Apprendimento di regole del primo ordine

Programmazione Logica Induttiva (Inductive Logic Programming, ILP)

- E' il settore dell'Apprendimento Automatico che impiega la programmazione logica come linguaggio di rappresentazione degli esempi e dei concetti

Definizione del problema di ILP

- Formalmente:
- Dati:

  un insieme $\mathcal{P}$ di possibili programmi

  un insieme E^+ di esempi positivi

  un insieme E^- di esempi negativi

  un programma logico consistente B, tale che

  $B \not\vdash e^+$, per almeno un $e^+ \in E^+$
- Trovare:

  un programma logico $H \in \mathcal{P}$ tale che

  $\forall e^+ \in E^+, B,H \vdash e^+$ (H è **completo**)

  $\forall e^- \in E^-, B,H \not\vdash e^-$ (H è **consistente**)

Terminologia

- Terminologia:
- B: **background knowledge**, conoscenza a priori, predicati di cui conosciamo già la definizione.
- E^+ E^- costituiscono il **training set**
- H programma **target**, B,H è il programma ottenuto aggiungendo le clausole di H dopo quelle di B.
- $\mathcal{P}$ viene detto **hypothesis space**. Definisce lo spazio di ricerca. La descrizione di tale spazio delle ipotesi prende il nome di **language bias**.
- copre(H,e) □ B,H ⊢ e

Esempio

- Apprendimento del predicato father.

Dati

$\mathcal{P}$: programmi logici contenenti clausole father(X,Y) :- $\Box$ con $\Box$ congiunzione di letterali scelti fra

parent(X,Y),parent(Y,X),male(X),male(Y),

female(X),female(Y)

B= {parent(john,mary),male(john),

parent(david,steve),male(david),

parent(kathy,ellen),female(kathy)}

Esempio

E^+={father(john,mary),father(david,steve)}

E^-={father(kathy,ellen),father(john,steve)}

Trovare:

- una definizione del predicato father che sia consistente e completo rispetto ad E^+, E^-

Possibile soluzione:

father(X,Y):-parent(X,Y),male(X).

Esempio (2)

- Apprendimento di intersection

Dati:

$\mathcal{P}$: programmi logici contenenti clausole int(X,Y,Z):-□
con

□ □ {null(X),null(Y),null(Z),cons(X1,X2,X),int(X2,Y,W)
int(X2,Y,Z),cons(X1,W,Z),member(X1,Y),notmember(X1
,Y)}

E^+={int([4,2,6],[5,2,8],[2])}

E^-={int([4,2,6],[5,2,8],[2,6]), int([4,2,6],[5,2,8],[4,2,6])}

Esempio (2)

B: null([]).

cons(X,Y,[X|Y]).

member(X,[X|Y]).

member(X,[Z|Y]) :- member(X,Y).

notmember(X,[]).

notmember(X,[Z|Y]) :- X □ Z,notmember(X,Y).

Trovare:

- un programma per calcolare l'intersezione che sia consistente e completo rispetto ad E^+, E^-

Esempio (2)

Possibile soluzione:

P1:

- int(X,Y,Z) :- null(X), null(Z).
- int(X,Y,Z) :- cons(X1,X2,X),member(X1,Y),int(X2,Y,W), cons(X1,W,Z).
- int(X,Y,Z) :- cons(X1,X2,X),int(X2,Y,Z).

Attenzione: P1 deriva int([1],[1],[]), che è falso.

Non è garantito che il programma sia consistente anche per gli esempi che non sono nel training set!

Aree di Applicazione [Lav94]

- acquisizione di conoscenza per sistemi esperti
- knowledge discovery nei database: scoperta di informazioni potenzialmente utili dai dati memorizzati in un database
- scientific knowledge discovery: generazione di una teoria scientifica a partire da osservazioni + generazione di esperimenti discriminanti + test della teoria
- software engineering: sintesi di programmi logici
- inductive engineering: costruzione di un modello di un dispositivo partendo da esempi del suo comportamento.

Applicazioni di successo

- predizione della relazione struttura - attività nella progettazione di medicine
- predizione della struttura secondaria delle proteine, importante per determinare l'attività di una medicina.
- predizione della mutagenesis dei composti aromatici, al fine di prevedere la carcinogenesi.
- classificazione dei documenti in base alla loro struttura
- diagnosi di guasti in apparati elettromeccanici
- progettazione del reticolo di elementi finiti
- regole temporali per la diagnosi di guasti ai componenti di satelliti

Classificazione dei sistemi

- Esistono diverse dimensioni di classificazione dei sistemi di ILP [Lav94].
- Possono richiedere che tutti gli esempi siano dati all'inizio (**batch learners**) oppure possono accettarli uno ad uno (**incremental learners**).
- Possono apprendere un solo predicato alla volta (**single predicate learners**) oppure più di uno (**multiple predicate learners**).
- Durante l'apprendimento possono fare domande ad un oracolo per chiedere la validità delle generalizzazioni e/o per classificare esempi generati dal sistema (**interactive learners**) oppure non avere interazione con l'esterno (**non-interactive learners**).

Classificazione dei sistemi

- Infine, un sistema può imparare una teoria da zero oppure partendo da una teoria preesistente incompleta e/o inconsistente (**theory revisors**).
- Sebbene si tratti di dimensioni indipendenti, i sistemi esistenti appartengono a due gruppi opposti.
- Da una parte ci sono i sistemi batch, non interattivi che apprendono da zero un concetto alla volta (**empirical systems**), dall'altro ci sono i sistemi incrementali, interattivi che fanno theory revision e imparano più predicati alla volta (**interactive or incremental systems**)

Relazione di generalita’

- I sistemi apprendono una teoria compiendo una ricerca nello spazio delle clausole ordinato secondo la relazione di generalità.
- Nel caso della programmazione logica induttiva, la relazione di generalità è basata su quella di □-**sussunzione**
- La clausola C1 □-**sussume** C2 se esiste una sostituzione □ tale che C1□ □ C2 [Plo69].
- Si dice che una clausola C1 è almeno tanto generale quanto C2 (C1 ≥ C2) se C1 □-sussume C2
- C1 è più generale di C2 (C1 > C2) se C1 ≥ C2 ma non viceversa.
- **Quindi se C1 □-sussume C2 allora C1 e’ piu’ generale di C2. Non e’ detto il viceversa.**

Relazione di generalita'

- Dato che la relazione di conseguenza logica e' semi-decidibile, al suo posto come relazione di generalita' si usa la θ-sussunzione. Percio':
  - Si dice che una clausola C1 e' almeno tanto generale quanto una clausola C2 (e si scrive C1 $\geq$ C2) se C1 θ-sussume C2 (relazione di generalita' sintattica)
  - C1 e' piu' generale di C2 (C1>C2) se C1 $\geq$ C2 ma non C2 $\geq$ C1

Esempi di relazione di □-sussunzione

C1=grandfather(X,Y) ← father(X,Z).

C2=grandfather(X,Y) ← father(X,Z),parent(Z,Y).

C3=grandfather(john,steve) ← father(john,mary), parent(mary, steve).

C1 □-sussume C2 con la sostituzione vuota □=□

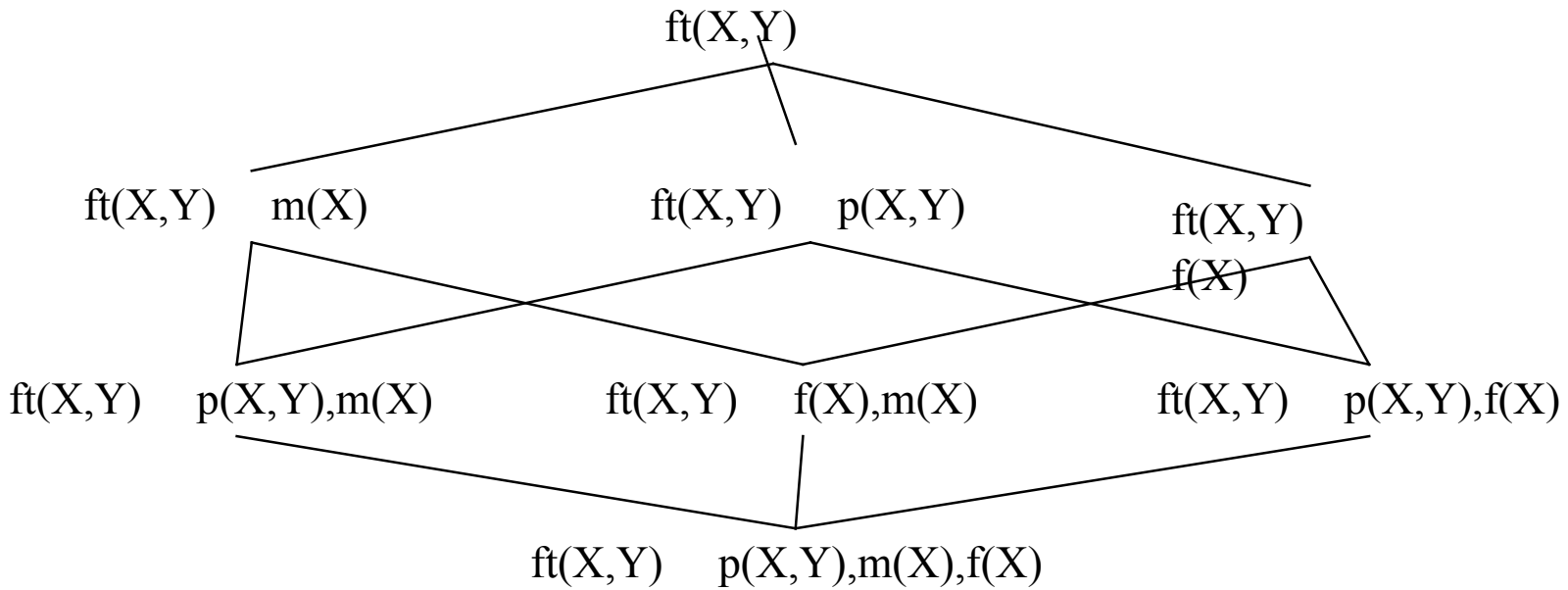
C1 □-sussume C3 con la sostituzione □={X/john,Y/steve,Z/mary}.

C2 □-sussume C3 con la sostituzione □={X/john,Y/steve,Z/mary}.

Proprieta' della □-sussunzione

- La □-sussunzione ha un'importante proprietà: introduce un reticolo nell'insieme delle clausole. Questo significa che ciascuna coppia di clausole ha un least upper bound (lub) e un greatest lower bound (glb).
- Questo non e' vero per la relazione di generalita' basata sull'implicazione logica

Reticolo di □-sussunzione



Algoritmi

- Gli algoritmi per ILP si dividono in **bottom-up** e **top-down** [Ber96] a seconda che utilizzino operatori di generalizzazione o specializzazione.
- Gli algoritmi bottom-up partono dagli esempi (specifici o bottom) e li generalizzano ottenendo una teoria che li copre.
  - Vengono utilizzati degli operatori di generalizzazione ottenuti invertendo le regole deduttive di unificazione, risoluzione e implicazione.

Algoritmi top down

- Gli algoritmi top-down imparano programmi generando le clausole in maniera sequenziale (una dopo l'altra). La generazione di una clausola avviene partendo da una clausola generale (con il body vuoto) e specializzandola (applicando un operatore di specializzazione o raffinamento) fino a che non copre più nessun esempio negativo.

Algoritmi top-down

- Operatore di raffinamento:

$\rho(C)=\{C' | C' \in L, C > C' \text{ e } \nexists C'' \in L, C>C'' \text{ e } C''>C'\}$

dove L e' lo spazio delle possibili clausole.

Tipicamente un operatore di raffinamento genera solo le clausole che sono specializzazioni minime (ovvero le piu' generali) della clausola.

Un operatore di raffinamento tipico applica due operazioni sintattiche ad una clausola:

- applica una sostituzione alla clausola
- aggiunge un letterale al body

Algoritmo top-down di base

Algoritmo ImparaTopDown(E,B;H): dato il training set E e la background B restituisce H

H := 0

repeat /* ciclo di covering */

GeneraClausola(E,B;C)

Aggiungi la clausola alla teoria H:=H□{C}

E:=E-covers(B,H,E) Rimuovi da E gli esempi positivi coperti da H

until il criterio di Sufficienza e' soddisfatto

Ciclo di specializzazione

GeneraClausola(E,B;C): dato il training set E e la background B restituisce la clausola migliore C

Scegli un predicato p da imparare

C := P(X) :- true.

repeat /* ciclo di specializzazione */

Calcola $\rho(C)$

Trova il raffinamento migliore $C_{best} \in \rho(C)$ utilizzando una funzione euristica

$C := C_{best}$

until il criterio di Necessita' e' soddisfatto

return C

Criteri di terminazione

- Gli algoritmo top-down differiscono tra di loro a seconda del criterio di necessita', di sufficienza e dell'euristica che utilizzano per la valutazione della clausola
- In genere, il criterio di Sufficienza richiede che tutti gli esempi positivi siano coperti
- In genere, il criterio di Necessità richiede che nessun esempio negativo sia coperto dalla clausola. Nel caso di dati rumorosi, questo criterio puo' essere rilassato e si puo' ammettere la copertura di un certo numero di esempi negativi.

Euristiche

- L'euristica piu' semplice e' l'accuratezza attesa della clausola

$$A(C)= n^{+}(C)/(n^{+}(C)+n^{-}(C))$$

- dove $n^{+}(C)$ e $n^{-}(C)$, sono, rispettivamente, gli esempi positivi e negativi coperti dalla clausola C (equivale alla Precision della clausola)
- Un'altra euristica e' l'informativita', definita come

$$I(C)=-\log_2(n^{+}(C)/(n^{+}(C)+n^{-}(C)))$$

- Va minimizzata. Il suo intervallo di variazione e' $[0,+\infty)$
- Altre euristiche sono:
- il guadagno di accuratezza GA(C',C)=A(C')-A(C)
- il guadagno di informazione GI(C',C)=I(C)-I(C')
- dove C' e' una specializzazione di C

Esempio

Esempio

$\mathcal{P}$: father(X,Y) :- □ con
□ □ {parent(X,Y),parent(Y,X),
male(X),male(Y),female(X),female(Y)}

B= { parent(john,mary),male(john),
parent(david,steve),male(david),
parent(kathy,ellen),female(kathy)}

E^+={father(john,mary),father(david,steve)}

E^-={father(kathy,ellen),father(john,steve)}

Esempio

1° passo di specializzazione

father(X,Y) :- true.

copre tutti gli e^+ ma anche gli e^-

2° passo

father(X,Y) :- parent(X,Y).

copre tutti gli e^+ ma anche l'e^- father(kathy,ellen).

3° passo

father(X,Y) :- parent(X,Y), male(X).

copre tutti gli e^+ e nessun e^-

Gli esempi positivi coperti vengono rimossi, E^+ diventa vuoto e l'algoritmo termina generando la teoria:

father(X,Y) :- parent(X,Y), male(X).

Strategie di ricerca

- L'algoritmo top-down mostrato utilizza una strategia di ricerca di tipo hill-climbing.
- Alcuni algoritmi invece utilizzano strategie di tipo beam search ovvero mantengono un insieme di clausole di dimensione fissa d

Beam:={C := P(X) :- true}.

repeat /* ciclo di specializzazione */

prendi la prima clausola M dal beam,

aggiungi al beam tutti i raffinamenti □(M)

ordina il beam secondo l'euristica e rimuovi le ultime clausole che eccedono la dimensione d

until il criterio di necessita' e' soddisfatto

Strategie di ricerca

- In questo caso il criterio di necessita' richiede che la migliore clausola del beam sia consistente

Osservazioni sugli algoritmi top-down

- **Ricerca non esaustiva**: non viene esplorato lo spazio dei possibili programmi ma solo quello delle possibili clausole. Ogni volta che viene trovata una clausola consistente viene aggiunta alla teoria e non viene più ritratta, non viene fatto il backtracking sulle clausole.
  - Problemi nel caso di predicati ricorsivi o nel caso di predicati multipli: non è detto che venga trovata una soluzione anche se questa esiste nello spazio delle ipotesi.
- Compromesso necessario per contenere il tempo di calcolo.

Osservazioni sugli algoritmi top-down

- **Copertura degli esempi**: alcuni sistemi utilizzano un differente criterio per la copertura degli esempi in cui per verificare la copertura di un esempio si usa una sola clausola appresa, la background knowledge e gli esempi (**extensional coverage**). Si parla di sistemi **estensionali** per distinguerli dagli **intensionali**.
- **P copre estensionalmente un esempio e** sse esiste una clausola $C = l{:-}l_1,l_2,...,l_n$ e una sostituzione θ tale che $e=[l]\theta$ e
- $B,E^+ \vdash H\ [l_1,l_2,...,l_n]\theta$

FOIL

- FOIL [Qui90,Qui93a,Cam94,Qui91] è tra i piu' diffusi sistemi di ILP. Il codice sorgente C è liberamente scaricabile dalla rete. http://www.cse.unsw.edu.au/~quinlan/foil6.sh
- E' dotato di una efficiente implementazione che lo rende in grado di affrontare problemi reali di apprendimento.
- FOIL è un sistema empirico top-down ed estensionale.
- La background knowledge è fornita a FOIL in forma estensionale, ovvero per ciascun predicato viene fornito l'insieme di tutti i fatti ground veri per quel predicato.

FOIL

- L'algoritmo di FOIL si differenzia dall'algoritmo top-down nei seguenti punti:
  - usa delle euristiche simili a quelle di C4.5, basate sulla teoria dell'informazione, per la selezione del letterale.
  - ricerca hill-climbing: una volta che un letterale è stato aggiunto ad una clausola, non vengono considerati in backtracking altri letterali;
  - extensional coverage degli esempi.
  - aggiunta di letterali determinati

Extensional coverage

- Grazie all'extensional coverage, la generazione di una clausola è completamente indipendente dalle altre e la ricerca nello spazio delle clausole è equivalente a quella nello spazio dei programmi.
- Questo si paga però con il fatto che i programmi appresi possono essere incompleti e inconsistenti:
  - Un programma completo e consistente secondo l'extensional coverage può non esserlo secondo la definizione di ILP (copertura intensionale)
  - Questo si verifica quando si apprendono programmi ricorsivi o contenenti più predicati

Progol

- Progol [Mug95] e' un algoritmo top-down con le seguenti caratteristiche
- utilizza una strategia di tipo beam-search.
- utilizza una clausola più specifica per limitare dal basso lo spazio di ricerca
- utilizza una euristica chiamata "compressione"
- e' scritto in C

Clausola più specifica

- Obiettivo dell'ILP è quello di trovare una teoria H tale che
- B¦HJE
- Ogni clausola nell'H più semplice deve spiegare almeno un esempio, altrimenti esiste un H' più semplice.
- Si consideri ora il caso che sia H che E siano singole clausole di Horn. Possiamo scrivere l'equazione sopra come
- B¦!EJ!H
- Dato che H e E sono clausole singole, !E e !H sono programmi logici che consistono solo di fatti ground skolemizzati

Clausola più specifica

- Sia !B la (potenzialmente infinita) congiunzione di fatti ground che sono veri in tutti i modelli di B¦!E.
- Dato che !H deve essere vero in ogni modello di B¦!E ed è un insieme di fatti ground, deve contenere un sottoinsieme dei letterali ground di !B. Quindi
- B¦!E J!B J!H
- Da cui per ogni H
- HJB
- Quindi un sottoinsieme delle soluzioni per H può essere trovato considerando le clausole che □- sussomono B
- B prende il nome di clausola più specifica

Clausola piu' specifica

- Consideriamo l'apprendimento del predicato father visto in precedenza. Abbiamo:

B= {parent(john,mary),male(john),

parent(david,steve),male(david),

parent(kathy,ellen),female(kathy)}

Considerando E=father(john,mary) abbiamo

B¦!E={parent(john,mary),male(john),
parent(david,steve), male(david), parent(kathy,ellen),
female(kathy), !father(john,mary)}

e B¦!E=!B

Clausola più specifica

Quindi B =father(john,mary):-
parent(john,mary),male(john),parent(david,steve),
male(david), parent(kathy,ellen), female(kathy).

- Come si vede, la soluzione del problema di apprendimento H=father(X,Y) :- parent(X,Y), male(X). è tale che H □-sussume B
- La clausola B serve a limitare la ricerca di H dal basso.
- L'algoritmo di apprendimento deve quindi cercare un H tale che sia meno generale della clausola vuota □ e più generale di B. □ e B servono a delimitare dall'alto e dal basso lo spazio di ricerca per H.

Clausola più specifica

- In generale però B può avere cardinalità infinita. Per questa ragione Progol genera una clausola più specifica B utilizzando alcune restrizioni:
  - Dichiarazioni di modo
  - Limite alla profondità delle variabili
  - Limite alla profondità della derivazione con cui viene ottenuta B

Dichiarazioni di modo

- Una dichiarazione di modo ha la forma
- modeh(n,atom) oppure modeb(n,atom)
- dove n, il recall, è un intero >0 o '*' e atom è un atomo ground.
- I termini nell'atomo sono normali o indicatori di posto.
- Un termine normale è una costante o un simbolo di funzione seguito da una tupla di termini.
- Un indicatore di posto è del tipo +tipo, - tipo o #tipo, dove tipo è una costante.

Dichiarazioni di modo

Esempi di dichiarazioni di modo:

modeh(1,plus(+int,+int,-int))

modeb(*,append(-list,+list,+list))

modeb(1,append(+list,[+any],-list)

modeb(4,(+int>#int))

- Il recall serve a limitare il numero di soluzioni alternative per istanziare l'atomo. '*' significa numero illimitato di soluzioni
- Ad esempio modeb(1,append(+list,[+any],-list) significa che append chiamato con i primi due argomenti ritorna un massimo di 1 istanziazione del terzo argomento

Definite mode language

- Sia C una clausola definita con un ordine totale definito sui letterali e sia M un insieme di dichiarazioni di modi.
- $C=h:-b_1,\ldots,b_n$ è nel definite mode language L(M) se e solo se
  1. h è un atomo di una dichiarazione modeh in M con ogni indicatore di posto +tipo e –tipo sostituito da variabili e ogni indicatore #tipo sostituito da un termine ground
  2. Ogni atomo b_i è l'atomo di una dichiarazione modeb in M con ogni indicatore +tipo e –tipo sostituito da una variabile e ogni indicatore #tipo sostituito da un termine ground

Definite mode language

1. Ogni variabile +tipo in ogni atomo b_i è +tipo in h o –tipo in qualche atomo b_j con 1<=j<i

- In altre parole: le variabili +tipo sono di ingresso e quelle –tipo di uscita: ogni variabile di ingresso in b_i
  - è di ingresso anche nella testa oppure
  - è di uscita in un letterale b_j che precede b_i

Definite mode language con profondità limitata

- Sia C una clausola definita con un ordine totale definito sui letterali e M un insieme di dichiarazioni di modo.
- C è in $L_i(M)$ se e solo se
  1. C è in L(M) e
  2. Tutte le variabili di C hanno profondità al massimo i secondo la seguente definizione

Profondità di una variabile

- La profondità d(v) di una variabile v in una clausola C è definita ricorsivamente come segue:
  - d(v)= 0 se v è nella testa di C
  - d(v)=($\max_{u \in U_V}$ d(u))+1 altrimenti
- Dove U_V sono le variabili diverse da v negli atomi del body di C che contengono v
- Ad esempio:
  - la variabile W nella clausola p(X,Y):-q(X,Z),s(Z,W),t(R,Y) ha profondita' 2
  - la variabile W nella clausola p(X,Y):-q(X,Z),r(X,W),s(Z,W),t(R,Y) ha profondita' 2
  - la variabile W nella clausola p(X,Y):-q(X,Z),s(X,Z,W),t(R,Y) ha profondita' 2

Clausole più specifiche in $L_i(M)$

- Progol utilizza nella ricerca una clausola più specifica che non è B ma è B_i ovvero l'elemento più specifico in $L_i(M)$ tale che

$$B \text{¦} B_i \text{¦!} e H_h \square$$

- dove e è un esempio e $H_h \square$ denota la derivazione della clausola vuota (equivalente a false) in al più h risoluzioni

Clausole più specifiche in $L_i(M)$

- Progol utilizza un algoritmo per calcolare B_i che è corretto e ha complessità

$O(r|M|j^{+}j^{-})^{ij+}$

- dove M è l'insieme delle dichiarazioni di modo, r è il recall massimo in M, j^{+} è il numero massimo di occorrenze di +tipo in ciascuna dichiarazione modeb e il numero massimo di occorrenze di –tipo in ciascuna dichiarazione modeh, j- è il numero massimo di occorrenze di -tipo in ciascuna dichiarazione modeb e il numero massimo di occorrenze di +tipo in ciascuna dichiarazione modeh

Algoritmo di Progol

- Ad ogni ciclo di covering, viene selezionato un esempio positivo e appartenente all'insieme E_{cur}
- Viene generata la clausola piu' specifica B_i (ammessa dal linguaggio) che copre e, ovvero tale che B¦B_i¦!eH$_h$ □
- B_i viene utilizzata da Progol nell'operatore di raffinamento □(C): in pratica C viene specializzata aggiungendo a C (varianti) di letterali presi dal body di B_i. Si veda [Mug95] per una definizione più formale di □(C).

Euristica

- Progol utilizza come euristica una funzione detta compressione.
- Per definire tale funzione occorre dare alcune definizioni preliminari.
- Progol mantiene insieme alla clausola corrente anche la sostituzione θ in base alla quale $C\theta \vdash$ sussume B_i. Quindi uno stato s della ricerca è definito come la coppia ordinata $\langle C, \theta \rangle$

Euristica

La funzione euristica compressione è definita da

$$comp(s)=n^{+}(C)-n^{-}(C)-(|C|-1)-h(s)$$

Osservazioni:

- (|C|-1) è il numero di letterali nel body
- h(s) serve a dare un costo all'introduzione di nuove variabili nella clausola C: è il minimo numero di letterali addizionali necessari per completare le catene di variabili di input/output in C (calcolate considerando le catene di variabili in B)

Definizione di h(s)

Dato s=<C,□>

V_S={v: u/v □ □ e u è nel body di C} insieme delle variabili di B_i a cui è stata sostituita una variabile in C

h(s)=min $_{v \square V_S}$ d'(v)

dove d'(v) indica la lunghezza della piu' corta catena di variabili che inizia con v e termina con una variabile –tipo nella testa di B_i

se l'unica catena che parte da v termina in una variabile che non compare nella testa, h(s)= □

Definizione di d’(v)

d’(v) è definita come segue:

d’(v)= 0 se non c’e’ una variabile –tipo nella testa di B_i
0 se v è –tipo nella testa di B_i
∞ se v non è in B_i
$(\min_{u \in U_v} d'(u))+1$ altrimenti

dove U_v sono le variabili –tipo in atomi del body di C
che contengono occorrenze +tipo di v

Esempio

Sia B_i =p(A,B):-q(A,C),t(A,C),s(C,B).

e M={p(+a,-a),q(+a,-a),t(+a,-a),s(+a,-a)}

s=<C, □>

C=p(A,B):-q(A,C),s(C,B),t(A,C). □={}

V_S=□

h(s)=0

Esempio

s'=<C', □'>

C'=p(A,B):-q(A,C),t(A,C),s(C,D). □'={D/B}

$V_{s'}$={B}

h(s')=d'(B)=0

s''=<C'', □''> C''=p(A,B):-q(A,C),t(A,E),s(C,B). □''={E/C}

$V_{s''}$={C}

h(s'')=d'(C). U_C={B}

d'(C)=d'(B)+1=0+1=1

Esempio

s’’’=<C’’’, □’’’> C’’’=p(A,B):-q(A,C),t(E,C),s(C,B). □’’’={E/A}

$V_{s'''}$={A}

h(s’’’)=d’(A). U_A={C}

d’(A)=d’(C)+1. U_C={B}

d’(A)=d’(B)+2=0+2=2

Considerazioni

- Si vede quindi che
  - introdurre nuove variabili costa
  - sostituire la A costa di piu' di sostituire la C, cioè costa di più sostituire una variabile che si trova all'inizio di una catena di variabili che dipendono l'una dall'altra di una che si trova alla fine

Esempio

s’’’’=<C’’’’, □’’’’> C’’’’=p(A,B):-q(A,C),t(A,E),s(C,D).

□’’’’={E/C, D/B}

$V_{s''''}$={C, B}

d’(C). U_C={D}

d’(C)=d’(D)+1=□+1=□

d’(B)=0

h(s’’’’)= 0

• **f:**

• E’ diverso dall’algoritmo di progol originale

itmo di specializzazione di Progol

L’algoritmo di specializzazione di Progol differisce dall’algoritmo top-down con beam search perche’ viene posto un limite al numero dei passi di specializzazione

GeneraClausola(E,B,n;C): n e’ il numero massimo di passi di specializzazione

seleziona un esempio e=p(X) da E

best=e, bestscore=comp(best), i=0, beam={p(X) }

Algoritmo di specializzazione di Progol

while (i<=n and beam□□)

incrementa i, seleziona la prima clausola M nel beam

genera l'insieme dei raffinamenti □(M) di M

valuta ciascuna clausola C in □(M). Sia h il valore migliore e hSet l'insieme delle clausole aventi valore h

Se h> bestscore ed esiste una clausola D in hSet che ha valore h ed e' consistente, allora poni best=D e bestscore=h

Aggiungi □(M) a beam.

Ordina beam secondo l'ordine decrescente dell'euristica

Elimina da beam le clausole che eccedono la dimensione massima

endwhile

return B

Aleph

- Vedremo l'impiego del sistema Aleph

http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph_toc.html

- Aleph è basato sullo stesso algoritmo di Progol ma
  - è scritto in Prolog invece che in C
  - utilizza una sintassi più semplice per i file di ingresso

Aleph

- Lanciare l'ambiente Prolog Yap
- Caricare aleph:

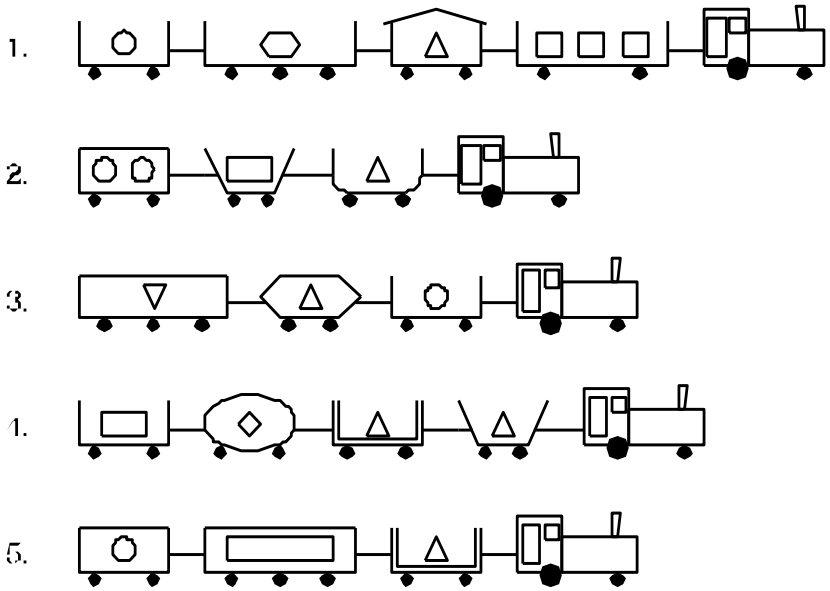
  ?- [aleph].
- Aleph richide 3 file per costruire teorie. L'uso più semplice di Aleph consiste nel
  1. Costruire 3 file di dati chiamati `filestem.b, filestem.f, filestem.n'.
  2. Leggere tutti i dati con il comando read_all(filestem).
  3. Costruire una teoria con il comando induce.

Un semplice esempio: train-spotting

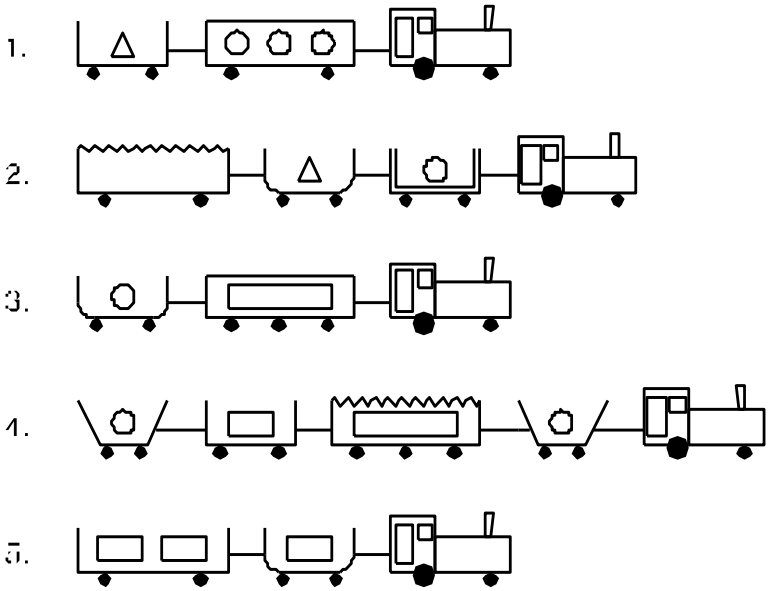
- Un semplice esempio di input per Aleph è la codifica di un problema proposto da Ryszard Michalski
- L'obiettivo è quello di trovare una spiegazione plausibile per distinguere tra due tipi di treni: treni che vanno ad est (eastbound) e treni che vanno ad ovest (westbound).
- Filestem= train

Trainspotting

1. TRAINS GOING EAST



2. TRAINS GOING WEST



File di background knowledge

- La background knowledge deve essere contenuta nel file filestem.b
- Ha la forma di clausole Prolog
- Il file può contenere inoltre direttive per il compilatore Prolog come :-consult(someotherfile).
- Il file contiene inoltre le restrizioni di linguaggio e di ricerca per Aleph:
  - *modi*
  - *tipi*
  - *determinazioni*

Esempio di background knowledge

- Trainspotting: background (nel file train.b):

```
% eastbound train 1
has_car(east1,car_11).
has_car(east1,car_12).
short(car_12).
closed(car_12).
long(car_11).
open_car(car_11).
shape(car_11,rectangle).
load(car_11,rectangle,3).
wheels(car_11,2).
wheels(car_12,2).

....
```

Dichiarazioni di modi

- Invece di modeh e modeb c'è solo mode
- La distinzione tra predicati che possono apparire nella testa o nel body è fatta utilizzando le dichiarazioni di determinazione
- Sintassi

:- mode(RecallNumber,PredicateMode).

- RecallNumber può essere un numero oppure '*'. E' generalmente più semplice specificare RecallNumber come *.

Dichiarazioni di modi

- PredicateMode è un template della forma:

p(ModeType, ModeType,...)

- Ecco alcuni esempi di come appaiono in un file:

```
:- mode(1,mem(+number,+list)).
:- mode(1,dec(+integer,-integer)).
:- mode(1,mult(+integer,+integer,-integer)).
:- mode(1,plus(+integer,+integer,-integer)).
:- mode(1,(+integer)=(#integer)).
:- mode(*,has_car(+train,-car)).
```

Dichiarazioni di modi

- Ogni ModeType può essere:
  - **Semplice:**
    - (a) **+T** variabile di “input”di tipo **T**;
    - (b) **-T** variabile di “output” di tipo **T**; oppure
    - (c) **#T** costante di tipo **T**.
  - **Strutturato:** è della forma **f(..)** dove **f** è un simbolo di funzione e ogni argomento può essere sia semplice che strutturato. Ecco un esempio:

    :- mode(1,mem(+number,[+number|+list])).

Dichiarazioni di modi

- Con queste direttive Aleph assicura che, per ciascuna clausola imparata della forma H:- B1, B2, ..., Bc:
  1. **Variabili di Input.** Ogni variabile di input di tipo **T** in un letterale del body Bi compare come variabile di output di tipo **T** in un letterale del body che appare prima di Bi o appare come variabile di ingresso di tipo **T** in H
  2. **Variabili di output.** Ogni variabile di output di tipo **T** in H compare come variabile di output di tipo **T** in almeno un Bi.
  3. **Costanti.** Ogni argomento denotato da **#T** nei modi ha solo termini ground di tipo **T**.

Dichiarazioni di tipi

- I tipi devono essere specificati per ciascun argomento di tutti i predicati che devono essere usati nella costruzione di una ipotesi. Questa specifica è fatta all'interno dei comandi mode(...,...).
- Per Aleph i tipi sono semplicemente nomi e nessun type-checking è fatto. Variabili di tipo diverso sono trattate distintamente anche se una è un sub-tipo dell'altra.

Dichiarazioni di determinazione

- Le dichiarazioni di determinazione dichiarano i predicati che possono essere utilizzati nelle teste e nei body. Prendono la forma

:-determination(TargetName/Arity,BackgroundName/Arity).

  - Significa che devo imparare il predicato TargetName/Arity e che nelle regole per TargetName/Arity puo' comparire il predicato BackgroundName/Arity nel body
- Tipicamente ci saranno tante dichiarazioni di determinazione per un predicato target, corrispondenti ai predicati che si pensa siano rilevanti nel costruire ipotesi.

Dichiarazioni di determinazione

- Se nessuna determinazione è presente, Aleph non costruisce alcuna ipotesi.
- Le determinazioni sono ammesse per un solo predicato target per ciascuna esecuzione di Aleph: se compaiono determinazioni multiple viene scelta la prima. Esempi:

```
:- determination(eastbound/1,has_car/2).
:- determination(mult/3,mult/3).
:- determination(p/1,'='/2).
```

File di esempi

- **File di esempi positivi:** ha estensione **.f** (forward, es. train.f). Gli esempi positivi sono semplicemente dei fatti ground. Ad esempio
- eastbound(east1). eastbound(east2). eastbound(east3).
- **File di esempi negativi**: ha estensione .n (es. train.n). Gli esempi negativi sono semplicemente dei fatti ground. Ad esempio
- eastbound(west1). eastbound(west1). eastbound(west1).

Lettura dei file di input

- Una volta che i file `filestem.b, filestem.f' e `filestem.n' sono pronti, possono essere letti con:

read_all(filestem).

Es.: read_all(train).

- Questo tipicamente produce una lunga lista delle impostazioni correnti

Costruire una teoria

- Il comando di base per costruire una teoria è:

induce.

- Il risultato è un elenco che mostra le clausole esplorate insieme alla loro copertura di esempi positivi e negativi, come

eastbound(A) :- has_car(A,B). [5/5]

eastbound(A) :- has_car(A,B), short(B). [5/5]

- Il risultato finale ha l'aspetto

[theory] [Rule 1][Pos cover = 5 Neg cover = 0]

eastbound(A) :- has_car(A,B), short(B), closed(B).

[pos-neg] [5]

Costruire una teoria

- induce stampa anche la prestazione della teoria sui dati di training come una matrice di confusione

```
[Training set performance]
        Actual
       +  -
     + 5  0  5
Pred - 0  5  5
       5  5  10
Accuracy = 100%
```

- Anche la prestazione su dati di test è riportata se i valori per i parametri test_pos e test_neg sono settati

Altri comandi

- **Salvare una teoria:** con

write_rules(FileName).

- memorizza in FileName la teoria prodotta.
- **Valutare una teoria:** la teoria prodotta può essere valutata su esempi in un qualunque file di dati con il comando

test(File,Flag,Covered,Total)

  - File è il nome del file contenente gli esempi.
  - Flag può essere show o noshow per mostrare gli esempi coperti o non coperti. Sia File che Flag devono essere forniti.
- test/4 restituisce:
  - Covered è il numero degli esempi coperti dalla teoria corrente
  - Total è il numero totale di esempi nel file di dati

Parametri

- Aleph ha una serie di parametri che ne regolano il funzionamento.
- Possono essere settati con

:-set(Parameter,Value).

- Parametri importanti
  - nodes: è il numero massimo di nodi da esplorare quando si cerca una clausola accettabile. Di default vale 5000.
  - clauselength: numero massimo di letterali nel body di una clausola (default 4).
  - depth: valore massimo della profondità delle dimostrazioni (default 5).

Parametri

- i: profondita' massima delle variabili (default 2)
- test_pos: il valore assegnato al parametro è il nome di un file contenente un elenco di esempi positivi da testare
- test_neg: il valore assegnato al parametro è il nome di un file contenente un elenco di esempi negativi da testare

Esempio: father

- Apprendimento del predicato father:
- father.b:

```
:- set(i,2).  :- set(verbose,1).
:- mode(*,father(+person,-person)).
:- mode(*,parent(+person,-person)).
:- mode(*,male(+person)).  :- mode(*,female(+person)).
:- determination(father/2,parent/2).:- determination(father/2,male/1).
:- determination(father/2,female/1).
% type definitions
person(john). person(mary). person(david). person(steve).
    person(kathy).person(ellen).

parent(john,mary). male(john).parent(david,steve).
    male(david).parent(kathy,ellen). female(kathy).
```

Esempio: father

- father.f:

father(john,mary).

father(david,steve).

- father.n:

father(kathy,ellen).

father(john,steve).

Esempio: father

- Risultato:

[theory]
[Rule 1] [Pos cover = 2 Neg cover = 0]
father(A, B) :-
  parent(A, B), male(A).
[Training set performance]

		Actual		
		+	-	
Pred	+	2	0	2
	-	0	2	2
		2	2	4

Accuracy = 1
[Training set summary] [[2, 0, 0, 2]]
[time taken] [0]
[total clauses constructed] [4]

Algoritmi bottom-up

- Gli algoritmi bottom-up sono basati sul concetto di least general generalization (lgg). L'lgg di due clausole C1 e C2 è una clausola C che:
  - è più generale di C1 e C2,
  - è la più specifica tra le clausole più generali di C1 e C2.

Least general generalization

- La θ-sussunzione ha l'importante proprietà che introduce un reticolo nell'insieme delle clausole. Questo significa che ciascuna coppia di clausole ha un least upper bound (lub) e un greatest lower bound (glb).

- **Definizione: least general generalization [Plo69]**
- La least general generalization di due clausole C1 e C2, denotata da lgg(C1,C2), è il least upper bound di C1 e C2 nel reticolo della θ-sussunzione.

Algoritmo per calcolare la lgg

- L'lgg di due **termini** f1(s1,...,sn) e f2(t1,...,tn) è:
  - f1(lgg(s1,t1),...,lgg(sn,tn)) se f1=f2 oppure
  - una nuova variabile V se f1 ≠ f2.
- V è usata per rappresentare l'lgg di f1(s1,...,sn),f2(t1,...,tn) e deve essere utilizzata per tutte le occorrenze dell'lgg degli stessi termini.
- Esempi:
  - lgg(f(a,b,c),f(a,c,d))=f(a,X,Y)
  - lgg(f(a,a),f(b,b))=f(lgg(a,b),lgg(a,b))=f(X,X)
- Si noti che la stessa variabile X è utilizzata in entrambi gli argomenti del secondo esempio perché indica l'lgg degli stessi due termini a e b.

Algoritmo per calcolare la lgg

- Si noti che se uso la variabile X per lgg(a,b) e poi devo calcolare lgg(b,a) devo usare una variabile diversa.
- Ad esempio
  - lgg(f(a,b),f(b,a))=f(lgg(a,b),lgg(b,a))=f(X,Y)
- Se cosi' non facessi avrei
  - lgg(f(a,b),f(b,a))=f(X,X)
- Ma f(X,X) non puo' essere istanziato in modo da ottenere f(a,b) o f(b,a)

Algoritmo per calcolare la lgg

- L'lgg di due **letterali** L1=(~)p(s1,...,sn) e L2=(~)q(t1,...,tn) è indefinito se L1 e L2 non hanno lo stesso simbolo predicativo e segno, altrimenti è definito come

- lgg(L1,L2)=(~) p(lgg(s1,t1),... lgg(sn,tn))
- **Esempi:**
- lgg(parent(john,mary),parent(john,steve))=parent(joh n,X)
- lgg(parent(john,mary),~ parent(john,steve))=indefinito
- lgg(parent(john,mary),father(john,steve))=indefinito

Algoritmo per calcolare la lgg

L'lgg di due clausole C1={L1,...,Ln} C2={K1,...,Km} è definito come:

lgg(C1,C2)={lgg(Li,Kj) | Li □ C1, Kj □ C2 e lgg(Li,Kj) è definito}

Lgg di due clausole

Esempi:

C1=father(john,mary) parent(john,mary),male(john)

C2=father(david,steve) parent(david,steve),male(david)

lgg(C1,C2)=father (X,Y) parent(X,Y),male(X)

C1=win(conf1) occ(place1,x,conf1),occ(place2,o,conf1)

C2=win(conf2) occ(place1,x, conf2),occ(place2,x, conf2)

lgg(C1,C2)=win(Conf) occ(place1,x,Conf),occ(L,x,Conf),
occ(M,Y,Conf), occ(place2,Y,Conf)

Relative Least General Generalization (rlgg).

- **Relative Least General Generalization (rlgg) [Plo71].**
- È un esempio di tecnica bottom-up.

- L'lgg non può essere utilizzato direttamente come operatore di generalizzazione in quanto non prende in considerazione la conoscenza di background. Per questo si introduce la rlgg che è utilizzata, per esempio, nel sistema GOLEM [Mug90].

Sussunzione relativa

- Definizione: Siano C1 e C2 due clausole e B un insieme di clausole. Si dice che **C1 θ-sussume C2 relativamente a B**, denotato da $C1 \preceq_B C2$, se esiste una sostituzione θ tale che $B \models (C1\theta \subseteq C2)$. L'ordinamento $\preceq_B$ prende il nome di sussunzione relativa e B e' la conoscenza di background [Plo71].
- La relative least general generalization di due clausole C1 e C2 relativamente alla background B e' quella clausola C tale che
  - $C \preceq_B C1$, $C \preceq_B C2$
  - $\forall D$ $D \preceq_B C1$, $D \preceq_B C2$ e $C \preceq D$

Rlgg

- La rlgg non esiste in tutti i casi.
- Nel caso in cui la conoscenza di background e' costituita da un insieme di fatti ground K, la rlgg di due clausole C1=H1 B1 e C2=H2 B2 esiste sempre e può calcolata in questo modo

  rlgg(C1,C2) lgg((H1 B1,K),(H2 B2,K))

- Quindi il problema di calcolare l'rlgg di due clausole può essere ricondotto al problema di calcolare l'lgg di due clausole trasformate.

Esempi di rlgg

Esempio 1:

Si considerino i due esempi positivi

e1=father(john,mary) e2=father(david,steve)

e la conoscenza di background

parent(john,mary). parent(david,steve).
parent(kathy,ellen).

male(john). male(david), female(kathy).

Esempi di rlgg

Abbreviando gli atomi:

e1=f(j,m) e2=f(d,s).

p (j,m). p(d,s). p(k,e). m(j). m(d), f(k).

L'rlgg di e1 ed e2, rispetto alla conoscenza di background, è

$\underline{f(V_{j,d},V_{m,s})}$ p(j,m),$\underline{p(V_{j,d},V_{m,s})}$,p($V_{j,k}$,$V_{m,e}$),

p($V_{d,j}$,$V_{s,m}$),p(d,s),p($V_{d,k}$,$V_{s,e}$),

p($V_{k,j}$,$V_{e,m}$),p($V_{k,d}$,$V_{e,s}$),p(k,e),

m(j),$\underline{m(V_{j,d})}$,m($V_{d,j}$),m(d),f(k).

Eliminando dal body i letterali A veri indipendentemente dai valori per le variabili nella testa (cioe' tali che BJ⊨A)

father(X,Y) parent(X,Y),male(X).

Esempi di rlgg

Esempio 2:

C1=uncle(X,Y) brother(X,father(Y)).

C2=uncle(X,Y) brother(X,mother(Y))

Conoscenza di background:

parent(father(X),X).

parent(mother(X),X).

L'rlgg di C1 e C2, rispetto alla conoscenza di background, è

uncle(X,Y) brother(X,Z), parent(Z,Y),

parent(father(X),X),parent(mother(X),X)

Eliminando i letterali A sempre veri (cioe' tali che BJ□A)

uncle(X,Y) brother(X,Z), parent(Z,Y).

Metodi bottom-up

- Nei metodi bottom-up, le clausole sono generate partendo dalla clausola più specifica che copre uno o più esempi positivi e nessun esempio negativo e applicando ripetutamente operatori di generalizzazione alla clausola finchè non può più essere ulteriormente generalizzata senza coprire esempi negativi.
- In genere inoltre è presente un passo ulteriore in cui si eliminano letterali dal body della clausola generata finchè la clausola non copre degli esempi negativi

GOLEM

- GOLEM è disponibile a
- ftp://ftp.mlnet.org/ml-archive/ILP/public/software/golem
- utilizza il seguente algoritmo:
  - Seleziona a caso alcune coppie di esempi da E^+
  - Calcola le loro rlgg
  - Ripeti
    - Sia C l'rlgg che copre più positivi senza coprire nessun negativo
    - Elimina da E^+ gli esempi coperti da C
    - Seleziona a caso alcuni esempi positivi da E_+
    - Calcola l'rlgg tra C e gli esempi positivi
  - Finchè l'rlgg non copre esempi negativi
  - Elimina letterali dal body della clausola finchè questa non copre esempi negativi

GOLEM

- La procedure precedente è ripetuta rimuovendo ad ogni passo gli esempi positivi coperti dalla clausola generata

- GOLEM accetta come Aleph 3 file:
  - Filestem.b: contiene la background knowledge che deve essere fatta di fatti ground!
  - Filestem.f: esempi positivi
  - Filestem.n: esempi negativi

Esempio: father con GOLEM

- father.b:

```
parent(john,mary).
male(john).

parent(david,steve).
male(david).

parent(kathy,ellen).
female(kathy).
```

Esempio: father con GOLEM

- father.f:

father(john,mary).

father(david,steve).

- father.n:

father(kathy,ellen).

father(john,steve).

Esempio: father con GOLEM

- Output:

[Read in 0ms]
[Rlgg of pair:]
father(john,mary).
father(david,steve).
[is]
father(A,B) :- parent(A,B), male(A).
[Number of negatives covered=0]
[Number of positives covered=2]
[Pos-neg cover=2,potential-examples=0]
[Cover=2]
[Reducing clause]
[Adding clause father(A,B) :- parent(A,B), male(A).]
[REMOVED: 2 FORES, 0 NEGS]
[FORES: 0, BACKS: 8, NEGS: 2, RULES: 1]
[Induction time 0ms]
[Clauses written to father.r]

Risorse disponibili in rete

Sito dell'organizzazione MLNET: www.mlnet.org

Contiene link a conferenze, gruppi di ricerca, materiale didattico, sistemi e dataset relativi all'apprendimento automatico

In particolare: materiale didattico

http://kiew.cs.uni-dortmund.de:8001/

Sito specifico sulla ricerca in ILP in Europa (ILNet2)

http://www.cs.bris.ac.uk/~ILPnet2/

Libro:

"Inductive Logic Programming Techniques and Applications" Nada Lavrac and Saso Dzeroski

http://obelix.ijs.si/SasoDzeroski/ILPBook/

Archivi di set di dati in rete

Dataset si possono trovare sul sito di MLNET e su quello di ILPNet2.

Inoltre:

Raccolta di dati di tipo attributo valore:

University of California at Irvine Machine Learning Repository

http://www.ics.uci.edu/~mlearn/MLRepository.html

Motore di ricerca di articoli su Machine Learning (e in generale su computer science)

http://researchindex.org/cs

Bibliografia

[Ber96] F. Bergadano e D. Gunetti, *Inductive Logic Programming - From Machine Learning to Software Engineering*, MIT Press, Cambridge, Massachusetts, 1996

[Cam94] R. M. Cameron-Jones e J. Ross Quinlan, *Efficient Top-down Induction of Logic Programs*, SIGART, 5, pag. 33—42, 1994.

[Lav94] N. Lavrac and S. Dzeroski, *Inductive Logic Programming Techniques and Applications*, Ellis Horwood, 1994

Bibliografia

[Mug95] Stephen Muggleton, *Inverse Entailment and Progol*, New Gen. Comput., 13:245-286, ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/InvEnt.ps.gz.

[Mug90] S. Muggleton e C. Feng, *Efficient induction of logic programs*, Proceedings of the 1st Conference on Algorithmic Learning Theory Ohmsma, Tokyo, pag. 368-381, 1990.

[Plo69] G.D. Plotkin. *A note on inductive generalisation*. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153-163. Edinburgh University Press, Edinburgh, 1969.

Bibliografia

[Plo71] G.D. Plotkin. *Automatic Methods of Inductive Inference,* PhD thesis, Edinburgh University, 1971.

[Qui91] J. R. Quinlan, *Determinate literals in inductive logic programming*, Proceedings of Twelfth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1991.

[Qui90] J. R. Quinlan, *Learning logical definitions from relations*, Machine Learning, 5:239-- 266, 1990.

[Qui93a] J. R. Quinlan e R. M. Cameron-Jones, *FOIL: A Midterm Report*, Proceedings of the 6th European Conference on Machine Learning, Springer-Verlag", 1993.