Programación Lógica Inductiva (ILP)

Eduardo F. Morales[1]

INAOE, Sta. María Tonantzintla, Puebla, 72840, México,
emorales@inaoep.mx,
WWW home page: http://ccc.inaoep.mx/~emorales

1 Introducción

Dentro de los algoritmos de aprendizaje computacional más exitosos, se encuentran los que inducen árboles de decisión (v.g., C4.5 [65, 66]) o reglas de clasificación (v.g., CN2 [10]), sin embargo, su lenguaje de representación o expresividad es escencialmente proposicional. Esto es, cada prueba que se hace sobre un atributo en un árbol o en una condición de una regla se puede ver como una proposición. Por lo mismo, hablan de un solo objeto a la vez y no podemos relacionar propiedades de dos o más objetos a menos que definamos una propiedad que exprese esa relación para todos los objetos de nuestro dominio.

La Programación Lógica Inductiva o ILP (*Inductive Logic Programming*) combina los resultados experimentales y métodos inductivos del aprendizaje computacional con el poder de representación y formalismo de la lógica de primer orden [50], para poder inducir conceptos representados por programas lógicos.

Para entender las ventajas que tiene aprender representaciones relaciones, supongamos que queremos aprender (y por lo tanto representar con nuestro sistema de aprendizaje) los movimientos de una torre en ajedrez. Si asumimos que representamos los movimientos de las piezas de ajedrez con cuatro atributos, $col1, ren1, col2$ y $ren2$, representando la columna y renglón de una pieza antes y después del movimiento, un sistema proposicional aprendería algo parecido a esto:
If $col1 = 1$ and $col2 = 1$ Then mov_torre = true
If $col1 = 2$ and $col2 = 2$ Then mov_torre = true
...
If $col1 = 8$ and $col2 = 8$ Then mov_torre = true
If $ren1 = 1$ and $ren2 = 1$ Then mov_torre = true
If $ren1 = 2$ and $ren2 = 2$ Then mov_torre = true
...
If $ren1 = 8$ and $ren2 = 8$ Then mov_torre = true

Representando que la torre se puede mover sólo sobre el mismo renglón o sobre la misma columna. En una representación relacional, si asumimos que tenemos un predicado $mov(X, Y, Z, W)$ cuyos argumentos representan igualmente la posición en columna y renglón de cada pieza antes y después del movimiento, nuestro sistema aprendería lo siguiente:
$mov(X, Y, X, Z) : -Y \neq Z.$
$mov(X, Y, Z, Y) : -Y \neq Z.$

Además de aprender una representación más compacta y contar con la capacidad de relacionar propiedades de más de un objeto a la vez, otra ventaja de un sistema de ILP es que puede incluir conocimiento del dominio dentro del proceso de aprendizaje. Consideremos el problema de aprender el concepto de *hija* definida entre dos personas. $hija(X, Y)$ es verdadero si X es hija de Y[1]. Podemos definir la relación $hija(X, Y)$, en términos de las relaciones como *padre* y *femenino*.

En ILP, el problema se plantea de la siguiente forma:

Ejemplos positivos ($\oplus$) y negativos ($\ominus$):
$hija(fernanda, eduardo).\oplus$
$hija(camila, rodrigo).\oplus$
$hija(eugenia, ernesto).\ominus$
$hija(valentina, roberto).\ominus$
...
Conocimiento del Dominio:
$femenino(fernanda).$
$femenino(camila).$
$femenino(eugenia).$
$femenino(valentina).$
...
$padre(eduardo, fernanda).$
$padre(rodrigo, camila).$
$padre(roberto, eugenia).$
$padre(ernesto, valentina).$
...
Resultado:
$hija(X, Y) :- femenino(X), padre(Y, X).$

Finalmente, algunos sistemas de ILP pueden introducir nuevos predicados automáticamente durante el aprendizaje, simplificando la representación de los conceptos aprendidos. Por ejemplo, introducir el predicado *progenitor* refiriendose a *padre* o *madre*, para simplificar una representación de un concepto que utilice indistintantemente a las relaciones de *padre* y *madre*. En la sección 3.2 proporcionamos un ejemplo de esto.

Estos ejemplos ilustran algunas limitaciones de muchos de los sistemas de aprendizaje actuales:

- *Representación Restringida*: inadecuados en áreas que requieren expresar conocimiento relacional (v.g., razonamiento temporal y/o espacial, planificación, lenguaje natural, razonamiento cualitativo, etc.).
- *Conocimiento del Dominio*: son incapaces de incorporar conocimiento del dominio (utilizan un conjunto fijo de atributos).

[1] Aquí asumimos la notación utilizada en Prolog, donde las predicados empiezan con minúsculas y las variables con mayúsculas.

– *Vocabulario Fijo*: no pueden introducir nuevo vocabulario con conocimiento insuficiente del dominio.[2]

Antes de revisar los algoritmos y técnicas principales de ILP que permiten inducir programas lógicos como los mostrados anteriormente, daremos primero algunas nociones de programación lógica y la notación utilizada en este capítulo, necesarias para su entendimiento (ver [37] o [58] para mayor información).

2 Nociones de Lógica y Notación Utilizada

Una *variable* se representa como una cadena de letras o digitos empezando con una letra mayúscula. Un *símbolo funcional* se representa como una letra minúscula seguida de una cadena de letras y números. Un *término* es una constante, una variable o la aplicación de un símbolo funcional a un número determinado de términos. Un *átomo* o *fórmula atómica* es la aplicación de un predicado a un número de términos. Una *literal* es un átomo o su negación. Dos literales son *compatibles* si se llaman igual, tienen el mismo signo y el mismo número de argumentos. La negación se denota como $\neg$. Una *cláusula* es una disjunción de un conjunto finito de literales, que puede representarse como:
$\{A_1, A_2, \ldots, A_n, \neg B_1, \ldots, \neg B_m\}$.
La siguiente notación es equivalente:
$A_1, A_2, \ldots, A_n \leftarrow B_1, B_2, \ldots, B_m$
donde las comas del lado izquierdo se interpretan como disjunciones y las del lado derecho como conjunciones.

Una *cláusula de Horn* es una cláusula con a lo más una literal positiva (v.g., $H \leftarrow B_1, \ldots, B_m$). La literal positiva ($H$) se llama la *cabeza* y las literales negativas (todas las B_is), el *cuerpo*. Una cláusula es *definitiva* si tiene una literal positiva. Una cláusula con el cuerpo vacío es una *cláusula unitaria*. Un conjunto de cláusulas de Horn es un *programa lógico*. F_1 *implica sintácticamente* F_2 (o $F_1 \vdash F_2$) sii F_2 puede derivarse de F_1 usando reglas de inferencia deductivas. Una *substitución* $\Theta = \{V_1/t_1, V_2/t_2, \ldots, V_n/t_n\}$ consiste de una secuencia finita de variables distintas asociadas con términos. Una *instancia* de una cláusula C con substitución Θ, representada como $C\Theta$, se obtiene al remplazar simultaneamente cada ocurrencia de una variable de Θ en C por su término correspondiente. Una substitución θ es un *unificador* de un conjunto de expresiones $\{E_1, \ldots, E_m\}$ si $E_1\theta = \ldots = E_m\theta$. Un unificador θ, es el *unificador más general* (*mgu*) de un conjunto de expresiones E, si para cada unificador σ de E, existe una substitución λ tal que $\sigma = \theta\lambda$

Resolución es una regla de inferencia que se puede usar para probar, por medio de refutación, implicaciones sintácticas que aplica sólo para fórmulas descritas en forma de cláusulas. Una contradicción se denota como: $\square$. Para hacer resolución en lógica de primer orden tenemos que comparar si dos literales complementarias *unifican*. El algoritmo de *unificación* construye el *mgu* de un

[2] Aunque existen sistemas proposicionales de *feature construction* que permiten crear nuevos atributos como combinaciones de atributos existentes [55, 68, 74].

conjunto de expresiones. Sean C_1 y C_2 dos cláusulas con literales L_1 y L_2 respectivamente. Si L_1 y $\neg L_2$ tienen un *mgu* σ, el *resolvente* C de C_1 y C_2 es la cláusula:

$$C = (C_1\sigma - \{L_1\sigma\}) \cup (C_2\sigma - \{L_2\sigma\}) \quad (1)$$

La figura 1 muestra un ejemplo de un árbol de derivación en donde se aplica resolución dos veces. Primero entre las cláusulas $femenino(ana)$ e $hija(X,Y) \leftarrow femenino(X), padre(Y,X)$ con un unificador $\theta_1 = \{X/ana\}$, y después el resolvente de estas dos con la cláusula $padre(juan, ana)$ con el unificador $\theta_2 = \{Y/juan\}$.

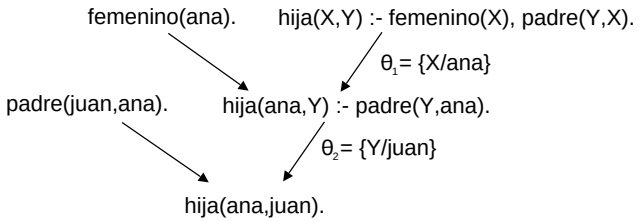

Figura 1. Un árbol de derivación lineal de primer orden.

Un *modelo* de un progama lógico es una interpretación para la cual las cláusulas asumen un valor de verdad. Decimos que F_1 *implica semanticamente a* F_2 (o $F_1 \models F_2$), o también F_1 implica lógicamente a F_2, o F_2 es consecuencia lógica de F_1, sii todo modelo de F_1 es un modelo de F_2.

3 Programación Lógica Inductiva (ILP)

La idea en ILP, como en aprendizaje inductivo, es aprender una hipótesis que cubra los ejemplos positivos y no cubra los negativos. Para verificar la covertura de ejemplos en ILP, se usa normalmente algún algoritmo de inferencia basado en resolución.

- Un programa lógico P se dice *completo* (con respecto a un conjunto de ejemplos positivos $\mathcal{E}^+$) sii para todos los ejemplos $e \in \mathcal{E}^+$, $P \vdash e$
- Un programa lógico P se dice *consistente* (con respecto a un conjunto de ejemplos negativos $\mathcal{E}^-$) sii para ningún ejemplo $e \in \mathcal{E}^-$, $P \vdash e$

El entorno teórico de ILP lo podemos caracterizar entonces como sigue:
Dados
• un conjunto de ejemplos positivos $\mathcal{E}^+$
• un conjunto de ejemplos negativos $\mathcal{E}^-$
• un programa lógico consistente, T, tal que $T \nvdash e^+$ para al menos un $e^+ \in \mathcal{E}^+$
Encontrar un programa lógico H tal que H y T sea completo y consistente: $T \cup H \vdash \mathcal{E}^+$ y $T \cup H \nvdash \mathcal{E}^-$.

T normalmente se refiere a conocimiento del dominio o conocimiento *a priori*. Desde un punto de vista semántico la definición de ILP es:

- Satisfactibilidad previa: $T \wedge E^- \not\models \square$
- Satisfactibilidad posterior (correcto o consistente): $T \wedge H \wedge E^- \not\models \square$
- Necesidad previa: $T \not\models E^+$
- Suficiencia posterior (completo): $T \wedge H \models E^+$

3.1 Búsqueda de Hipótesis

El proceso de inducción puede verse como un proceso de búsqueda de una hipótesis dentro del espacio de hipótesis $H = \{\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_n\}$, esto es dentro del conjunto de todas las hipótesis que el algoritmo de aprendizaje está diseñado a producir [42]. En ILP este espacio puede ser demasiado grande por lo que normalmente se diseñan estrategias de búsqueda que consideren sólo un número limitado de alternativas.

Para realizar una búsqueda eficiente de hipótesis, normalmente es necesario estructurar el espacio de hipótesis, lo cual se puede hacer con un modelo de generalización. Esto es, con un modelo que me diga si una hipótesis es más general o más específica que otra. Esta estructuración permite cortar ramas durante la búsqueda sabiendo que especializaciones o generalizaciones de hipótesis hereden alguna propiedad. Las propiedades más comunes son: incapacidad de cubrir un ejemplo conocido como verdadero o probar un ejemplo conocido como falso. Por ejemplo, si sabemos que una hipótesis cubre un ejemplo negativo, podemos eliminar del espacio de búsqueda todas sus generalizaciones ya que van a seguir cubriendo ese ejemplo. Por el contrario, si una hipótesis no cubre un ejemplo positivo, podemos eliminar del espacio de búsqueda todas sus especializaciones ya que tampoco lo van a cubrir.

Esta estructuración del espacio de hipótesis se puede hacer utilizando $\Theta-$*subsumption*. Una cláusula C, θ–subsume (o es una generalización de) una cláusula D si existe una substitución θ tal que $C\theta \subseteq D$. Usualmente se escribe como $C \preceq D$. Por ejemplo: Sea $C = hija(X,Y) \leftarrow padre(Y,X)$. Con la substitución vacía, C subsume a $hija(X,Y) \leftarrow femenino(X), padre(Y,X)$. Con la substitución $\Theta = \{Y/X\}$, C subsume a $hija(X,X) \leftarrow femenino(X), padre(X,X)$, y con la substitución $\Theta = \{X/ana, Y/luis\}$, C subsume a $hija(ana, luis) \leftarrow femenino(ana)$, $padre(luis, ana), padre(luis, pepe)$.

$\Theta-$*subsumption* introduce una noción de generalización. Una cláusula C es más general que C' si $C\Theta$-subsume a C' y no al revés. También se dice que C' es una especialización (o refinamiento) de C.

Si $C\Theta$-subsume a C', entonces C' es una consecuencia lógica de C, $C \models C'$, pero al revés no se cumple. Por ejemplo: $C = par(X) \leftarrow par(mitad(X))$ y $D = par(X) \leftarrow par(mitad(mitad(X)))$. $C \models D$ pero C no Θ-subsume D ($C\Theta \not\subseteq D$). En particular, C no puede aplicarse a si mismo directa o indirectamente durante la prueba.

El uso de $\Theta-$*subsumtion* se justifica por el hecho de que es decidible entre cláusulas, es fácil de calcular (aunque es NP) y crea un *lattice*. Esto es importante

porque permite buscar en ese *lattice* por hipótesis. La búsqueda puede hacerse: (i) de específico a general, buscando cláusulas que subsuman a la hipótesis actual, (ii) de general a específico, buscando cláusulas subsumidas por la hipótesis actual, ó (iii) en ambos sentidos.

3.2 Sistemas de Específico a General

Genearalización menos general. Una forma de ir búscando hipótesis es generalizando cláusulas gradualmente. La *generalización menos general* (*lgg*) de dos cláusulas C y C' es la generalización más específica de las cláusulas C y C' dentro del *lattice* generado por $\Theta-subsumtion$.

C es la generalización menos general (*lgg*) de D bajo $\theta-subsumtion$ si $C \preceq D$ y para cualquier otra E tal que $E \preceq D$, $E \preceq C$. Plotkin [61–63] fué uno de los pioneros en usar *lgg* como mecanismo de aprendizaje en lógica de primer orden. El algoritmo para evaluar el *lgg* entre dos términos viene descrito en la tabla 1.

Con respecto a átomos, *lgg* es el dual de *mgu*. Dados dos términos f_1 y f_2 y el orden impuesto por $\preceq$, entonces el *lgg* de f_1 y f_2 es su límite inferior más grande (glb) y el *mgu* es el límite superior más bajo (lub).

Tabla 1. Algoritmo de *lgg* entre dos términos.

Si L_1 y L_2 son dos términos o literales compatibles
1. Sea $P_1 = L_1$ y $P_2 = L_2$.
2. Encuentra dos términos, t_1 y t_2, en el mismo lugar en P_1 y P_2, tal que $t_1 \neq t_2$ y o los dos tienen un nombre de función diferente o por lo menos uno de ellos es una variable
3. Si no existe ese par, entonces acaba. $P_1 = P_2 = lgg(L_1, L_2)$.
4. Si existe, escoge una variable X distinta de cualquier variable que ocurra en P_1 o P_2, y en donde t_1 y t_2 aparezcan en el mismo lugar en P_1 y P_2, remplazalos con X.
5. Ve a 2.

Por ejemplo, si tenemos las siguientes dos literales (L_1, L_2) podemos encontrar el *lgg* y el *mgu* entre ellas, donde el *lgg* es la literal más específica que subsume a las dos literales, mientras que el *mgu* es la literal más general subsumida por L_1 y L_2.

$$\overbrace{\underbrace{L_1 = foo(a, f(a), g(X, b), Z)}_{mgu(L_1,L_2) = foo(a,f(a),g(c,b),Z).} \qquad \underbrace{L_2 = foo(Y, f(Y), g(c, b), Z)}}^{lgg(L_1,L_2) = foo(V,f(V),g(W,b),Z).}$$

El *lgg* de dos cláusulas C_1 y C_2 está definido por: $\{l : l_1 \in C_1$ y $l_2 \in C_2$ y $l = lgg(l_1, l_2)\}$. Por ejemplo, si:

$$C1 = hija(fernanda, eduardo) \leftarrow padre(eduardo, fernanda),$$
$$femenino(fernanda), pequeña(fernanda).$$
$$C2 = hija(camila, rodrigo) \leftarrow padre(rodrigo, camila),$$
$$femenino(camila), grande(camila).$$

$$lgg(C1, C2) = hija(X, Y) \leftarrow padre(Y, X), femenino(X).$$

Esto mismo se extiende para un conjunto de cláusulas.

***RLGG* o *lgg* relativo a una teoría.** En general nos interesa encontrar generalizaciones de un conjunto de ejemplos en relación a cierta teoría o conocimiento del dominio. Una cláusula C es más general que una D con respecto a una teoría T si $T \wedge C \vdash D$. Una cláusula C es un *lgg* de una cláusula D con respecto a una teoría T, si $T \vdash C\Theta \rightarrow D$ para alguna substitución Θ. Decimos que C es la generalización menos general de D relativa a T (*rlgg*). Esto es equivalente a decir que $C \wedge T \vdash D'$ donde D' subsume a D y C se usa sólo una vez en la derivación de D'.

En general, puede no existir un *rlgg*, pero si existe para teorías aterrizadas (sin variables). En particular, si T es un conjunto finito de literales aterrizadas, el *lgg* de C_1 y C_2 con respecto a T, es: $lgg(T \rightarrow C_1, T \rightarrow C_2)$.

Rlgg sin embargo, puede tener algunas conclusiones no intuititvas. Por ejemplo, es fácil de verificar que: $P \leftarrow Q$ es más general que $R \leftarrow S, Q$ relativa a $R \leftarrow P, S$.

Para mejorar esto, Buntine introdujo la noción de subsumción generalizada, el cual es un caso especial de *rlgg*, restringido a cláusulas definitivas [6, 7]. La idea es que C es más general que D con respecto a T, si cada vez que D se puede usar (junto con T) para explicar algún ejemplo, C también se pueda usar. Esto lo podemos expresar más formalmente como sigue: Una cláusula $C \equiv C_{cabeza} \leftarrow C_{cuerpo}$, subsume a otra cláusula $D \equiv D_{cabeza} \leftarrow D_{cuerpo}$ con respecto a T ($C \preceq_T D$) si existe una substitución mínima σ tal que $C_{cabeza}\sigma = D_{cabeza}$ y para cualquier substitución aterrizada (*ground*) θ con constantes nuevas para D, se cumple que: $T \cup D_{cuerpo}\theta \models \exists\, (C_{cuerpo}\sigma\theta)$.

Esto lo podemos ver como sigue. Sean: (i) C y D dos cláusulas con variables disjuntas, y T un programa lógico, (ii) θ_1 una substitución (ground) para las variables en C_{cabeza}, (iii) θ_2 una substitución para el resto de las variables en C, y (iv) similarmente, ϕ_1 y ϕ_2 para D. Si $lgg_T(C, D)$ existe, es equivalente al $lgg(C', D')$, donde:

$$C' \equiv C\,\theta_1 \cup \{\neg A_1, \ldots, \neg A_n\} \text{ y } D' \equiv D\,\phi_1 \cup \{\neg B_1, \ldots, \neg B_m\}$$

y para $1 \leq i \leq n$, $T \wedge C_{cuerpo}\theta_1\theta_2 \models A_i$, y A_i es un átomo aterrizado construido con símbolos que ocurren en T, C, θ_1, θ_2, y D. Similarmente para cada B_j.

Esto se puede utilizar dentro de un sistema de aprendizaje de la siguiente forma:

- Toma una cláusula ejemplo (C_1) y sea $\theta_{1,1}$ una substitución instanciando las variables en la cabeza de C_1 a nuevas constants y $\theta_{1,2}$ instanciando las variables que quedan a nuevas constantes.
- Construye una nueva cláusula *saturada* (NC) definida como: $NC \equiv C_1\theta_{1,1} \cup \{\neg A_{1,1}, \neg A_{1,2}, \ldots\}$ donde $T \wedge C_{1cuerpo}\theta_{1,1}\theta_{1,2} \models A_{1,i}$, y $A_{1,i}$ es una átomo instanciado.
- Construye para cada ejemplo, su cláusula saturada, y calcula el *lgg* entre ellas.

Por ejemplo, supongamos que queremos aprender una definición de *faldero* y tenemos las siguientes dos cláusulas ejemplo:
$C = faldero(fido) \leftarrow consentido(fido), pequeño(fido), perro(fido).$
$D = faldero(morris) \leftarrow consentido(morris), gato(morris).$
Entonces:

$$lgg(C, D) = faldero(X) \leftarrow consentido(X).$$

lo cual podría ofender a varias personas. Si por otro lado tenemos de conocimiento del dominio:
$mascota(X) \leftarrow perro(X).$
$mascota(X) \leftarrow gato(X).$
$pequeño(X) \leftarrow gato(X).$

Podemos añadir al cuerpo de C y D lo que podamos deducir del cuerpo de cada cláusula con el conocimiento del dominio. Esto es, añadir $mascota(fido)$ a C tomando $perro(fido)$ del cuerpo de C y la primera cláusula del conocimiento del dominio. De la misma forma, podemos añadir a D, $mascota(morris)$ y $pequeño(morris)$, con lo que nos quedarían las siguientes dos cláusulas satudadas:

$C' = faldero(fido) \leftarrow consentido(fido), pequeño(fido), perro(fido),$
$\quad mascota(fido).$
$D' = faldero(morris) \leftarrow consentido(morris), gato(morris), mascota(morris),$
$\quad pequeño(morris).$

Entonces:

$$rlgg_T(C, D) = lgg(C', D') =$$

$$faldero(X) \leftarrow consentido(X), pequeño(X), mascota(X).$$

que se acerca más a una definición plausible que toma en cuenta nuestro conocimiento del dominio. Sistemas como PAL [43–45] y Golem [49] estás basados en esta técnica.

Inversión de Resolución. Otra idea para aprender programas lógicos, es invertir el proceso de *resolución*. Para esto necesitamos definir una substitución inversa Θ^{-1} que mapea términos a variables. Por ejemplo, si $C = hija(X, Y) \leftarrow femenino(X), padre(Y, X)$, la substitución: $\Theta = \{X/ana, Y/juan\}$ nos da: $C' = C\Theta = hija(ana, juan) \leftarrow femenino(ana), padre(juan, ana)$ y la substitución inversa: $\Theta^{-1} = \{ana/X, juan/Y\}$ nos da: $C'\Theta^{-1} = hija(X, Y) \leftarrow femenino(X)$, $padre(Y, X)$. De forma similar, si conocemos $hija(ana, juan)$ y $padre(juan, ana)$ (figura 2), podríamos aplicar un paso inverso de resolución para obtener $hija(ana, Y)$ $\leftarrow padre(Y, ana)$, con una substitución inversa de $\Theta_2^{-1} = \{juan/Y\}$. Si además sabemos que $femenino(ana)$, podríamos aplicar otro proceso inverso de resolución para obtener $hija(X, Y) \leftarrow femenino(X), padre(Y, X)$ con $\Theta_1^{-1} = \{ana/X\}$.

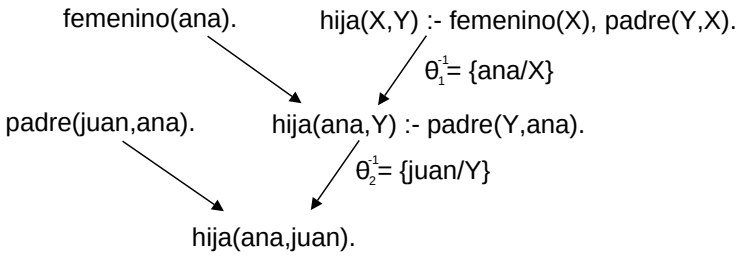

Figura 2. Un árbol de derivación inversa.

El tratar de invertir resolución presenta algunos problemas:

- En general no existe una solución única.
- Tenemos que decidir si vamos a cambiar términos a variables y cómo.

Dado un árbol de derivación de dos cláusulas C_1 y C_2 para obtener C, el operador de *absortion*, construye C_2, dados C y C_1. De la ecuación (1) vista en la sección 2, podemos despejar C_2:

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{L_2\}$$

donde θ_1 y θ_2 son substituciones involucrando únicamente las variables de las cláusulas C_1 y C_2 respectivamente.

Para ésto, se tiene que decidir qué términos y subtérminos se deben de remplazar por la misma variable y cuáles por variables diferentes. Cigol [51] resuelve parcialmente esto, asumiendo que C_1 es una cláusula unitaria (i.e., $C_1 = L_1$). Dado que $\neg L_1\theta_1 = L_2\theta_2$, y por lo tanto $L_2 = (\neg L_1\theta_1)\theta_2^{-1}$, con $C_1 - \{L_1\}$ obtenemos:

$$C_2 = (C \cup \{\neg L_1\}\theta_1)\theta_2^{-1}$$

El problema está con θ_2^{-1}. Si C_1 es una cláusula aterrizada (ejemplo positivo) entonces θ_1 es vacía. Por ejemplo, si $C = menor(A, suc(suc(A)))$ y $C_1 = menor(B, suc(B))$. Si definimos: $\theta_1 = \{B/suc(A)\}$, obtenemos: $(C \cup \{\neg l_1\}\theta_1) =$

$menor(A, suc(suc(A))) \leftarrow menor(suc(A), suc(suc(A)))$. Ahora, para calcular θ_2^{-1}, debemos decidir cómo cambiar las dos ocurrencias del término $suc(suc(A))$ en variables. Supongamos que ambas ocurrencias las cambiamos por D, entonces: $C_2 = (C \cup \{\neg l_1\}\theta_1)\theta_2^{-1} = menor(A, D) \leftarrow menor(suc(A), D)$.

Con cláusulas de Horn en general, el cuerpo de C_1 es "absorbido" en el cuerpo de C (después de la aplicación de una unificación adecuada) y remplazada con su cabeza. Por ejemplo:

$$C = ave(tweety) \leftarrow plumas(tweety), alas(tweety), pico(tweety).$$

$$C_1 = vuela(X) \leftarrow plumas(X), alas(X).$$

El cuerpo de C_1 es "absorbido" en el cuerpo de C después de la substitución $\theta = \{X/tweety\}$ dando una posible solución: $C_2 = ave(tweety) \leftarrow vuela(tweety), pico(tweety)$.

El problema de *absoption* es que es destructiva, en el sentido de que las literales remplazadas se pierden y no pueden usarse para futuras generalizaciones (es problema cuando los cuerpos de las cláusulas se traslapan parcialmente), por lo que las generalizaciones dependen del orden de estas. Por ejemplo, si tenemos:
$C_1 = P \leftarrow Q, R.$
$C_2 = S \leftarrow R, T.$
$C_3 = V \leftarrow Q, R, T, W.$

C_1 y C_2 comparten una literal y ambas se pueden usar para hacer *absorption* con respecto a C_3. *Absorption* de C_3 con C_1 nos da: $C_4 = V \leftarrow P, T, W$, pero ahora no se puede hacer *absorption* de C_4 con C_2.

Este problema se puede resolver si usamos *saturación* [69], en donde la diferencia es que mantenemos todas las literales (las literales usadas en paréntesis, indicando que son opcionales). Saturación hace todas las posibles deducciones en el cuerpo de una cláusula de entrada usando el conocimiento del dominio, y viene la generalización al eliminar las literales redundantes.
C_1 y C_3: $C_4 = V \leftarrow [Q, R], P, T, W.$
C_4 y C_2: $C_5 = V \leftarrow [Q, R, T], W, P, S.$

Otro operador que invierte el proceso de resolución es el operador "W" que se obtiene al combinar dos operadores "V", como el que acabamos de ver. Supongamos que C_1 y C_2 resuelven en una literal común l dentro de la cláusula A para producir B_1 y B_2. Entonces el operador W construye A, C_1 y C_2 dados B_1 y B_2. Cuando l es negativo se llama *intraconstruction* y cuando es positivo *interconstruction*.

Como la literal l en A es eliminada en la resolución y no se encuentra en B_1 o B_2, se tiene que inventar un nuevo predicado.

$$B_1 = (A - \{l_1\})\theta_{A,1} \cup (C_1 - \{l_1\})\theta_{C,1}$$

$$B_2 = (A - \{l_1\})\theta_{A,2} \cup (C_2 - \{l_2\})\theta_{C,2}$$

Suponiendo otra vez que C_1 y C_2 son cláusulas unitarias:

$$A = B_1\theta_{A,1}^{-1} \cup \{l\} = B_2\theta_{A,2}^{-1} \cup \{l\} = B \cup \{l\}$$

donde B es una generalización común de las cláusulas B_1 y B_2.
Por ejemplo, supongamos que:
$B_1 = abuelo(X, Z)$:- $padre(X, Y), padre(Y, Z)$.
$B_2 = abuelo(A, C)$:- $padre(A, B), madre(B, C)$.
Podemos aplicar el operador "W" para obtener las siguientes cláusulas (ver figura 3):
$A = abuelo(L, N)$:- $padre(L, M), nvop(M, N)$.
$C_1 = nvop(Y, Z)$:- $padre(Y, Z)$.
$C_2 = nvop(B, C)$:- $madre(B, C)$.

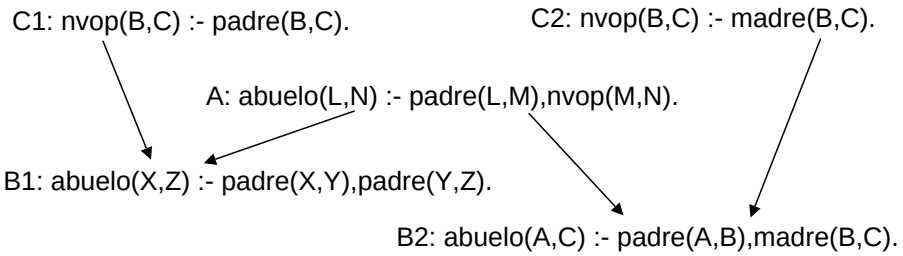

Figura 3. Un ejemplo del operador "W".

Un esquema común de generalización. En [50], Muggleton estableció una forma de relacionar resolución inversa y *rlgg* [50]. La idea es que para cada ejemplo que se tenga, realizar la resolución inversa (derivación inversa lineal) con la substitución inversa más específica (substitución vacía) y después hacer el *lgg* de las cláusulas resultantes. Esto es, las generalizaciones más específicas con respecto a conocimiento del dominio (*rlgg*) son las generalizaciones más específicas (*lgg*) de los árboles inversos de derivación más específicos (ver figura 4).

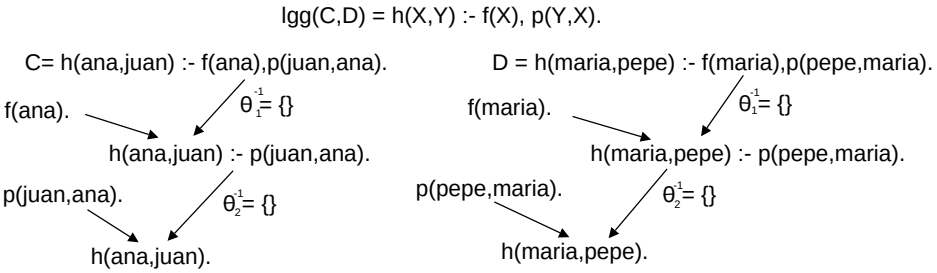

Figura 4. Esquema común de generalización, donde h se refiere a $hija$, f a $femenino$ y p a $padre$.

Inversión de Implicación. Finalmente, podemos tomar una interpretación semántica y pensar en invertir implicación. Sean C y D cláusulas. Decimos que C implica D, o $C \rightarrow D$, sii todo modelo de C también es modelo de D, i.e., $C \models D$. Decimos que C es una generalización (bajo implicación) de D. El problema de invertir implicación es que implicación es indecidible y computacionalmente es muy costoso, a menos, que se impongan ciertas restricciones.

Lo que se quiere encontrar es una H tal que: $T \wedge H \models E$. Por el teorema de deducción: $T \wedge \neg E \models \neg H$, donde $\neg E$ y $\neg H$ representan conjunciones de literales aterrizadas (Skolemizadas). Si $\neg \perp$ representa todas las literales aterrizadas (potencialmente infinitas) ciertas en todos los modelos de: $T \wedge \neg E$, $\neg H$ debe de ser un subconjunto de $\neg \perp$, por lo que: $T \wedge \neg E \models \neg \perp \models \neg H$ y para toda H, $H \models \perp$. Con esto se puede buscar a H en cláusulas que subsumen a $\perp$.

En la práctica para construir $\perp$ se utiliza resolución SLD[3] de profundidad limitada (h). Al resultado se le conoce como modelos *h-easy*. Una forma de encontrar H es construyendo gradualmente hipótesis que sean subconjuntos de $\perp$ siguiendo una estrategia de general a específico. Esto es básicamente lo que hace el sistema *Progol* [52].

Los algoritmos que buscan de específico a general pueden tener problemas en presencia de ruido. Lo mismo sucedió con los algoritmos de reglas proposicionales iniciales como AQ [40, 41], lo que originó el proponer algoritmos de general a específico que siguen una estrategia de *covering* como CN2 [10] o PART [26]. El mismo esquema de *covering* se propuso en ILP para lidear con ruido como se verá en la siguiente sección.

3.3 Sistemas de General a Específico

En general, los algoritmos de ILP de general a específico siguen el esquema descrito en la tabla 2. La idea es ir añadiendo incrementalmente literales (condiciones a reglas) siguiendo un proceso de búsqueda, generalmente tipo *hill-climbing*, usando una medida heurística (ver figura 5). Una vez que se cumple el criterio de necesidad por la hipótesis actual, se eliminan los ejemplos positivos cubiertos y se empieza a generar una nueva cláusula. El proceso continua hasta que se cumple un cierto criterio de suficiencia. En dominios sin ruido, el criterio de necesidad es de consistencia, esto es, no cubrir ningún ejemplo negativo, y el de suficiencia es de covertura, esto es, hasta cubrir todos los ejemplos positivos. En dominios con ruido, se deja de exigir que las hipótesis sean completas y consistentes y se utilizan medidas heurísticas. Estas medidas se basan en el número de ejemplos positivos y negativos cubiertos por las hipótesis, como se verá más adelante.

En esta forma de construcción de programas lógicos se tiene que especificar el criterio a utilizar para seleccionar una nueva literal y el criterio de paro.

Para añadir una nueva literal (especializar una cláusula) se puede hacer con operadores de refinamiento (*refinement operators*). A grandes rasgos, Q es un

[3] **S**eleccionar una literal, usando una estrategia *L*ineal, restringido a cláusulas **D**efinitivas.

Tabla 2. Algoritmo de construcción de cláusulas de general a específico.

Inicializa $\mathcal{E}_{actual} := \mathcal{E}$
Inicializa $H := \emptyset$
repite % *covering*
  Iniciaiza $C := T \leftarrow$
  repite % *especializa*
    Encuentra el mejor refinamiento (C_{mejor}) de C
    Sea $C := C_{mejor}$
  hasta criterio de paro (necesidad)
  Añade C a H, $H := H \cup \{C\}$
  Elimina ejemplos positivos cubiertos por C de $\mathcal{E}_{actual}$
hasta criterio de paro (suficiencia)
Regresa H

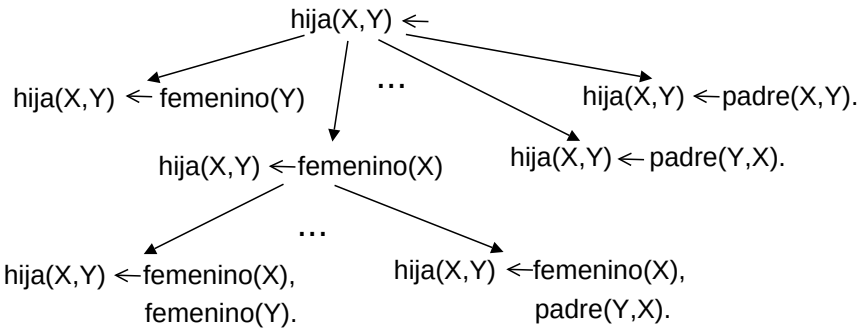

Figura 5. Proceso de construcción de programas lógicos siguiendo un esquema de general-a-específico.

refinamiento de T si T implica Q y $tamaño(T) < tamaño(Q)$, donde *tamaño* es una función que hace un mapeo de cláusulas a números naturales.

Un operador de refinamiento se dice completo sobre un conjunto de cláusulas, si podemos obtener todas las cláusulas por medio de refinamientos sucesivos a partir de la cláusula vacía. Un operador de refinamiento induce un orden parcial sobre el lenguaje de hipótesis. Al igual que con $\Theta-subsumption$, se puede hacer una grafo en donde nodos en capas inferiores son especializaciones de nodos en capas superiores.

Dado un operador de refinamiento completo para su lenguaje de hipótesis, estos sistemas recorren su grafo de refinamiento hasta encontrar la hipótesis deseada. Uno de los primeros sistemas en usar operadores de refinamiento fue MIS [70]. La diferencia entre muchos de los sistemas de ILP radica en qué operador de refinamiento utilizan y cómo recorren su grafo de refinamiento. Por ejemplo, el operador de refinamiento de Foil [67] considera añadir al cuerpo de la cláusula alguna de las siguientes literales: (i) $X_j = X_k$, (ii) $X_j \neq X_k$, (iii) $P(V_1, V_2, \ldots, V_n)$ y (iv) $\neg P(V_1, V_2, \ldots, V_n)$, donde las Xs y Vs son variables y P en uno de los predicados del conocimiento del dominio.

Además de utilizar un operador de refinamiento para especializar una hipótesis, se tiene que determinar la calidad de las hipótesis generadas. Dentro de las diferentes medidas, podemos mencionar las siguientes:

- Precisión: $A(c) = p(\oplus|c)$. Se pueden usar diferentes medidas para estimar esta probabilididad. La más usada, aunque no necesariamente la mejor es: $p(\oplus|c) = \frac{n^{\oplus}(c)}{n(c)}$. Donde $n^{\oplus}(c)$ se refiere a los ejemplos positivos cubiertos por la cláusula c y $n(c)$ se refiere a todos los ejemplos cubiertos por la cláusula c. Más adelante describiremos otras medidas para estimar probabilidades.
- Basada en información: $I(c) = -log_2 p(\oplus|c)$.
- Ganancia en precisión: Se puede medir el aumento de precisión que se obtiene al añadir una literal a la cláusula c para obtener c', $AG(c', c') = A(c') - A(c) = p(\oplus|c') - p(\oplus|c)$.
- Ganancia de información: Similarmente, se puede medir la disminución en información, $IG(c', c) = I(c) - I(c') = log_2 p(\oplus|c') - log_2 p(\oplus|c)$.
- Ganancia en precisión o en información pesada: Se pueden pesar las medidas anteriores por el siguiente factor $\frac{n^{\oplus}(c')}{n^{\oplus}(c)}$, para estimar lo que se gana en ejemplos positivos cubiertos por una especialización particular. Una medida parecida a esta es empleada por Foil [67].
- Ganancia de información mejorada: En [28] se define una variante de ganacia de información como sigue: $IG(c) = \frac{\frac{n^{\oplus}(c)+(|N^{\ominus}|-n^{\ominus}(c)}{|N|} * (I(\top) - I(c))}{|c|}$ donde $|N^{\ominus}|$ es el número de ejemplos negativos, $n^{\ominus}(c)$ es el número de ejemplos negativos cubiertos por la hipótesis c, $|N|$ es el número total de ejemplos, y $|c|$ es el número de literales en el cuerpo de c.

Para estimar las probabilidades usadas en estas medidas, se pueden utilizar diferentes estimadores. Los más comunes son:

- Frecuencia relativa, que como ya vimos antes es: $p(\oplus|c) = \frac{n^{\oplus}(c)}{n(c)}$. Esto es adecuado cuando se cubren muchos ejemplos. Cuando existen pocos ejemplos, esta medida deja de ser confiable [9].
- Estimador Laplaciano: $p(\oplus|c) = \frac{n^{\oplus}(c)+1}{n(c)+2}$. Esta medida aplica cuando se tienen dos clases. Esta medida también asume que se tiene una distribución uniforme de las dos clases [57].
- Estimador-m: $p(\oplus|c) = \frac{n^{\oplus}(c)+m\times p_a(\oplus)}{n(c)+m}$ donde $p_a(\oplus) = \frac{n^{\oplus}}{n}$, es la probabilidad *a priori* de la clase y m expresa nuestra confianza en la evidencia (ejemplos de entrenamiento). Se define subjetivamente de acuerdo al ruido en los ejemplos, entre más ruido más grande tiene que se ser el valor de m. Si $m = 0$ regresamos a frecuencia relativa y si $m = 2$ y $p_a(\oplus) = \frac{1}{2}$ regresamos al estimador Laplaciano.

Existe una gran cantidad de algoritmos de ILP que utilizan una estrategia *top-down* de general a específico (v.g., [3, 13, 21, 59, 67] entre otros).

Los sistemas que usan estrategias de general a específico tienden a tener dificultades con cláusulas que involucran muchas literales. Por otro lado, al utilizar una estrategia de búsqueda tipo *hill climbing* (como la mayoría de los sistemas de aprendizaje), pueden caer en mínimos locales y no llegar a encontrar la mejor hipótesis.

Se han propuesto varios esquemas en ILP para aliviar el problema de miopía que se origina por el esquema de búsqueda utilizado. Por ejemplo, Peña-Castillo y Wrobel [59, 60] proponen utilizar *macro-operadores* para poder añadir más de una literal al mismo tiempo. Otros sistemas, como m-Foil [21] e ICL [13] utilizan *beam-search* para tratar de aliviar el problema de la miopía. También se ha utilizado *fixed-depth look-ahead*, esto es, continuar el refinamiento varios pasos adelante. Esto puede ser utilizando *templates* para hacer la búsqueda más selectiva [3].

3.4 Restricciones y Técnicas Adicionales

En general todos los sistemas ILP introducen ciertas restricciones para generar sus hipótesis:

- Se le dice al sistema qué argumentos están determinados (argumentos de salida) en un predicado si el resto de los argumentos son conocidos (argumentos de entrada). Se pueden formar gráfos que ligan entradas y salidas guiando la construcción de las hipótesis. Se puede restringir aún más utilizando tipos, esto es, sólo puede existir una liga de variables de entrada - salida si los argumentos son del mismo tipo (v.g., [52, 71]).
- Considera sólo cláusulas en que todas las variables aparescan por lo menos 2 veces en la cláusula, o introduce una literal con al menos una variable existente (v.g., [67]).
- Construye hipótesis sólo de una clase de cláusulas definidas con esquemas o modelos de reglas o gramáticas (v.g., [11, 47]).

- Construye hipótesis siguiendo un operador de refinamiento particular (v.g., [70]).
- Utiliza predicados adicionales para determinar qué predicados del conocimiento del dominio son relevantes para la hipótesis actual (v.g., [2]).

Los programas lógicos pueden tener términos complejos usando símbolos funcionales. Muchos sistemas de ILP usan una representación aplanada o *flattened* para eliminar los símbolos funcionales. Esto es, cada término $f(X_1, \ldots, X_n)$ en cada cláusula C de un programa, se cambia por una nueva cláusula con variables X y se añade al cuerpo de C un nuevo predicado $P_f(X_1, \ldots, X_n, X)$ representando la función f.

Proposicionalización. La idea de proposicionalización [30] es la de transformar un problema relacional en una representación de atributo-valor que pueda ser usada por algoritmos más convencionales de aprendizaje computacional como son C4.5 [65, 66] y CN2 [10]. Las razones principales son: (i) utilizar algoritmos más eficientes de aprendizaje, (ii) contar con una mayor cantidad de opciones de algoritmos, y (iii) utilizar técnicas más maduras, por ejemplo, en el uso de funciones y regresiones, en el manejo de ruido, etc. Durante el proceso de transformación se construyen atributos a partir del conocimiento del dominio y de las propiedades estructurales de los individuos. Este proceso puede ser *completo* ó *parcial* (heurístico). Una proposicionalización completa no pierde información. En la parcial o incompleta se pierde información y el objetivo es el generar automáticamente un conjunto pequeño pero relevante de atributos estructurales. Este último enfoque es el más utilizado debido a que a veces el conjunto completo puede llegar a ser infinito.

Dedibo al incremento exponencial de atributos con respecto a factores como el número de predicados usados y el número máximo de literales a utilizar, algunos sistemas limitan el número de literales, de variables y de posibles valores por los tipos de atributos, así como la longitud de las cláusulas y el número de ocurrencias de ciertos predicados [32].

Otro enfoque recientemente utilizado es emplear funciones de agregación como las usadas en bases de datos. Se aplican agregaciones a columnas de tablas y se obtienen atributos como promedios, máximos, mínimos, y sumas, los cuales alimentan a un sistema proposicional [33].

Para ilustrar más cláramente el proceso de proposicionalización, la tabla 3 muestra cómo se podría expresar con Linus [35], el problema de aprender una definición para la relación *hija*, visto al inicio del capítulo, donde $f(X)$ significa $femenino$ y $p(X, Y)$ significa $progenitor$.

El resultado sería:

IF f(X) = true AND p(Y,X) = true
THEN Clase = $\oplus$.

lo que se puede transformar de regreso a una representación relacional como sigue: $hija(X, Y) \leftarrow femenino(X), progenitor(Y, X)$.

Tabla 3. Transformación a una representación proposicional para aprender la relación de *hija*.

Variables		Atributos proposicionales							Clase
X	Y	f(X)	f(Y)	P(X,X)	P(X,Y)	P(Y,X)	P(Y,Y)	X=Y	
fernanda	eduardo	true	false	false	true	false	false	false	$\oplus$
camila	carmen	true	true	false	true	false	false	false	$\oplus$
emiliano	ernesto	false	false	false	false	true	false	false	$\ominus$
valentina	roberto	true	false	false	false	false	false	false	$\ominus$

El problema con este enfoque esta en como generar un conjunto adecuado de atributos que sean manejables y la incapacidad de inducir definiciones recursivas.

3.5 Algunas extensiones recientes

El uso de representaciones relacionales ha permeado a prácticamente todas las áreas de aprendizaje computacional (v.g., [34]). Dentro de los desarrollos más importantes podemos mencionar:

- Aprendizaje de árboles de decisión relacionales. Se aprenden árboles binarios, donde cada nodo contiene una conjunción lógica y los nodos dentro de un camino del árbol, pueden compartir variables entre sí. El probar un nodo significa probar la conjunción en el nodo y las conjunciones en el camino del nodo raíz hasta nodo que se está probando [4]. Esto mismo se ha extendido a árboles de regresión [31].
- Definición de una medida de distancia relacional que permite calcular la similaridad entre dos objetos. Esta distancia puede tomar en cuenta la similaridad entre objetos relacionados, por ejemplo, entre "hijos" al comparar a dos personas. Esto se puede utilizar para aprendizaje basado en instancias relacional [25] y para realizar *clustering* [38, 39].
- Aprendizaje de reglas de asociación de primer orden. Esto extiende la expresividad de las reglas de asociación y permite encontrar patrones más complejos [17, 18].
- Aprendizaje por refuerzo relacional con lo que se puede aprender una política óptima de acciones relacionales en un ambiente relacional. Por ejemplo, aprender qué movidas realizar en ajedrez [46], cómo jugar Titris [24], etc.
- Aprendizaje de lenguajes lógicos (LLL) o gramáticas aprovechando la expresividad de la lógica de predicados [12].
- Combinar ideas de programación lógica inductiva con probabilidad [15, 16]. ILP se puede extender para considerar aspectos probabilísticos, en particular, aprendizaje basado en implicación probabilísta (v.g., [54, 64]), aprendizaje basado en interpretaciones probabilísticas [27, 56] y finalmente, aprendizaje basado en pruebas lógicas probabilísticas [29].
- Inducir ensambles de clasificadores, tales como *Bagging* y *Boosting* en ILP. La idea es combinar los resultados de varios clasificadores inducidos con algoritmos de ILP [20].

3.6 Aplicaciones

Aunque no se ha tenido el auge de otras áreas de aprendizaje, ILP ha tenido algunos resultados importantes que son difíciles de obtener con otras técnicas (ver p.ej, [22]). Dentro de las aplicaciones principales, podemos mencionar:

- Predicción de relaciones en estructura-actividad, incluyendo la mutagénesis de compuestos moleculares que pueden causar cancer [72].
- Predicción de la estructura tridimensional secundaria de proteinas a partir de su estructura primara o secuencia de aminoácidos [53].
- Diseño de elemento finito de malla para analizar tensión en estructuras físicas [19].
- Aprendizaje de reglas para diagnóstico temprano de enfermedades de reumatismo [36].
- Construcción de programas a partir de especificaciones de alto nivel [1, 2],
- Aprendizaje de reglas de control para sistemas dinámicos a partir de trazas [8, 23].
- Clasificación biológica de calidad de agua de ríos [14].
- Aprendizaje de modelos cualitativos de sistemas dinámicos [5, 21, 44, 73].

Referencias

1. F. Bergadano y D. Gunetti (1993). An interactive system to learn functional logic programs. En *Proc. of the 13th International Joint Conference on Artificial Intelligence*, pgs. 1044-1049. Morgan-Kaufmann, San Mateo, CA.
2. F. Bergadano y D. Gunetti (1994). Learning clauses by tracing derivations. En *Proc. 4th. International Workshop on Inductive Logicv Programming*, pgs. 11-30. Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn.
3. H. Blockeel y L. de Raedt (1997). Lookahead and discretization in ILP. En *Proc. of the 7th. International Workshop on ILP*, pgs. 77-84.
4. H. Blockeel y L. de Raedt (1998). Top-down induction of 1rst order logical decision trees. *Articial Intelligence* 101(1-2):285-297.
5. I. Bratko, S. Muggleton y A. Varsek (1992). Learning qualitative models of dynamic systems. En *Inductive Logic Programming*, S. Muggleton (editor), London: Academic Press.
6. W. Buntine (1988). Generalized subsumption and its applications to induction and redundancy, *Artificial intelligence*, 36(2), 149–176.
7. W. Buntine (1987). Induction of horn clauses: methods and the plausible generalization algorithm. *International Journal of Man-Machine Studies* 26(4):499-519.
8. R. Camacho (1994). Learning stage transition rules with INDLOG. En *Proc. of the Fourth International Workshop on Inductive Logic Programming*, pgs. 273-290. Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn.
9. B. Cestnik (1990). Estimating probabilities: A crucial task in machine learning. En *Proc. Ninth Europena Conference on Artificial Intelligence*, pgs. 147-149. Pitman, Londres.
10. P. Clark y T. Niblett (1989). The CN2 induction algorithm. *Machine learning* 3(4):261-283.
11. W. Cohen (1994). Grammatically biased learning: Learning logic programs using an explicit antecendent description language. *Artificial Intelligence* 68: 303-366.

12. J. Cussens y S. Dzeroski (editores) (2000). *Learning Language in Logic*. Lecture Notes in Artificial Intelligence, Vol. 1925.
13. L. de Raedt y W. van Laer (1995). Inductive Constraint Logic. *Proc. of the Sixth Conference on Algorithmic Learning Theory*, pgs. 80-94.
14. L. de Raedt y M. Bruynooghe (1993). A theory of clausal discovery. En *Proc. of the Thiteenth International Conference on Artificial Intelligence*, pgs. 1058-1063. Morgan Kaufmann, San Mateo, CA.
15. L. De Raedt, K. Kersting (2004). Probabilistic Inductive Logic Programming. En S. Ben-David, J. Case y A. Maruoka (editores), *Proceedings of the 15th International Conference on Algorithmic Learning Theory* (ALT-2004), pgs 19-36. Padova, Italy.
16. L. De Raedt, K. Kersting (2003). Probabilistic Logic Learning. *ACM-SIGKDD Explorations*, número especial en Multi-Relational Data Mining, Vol. 5(1), pp. 31-48.
17. L. Dehaspe y H. Toivonen (1999). Discovery of frequent DATALOG patterns. *Data Mining and Knowledge Discovery archive* 3(1): 7 - 36.
18. L. Dehaspe y H. Toivonen (2001). Discovery of Relational Association Rules. *Relational Data Mining*, S. Džeroski y N. Lavrac (editores). Springer, Berlín.
19. B. Dolšak y S. Muggleton (1992). The application of inductive logic programming to finite element mesh design. En *Inductive Logic Programming*, pgs. 452-472. S. Muggleton (editor), Academic Press.
20. I. Dutra, D. Page, V. Santos Costa y J. Shavlik (2002). An Empirical Evaluation of Bagging in Inductive Logic Programming. En *Proceedings of the Twelfth International Conference on Inductive Logic Programming*, pgs. 48-65, Sydney, Australia.
21. S. Džeroski e I Bratko (1992). Handling noise in inductive logic programming, *Proceedings of the Second International Workshop on Inductive Progamming*, Tokyo, Japan, ICOT TM-1181, Institute for New Generation Computer Technology.
22. S. Džeroski e I Bratko (1996). Applications of inductive logic programming. En *Advances in Inductive Logic Progamming*, L. De Raedt (editor), IOS Press.
23. S. Džeroski, L. Todorovski y Y. Urbančic (1995). Handling real numbers in ILP: A step towards successful behavioural cloning. En *Proc. of the Eigth European Conference on Machine Learning*, pgs. 283-286. Springer-Verlag, Berlín.
24. S. Džeroski, L. De Raedt y K. Driessens (2001). Relational reinforcement learning. *Machine Learning* 43(2): 5-52.
25. W. Emde y D. Wettschereck (1996). Relational Instance Based Learning. En *Proceedings of the 13th International Conference on Machine Learning*, pgs 122 - 130. L. Saitta (editora). Morgan Kaufmann.
26. E. Frank e I.H. Witten (1998). Generating accurate rule sets without global optimization. En *Proc. Fifteenth International Conference on Machine Learning*, pgs. 144-151. J. Sahvlik (editor). Morgan-Kaufmann, San Francisco.
27. N. Friedman, L Getoor, D. Koller y A. Pfeffer (1999). Learning Probabilistic Relational Models. En *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (IJCAI-99), pgs. 1300-1309.
28. J. Fürnkranz t P. Flach (2003). An analysis of rule evaluation metrics. En }emphProc. of the 20th. International Conference on Machine Learning, pgs. 202-209.
29. K. Kersting y L. De Raedt (2002). Basic principles of learning Bayesian logic programs. Tech Report 174, University of Freiburg, Germany.
30. S. Kramer, N. Lavrac y P. Flach (2001). Propositionalization approaches to relational data mining. En *Relational Data Mining*, pgs. 262-291, N. Lavrac y S. Dzeroski (editores) Springer.

31. S. Kramer y G. Widmer (2001). Inducing Classification and Regression Trees in First Order Logic. En *Relational Data Mining* S. Džeroski y N. Lavrac (editores). Springer, Berlín.
32. M.A. Krogel, S. Rawlwa, F. Zelezny, P. Flach, N. Lavrac y S. Wrobel (2003). Comparative Evaluation of Approaches to Propositionallization. En *Proc. of the 13th International Conference on Inductive Logic Programming*, T. Horvath y A. Yamamoto (editores), pgs 194–217. Springer-Verlag Heidelberg.
33. M.A. Krogel y S. Wrobel (2001). Transformation-based learning using multirelational aggergation. En *Proc. of the 11th. International Conference on Inductive Logic Programming* LNAI 2157, C. Rouveirol y M. Sebag (editores), pgs. 142-155, Springer.
34. W. van Laer y L. de Raedt (2001). How to Upgrade Propositional Learners to First Order Logic: A Case Study. *Relational Data Mining*, S. Džeroski y N. Lavrac (editores). Springer, Berlín.
35. N. Lavrac, S. Džeroski y M. Grobelnik. (1991). Learning nonrecursive definitions of relations with Linus, *Proceedings of the European Working Session on Learning*, (265–281), Berlin: Springer-Verlag.
36. N. Lavrac y S. Džeroski (1994). Inductive Logic Programming: Techniques and Applications. Ellis Horwood series in Artificial Intelligence.
37. J.W. Lloyd (1987). *Foundations of Logic Programming*. Springer-Verlag, 2a. edición.
38. M. Kirsten y S. Wrobel (1998). Relational Distance-Based Clustering. En *Proc. Fachgruppentreffen Maschinelles Lernen*, F. Wysotzki, P. Geibel C. Schädler (editores), pgs. 119-124. Techn. Univ. Berlin, Technischer Bericht 98/11.
39. M. Kirsten, S. Wrobel y T. Horvath (2001). Distance Based Approaches to Relational Learning and Clustering. *Relational Data Mining*, S. Džeroski y N. Lavrac (editores). Springer, Berlín.
40. R.S. Michalski (1969). On the quasi-minimal solution of the general covering problem. En *Proceedings of the First International Symposium on Information Processing* pgs. 125-128. Bled. Yugoslavia.
41. R.S. Michalski, I. Mozetic, J. Hong y N. Lavrac (1986). The multi-purpose incremental learning system AQ15 and its testing application on three medical domains. En *Proc. of the Fifth National Conference on AI*, pgs. 1041-1045. Morgan-Kaufmann, Philadelphia.
42. T.M. Mitchell (1982). Generalization as search. *Artificial Intelligence* 18(2):203-226.
43. E. Morales (1991). Learning features by experimentation in chess. En *Proceedings of the European Working Session on Learning*, pgs. 494-551, Y. Kodratoff (editor), Springer-Verlag, Berlin.
44. E. Morales (1997). PAL: A pattern-based first-order inductive system. *Machine Learning* 26: 227-252.
45. E. Morales (1992). Learning Chess patterns, en S. Muggleton (editor), *Inductive Logic Programming*, Academic Press, pp 517-537.
46. E. Morales (2003). Scaling up reinforcement learning with a relational representation. En *Proc. of the Workshop on Adaptability in Multi-agent Systems*, pgs. 15-26, Sydney, Australia.
47. K. Morik, S. Wrobel, J. Kietz t W. Emde (1994). *Knowledge Acquisition and Machine Learning: Theory Methods and Applications*. Academic Press.
48. Muggleton, S. (1992) *Inductive Logic Programming*, London: Academic Press.
49. S. Muggleton y C. Feng (1990). Efficient Induction of Logic Programs. En *Inductive Logic Programming*, S. Muggleton (editor.),London: Academic Press.

50. S. Muggleton (1991). Inductive Logic Programming. *New Generation Computing* 8:295-318, Ohmsha, Ltd.
51. S. Muggleton y W. Buntine (1988). Machine invention of first-order predicates by inverting resolution. En *Proceedings of the Fifth International Conference on Machine Learning*, pgs. 339-352. J. Laird (editor). Morgan Kaufmann, San Mateo, CA.
52. S. Muggleton (1995). Inverse entailmente and Progol. *New Generation Computing* Special issue on Inductive Logic Programming 13(3-4): 245-286.
53. S. Muggleton, R. King y M. Stenberg (1992), Protein secondary structure prediction using logic. *Protein Engineering* 5: 647-657.
54. S. Muggleton (1995). Stochastic Logic Programs. En *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pgs. 29, L. de Raedt (editor), Department of Computer Science, Katholieke Universiteit Leuven.
55. P.M. Murphy y M.J. Pazzani (1991). ID2-of-3: Constructive induction of M-of-N concepts for discriminators in decision trees. En *Proc. of the Eighth International Workshop on Machine Learning*, pgs. 183-187, Morgan Kaufmann, San Francisco, CA.
56. L. Ngo y P. Haddawy (1995). Probabilistic Logic Programming and Bayesian Networks. En *Proc. Asian Computing Science Conference*, pgs. 286-300.
57. T. Niblett e I. Bratko (1986). Learning decision rules in noisy domains. En *Research and development in Expert Systems III*, pgs. 24-25. M. Bramer (editor), Cambridge University Press, Cambridge.
58. S.H. Nienhuys-Cheng y R. de Wolf (1997). *Founbdations of Inductive Logic Programming*. Springer-Verlag, Berllin.
59. L. Peña-Castillo y S. Wrobel (2002). Macro-operators in multirelational learning: a search-space reduction technique. En *Proc. of the 13th. European Conference on Machine Learning*, pgs. 357-368.
60. L. Peña-Castillo y S. Wrobel (2004). A comparative study on methods for reducing myopia of hill-climbing search in multirelational learning. En *Proc. of the 21st. International Conference on Machine Learning*, pgs. 145-152. G. Greiner y D. Schuurmans (editores).
61. G.D. Plotkin (1969). A note on inductive generalization. En *Machine intelligence 5*, B. Meltzer y D. Michie (editores.), Edinburgh: Edinburgh University Press.
62. G.D. Plotkin (1971). A further note of inductive generalization. En *Machine intelligence 6*, B. Meltzer y D. Michie (editores.), Edinburgh: Edinburgh University Press.
63. Plotkin, G. D. (1971). Automatic methods of inductive inference, *PhD Thesis*, University of Edinburgh, Edimburgo.
64. D. Poole (1993). Probabilistic Horn abduction. *Artificial Intelligence* 64(1): pgs 81-129.
65. J.R. Quinlan (1983). Learning efficient classification procedures and their application to chess end games. En *Machine Learning: an artificial intelligence approach*, R.S. Michalski, J.G. Carbonell, y T.M. Mitchell (editores.), Palo Alto, CA: Tioga.
66. J.R. Quinlan (1993). *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Mateo, CA.
67. J.R. Quinlan (1990). Learning logical definitions from relations. *Machine learning*, 5(3), 239–266.
68. H. Ragavan y L. Rendell (1993). Lookahead feature construction for learning hard concepts. En *Proc. of the Tenth International Conference on Machine Learning*, pgs. 252-259, Morgan Kaufmann, San Francisco, CA.

69. C. Rouveirol (1991). ITOU: Induction of first order theories. En *Proceedings of the International Workshop of Inductive Logic Programming*, pgs. 127-157, S. Muggleton (editor), Viana de Castelo, Portugal.
70. E.Y. Shapiro (1983) *Algorithmic program debugging*. MIT Press.
71. A. Srinivasan y R. King (1996). Feature construction with inductive logic programming: A study of quantitative predictions of biological activity aided by structural attributes. En *Proc. of the Sixth International Conference on Inductive Logic Programming* LNAI 1314, pgs. 89-104, S. Muggleton (editor), Springer.
72. A. Srinivasan, S. Muggleton, M. Sternberg y R. King (1996). Theories for mutagenecity: a study in first-order and feature-based induction. *Artificial Intelligence* 85 (1,2): 227-299.
73. A. Varsek (1991). Qualitative model evolution. En *Proceedings IJCAI-91: Twelfth International Joint Conference on Artificial Intelligence*, pgs. 1311 - 1316 J. Mylopoulos y R. Reiter (editores), Morgan Kaufman, San Mateo, CA.
74. Z. Zheng (1998). A comparison of constructing different types of new feature for decision tree learning. En *Feature Extraction, Construction and Selection: A data mining perspective*, Kluwer Academic.