conoce como regresión lineal múltiple.

[4] Un hiperplano es una extensión del concepto del plano, este tiene una dimensión menos que el ambiente en el cual reside, por ejemplo en un espacio tridimensional, el hiperplano correspondiente sería un plano.

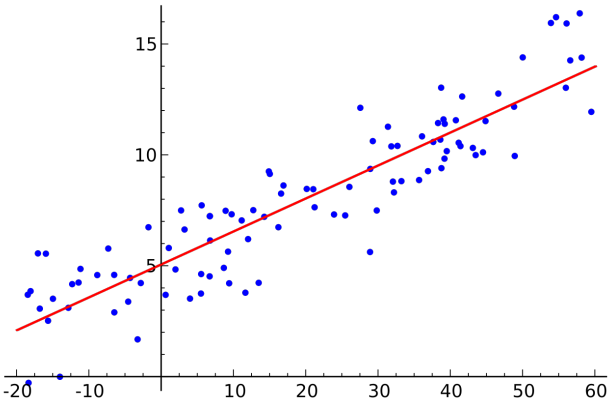


Figura 8.11: Ejemplo de una regresión lineal con una variable dependiente y una variable independiente.

La fórmula 8.22 puede ser vectorizada de la siguiente manera, asumiendo que:

$X_0 = 1$
Los atributos de nuestros datos de entrada se encuentran en una matriz X de dimensión (m x n)
Las etiquetas de nuestros datos de entrada se encuentran en una matriz Y de dimensión (m x 1)
Los parámetros de nuestro modelo se encuentran en una matriz β de dimensión (n x 1)

Donde m es el número de instancias y n el número de atributos o características más uno (se suma uno por X_0)

$$X = \begin{bmatrix} X_0^1 & X_1^1 & \dots & X_n^1 \\ X_0^2 & X_1^2 & \dots & X_n^2 \\ \dots & \dots & \dots & \dots \\ X_0^m & X_1^m & \dots & X_n^m \end{bmatrix} \tag{8.6}$$

$$Y = \begin{bmatrix} Y^1 \\ Y^2 \\ \dots \\ Y^m \end{bmatrix} \tag{8.7}$$

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \dots \\ \beta_n \end{bmatrix} \tag{8.8}$$

$$\hat{Y} = X\beta \tag{8.9}$$

Solución mediante la forma cerrada

Dada la ecuación 8.23 se pueden calcular los parámetros β de la siguiente forma:

$$\beta = (X'X)^{-1}X'Y \tag{8.10}$$

Donde:

X' es la matriz transpuesta de X
$(X'X)^{-1}$ es la matriz inversa de $(X'X)$

Esta es una manera simple de obtener el valor de los parámetros del modelo, sin embargo no escala bien cuando utilizamos grandes cantidades de datos, es por eso que generalmente se suele utilizar el descenso del gradiente para el entrenamiento del modelo.

Solución mediante descenso del gradiente

Como se describió en la ecuación 8.3 mediante el descenso del gradiente podemos actualizar el valor de los parámetros para optimizar una función de coste. Como función de coste vamos a utilizar el Error cuadrático medio (Mean Squared error - MSE), esta función tiene la siguiente forma:

$$J(\beta) = \frac{1}{n}\sum_{i=1}^{n}\left(Y - \hat{Y}\right)^2 \tag{8.11}$$

Vamos a modificar un poco la ecuación con el fin de que al calcular el gradiente resulte una ecuación más sencilla, la ecuación quedaría de la siguiente manera:

$$J(\beta) = \frac{1}{2n}\sum_{i=1}^{n}\left(\hat{Y} - Y\right)^2 \tag{8.12}$$

El gradiente de la ecuación 8.12 es el siguiente: (Su cálculo queda fuera del alcance de este libro)

$$\nabla_\beta J(\beta) = \frac{1}{n}\left(X'(\hat{Y} - Y)\right) \tag{8.13}$$

Por ende la actualización de los parámetros quedaría de la siguiente forma:

$$\theta = \theta - \alpha \cdot \frac{1}{n}\left(X'(\hat{Y} - Y)\right) \tag{8.14}$$

Ridge Regression

Para implementar regresión lineal con regularización L2 basta con modificar la función de coste de la siguiente manera:

$$J(\beta) = \frac{1}{2n}\sum_{i=1}^{n}\left(\hat{Y} - Y\right)^2 + \lambda\sum_{j=1}^{n}{\beta_j}^2 \tag{8.15}$$

Por lo cual, la actualización de los pesos sería la siguiente:

$$\beta = \beta - \alpha \cdot \frac{1}{n}\left(X'(\hat{Y} - Y) + 2\lambda\beta\right) \tag{8.16}$$

Lasso Regression

Para implementar regresión lineal con regularización L1 tenemos que modificar la función de coste de la siguiente manera:

$$J(\beta) = \frac{1}{2n}\sum_{i=1}^{n}\left(\hat{Y} - Y\right)^2 + \lambda\sum_{j=1}^{n}|\beta_j| \tag{8.17}$$

En este caso la actualización de los pesos dependerá del valor de β_j

$$\beta_j = \begin{cases} \beta_j - \alpha \cdot \frac{1}{n}\left(X_j(\hat{Y} - Y) + \lambda\right), & \text{si } \beta_j \geq 0 \\ \beta_j - \alpha \cdot \frac{1}{n}\left(X_j(\hat{Y} - Y) - \lambda\right), & \text{si } \beta_j < 0 \end{cases} \tag{8.18}$$

Ejercicio de programación:

Ejercicio para fortalecer los conocimientos adquiridos:

1. En un lenguaje de programación matemático como Octave, Julia o Matlab resolver un problema de regresión lineal implementando la solución de forma cerrada.
2. En un lenguaje de programación matemático como Octave, Julia o Matlab resolver un problema de regresión lineal implementando la solución mediante el descenso del gradiente.
3. En cualquier lenguaje de programación utilizar alguna librería como scikit-learn para resolver un problema de regresión lineal.

A continuación se presenta un link a las soluciones en caso de que el lector lo requiera:

`https://github.com/amr205/Introduccion-a-la-IA---Libro`

8.5.2 Regresión polinomial

En regresión polinomial se ajusta un modelo no lineal entre las variables independientes X y la variable dependiente y, a pesar de esto todos los parámetros de este modelo son lineales y la regresión polinomial puede considerarse un caso especial de regresión lineal múltiple.

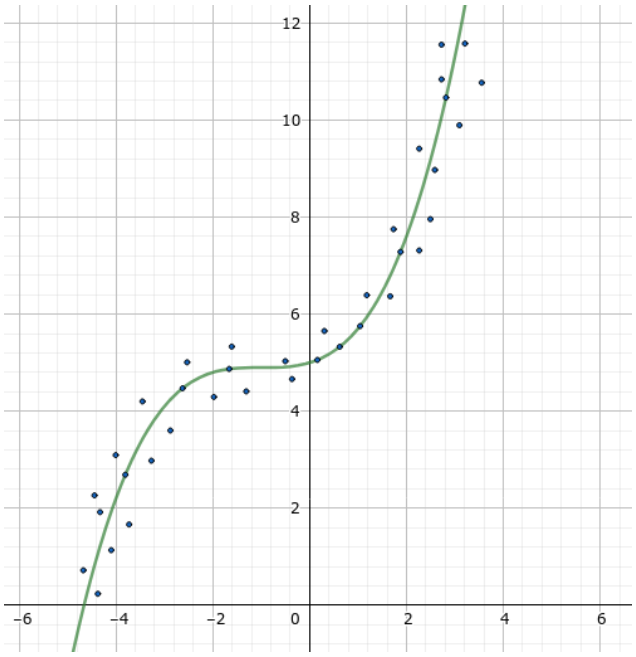


Figura 8.12: Ejemplo de una un modelo de regresión polinomial de grado 3

En la figura 8.1 se puede observar como nuestras variables de entrada X pasan por un proceso de extracción de características para obtener $h(X)$, en el tema anterior $X = h(X)$, en regresión polinomial cada variable x_j^i generaremos características elevando esta variable hasta la k potencia. De tal forma que:

$$X = \begin{bmatrix} X_0^1 & X_1^1 & \dots & X_n^1 \\ X_0^2 & X_1^2 & \dots & X_n^2 \\ \dots & \dots & \dots & \dots \\ X_0^m & X_1^m & \dots & X_n^m \end{bmatrix} \tag{8.19}$$

$$h(X) = \begin{bmatrix} X_0^1 & X_1^1 & \dots & X_n^1 & (X_1^1)^2 & \dots & (X_n^1)^2 & \dots & (X_1^1)^k & \dots & (X_n^1)^k \\ X_0^2 & X_1^2 & \dots & X_n^2 & (X_1^2)^2 & \dots & (X_n^2)^2 & \dots & (X_1^2)^k & \dots & (X_n^2)^k \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ X_0^m & X_1^m & \dots & X_n^m & (X_1^m)^2 & \dots & (X_n^m)^2 & \dots & (X_1^m)^k & \dots & (X_n^m)^k \end{bmatrix} \tag{8.20}$$

Hay que considerar que por la vectorización de nuestro modelo, $X_0 = 1$. Por poner un ejemplo el siguiente modelo lineal:

$$\hat{Y} = \beta_1 x_1 + \beta_2 x_2 + \beta_0 \tag{8.21}$$

Con $k = 3$ sería de la siguiente forma:

$$\hat{Y} = \beta_1 x_1 + \beta_2 x_2 + \beta_3 (x_1)^2 + \beta_4 (x_2)^2 + \beta_5 (x_1)^3 + \beta_6 (x_2)^3 + \beta_0 \tag{8.22}$$

Nuestro modelo quedaría descrito de la siguiente forma:

$$\hat{Y} = h(X)\beta \tag{8.23}$$

Y todos los métodos para obtener los parámetros del modelo descritos en el tema de regresión lineal pueden ser aplicados reemplazando X por $h(X)$.

Nota para regresión lineal múltiple

Mediante la generación de características derivadas de nuestras variables independientes es posible modelar diferentes patrones presentes en los datos, un ejemplo de ello es modelar estacionalidad, si una variable de entrada x presenta este patrón la estacionalidad puede modelarse de la siguiente manera:

$w_1 * sin(2\pi x/periodo) + w_2 * cos(2\pi x/periodo)$

Overfitting en regresión polinomial

Mientras mayor sea el valor de k, más grande es el riesgo de que nuestro modelo sufra del problema de overfitting, más adelante se explorarán técnicas para escoger hiperparámetros, de momento una solución viable es hacer uso de regularización L2 si se presenta este problema.

Ejercicio de programación:

Ejercicio para fortalecer los conocimientos adquiridos:

1. En un lenguaje de programación matemático como Octave, Julia o Matlab resolver un problema de regresión polinomial.
2. En cualquier lenguaje de programación utilizar alguna librería como scikit-learn para resolver un problema de regresión polinomial.

A continuación se presenta un link a las soluciones en caso de que el lector lo requiera:

`https://github.com/amr205/Introduccion-a-la-IA---Libro`

8.5.3 Regresión mediante K-Nearest Neighbors

El algoritmo de K-Nearest Neighbors (K-Vecinos más cercanos) es una técnica no paramétrica [5] que puede ser utilizada para resolver tareas de regresión y clasificación [1], en este tema se explorará su uso en el área de regresión.

Funcionamiento del algoritmo de K-NN

Tendremos los siguientes elementos:

- Un set de entrenamiento compuesto de las características X y el vector columna Y que contiene la variable dependiente que queremos predecir asociado a cada instancia en X.
- Una nueva instancia b a clasificar.
- El hiperparámetro k que determina el número de vecinos a seleccionar.

El algoritmo es el siguiente:

1. Calcular la distancia entre la nueva instancia b y cada una de las instancias en X.
2. Seleccionar los k elementos presentes en X más cercanos a b.
3. Devolver un resultado basado en el valor presente en Y de los k elementos más cercanos.

Es un algoritmo bastante simple, vamos a explorar cada uno de estos pasos de manera más detallada para que después de leer este tema el lector sea capaz de realizar la implementación del algoritmo.

1. Calcular la distancia entre b y las demás instancias en X

Existen diferentes funciones para calcular la distancia entre vectores con valores continuos, algunas de las más comunes son las siguientes:

Distancia euclidiana o euclídea

$$d(x_i, y_i) = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2} \tag{8.24}$$

Distancia Manhattan

$$d(x_i, y_i) = \sum_{i=1}^{k} |x_i - y_i| \tag{8.25}$$

Distancia Minkowski

$$d(x_i, y_i) = \left(\sum_{i=1}^{k} (|x_i - y_i|)^q \right)^{1/q} \tag{8.26}$$

Para vectores con datos categóricos se tiene que usar la distancia Hamming, la distancia Hamming entre dos vectores es igual al número de posiciones donde los elementos correspondientes en los vectores son diferentes.

Distancia Hamming

$$d(x_i, y_i) = \sum_{i=1}^{k} h(x_i, y_i) \tag{8.27}$$

[5] La regresión no paramétrica comprende un conjunto de técnicas para estimar una curva de regresión sin realizar fuertes suposiciones acerca de la forma de la señal que pretendemos extaer de los datos

$$h(x_i, y_i) = \begin{cases} 0, & \text{si } x_i = y_i \\ 1, & \text{si } x_i \neq y_i \end{cases} \tag{8.28}$$

2. Seleccionar los k elementos presentes en X más cercanos a b.

Para esto se puede hacer uso de una cola de prioridad de k elementos, añadiendo cada elemento al calcular la distancia. Otra opción es primero calcular todas las distancias y posteriormente ordenarla para seleccionar los primeros k elementos.

3. Devolver un resultado basado en el valor presente en Y de los k elementos más cercanos.

Hay diferentes opciones para calcular el valor que se va a devolver:

- **Promedio:** Simplemente se regresa el promedio de los valores en Y de los k elementos más cercanos.

  $$\hat{Y}_b = \frac{1}{k} \sum_{i=1}^{k} Y_{NNi} \tag{8.29}$$

  Donde Y_{NNi} corresponde al valor de Y del vecino más cercano i.

- **Vecinos más cercanos con distancia ponderada (weighted knn):** El valor devuelto se calcula tomando en cuenta la distancia los vecinos más cercanos a b, mientras más cercano sea, su valor en Y tendrá mayor contribución al valor devuelto.
  La fórmula general para calcular el valor sería la siguiente:

  $$\hat{Y}_b = \frac{\sum_{i=1}^{k} C_{bNNi} * Y_{NNi}}{\sum_{i=1}^{k} C_{bNNi}} \tag{8.30}$$

  Donde Y_{NNi} corresponde al valor de Y del vecino más cercano i.
  Donde C_{bNNi} corresponde al valor de la contribución que el vecino más cercano i aportará al calculo del valor que se quiere predecir de b.

  Para calcular el valor de C_{bNNi} se utilizan diversas funciones (también llamadas kernels) que toman en cuenta la distancia entre el vecino más cercano i y la instancia b para determinar el nivel de contribución. Algunas de estas funciones son las siguientes (λ es un hiper-parámetro utilizado para determinar que tan rápido decae la contribución respecto a la distancia):

  Distancia ponderada simple:

  $$C_{bNNi} = \frac{1}{d(X_b, X_{NNi})^2} \tag{8.31}$$

  Kernel gaussiano:

  $$C_{bNNi} = exp(-d(X_b, X_{NNi})^2/\lambda) \tag{8.32}$$

  Kernel uniforme:

  $$C_{bNNi} = \begin{cases} 0, & \text{si } |d(X_b, X_{NNi})| > \lambda \\ 1/2, & \text{si } |d(X_b, X_{NNi})| \leq \lambda \end{cases} \tag{8.33}$$

Kernel triangular:

$$C_{bNNi} = \begin{cases} 0, & \text{si } |d(X_b, X_{NNi})| > \lambda \\ (\lambda - |d(X_b, X_{NNi})|), & \text{si } |d(X_b, X_{NNi})| \leq \lambda \end{cases} \tag{8.34}$$

Kernel Epanechnikov:

$$C_{bNNi} = \begin{cases} 0, & \text{si } |d(X_b, X_{NNi})| > \lambda \\ \frac{3}{4}(\lambda^2 - d(X_b, X_{NNi})^2), & \text{si } |d(X_b, X_{NNi})| \leq \lambda \end{cases} \tag{8.35}$$

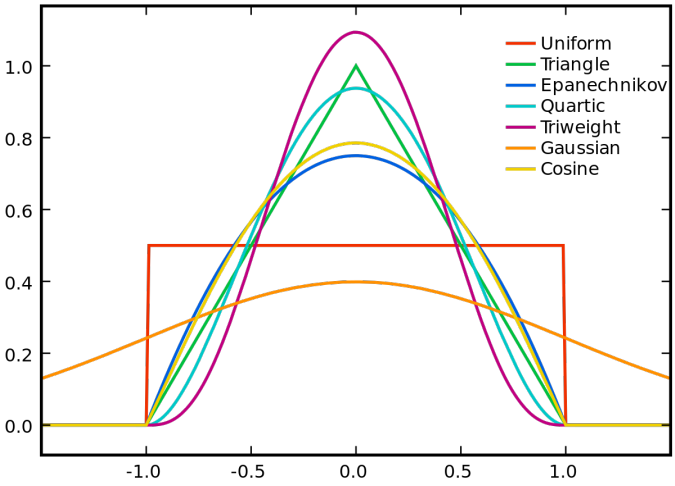


Figura 8.13: Diferentes kernels con $\lambda = 1$ por Brian Amberg licencia CC BY-SA 3.0

Ejemplo de solución de un problema

Se busca predecir el precio de una casa con 4 recámaras, 2 baños y 5 años de antigüedad, se disponen de los siguientes datos:

recámaras	baños	antigüedad	precio
4	1	2	224,000
2	2	1	113,000
2	1	4	144,000
3	2	5	212,000
1	1	3	92,000
5	1	5	260,000
5	2	4	300,000
3	4	2	175,000
3	2	6	224,000
2	2	8	194,000
3	4	2	178,000

Se aplicará el algoritmo de KNN con $k = 5$.
El primer paso es calcular la distancia, para este ejemplo se usará la distancia euclidiana:

recámaras	baños	antigüedad	precio	distancia
4	1	2	224,000	3.1623
2	2	1	113,000	4.4721
2	1	4	144,000	2.4495
3	2	5	212,000	1.0000
1	1	3	92,000	3.7417
5	1	5	260,000	1.4142
5	2	4	300,000	1.4142
3	4	2	175,000	3.7417
3	2	6	224,000	1.4142
2	2	8	194,000	3.6056
3	4	2	178,000	3.7417

Debido a que $k = 2$ se seleccionarán los dos vecinos más cercanos, en este caso se seleccionaron los vecinos 4 y 6.

recámaras	baños	antigüedad	precio	distancia
3	2	5	212,000	1.0000
5	1	5	260,000	1.4142

En este caso se devolverá el promedio de los precios, dando como resultado: 236,000.

Ejercicio de programación:

1. En un lenguaje de programación matemático como Octave, Julia o Matlab resolver un problema de regresión mediante knn.
2. En cualquier lenguaje de programación utilizar alguna librería como scikit-learn para resolver un problema de regresión mediante knn.

A continuación se presenta un link a las soluciones en caso de que el lector lo requiera:

`https://github.com/amr205/Introduccion-a-la-IA---Libro`

8.5.4 Kernel Regression

Kernel Regression es otra técnica para regresión no parámetrica muy similar a vecinos más cercanos con distancia ponderada, la principal diferencia es que no se eligen un k número de vecinos sino todos aquellos en los cuales $|d(v,b)| \leq \lambda$, donde b es la instancia que queremos predecir, v es una instancia cualquiera y λ es un hiperparámetro que limita la distancia máxima a considerar para el algoritmo.

Funcionamiento del algoritmo Kernel Regression

Tendremos los siguientes elementos:

- Un set de entrenamiento compuesto de las características X y el vector columna Y que contiene la variable dependiente que queremos predecir asociado a cada instancia en X.
- Una nueva instancia b a clasificar.
- El hiperparámetro λ que determina la distancia máxima que debe tener un vecino cualquiera v respecto a b para ser considerado.

El algoritmo es el siguiente:

1. Calcular la distancia entre la nueva instancia b y cada una de las instancias en X.
2. Seleccionar los elementos V presentes en X dado que $|d(v,b)| \leq \lambda$.
3. Devolver un resultado basado en el valor presente en Y de los elementos presentes en V.

1. Calcular la distancia entre la nueva instancia b y cada una de las instancias en X.

Se pueden utilizar cualquiera de las funciones para calcular distancias descritas en el tema anterior (8.5.3).

2 y 3. Seleccionar los elementos V y devolver un resultado basado en su valor presente en Y

Se podría aplicar la siguiente fórmula:

$$\hat{Y}_b = \frac{\sum_{i=1}^{n} C_{bNNi} * Y_{NNi}}{\sum_{i=1}^{n} C_{bNNi}} \tag{8.36}$$

Donde Y_{NNi} corresponde al valor de Y de la instancia i.
Donde C_{bNNi} corresponde al valor de la contribución que la instancia i aportará al calculo del valor que se quiere predecir de b.
Donde n corresponde al número de instancias en X.

Esto puede hacerse de esta manera si los kernels devuelven 0 cuando $|d(X_i,b)| > \lambda$. Para aquellos kernels que no lo hagan puede agregarse una condición (en este libro los kernels que no tienen la condición ya implementada son el kernel gaussiano y distancia ponderada simple).

Elección de kernel y λ

Más adelante en este libro se explorarán técnicas para la selección de hiperparámetros, por ahora considero útil que el lector vea visualmente los efectos del uso de diferentes kernels con diferentes valores de λ.

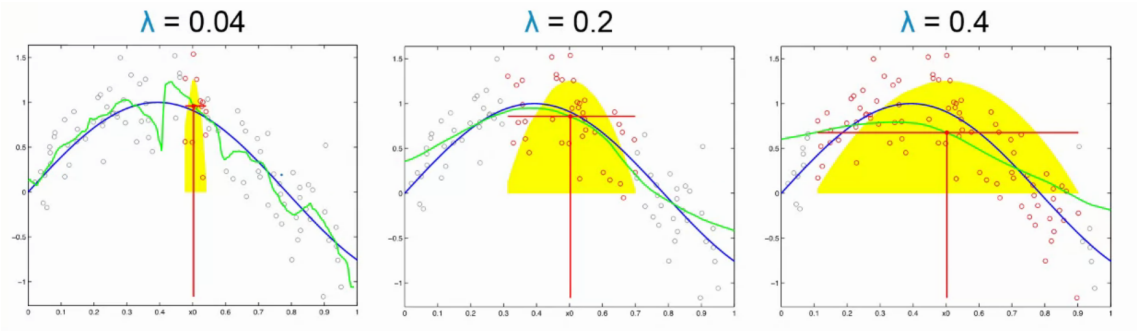


Figura 8.14: Kernel regression con kernel Epanechnikov y diferentes valores de lambda, azul-señal, verde-predicción, imagen tomada de Fox, Emily y Guestrin, Carlos (2015). Machine Learning: Regression [MOOC]. Coursera. `https://www.coursera.org/learn/ml-regression/`

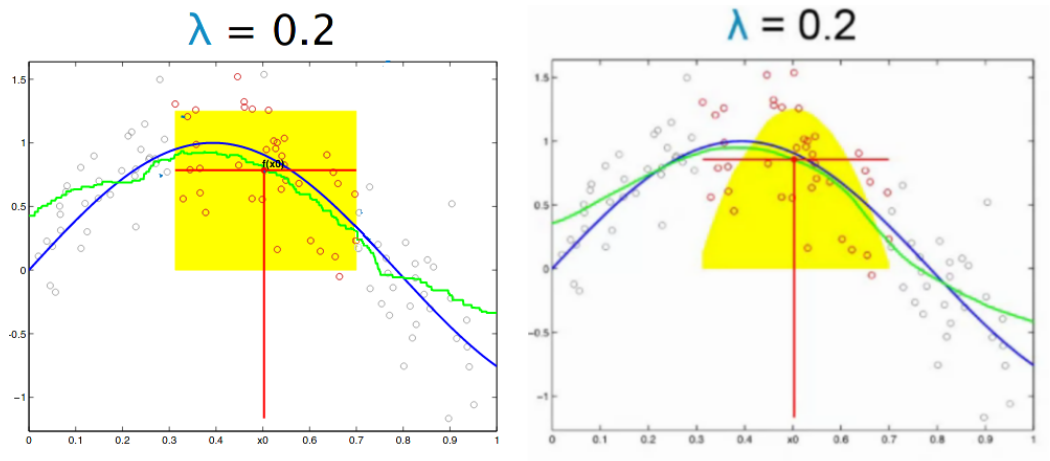


Figura 8.15: Kernel regression con diferentes kernels, izquierda-uniforme, derecha-Epanechnikov, imagen tomada de Fox, Emily y Guestrin, Carlos (2015). Machine Learning: Regression [MOOC]. Coursera. `https://www.coursera.org/learn/ml-regression/`

Ejercicio de programación:

En un lenguaje de programación matemático como Octave, Julia o Matlab resolver un problema de regresión mediante kernel regression.

A continuación se presenta un link a las soluciones en caso de que el lector lo requiera:

`https://github.com/amr205/Introduccion-a-la-IA---Libro`

Bibliography

Articles

[1] N. S. Altman. “An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression”. En: *The American Statistician* 46.3 (1992), páginas 175-185. DOI: 10.1080/00031305.1992.10475879. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00031305.1992.10475879. URL: https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879 (véase página 104).

[4] Tomáš Cádrik y Marian Mach. “Usage of ZCS Evolutionary Classifier System as a Rule Maker for Cleaning Robot Task”. En: (2015). Editado por Peter Sinčák y col., páginas 113-119 (véanse páginas 55, 56).

[5] Murray Campbell, A.Joseph Hoane y Feng-hsiung Hsu. “Deep Blue”. En: *Artificial Intelligence* 134.1 (2002), páginas 57-83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: http://www.sciencedirect.com/science/article/pii/S0004370201001291 (véase página 22).

[7] Marta Garnelo y Murray Shanahan. “Reconciling deep learning with symbolic artificial intelligence: representing objects and relations”. En: *Current Opinion in Behavioral Sciences* 29 (2019). SI: 29: Artificial Intelligence (2019), páginas 17-23. ISSN: 2352-1546. DOI: https://doi.org/10.1016/j.cobeha.2018.12.010. URL: http://www.sciencedirect.com/science/article/pii/S2352154618301943 (véase página 61).

[11] Khali Jebari. “Parent Selection Operators for Genetic Algorithms”. En: *International Journal of Engineering Research and Technology* 12 (nov. de 2013) (véanse páginas 34, 37).

[12] Andreas Kaplan y Michael Haenlein. “Siri, Siri, in my hand: Who’s the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence”. En: *Business Horizons* 62.1 (2019), páginas 15-25. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2018.08.004. URL: http://www.sciencedirect.com/science/article/pii/S0007681318301393 (véase página 17).

[13] Padmavathi Kora y Priyanka Yadlapalli. “Crossover Operators in Genetic Algorithms: A Review”. En: *International Journal of Computer Applications* 162 (mar. de 2017), páginas 34-36. DOI: 10.5120/ijca2017913370 (véase página 37).

[16] Philip Leith. "The rise and fall of the legal expert system Previously published in Leith P., 'The rise and fall of the legal expert system', in European Journal of Law and Technology, Vol 1, Issue 1, 2010.View all notes". En: *International Review of Law, Computers and Technology* 30 (sep. de 2016), páginas 94-106. DOI: 10.1080/13600869.2016.1232465 (véase página 78).

[17] Sean Luke y Lee Spector. "A comparison of crossover and mutation in genetic programming". En: *Genetic Programming* 97 (1997), páginas 240-248 (véase página 47).

[19] J. Mccarthy y col. "A Proposal for the Dartmouth Summer Research Project on Arti cial Intelligence". En: *AI Magazine* 27 (dic. de 2006) (véase página 14).

[22] Carlos Muñoz Gutiérrez. "Introducción a la lógica". En: *Recuperado de http://pendientedemigracion. ucm. es/info/pslogica/cdn. pdf* (2013) (véanse páginas 64, 65).

[24] Albert Orriols-Puig y Ester Bernadó-Mansilla. "A further look at UCS classifier system". En: (2006) (véase página 58).

[27] Gil Press. "The Brute Force of IBM Deep Blue And Google DeepMind". En: *Forbes* (feb. de 2018) (véase página 22).

[28] Shweta Rani, Bharti Suri y Rinkaj Goyal. "On the effectiveness of using elitist genetic algorithm in mutation testing". En: *Symmetry* 11.9 (2019), página 1145 (véase página 40).

[29] Ramprasaath Rs y col. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization". En: *International Journal of Computer Vision* 128 (oct. de 2019). DOI: 10.1007/s11263-019-01228-7 (véase página 18).

[30] Sebastian Ruder. "An overview of gradient descent optimization algorithms". En: (2017). arXiv: 1609.04747 [cs.LG] (véase página 95).

[32] Olivier Sigaud y Stewart Wilson. "Learning classifier systems: A survey". En: *Soft Comput.* 11 (mayo de 2007), páginas 1065-1078. DOI: 10.1007/s00500-007-0164-0 (véase página 51).

[34] Andrew Sloss y Steven Gustafson. "2019 Evolutionary Algorithms Review". En: (jun. de 2019) (véase página 59).

[35] S. F. SMITH. "A Learning system based on genetic adaptive algorithms". En: *Ph. D. Thesis, Univ. of Pittsburgh* (1980). URL: https://ci.nii.ac.jp/naid/10010118042/en/ (véase página 51).

[36] William M. Spears y Vic Anand. "A study of crossover operators in genetic programming". English (US). En: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (ene. de 1991). Editado por Zbigniew W. Ras y Maria Zemankova. 6th International Symposium on Methodologies for Intelligent Systems, ISMIS 1991 ; Conference date: 16-10-1991 Through 19-10-1991, páginas 409-418. DOI: 10.1007/3-540-54563-8_104 (véase página 37).

[37] Felix Streichert. "Introduction to evolutionary algorithms". En: (2002) (véase página 29).

[39] A. M. TURING. "I.—COMPUTING MACHINERY AND INTELLIGENCE". En: *Mind* LIX.236 (oct. de 1950), páginas 433-460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433 (véase página 13).

[40] Tzung-Pei Hong y Hong-Shung Wang. "A dynamic mutation genetic algorithm". En: 3 (1996), 2000-2005 vol.3 (véase página 38).

[41] Ryan Urbanowicz y Jason Moore. "Learning Classifier Systems: A Complete Introduction, Review, and Roadmap". En: *Journal of Artificial Evolution and Applications* 2009 (sep. de 2009). DOI: 10.1155/2009/736398 (véanse páginas 52, 59).

[42] Stewart Wilson. "ZCS: A zeroth level classifier system". En: *Evolutionary Computation* 2 (feb. de 1970). DOI: 10.1162/evco.1994.2.1.1 (véanse páginas 55, 56).

Books

[3] A. Burkov. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019. ISBN: 9781999579517. URL: https://books.google.com.mx/books?id=0jbxwQEACAAJ (véanse páginas 86, 89, 90).

[6] Mariusz Flasiński. *Symbolic Artificial Intelligence*. Springer, 2016, páginas 15-22 (véanse páginas 61, 62).

[8] Alejandro Guerra Hernández. *Representación del Conocimiento*. 2018 (véanse páginas 69-71).

[9] John H. Holland y Judith S. Reitman. *COGNITIVE SYSTEMS BASED ON ADAPTIVE ALGORITHMS11Research reported in this paper was supported in part by the National Science Foundation under grant DCR 71-01997 and by the Horace H. Rackham School of Graduate Studies under grant 387156.* Editado por D.A. WATERMAN y FREDERICK HAYES-ROTH. Academic Press, 1978, páginas 313-329. ISBN: 978-0-12-737550-2. DOI: https://doi.org/10.1016/B978-0-12-737550-2.50020-8. URL: http://www.sciencedirect.com/science/article/pii/B9780127375502500208 (véase página 51).

[10] F.H. Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton paperbacks. Princeton University Press, 2004. ISBN: 9780691118185. URL: https://books.google.com.sb/books?id=t71fPwAACAAJ (véase página 22).

[14] J.R. Koza, J.R. Koza y J.P. Rice. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book. Bradford, 1992. ISBN: 9780262111706. URL: https://books.google.com.mx/books?id=Bhtxo60BV0EC (véanse páginas 42, 44, 45).

[15] Pat Langley. *Elements of machine learning*. eng. San Francisco (Calif.) : Morgan Kaufmann, 1996. ISBN: 1558603018. URL: http://lib.ugent.be/catalog/rug01:000857792 (véase página 52).

[18] R. Marin y P. Jose. *Inteligencia artificial. Técnicas, métodos y aplicaciones*. McGraw-Hill Interamericana de España S.L., 2008. ISBN: 9788448156183. URL: https://books.google.com.mx/books?id=cB8PPwAACAAJ (véase página 18).

[20] Pamela McCorduck y Cli Cfe. *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. CRC Press, 2004, página 124 (véase página 62).

[21] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. ISBN: 9780071154673. URL: https://books.google.com.mx/books?id=EoYBngEACAAJ (véase página 85).

[23] Marisa Navarro. *Curso de programación lógica*. Facultad de Informática de San Sebastián, 2008 (véanse páginas 72, 78).

[25] Riccardo Poli, William Langdon y Nicholas Mcphee. *A Field Guide to Genetic Programming*. Ene. de 2008. ISBN: 978-1-4092-0073-4 (véase página 48).

[26] David Poole, Alan Mackworth y Randy Goebel. *Computational Intelligence: A Logical Approach*. Ene. de 1998. ISBN: 978-0-19-510270-3 (véase página 17).

[31] S.J. Russell, P. Norvig y J.M.C. Rodríguez. *Inteligencia artificial: un enfoque moderno*. Colección de Inteligencia Artificial de Prentice Hall. Pearson Educación, 2004. ISBN: 9788420540030 (véanse páginas 70-74, 76).

[33] Jed Simson. *Open-Source Linear Genetic Programming*. Faculty of Computing y Mathematical SciencesUniversity of Waikato, Waikato, New Zealand, 2017 (véanse páginas 43, 46).

[38] Mehmet Tolun, Seda Sahin y Kasim Oztoprak. *Expert Systems*. Dic. de 2016. DOI: 10.1002/0471238961.0524160518011305.a01.pub2 (véase página 78).

Índice alfabético