

Modelos de Inteligencia Artificial

*Conforme a contenidos del «Curso de Especialización
en Inteligencia Artificial y Big Data»*



**Modelos de
Inteligencia_Artificial**

Universidad de Castilla-La Mancha

Escuela Superior de Informática
Ciudad Real

Capítulo 2

Utilización de los modelos de la Inteligencia Artificial

Santiago Sánchez Sobrino

Los problemas de IA pueden abordarse utilizando diferentes enfoques dependiendo de la naturaleza del problema al que nos enfrentamos. Estos enfoques se basan en el concepto de agente racional. Como se mencionó en el Capítulo 1, un agente racional es aquel que hace lo correcto dado el objetivo que se le proporciona al agente, es decir, maximiza el objetivo (meta) con la información de la que dispone. Esta definición se completa enumerando los cuatro elementos que debe tener un agente racional:

- La medida de rendimiento que define lo que es correcto (objetivo).
- El conocimiento previo que el agente posee del mundo (entorno).
- Las acciones que el agente puede realizar (actuadores).
- Lo que el agente ha percibido hasta el momento (sensores).

Por ejemplo, un tipo de agente racional que solucione un problema de robot aspirador podría tener los siguientes elementos:

- **Objetivo:** maximizar superficie limpia.
- **Entorno:** superficie, personas, animales, muebles.
- **Actuadores:** dirección, acelerador.
- **Sensores:** cámaras, infrarrojos, sistema de posicionamiento, acelerómetro, batería.

El comportamiento del agente anterior viene definido por su **función agente**, la cual es una descripción matemática abstracta que relaciona las percepciones recibidas del entorno con acciones concretas. La implementación de dicha función sobre algún sistema se denominaría **programa agente**.

Hay muchas formas de dotar de inteligencia a un agente racional. Las técnicas más utilizadas son el aprendizaje automático, los sistemas basados en el conocimiento, y los basados en reglas, entre otros. En este capítulo vamos a introducir en detalle los agentes racionales, los tipos que existen para construir sistemas de resolución de problemas y algunos de los que existen orientados a modelar sistemas de IA mediante reglas.

2.1. Entornos de trabajo

La gran cantidad de problemas de IA que pueden surgir es bastante amplia, sin embargo, podemos describirlos dependiendo de cómo sean los entornos de trabajo donde se ejecutarán dichos agentes, e identificar las dimensiones que sean comunes. Estas dimensiones determinan el diseño apropiado del agente y la aplicabilidad de cada una de las principales familias de técnicas para su implementación:

- **Completamente observable o parcialmente observable:** si los sensores de un agente proporcionan acceso al estado completo del entorno en cada momento, decimos que el entorno es completamente observable. Un entorno es completamente observable si los sensores detectan todos los aspectos que son relevantes para la elección de las acciones. Si el agente carece de sensores, decimos que el entorno en el que se encuentra no es observable.
- **Agente individual o multi-agente:** si en el entorno existe más de un agente, entonces decimos que nos encontramos en un entorno multi-agente; en caso contrario, sería un entorno con un agente individual. Podemos detectar si los otros elementos del entorno son agentes realmente o no, en función de si a través de su comportamiento trata de maximizar la métrica de rendimiento dada, cuyo valor depende de otro agente. Así, un juego de ajedrez será un entorno multi-agente (2 agentes), mientras que un entorno donde haya que resolver un *Sudoku* únicamente necesitará un agente.
- **Determinista o no determinista:** si el siguiente estado del entorno está completamente determinado por el estado actual y la acción ejecutada por el agente (o agentes), entonces decimos que el entorno es determinista; en caso contrario, es no determinista.
- **Episódico o secuencial:** en un entorno episódico, la experiencia del agente se divide en episodios atómicos. En cada episodio, el agente percibe algo y realiza una única acción. Lo más importante es que el siguiente episodio no dependerá de las acciones realizadas en los episodios anteriores. En cambio, en los entornos secuenciales, la decisión actual podría afectar a todas las decisiones futuras. Los entornos episódicos son mucho más sencillos que los secuenciales porque el agente no necesita pensar en el futuro.

- **Estático o dinámico:** si el entorno puede cambiar mientras un agente está decidiendo, entonces decimos que el entorno es dinámico para ese agente; de lo contrario, es estático. Los entornos estáticos son fáciles de manejar porque el agente no necesita seguir observando el entorno mientras decide una acción, ni tiene que preocuparse por el paso del tiempo. Los entornos dinámicos, en cambio, no dejan de cuestionar al agente lo que quiere hacer; si aún no lo ha decidido, cuenta como si no hubiera decidido hacer nada.
- **Discreto o continuo:** la distinción entre discreto y continuo se aplica al estado del entorno, a la forma de manejar el tiempo y a las percepciones y acciones del agente. Por ejemplo, el entorno del ajedrez tendría un número finito de estados diferentes, así como un conjunto discreto de percepciones y acciones.

Conociendo las dimensiones del entorno es posible crear entornos simulados que sirvan para experimentar con el agente y construir así su función agente con el objetivo de llegar a implementar programas agente concretos.

A menudo, los experimentos no se llevan a cabo para un único entorno, sino para muchos entornos extraídos de una clase de entorno. Por ejemplo, para evaluar a un conductor de taxi en un tráfico simulado, querríamos realizar muchas simulaciones con diferentes condiciones de tráfico, iluminación y clima, por lo que nos interesa el rendimiento medio del agente en la clase de entorno.

2.2. Sistemas de resolución de problemas

Los problemas a resolver mediante IA pueden presentarse en diferentes entornos de trabajo, los cuales pueden ser abordados mediante agentes racionales. En esta sección vamos a introducir los tipos de problemas que nos podemos encontrar en los entornos y los tipos de programas agente que podemos construir.

2.2.1. Tipos de problemas en entornos de trabajo

Los agentes pueden enfrentarse a diversos tipos de problemas dependiendo del entorno de trabajo. En esta sección se realiza una clasificación de los distintos tipos de problemas.

Búsqueda de estado frente a la búsqueda de secuencia de acciones

En los problemas de búsqueda de estado, sólo se conocen las propiedades que debe tener un estado objetivo, pero ni siquiera sabemos si existe dicho estado. Sólo necesitamos encontrar un estado que satisfaga ciertas restricciones; no importa qué secuencia de acciones nos lleve hasta allí. La definición de lo que es óptimo viene dada por encontrar el *mejor estado posible*.

En los problemas de búsqueda de secuencia de acciones, se conoce el espacio de estados de antemano, por lo que sabemos qué estados son los objetivos. En este caso, sí que tenemos que encontrar la secuencia de acciones que nos llevan a esos estados objetivos. Lo óptimo en este caso significa encontrar la *ruta menos costosa*.

Problemas en línea frente a problemas fuera de línea

En un problema en línea, el agente no conoce el espacio de estados y tiene que construir un modelo del mismo mientras actúa. Por otra parte, en un problema fuera de línea, las percepciones no importan en absoluto, por lo que un agente podría descubrir toda la secuencia de acciones antes de hacer nada.

Problemas de ausencia de sensores

Se trata de un tipo de problema que se da en entornos deterministas y no observables. En este caso, el agente no sabe donde se encuentra, por lo que abordan mediante conjuntos de estados en los que el agente podría encontrarse.

Problemas de contingencia

En este tipo de problemas, el agente no sabe qué efecto tendrán las acciones que realice. Esto puede deberse a que el entorno sea parcialmente observable o no determinista (otros agentes afectan al entorno). Una forma de abordar este tipo de problemas es mediante la construcción de la solución a través de una lista de acciones a ejecutar condicionalmente.

2.2.2. Programas agente

Hasta ahora se han descrito los agentes racionales a partir de su comportamiento, es decir, la acción que se lleva a cabo después de una secuencia dada de percepciones del entorno. Para que esta acción sea llevada a cabo, necesitamos comprender cómo funciona internamente un agente racional, es decir, dada una función agente, cómo podemos construir un programa agente que la implemente. Este programa agente se ejecutará sobre algún dispositivo con sensores físicos y actuadores, lo que se conoce como la arquitectura del agente. Así, un agente vendrá definido por una arquitectura y un programa.

El programa agente percibirá como entrada lo que está sucediendo en el momento actual en el entorno y actuará en consecuencia. Si la acción que el agente tiene que realizar dependiera de percepciones pasadas, entonces el agente necesitaría guardar ese histórico de percepciones en memoria para recuperarlo y poder revisarlo.

El programa agente más básico que podemos diseñar se logra a partir de la construcción de una tabla que relacione todas las secuencias de percepciones que el agente puede interpretar, con su acción correspondiente. Podemos denominar a este agente como *agente dirigido por tablas*. Este enfoque para construir programas agente es funcional, aunque no realista; la tabla que habría que construir para un problema tan simple como el ajedrez (donde todas las reglas y resultados de las acciones están bien definidos) sería enorme, contando al menos con 10^{150} filas. Dado el enorme tamaño de estas tablas, nos encontraríamos con que no existe ningún dispositivo físico capaz de almacenarlas, el diseñador del programa agente no tendría tiempo para crear las tablas y ningún tipo de agente racional sería capaz de encontrar las filas correctas a partir de la experiencia acumulada. A pesar de todo, el *agente dirigido por tablas* es teóricamente funcional, asumiendo que la tabla se construya correctamente.

La implementación en Python de un programa agente dirigido por tablas puede verse en el Listado 2.1. El programa agente toma la percepción actual como entrada de los sensores y devuelve una acción a los actuadores. La tabla vendrá dada como un diccionario de pares tupla-acción, siendo las tuplas la secuencia de percepciones a la que el agente reaccionará. El programa agente recordará la secuencia de percepciones, lo que implicará aumentar el tamaño de la tabla con nuevas filas que contemplen las nuevas secuencias de percepciones. En caso de no existir una acción para una secuencia dada, el agente ejecutará alguna acción por defecto definida en la tabla (ver líneas ⑤-6). Como se ha comentado, este agente es teóricamente funcional, siempre y cuando podamos construir una tabla que contemple todas las secuencias de percepciones existentes para el entorno donde se ejecute el agente. Así, un agente de este tipo sería factible para dominios pequeños.

Listado 2.1: Implementación de un programa *agente dirigido por tablas*

```
1 def table_driven_agent_program(table, current_percept, past_percepts=[]):
2     past_percepts.append(current_percept)
3     percept = tuple(past_percepts)
4     action = table.get(percept)
5     if action is None:
6         action = table.get(tuple())
7     return action
```

En el resto de esta sección describimos cuatro tipos básicos de programas agentes que recogen los principios que fundamentan casi todos los sistemas inteligentes. En la última parte, veremos también como cualquiera de estos cuatro agentes puede mejorar su rendimiento mediante el aprendizaje.

Agentes reactivos simples

El tipo de agente más sencillo es el *agente reactivo simple*. Estos agentes deciden qué acción realizar en función de la percepción actual, ignorando las percepciones anteriores. Por ejemplo, el agente de un robot aspirador que solo puede limpiar en dos ubicaciones distintas sería un agente reactivo simple, porque su decisión se

basa sólo en la superficie actual y en si esa superficie contiene suciedad, es decir, no tiene en cuenta las superficies que ya ha limpiado (y que podrían encontrarse sucias de nuevo). En el Listado 2.2 se muestra una posible implementación mínima para este programa agente.

Listado 2.2: Implementación de un programa *agente reactivo simple* para un robot aspirador

```
1 def vacuum_reflex_agent_program(location, status):
2     if status == 'Dirty':
3         return 'vacuum'
4     elif location == 'A':
5         return 'right'
6     elif location == 'B':
7         return 'left'
```

La implementación del programa agente anterior es específico para un entorno de trabajo concreto al del robot aspirador. Un enfoque más general y flexible es construir primero un intérprete de propósito general para las reglas de condición-acción y luego crear conjuntos de reglas para entornos de trabajo específicos. La Figura 2.1 presenta la estructura de este programa general, mostrando cómo las reglas de condición-acción permiten al agente realizar la conexión entre lo que percibe y la acción a realizar. Un programa agente que utilice dicho intérprete procesaría lo percibido y ejecutaría la primera regla del conjunto de reglas disponible que coincida con la percepción analizada. La implementación de esta lógica podría realizarse utilizando secuencias de expresiones condicionales *if-then-else*, circuitos de puertas lógicas Booleanas, o redes neuronales artificiales.

Este tipo de agentes tan simples presentan ciertos problemas en caso de que el agente presente alguna dificultad a la hora de observar el entorno. Por ejemplo, supongamos que el programa agente del robot aspirador (ver Listado 2.2) tiene el sensor de ubicación roto y sólo dispone del sensor de suciedad; este agente sólo tendrá dos percepciones posibles: *sucio* y *limpio*. Puede aspirar en respuesta a *sucio* pero, ¿qué debería hacer en respuesta a *limpio*? Moverse a la izquierda fallará si resulta que empieza en la ubicación A, y moverse a la derecha también fallará si resulta que empieza en la ubicación B, dando lugar a un bucle infinito. Esta problemática suele ser inevitable para este tipo de agentes que operan en entornos parcialmente observables. Una posible solución para romper salir de posibles bucles infinitos es haciendo que el agente realice acciones aleatorias. Por ejemplo, si el agente percibe la superficie como *limpia*, podríamos definir que la mitad de las veces el agente se dirija a la izquierda y la otra mitad a la derecha. En entornos multi-agente este tipo de comportamientos aleatorios puede estar justificado, pero no en entornos de agente individual, donde estos comportamientos no suelen ser racionales.

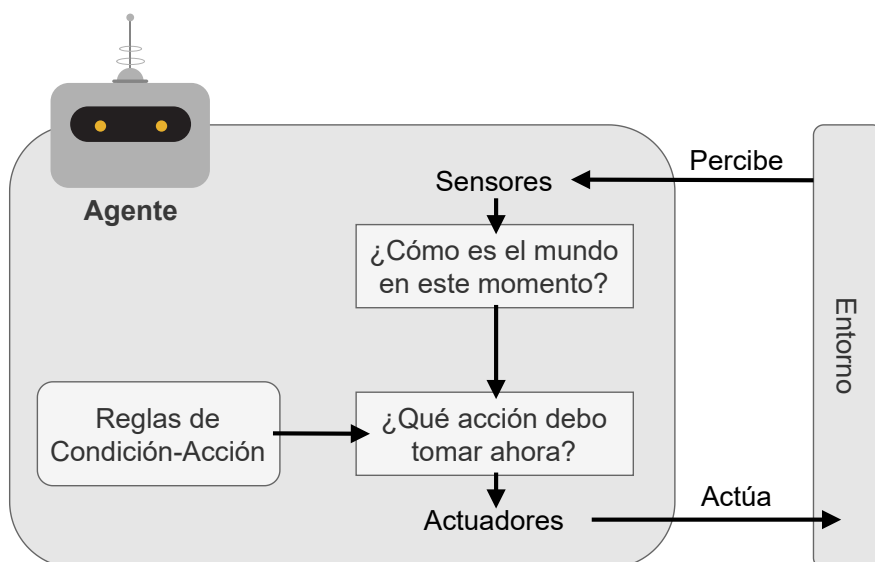


Figura 2.1: Estructura de un agente reactivo simple.

Agentes reactivos basados en modelos

Los agentes reactivos simples no son los más adecuados cuando nos enfrentamos a entornos parcialmente observables. Para enfrentarnos a estos entornos, podemos construir agentes que mantengan un estado interno que dependa del histórico de percepciones pasado y que por lo tanto sirva para reaccionar ante algún posible aspecto no observado del estado actual.

La actualización de la información del estado interno a medida que pasa el tiempo requiere que el programa agente codifique de alguna forma dos tipos de conocimiento. En primer lugar, necesitamos información sobre cómo cambia el mundo a lo largo del tiempo, que puede dividirse aproximadamente en dos partes: los efectos de las acciones del agente y cómo evoluciona el mundo independientemente del agente. Este conocimiento sobre *cómo evoluciona el mundo* se denomina *modelo de transición* del mundo.

En segundo lugar, necesitamos información sobre cómo se refleja el estado del mundo en las percepciones del agente. Este tipo de conocimiento se denomina *modelo de sensor*.

Estos dos modelos anteriores permiten que el agente mantenga un estado interno del mundo que le rodea (hasta un cierto límite dado por los sensores disponibles del agente). Los agentes que usan estos dos modelos se denominan *agentes reactivos basados en modelos*. En la Figura 2.2 se puede ver la estructura de este tipo de agentes, donde se muestra cómo el estado interno antiguo se combina con lo que se está percibiendo actualmente para generar el estado actualizado. De igual forma, en el Listado 2.3 se muestra una posible implementación del programa

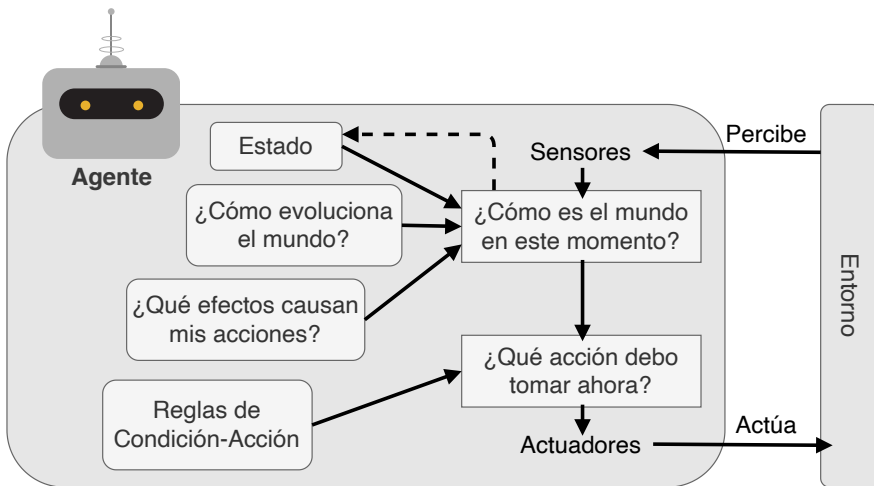


Figura 2.2: Estructura de un agente reactivo basado en modelos.

agente correspondiente; en las líneas ② 1-2 se crea el nuevo estado interno a partir del estado actual (*state*), la descripción de cómo el siguiente estado dependerá del estado actual y la acción ejecutada (*transition_model*), la descripción de cómo el estado del mundo actual se refleja en las percepciones recibidas por el agente (*sensor_model*), el conjunto de reglas de pares condición-acción (*rules*), y la acción más reciente que inicialmente no será ninguna (*action*).

Listado 2.3: Implementación teórica de un programa *agente reactivo basado en modelos*.

```

1 def model_based_reflex_agent_program(percept):
2     state = update_state(state, action, percept,
3                           transition_model, sensor_model)
4     rule = rule_match(state, rules)
5     action = rule.action
6     return action

```

Agentes basados en objetivos

Saber algo sobre el estado actual del entorno no siempre es suficiente para decidir qué hacer. Además de una descripción del estado actual, el agente necesitará algún tipo de información sobre el objetivo que describa situaciones deseables. El programa agente puede combinarla con el modelo (la misma información que se utilizó en el agente reactivo basado en el modelo) para elegir las acciones que permiten alcanzar el objetivo. La Figura 2.3 muestra la estructura del *agente basado en objetivos*.

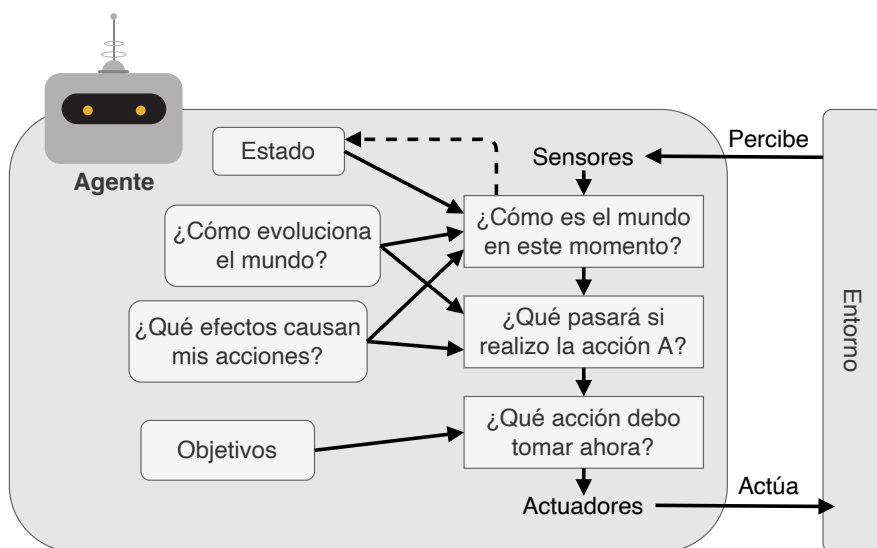


Figura 2.3: Estructura de un agente basado en objetivos.

A diferencia de los agentes reactivos, este tipo de agentes describe las acciones a realizar en base a un objetivo final y la secuencia de acciones que se deben realizar para llegar a él, en lugar de que las acciones vengan relacionadas directamente con las percepciones recibidas. Así, aunque puedan parecer menos eficientes, resultan más flexibles porque el conocimiento que motiva la toma de decisiones se encuentra representado explícitamente y puede ser modificado. En el caso de los agentes reactivos, este conocimiento es específico para un objetivo concreto, y debe ser modificado manualmente en caso de que cambiemos el objetivo.

Los algoritmos de búsqueda (introducidos en el capítulo anterior) y planificación (presentados en este capítulo) emplean agentes de este tipo.

Agentes basados en utilidad

Definir objetivos no es suficiente para producir comportamientos de alta calidad en la mayoría de entornos. Los objetivos solo permiten diferenciar entre dos estados: si se han cumplido, o no. Una medida de rendimiento más general debería permitir la comparación entre diferentes estados del mundo según el grado de completitud del objetivo, para lo que se utiliza el término de *utilidad*. Utilizando esta medida de utilidad, podríamos instruir al agente en que complete su objetivo en base a algún criterio adicional, en el caso de un vehículo autónomo, eligiendo rutas que sean más rápidas, seguras, fiables o baratas.

Los agentes basados en utilidad presentan algunas ventajas sobre aquellos basados en objetivos en dos escenarios distintos: i) cuando existe un conflicto con la definición de los objetivos, la medida de utilidad ayudaría a tomar la mejor decisión (por ejemplo, a la hora de definir un objetivo que contemple al mismo tiempo

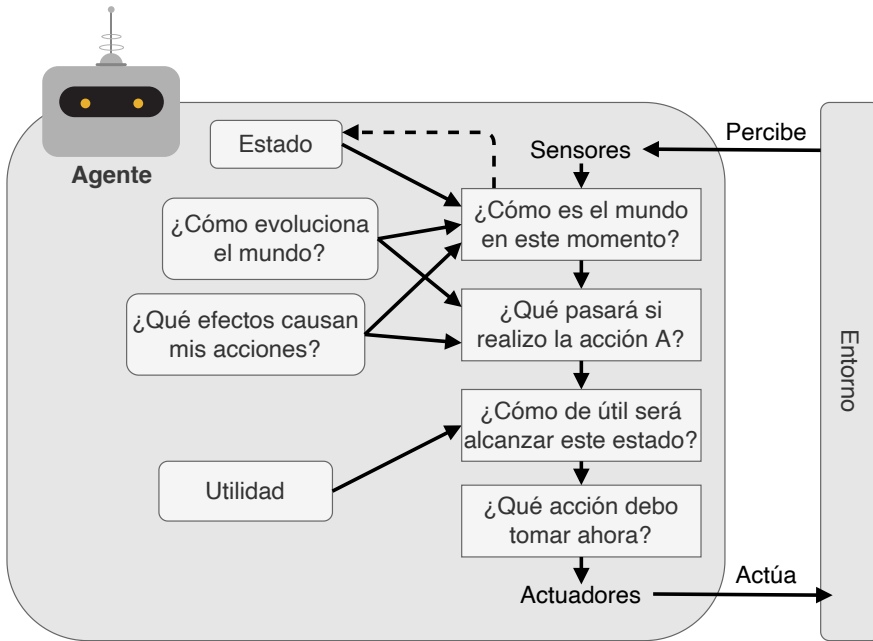


Figura 2.4: Estructura de un agente basado en utilidad.

una alta velocidad y gran seguridad, en el caso del vehículo autónomo), y ii) cuando existen varios objetivos pero ninguno de ellos puede ser alcanzado con certeza, la medida de utilidad proporciona una forma de sopesar la probabilidad de éxito frente a la importancia de los objetivos.

En la Figura 2.4 se muestra la estructura de este tipo de agentes, donde se muestra la utilidad empleada para medir la preferencia entre los estados disponibles. Utilizando esta medida de utilidad, el agente seleccionará aquella acción que sirva para alcanzar la mejor utilidad posible.

Agentes que aprenden

Los tipos de agente que hemos visto hasta ahora pueden ser contruidos como *agentes que aprenden*. De hecho, la mayoría de sistemas de IA actuales cuentan con algún mecanismo de aprendizaje automático, ya sea mediante aprendizaje supervisado, no supervisado, profundo, por refuerzo o por modelos probabilísticos, entre otros. Los agentes que aprenden presentan ciertas ventajas, como permitir al agente operar en entornos inicialmente desconocidos y hacerse más competente de lo que su conocimiento inicial le podría permitir.

Un agente que aprende puede ser dividido en cuatro componentes conceptuales, tal y como se muestra en la Figura 2.5:

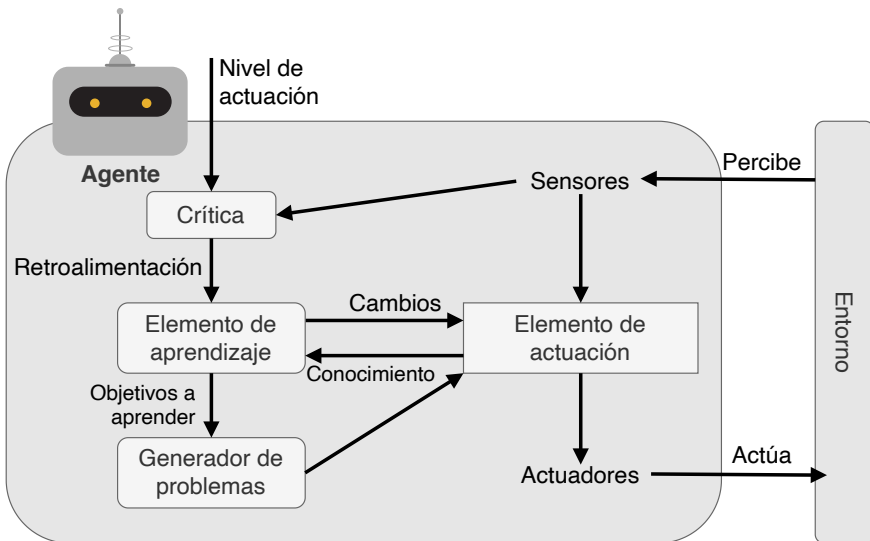


Figura 2.5: Estructura de un agente que aprende; el *elemento de actuación* representa alguno de los programas agente que hemos visto hasta ahora, mientras que el *elemento de aprendizaje* se encarga de modificar el programa para mejorar su desempeño.

- **Elemento de aprendizaje:** se encarga de modificar el programa para mejorar su desempeño.
- **Elemento de actuación:** se encarga de seleccionar las acciones externas a realizar. Se trata de alguno de los programas agente que hemos visto en las secciones anteriores.
- **Crítica:** proporciona retroalimentación al elemento de aprendizaje sobre el rendimiento del agente con respecto del nivel de actuación y determina cómo se debería de modificar el elemento de actuación para que funcione mejor.
- **Generador de problemas:** se encarga de sugerir acciones que conduzcan a experiencias nuevas e instructivas. Si el elemento de actuación pudiera, seguiría haciendo las mejores acciones posibles, pero si el agente está dispuesto a explorar un poco y hacer algunas acciones un poco peores a corto plazo, podría descubrir acciones mucho mejores para el largo plazo. El trabajo del generador de problemas es sugerir estas acciones exploratorias.

Así, los agentes que aprenden pueden verse como un proceso de modificación de los componentes del agente para lograr una mayor concordancia con la información de retroalimentación disponible, mejorando así el rendimiento general del agente.

2.3. Modelos de sistemas de IA

Utilizando el concepto de programa agente, podemos construir sistemas de IA que resuelvan problemas utilizando diferentes enfoques. En esta sección se presentan algunos de ellos.

2.3.1. Planificación automática

Se entiende como *planificación* a la tarea de encontrar una secuencia de acciones para lograr una meta en un entorno discreto, determinista, estático y completamente observable. En un entorno así, podemos diseñar agentes que apliquen un proceso de resolución del problema en cuatro pasos:

1. **Planteamiento del objetivo:** los objetivos permiten al agente organizar su comportamiento al restringir las acciones que debe considerar.
2. **Formulación del problema:** el agente plantea una descripción de los estados y acciones necesarios para alcanzar el objetivo.
3. **Búsqueda de soluciones:** antes de realizar cualquier acción en el mundo real, el agente simula secuencias de acciones en su modelo, buscando hasta encontrar una secuencia de acciones que alcance el objetivo.
4. **Ejecución:** el agente ya puede ejecutar las acciones de la solución, de una en una.

Sin embargo, este proceso plantea dos problemas. Por una parte, se requiere una heurística propia para cada nuevo dominio: una función de evaluación heurística para la búsqueda y un código concreto en algún lenguaje de programación para el agente. Por otra parte, requiere representar de manera explícita un espacio de estados exponencialmente grande.

Debido a estas limitaciones, surgió una familia de lenguajes de definición del dominio de planificación o PDDL por sus siglas en inglés, los cuales permiten expresar todas las acciones posibles de manera única y no necesitan conocimiento específico del dominio del problema. La sintaxis de PDDL está basada en *Lisp*, un lenguaje de programación cuya sintaxis está basada en listas. Sin embargo, no es el objetivo de este capítulo detallar dicha sintaxis, por lo que nos abstraeremos de ella y en su lugar se introducirá PDDL mediante una abstracción basada en la lógica de primer orden.

Lógica de primer orden

A diferencia de la lógica proposicional, que solo permite describir hechos que sean verdaderos, falsos o desconocidos, la **lógica de primer orden** permite, adicionalmente, describir objetos del mundo y sus relaciones, ya sea para algunos o todos los objetos.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \implies Q$	$P \iff Q$
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

Tabla 2.1: Tabla de verdad para las 5 conectivas lógicas. En la parte izquierda se definen los posibles valores que tomarán los símbolos P y Q . En la parte derecha se muestra el resultado para cada conectiva utilizando los valores de los símbolos en una fila concreta.

La sintaxis en ambos tipos de lógica es común. En la lógica proposicional, la sintaxis describe las sentencias que se admiten. Las sentencias atómicas están formadas por un único símbolo de proposición, el cual puede ser verdadero o falso. Estos símbolos vienen representados por cualquier palabra que empiece por una letra mayúscula o por una única letra, y pueden incluir o no subíndices, por ejemplo, P , Q , $S_{1,3}$. Las sentencias complejas se construyen a partir de otras sentencias más simples, utilizando paréntesis y operadores llamados conectivas lógicas. Normalmente se utilizan cinco de ellas:

- \neg (*no*): sirve para negar una sentencia. Por ejemplo, $\neg Q$ se denominaría como la negación de Q .
- \wedge (*y*): sirve para construir conjunciones. Por ejemplo, $P \wedge Q$ se denominaría como PyQ .
- \vee (*o*): sirve para construir disyunciones. Por ejemplo, $P \vee Q$ se denominaría como PoQ .
- \implies (*implica*): sirve para construir implicaciones, también llamadas condicionales. Por ejemplo, $(P \wedge Q) \implies R$ se denominaría como que PyQ (premisa o antecedente) implica R (conclusión o consecuente). Las implicaciones son también conocidas como *reglas* o sentencias *if-then*.
- \iff (*si y solo si*): permite establecer relaciones bicondicionales de equivalencia entre dos expresiones. Significa que una es verdadera cuando la otra es verdadera (y al contrario). Por ejemplo, la sentencia $P \iff Q$ se denominaría como bicondicional.

Utilizando esta sintaxis podemos especificar la semántica relacionada definiendo las reglas que determinan la verdad de una sentencia con respecto a un modelo concreto. Por ejemplo, un posible modelo para una base de conocimiento que use los símbolos proposicionales P , Q y R podría ser $m = \{P = \textit{false}, Q = \textit{false}, R = \textit{true}\}$. El listado de reglas que indica todas las posibles combinaciones de verdad para un conjunto finito de símbolos proposicionales se denominan tablas de verdad. En la Tabla 2.1 se muestran todas las reglas de verdad que existen para los operadores anteriores.

$(P \wedge Q) \equiv (Q \wedge P)$	conmutatividad de \wedge
$(P \vee Q) \equiv (Q \vee P)$	conmutatividad de \vee
$((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$	asociatividad de \wedge
$((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$	asociatividad de \vee
$\neg(\neg P) \equiv P$	eliminación de la doble negación
$(P \implies Q) \equiv (\neg Q \implies \neg P)$	contraposición
$(P \implies Q) \equiv (\neg P \vee Q)$	eliminación de la implicación
$(P \iff Q) \equiv ((P \implies Q) \wedge (Q \implies P))$	eliminación de la bicondicional
$\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$	De Morgan
$\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$	De Morgan
$(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$	distribución de \wedge sobre \vee
$(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$	distribución de \vee sobre \wedge

Tabla 2.2: Equivalencias lógicas estándar.

De esta forma podemos definir y evaluar sentencias que sean más o menos complejas. Así, continuando con el ejemplo y los símbolos definidos anteriormente, podríamos aplicar las reglas de verdad para realizar la evaluación: $\neg P \wedge (Q \vee R) = \text{true} \wedge (\text{false} \vee \text{true}) = \text{true} \wedge \text{true} = \text{true}$.

La evaluación anterior puede complicarse en función del número de símbolos y de hechos que existan, por lo que existen varias técnicas para automatizar dicha evaluación, lo que se denomina *inferencia*. Aplicando reglas de inferencias podemos derivar una *prueba*, o lo que es lo mismo, una cadena de conclusiones que nos permita alcanzar el objetivo deseado. Algunas de las reglas de inferencia más conocidas son i) *Modus Ponens*, definido como $P \implies Q, P \vdash Q$ y que significa que en caso de tener sentencias $P \implies Q$ y P , entonces podemos inferir la sentencia Q ; y ii) *simplificación*, definido como $P \wedge Q \vdash P$ y que significa que dada una conjunción, cualquiera de las sentencias que forman parte de la conjunción puede ser inferida. Adicionalmente a estas reglas de inferencia, existe un conjunto de equivalencias lógicas (ver Tabla 2.2) que resulta útil para simplificar las sentencias y facilitar la aplicación de las reglas de inferencia. Un agente podría beneficiarse de esto dados una base de conocimiento que utilice sentencias en esta sintaxis y un motor de inferencia: almacenando sentencias sobre el entorno en su base de conocimiento, usando el motor de inferencias para inferir nuevas sentencias y utilizando estas sentencias para decidir qué acción realizar.

En el lenguaje lógico, los *modelos* son las estructuras formales que constituyen los entornos posibles a considerar. Cada modelo vincula el vocabulario de las sentencias lógicas con elementos del entorno, de manera que se pueda determinar la verdad de cualquier sentencia. Así, los modelos de la lógica proposicional vinculan los símbolos proposicionales con valores de verdad predefinidos. Por tanto, la lógica proposicional no es capaz de adaptarse a entornos de tamaño ilimitado porque carece de la capacidad expresiva necesaria para tratar de forma concisa el tiempo, el espacio y las relaciones entre objetos.

En la lógica de primer orden los *modelos* son más realistas, ya que permiten incluir objetos y sus relaciones. El conjunto de objetos de un modelo se denomina *dominio* del modelo. Los elementos de un modelo se representan mediante símbolos, los cuales pueden ser: i) *constantes*, para definir los objetos; ii) *predicados*, para definir las relaciones; y iii) *funciones*, para definir expresiones que relacionan una entrada con una salida (otro objeto). Por ejemplo, *Juan* y *Pepe* podrían ser símbolos constantes; *Hermano*, *Persona*, *Ingeniero*, *EnLaCabeza* y *Sombrero* podrían ser predicados; y *PiernaDerecha* podría ser una función. Podemos referirnos comúnmente a los símbolos constantes y a las funciones como *términos*, ya que ambos hacen referencia a objetos.

Además de lo anterior, cada modelo incluye una *interpretación* que especifica exactamente a qué objetos, relaciones y funciones se refieren los términos y los predicados, es decir, el conocimiento del mundo real que representan.

A partir de aquí, podemos construir sentencias que utilicen objetos y establezcan relaciones entre ellos mediante el uso de predicados seguidos de una lista de términos, por ejemplo, la sentencia $Hermano(Juan, Pepe)$ significaría (dada una posible interpretación que diga quienes son Juan y Pepe) que Juan y Pepe son hermanos. Otro ejemplo de sentencia podría ser $Casados(Padre(Juan), Madre(Pepe))$, que significaría que el padre de Juan y la madre de Pepe están casados. También podemos utilizar conectivas lógicas para construir sentencias más complejas como, por ejemplo, $Hermano(Juan, Pepe) \wedge Hermano(Pepe, Juan)$. Las relaciones entre los objetos son consistentes entre las sentencias de ejemplo, por lo que podemos decir que todas las sentencias son verdad.

Por último, restaría destacar la posibilidad de utilizar dos cuantificadores en la lógica de primer orden, es decir, poder expresar propiedades sobre colecciones de objetos completas. El primero de ellos es el **cuantificador universal**, denotado como \forall , típicamente pronunciado como *para todo*, y que permite establecer reglas del tipo *todos los ingenieros son personas*: $\forall x Ingeniero(x) \implies Persona(x)$. En este caso el símbolo x es una variable que podría tomar el valor de cualquier término del dominio. La sentencia anterior será verdad si la misma sentencia sin cuantificar es verdad para todos los términos del dominio. Así, podríamos reescribir esta sentencia como la conjunción de todas las sentencias posibles del dominio:

$$\begin{aligned} &(Ingeniero(Juan) \implies Persona(Juan)) \wedge \\ &(Ingeniero(Pepe) \implies Persona(Pepe)) \wedge \\ &(Ingeniero(PiernaDerecha(Juan)) \implies \\ &Persona(PiernaDerecha(Juan))) \wedge \\ &\dots \end{aligned}$$

En este ejemplo concreto, vemos que la sentencia no se cumple para todos los casos posibles, por lo que el resultado de su evaluación será *falso*. El otro cuantificador disponible es el **cuantificador existencial**, denotado como \exists , típicamente pronunciado como *para algún*, y que sirve para establecer reglas del tipo *Pepe tiene un hermano*: $\exists x Persona(x) \wedge Hermano(x, Pepe)$. La sentencia anterior será ver-

dad si la misma sentencia sin cuantificar es verdad para alguno de los términos del dominio:

$$\begin{aligned}
 & (Persona(Pepe) \wedge Hermano(Pepe, Pepe)) \vee \\
 & (Persona(Juan) \wedge Hermano(Juan, Pepe)) \vee \\
 & (Persona(PiernaDerecha(Pepe)) \wedge \\
 & Hermano(PiernaDerecha(Pepe), Pepe)) \vee \\
 & \dots
 \end{aligned}$$

En este ejemplo concreto vemos como la sentencia se cumple al menos para el segundo caso, por lo que el resultado de su evaluación será *verdadero*.

La lógica de primer orden también incluye otra forma de construir sentencias, adicionalmente a utilizar predicados y términos como se ha visto hasta ahora. Se puede utilizar el *símbolo de igualdad* (denotado por $=$) para indicar que dos términos se refieren al mismo objeto y poder así establecer hechos, por ejemplo, $Hermano(Juan) = Pepe$. Del mismo modo, también podemos definir la desigualdad, por ejemplo $\exists x, y Hermano(x, Pepe) \wedge Hermano(x, Pepe) \wedge \neg(x = y)$, para indicar que *Pepe* tiene al menos dos hermanos distintos. La notación $x \neq y$ también se puede utilizar como abreviación de $\neg(x = y)$.

Con esta breve introducción a la sintaxis de la notación lógica, podemos continuar la siguiente sección relativa a los lenguajes PDDL.

PDDL

En PDDL, un *estado* se representa mediante una conjunción de aspectos del mundo que cambian a lo largo del tiempo, mediante un único predicado y, si dicho predicado admitiera argumentos, deberán ser constantes. Por ejemplo, un posible estado podría ser $En(Camion_1, Madrid) \wedge En(Camion_2, Barcelona)$, el cual es una conjunción que emplea un predicado y cuatro objetos, que podrían pertenecer al dominio de un problema de entrega de pedidos.

Podemos definir *esquemas de acciones* que compartan la misma temática. Por ejemplo, a continuación se define un esquema de acciones para que un avión vuele de una ubicación a otra:

$$\begin{aligned}
 & Action(Volar(a, origen, destino), \\
 & PRECOND : En(a, origen) \wedge Avion(a) \wedge Aeropuerto(origen) \\
 & \wedge Aeropuerto(destino) \\
 & EFFECT : \neg En(a, origen) \wedge En(a, destino))
 \end{aligned}$$

El esquema de acciones está formado por el nombre de la acción, una lista de todas las variables usadas en el esquema, una precondition y un efecto. Los dos últimos se definen mediante conjunciones de sentencias que pueden ser positivas o negativas. Así, dada una acción *a* cuyas variables hayan sido reemplazadas por

constantes, decimos que es *aplicable* a un estado s si dicho estado cumple las precondiciones de la acción. El resultado de ejecutar una acción aplicable a a un estado s se define como un estado s' representado por el conjunto de sentencias originales de s al que se le elimina las sentencias negativas y se le añade las sentencias positivas del *efecto*. Por ejemplo, ejecutar la acción $Volar(A_1, MAD, BCN)$ eliminaría la sentencia $En(A_1, MAD)$ del estado y añadiría la sentencia $En(A_1, BCN)$ al estado, tal y como indica el campo *EFFECT* de la acción.

Un conjunto de estos esquemas de acciones sirve para definir un *dominio de planificación*, que engloba varios problemas específicos definidos mediante un estado inicial y un objetivo. El estado inicial se especifica mediante la conjunción de sentencias que no incluyan variables y el objetivo mediante una conjunción de sentencias positivas o negativas.

A continuación se define un ejemplo completo de problema en el dominio del transporte aéreo de mercancías, que incluye tres acciones: *Cargar*, *Descargar* y *Volar*. Estas acciones afectan a dos predicados: $Dentro(c, a)$ para indicar que el cargamento c está dentro del avión a y $En(x, a)$ para indicar que el objeto x (que puede ser un avión o un cargamento) está en el aeropuerto a :

$$\begin{aligned}
 &Init(En(C_1, MAD) \wedge En(C_2, BCN) \wedge En(A_1, MAD) \wedge En(A_2, BCN) \\
 &\quad \wedge Cargamento(C_1) \wedge Cargamento(C_2) \wedge Avion(A_1) \wedge Avion(A_2)) \\
 &\quad \wedge Aeropuerto(MAD) \wedge Aeropuerto(BCN)) \\
 &Goal(En(C_1, BCN) \wedge En(C_2, MAD)) \\
 &Action(Cargar(c, a, r), \\
 &\quad PRECOND : En(c, r) \wedge En(a, r) \wedge Cargamento(c) \wedge Avion(a) \\
 &\quad \wedge Aeropuerto(r) \\
 &\quad EFFECT : \neg En(c, r) \wedge Dentro(c, a)) \\
 &Action(Descargar(c, a, r), \\
 &\quad PRECOND : Dentro(c, a) \wedge En(a, r) \wedge Cargamento(c) \\
 &\quad \wedge Avion(a) \wedge Aeropuerto(r) \\
 &\quad EFFECT : En(c, r) \wedge \neg Dentro(c, a)) \\
 &Action(Volar(a, origen, destino), \\
 &\quad PRECOND : En(a, origen) \wedge Avion(a) \wedge Aeropuerto(origen) \\
 &\quad \wedge Aeropuerto(destino) \\
 &\quad EFFECT : \neg En(a, origen) \wedge En(a, destino))
 \end{aligned}$$

Hay que tener especial cuidado a la hora de definir las precondiciones de las acciones y asegurarse de que se es completamente explícito con las sentencias. Cuando un avión vuela de un aeropuerto a otro, el cargamento dentro del avión también vuela con él. Así, decimos que un cargamento solo se encontrará en un aeropuerto cuando sea explícitamente descargado. Una planificación posible para


el problema anterior podría ser:

$Cargar(C_1, A_1, MAD), Volar(A_1, MAD, BCN), Descargar(C_1, A_1, BCN),$
 $Cargar(C_2, A_2, BCN), Volar(A_2, BCN, MAD), Descargar(C_2, A_2, MAD)$

Esta planificación cumpliría con el objetivo del problema, de tal forma que ambos cargamentos acabarán en el aeropuerto que les corresponde.

Como se ha comentado, PDDL utiliza su propia sintaxis basada en listas. Podemos ver un ejemplo completo del problema anterior implementado en dicha sintaxis en los Listados 2.4 y 2.5. El primero de ellos (*domain.pddl*) especificará el dominio, mientras que el segundo (*problem.pddl*) especificará el problema concreto a resolver en dicho dominio. Puede observarse como la traducción de la sintaxis de símbolos y conectivas lógicas a PDDL es directa. Es posible resolver dicho problema en cualquier entorno PDDL.



En  Link: <http://editor.planning.domains> se encuentra un entorno donde podemos cargar archivos en lenguaje PDDL (dominio y problema), resolver el problema en un dominio concreto y obtener una posible planificación. Nótese que la planificación resultante podrá variar en función de la implementación interna del planificador utilizado.

Listado 2.4: Implementación en PDDL del dominio del problema del transporte aéreo.

```

1 (define (domain transporte-de-mercancias)
2   (:predicates
3     (En ?c ?r)
4     (Dentro ?c ?a)
5     (Cargamento ?c)
6     (Avion ?a)
7     (Aeropuerto ?r))
8
9   (:action Cargar
10    :parameters (?c ?a ?r)
11    :precondition (and (En ?c ?r) (En ?a ?r) (Cargamento ?c) (Avion ?a) (Aeropuerto ?
12    r))
13    :effect (and (not (En ?c ?r)) (Dentro ?c ?a)))
14
15   (:action Descargar
16    :parameters (?c ?a ?r)
17    :precondition (and (Dentro ?c ?a) (En ?a ?r) (Cargamento ?c) (Avion ?a) (
18    Aeropuerto ?r))
19    :effect (and (En ?c ?r) (not (Dentro ?c ?a))))
20
21   (:action Volar
22    :parameters (?a ?origen ?destino)
23    :precondition (and (En ?a ?origen) (Avion ?a) (Aeropuerto ?origen) (Aeropuerto ?
24    destino))
25    :effect (and (not (En ?a ?origen)) (En ?a ?destino)))

```

Listado 2.5: Implementación en PDDL del problema del transporte aéreo.

```
1 (define (problem transporte-de-mercancias-madrid-barcelona)
2   (:domain transporte-de-mercancias)
3   (:objects A1 A2 C1 C2 MAD BCN)
4   (:init
5     (Avion A1)
6     (Avion A2)
7     (Cargamento C1)
8     (Cargamento C2)
9     (Aeropuerto MAD)
10    (Aeropuerto BCN)
11    (En A1 MAD)
12    (En A2 BCN)
13    (En C1 MAD)
14    (En C2 BCN))
15   (:goal (and (En C1 BCN) (En C2 MAD))))
```

2.3.2. Sistemas de razonamiento impreciso

En la sección anterior se ha descrito PDDL, un lenguaje basado en listas capaz de proporcionar soluciones a partir de la descripción de un problema en un dominio concreto. Dicho sistema está basado en la lógica clásica, donde las sentencias pueden tomar un valor *verdadero* o *falso*, sin más opciones posibles.

Los sistemas de razonamiento impreciso permiten resolver problemas de forma imprecisa con el uso de términos lingüísticos. Concretamente, aplicando lo que se conoce como **lógica difusa**, es posible definir elementos que puedan pertenecer a diferentes conjuntos con grados de pertenencia comprendidos en el intervalo $[0, 1]$, donde 1 representa pertenencia absoluta y 0 pertenencia nula. Así, podemos decir que utilizando la lógica difusa, es posible expresar algo que sea *parcialmente verdadero* o *parcialmente falso*.

El esquema general para construir este tipo de sistemas comprende cuatro pasos bien diferenciados basados en el método de Mamdani [MA75]: i) modelado, donde se definen las variables y sus dominios; ii) proceso de *fuzzificación*, donde se mide el grado de pertenencia de las variables de entrada a los conjuntos difusos; iii) motor de inferencia, donde se realiza el razonamiento; y iv) proceso de *defuzzificación*, donde se realiza el paso contrario al primero, aunque solo se realizará si se desea un valor de salida real en nuestro sistema.

Fase de modelado

En este paso se definen las variables de entrada, las variables de salida y el dominio de definición para cada variable. Este último estará constituido por el rango de valores numéricos que la variable puede tomar y los conjuntos difusos. Los conjuntos difusos estarán formados por rangos de valores asociados con una etiqueta lingüística (denominadas *variables difusas*). Por ejemplo, la Figura 2.6a podría servir para clasificar a las personas de una determinada población en función de su

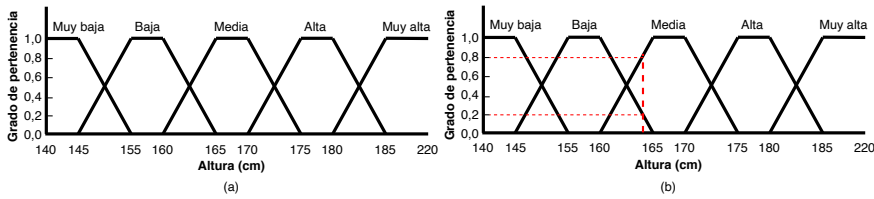


Figura 2.6: (a) Dominio de definición para la variable de la altura y (b) ejemplo de pertenencia a un conjunto difuso dada una altura de 164 centímetros.

altura. Computacionalmente, los conjuntos difusos suelen representarse mediante *funciones de ajuste lineales* en lugar de otras más complejas por cuestiones de rendimiento. Así, el conjunto difuso de las personas con una altura *alta* puede definirse como $(0/170, 1/175, 1/180, 0/185)$, donde el separador / sirve para asociar el grado de pertenencia con la altura en el eje horizontal.

Tradicionalmente los conjuntos difusos se han definido involucrando a un experto en el dominio del problema o combinando la opinión de varios expertos. Recientemente se ha introducido una nueva técnica basada en las ANNs, que aprenden a partir de los datos disponibles del funcionamiento del sistema y luego derivan los conjuntos difusos automáticamente.

Fuzzificación

En este paso se convierten las variables de entrada en grados de pertenencia que servirán para cuantificar el grado de posesión hacia su correspondiente variable difusa. Para el ejemplo de la Figura 2.6, estas variables representan las etiquetas *bajo*, *alto*, *muy alto*, etc. Por ejemplo, si una persona mide 164 centímetros, podemos decir que dicho valor de altura estaría comprendido entre los conjuntos *bajo* y *medio*, pero con una pertenencia mayor al conjunto *medio* (ver Figura 2.6b). En la práctica, se utilizan funciones de pertenencia para obtener los grados de pertenencia en función de la forma que tengan los conjuntos del dominio de definición.

Por ejemplo, la función de pertenencia de una entrada x para un conjunto difuso triangular A definido por un límite inferior a , un límite superior b y un valor m donde $a < m < b$, viene definida por:

$$\mu_A(x) = \begin{cases} 0, & x \leq a \\ \frac{x-a}{m-a}, & a < x \leq m \\ \frac{b-x}{b-m}, & m < x < b \\ 0, & x \geq b \end{cases}$$

Por otra parte, para el caso del conjunto difuso trapezoidal definido por un límite inferior a , un límite superior d , un límite de apoyo inferior b y un límite de apoyo superior c donde $a < b < c < d$, podemos definir su función como:

$$\mu_A(x) = \begin{cases} 0, & (x < a) \vee (x > d) \\ \frac{x-a}{b-a}, & a \leq x \leq b \\ 1, & b \leq x \leq c \\ \frac{d-x}{d-c}, & c \leq x \leq d \end{cases}$$



Los conjuntos difusos pueden ser alterados utilizando **funciones de cobertura**, relaciones entre palabras (*muy, bastante, extremadamente...*) y funciones concretas que servirán para pronunciar o suavizar en menor o mayor medida las pendientes de las rectas que forman los conjuntos.



Se pueden manipular los conjuntos difusos aplicando operaciones de la teoría clásica de conjuntos, por ejemplo: *complemento, subconjunto, intersección o unión*.

Independientemente de la variable de entrada, la suma de todas las pertenencias a los conjuntos siempre será 1. Para el ejemplo de la altura, y aplicando la función de pertenencia trapezoidal, podemos ver como el valor de 164 centímetros tiene una pertenencia de 0,2 al conjunto de la altura *baja* ($a = 145, b = 155, c = 160, d = 165$; $(165 - 164)/(165 - 160) = 0,2$) y de 0,8 al conjunto de la altura *media* ($a = 160, b = 165, c = 170, d = 175$; $(164 - 160)/(165 - 160) = 0,8$), cuya suma total es 1. Esto significaría que una persona con dicha altura tendría una pertenencia parcial a varios conjuntos.

Motor de inferencia

Una vez fuzzificadas las variables de entrada y haber determinado el grado de pertenencia de las variables a los conjuntos difusos, se realiza el razonamiento para la toma de decisiones. Esto puede realizarse mediante la generación de reglas difusas del tipo *if-then*: *SI x es A, ENTONCES y es B*, donde x e y son variables difusas y A y B son valores lingüísticos determinados por los conjuntos difusos definidos por los rangos de valores posibles; la parte del *SI* de la regla se denomina *antecedente*, mientras que la parte del *ENTONCES* se denomina *consecuente*.

La diferencia entre las reglas clásicas y las reglas difusas es que las primeras utilizan valores discretos para definir las condiciones, mientras que las segundas utilizan valores difusos para ello. Por ejemplo, la regla clásica *SI velocidad es >100, ENTONCES distancia_de_frenado es larga* y la regla difusa *SI velocidad es rápida, ENTONCES distancia_de_frenado es larga* se diferencian en que la primera definirá únicamente un resultado u otro en función de la *velocidad*, la cual puede variar en un rango de 0 a 220 km/h; la segunda regla define también el mismo rango

de valores para la variable *velocidad*, pero en este caso, dicho rango incluye conjuntos difusos como *lenta*, *media* y *rápida*. Así, la variable *distancia_de_frenado* puede variar entre 0 y 300 m, y puede incluir conjuntos difusos como *corta*, *media* y *larga*, por lo que las reglas difusas se relacionan con conjuntos difusos.



Los sistemas de razonamiento imprecisos basados en lógica difusa combinan las reglas posibles y reducen el número de reglas a codificar en al menos un 90 % con respecto de los sistemas tradicionales basados en reglas clásicas.

Continuando con el ejemplo de la altura, podemos construir una regla que sea: *SI la altura es {muy bajo, bajo} y el peso es {elevado, sobrepeso, obesidad} ENTONCES RecomendarDieta {estricta}*. Esta regla se activará cuando el grado de pertenencia de la altura sea superior a 0 para los conjuntos *muy bajo* o *bajo*, así como el peso sea al menos uno de los tres: *elevado*, *sobrepeso* u *obesidad*. Cuando se active, entonces el sistema ejecutará la acción *RecomendarDieta* con un determinado grado de pertenencia a *estricta*.

Los antecedentes de las reglas difusas se pueden evaluar asociando funciones a los operadores Booleanos (*AND*, *OR*...). Típicamente, la función asociada al operador *OR* suele evaluarse como el máximo de los valores dados, mientras que la asociada al operador *AND* suele evaluarse como el mínimo de los valores dados.

Una vez evaluadas todas las reglas, se agregarán todos los resultados obtenidos unificándolos, es decir, produciendo un conjunto difuso final.

Defuzzificación

Este último paso es opcional, y solo será necesario en caso de que queramos obtener un valor discreto de salida en nuestro sistema. Normalmente los métodos más comunes son i) tomar el centro de masas del conjunto difuso de la salida, ii) tomar el valor de la máxima pertenencia del conjunto difuso de la salida.

El primer caso se puede estimar simplemente tomando una muestra de puntos del eje horizontal del conjunto difuso, sumando cada serie de muestras de manera individual y multiplicándola por su grado de pertenencia, y finalmente dividiendo el resultado entre la suma de todos los grados de pertenencia utilizados.

Ejemplo: problema de la propina

El problema de la propina se utiliza normalmente para ilustrar la potencia de los sistemas de razonamiento impreciso basados en lógica difusa. El problema consiste en diseñar un sistema de control difuso que modele cuánta propina deberías dejar en un restaurante. Para ello, el sistema deberá considerar la calidad del servicio y de la comida, puntuados entre 0 y 10. Usando este criterio, se puede dejar una propina comprendida entre el 0 % y el 25 % de la cuenta.

Así, podemos formular el problema de esta forma:

■ Entradas (antecedentes):

- *calidad del servicio*: en una escala del 0 al 10, ¿cómo de buena ha sido el servicio del personal?. Conjunto difuso: *pobre, aceptable, increíble*.
- *calidad de la comida*: en una escala del 0 al 10, ¿cómo de buena estaba la comida?. Conjunto difuso: *mala, decente, buena*.

■ Salidas (consecuentes):

- *propina*: en una escala del 0 % al 25 %, ¿cuánta propina deberíamos dejar?. Conjunto difuso: *baja, media, alta*.

■ Reglas:

- SI la calidad de la comida fue mala o la calidad del servicio fue pobre, ENTONCES la propina será baja.
- SI la calidad del servicio fue aceptable, ENTONCES la propina será media.
- SI la calidad de la comida fue buena o la calidad del servicio fue increíble, ENTONCES la propina será alta.

Aplicando el método de Mamdani para las siguientes entradas:

- **calidad de la comida:** 6,5
- **calidad del servicio:** 9,8

Podemos obtener un resultado final discreto (defuzzificado) que nos recomiende dejar una propina del 19,66 % (ver Figura 2.7 con el proceso completo).

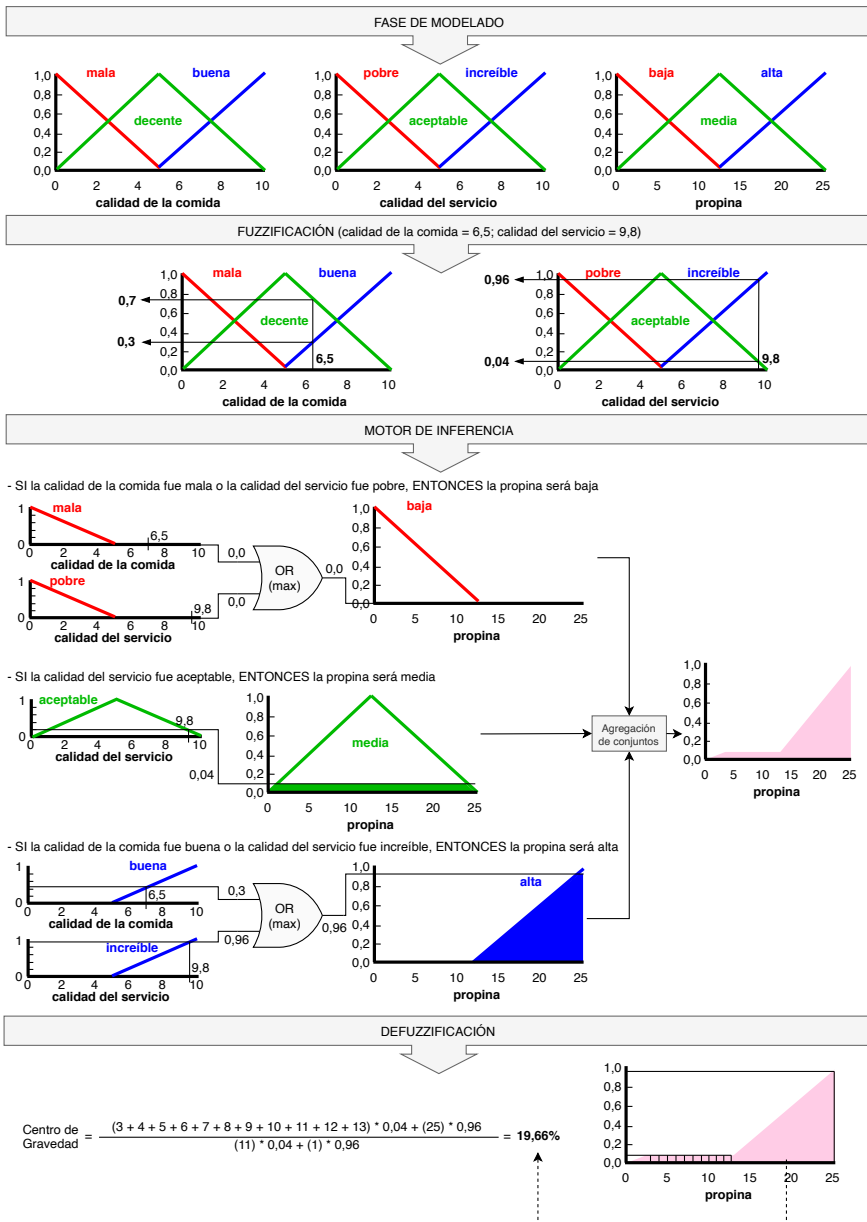


Figura 2.7: Método de Mamdani aplicado al problema de la propina.


Podemos codificar este dominio utilizando alguna biblioteca que nos permita trabajar con sistemas de lógica difusa. En nuestro caso, utilizaremos la biblioteca *scikit-fuzzy*¹ para Python por ser una de las más populares y utilizadas por la comunidad. Así, nos quedaría un código fuente como el que se muestra en el Listado 2.6.

Listado 2.6: Código fuente del problema de la propina implementado utilizando la biblioteca *scikit-fuzzy*.

```

1 import numpy as np
2 import skfuzzy as fuzz
3 from skfuzzy import control as ctrl
4
5 # Rangos de valores que pueden tomar las variables difusas
6 food_quality = ctrl.Antecedent(np.arange(0, 11, 1), 'calidad de la comida')
7 service_quality = ctrl.Antecedent(np.arange(0, 11, 1), 'calidad del servicio')
8 tip = ctrl.Consequent(np.arange(0, 26, 1), 'propina')
9
10 # Indicamos que genere 3 funciones de pertenencia automaticamente para cada
11 # antecedente con las etiquetas lingüísticas indicadas
12 food_quality.automf(names=['mala', 'decente', 'buena'])
13 service_quality.automf(names=['pobre', 'aceptable', 'increible'])
14
15 # Definimos manualmente las funciones de pertenencia utilizando conjuntos
16 # triangulares para el consecuente; los puntos del triángulo se introducen en
17 # sentido horario comenzando por el extremo izquierdo
18 tip['baja'] = fuzz.trimf(tip.universe, [0, 0, 13])
19 tip['media'] = fuzz.trimf(tip.universe, [0, 13, 25])
20 tip['alta'] = fuzz.trimf(tip.universe, [13, 25, 25])
21
22 # Definimos las reglas y las asociamos con el sistema
23 tipping = ctrl.ControlSystemSimulation(ctrl.ControlSystem([
24     ctrl.Rule(food_quality['mala'] | service_quality['pobre'], tip['baja']),
25     ctrl.Rule(service_quality['aceptable'], tip['media']),
26     ctrl.Rule(food_quality['buena'] | service_quality['increible'], tip['alta'])
27 ]))
28
29 # Entradas para nuestro problema concreto
30 tipping.inputs({
31     'calidad de la comida': 6.5,
32     'calidad del servicio': 9.8
33 })
34
35 # Inferencia y defuzzificación
36 tipping.compute()
37
38 print(tipping.output['propina'])

```

¹  Link: <http://github.com/scikit-fuzzy/scikit-fuzzy>

Bibliografía

- [BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.
- [BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, Boston, MA, USA, 1999.
- [HG10] Ralf Herbrich and Thore Graepel. *Handbook of Natural Language Processing, Second Edition*. Chapman and Hall/CRC, Machine Learning and Pattern Recognition Series, 2010.
- [MA75] Ebrahim H Mamdani and Sedrak Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *International journal of man-machine studies*, 7(1):1–13, 1975.
- [Mil95] G. A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.
- [Sar19] D. Sarkar. Python for natural language processing. In *Text Analytics with Python*. 2019.
- [Sch13] Bill Schmarzo. *Big Data: Understanding how data powers big business*. John Wiley & Sons, 2013.
- [Sch15] Bill Schmarzo. *Big Data MBA: Driving business strategies with data science*. John Wiley & Sons, 2015.