ILP: Induction Logic Programming

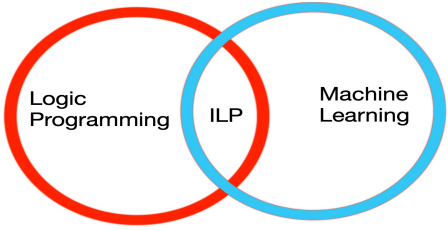
Francesco Cassano

2022

Argomenti

Introduzione a ILP

- La **Programmazione Logica Induttiva** in inglese **Inductive Logic Programming** (**ILP**) è una branca relativamente nuova nel Machine Learning.
- ILP nasce come l'unione tra Programmazione Logica e Machine Learning.
- L'obiettivo è **indurre un ipotesi** in grado di **generalizzare gli esempi** partendo da una conoscenza di background.



Il ragionamento in ILP

Esistono 3 tipi di ragionamento

- Ragionamento deduttivo
- Ragionamento induttivo
- Ragionamento abduttivo

Di base ILP utilizza il ragionamento **INDUTTIVO**
In alcuni casi può valere la relazione Induzione = Abduzione + Giustificazione

I programmi logici

I programmi logici non agiscono come la programmazione tradizionale

- Utilizzano linguaggi logici del primo ordine per rappresentare ed elaborare informazioni
- Il programmatore descrive la struttura del programma e dei dati, ma non deve specificare come risolverlo
- Usato principalmente per provare teoremi matematici
- Utilizza deduzione

L'importanza dei sistemi ILP

Sebbene ILP sia un'area quasi di nicchia nel ML, è molto apprezzata nella ricerca e in ambiti operativi come:

- Formulazione di teorie scientifiche
- Robotica
- Visione artificiale
- Cura e trasformazione dei dati
- Apprendimento dalle traiettorie

ILP vs ML

Pur essendo un algoritmo di ML, i principali ”concorrenti” di ILP sono altri algoritmi di ML:

- SVM
- Decision Tree
- Neural Network

ILP vs ML

Pro di ILP

- Induce regole tramite esempi e conoscenza pregressa
- Struttura simbolica
- Utilizzo efficiente dei dati
- Conoscenza human-friendly e trasferibile

Contro di ILP

- Acquisisce informazioni solo in linguaggio logico
- Soffre molto della presenza di errori negli esempi
- Soffre molto la mancanza di parte dei dati negli esempi
- Individuare i giusti Bias è difficile

ILP vs ML

Pro di ML

- Imparano tramite inferenza statistica
- Tolleranti agli errori o ai dati mancanti
- Accettano anche dati grezzi non formattati
- Non bisogna definire dei formalismi (Assenza di Bias) difficili da individuare

Contro di ML

- Servono molti dati
- Sono una scatola nera

I Fondamentali di ILP

Le componenti fondamentali di ILP sono:

- Background Knowledge, conoscenza pregressa, B
- Positive Examples, esempi positivi, E^+
- Negative Examples, esempi negativi, E^-
- Examples set, insieme degli esempi, $E = E^+ \cup E^-$
- Hypotesis, programma target, H
- Hypotesis space, spazio delle ipotesi, $\mathcal{H}$
- Programma logico, programma ottenuto dall'unione di B e H, (B, H)

I Fondamentali di ILP

Condizioni fondamentali di ILP

- Proprietà di copertura degli esempi: $B \cup H \vdash e^+$
- Proprietà di completezza del programma logico: $\forall e^+ \in E^+, B \cup H \models e^+$
- Proprietà di consistenza del programma logico: $\forall e^- \in E^-, B \cup H \not\models e^-$

Inoltre bisogna che il Background Knowledge B sia consistente:

- $\exists e^+ \in E^+ | B \not\models e^+$

Le basi del linguaggio logico

ILP utilizza il linguaggio logico che assicura determinati vantaggi:

- permette programmazione dichiarativa (Prolog)
- formalismi validi per la rappresentazione della conoscenza (**logica del primo ordine**), e la semantica non ha ambiguità, il formalismo utilizzato è quello delle clausole della logica di Horn.
- fondamenti di analisi e verifica dei sistemi, quali **model checking** e **theorem proving**

I linguaggi logici permettono, tramite il motore di inferenza, il controllo dei fatti in modo che la conoscenza sia consistente

Le componenti del linguaggio logico

- **Costanti**: entità del dominio note a priori, sono considerate funzioni con arità 0.
- **Variabili**: entità del dominio **non** note a priori.
- **Funzioni**: **identifica univocamente un oggetto** del dominio mediante una relazione tra altri n oggetti del dominio.
- **Predicato**: **definisce una relazione generica** tra più oggetti del dominio, che può essere vera o falsa.

Le componenti del linguaggio logico

- **Atomi o formule atomiche**: applicazione di un simbolo predicato con n termini, $p(T_1, ..., T_n)$
- **Espressione o formula**: sequenza di simboli appartenenti all'alfabeto.
- **Formule ben formate (fbf)**: formule composte da una o più formule atomiche, usando connettivi e quantificatori.
- **Clausole**: sono espresse nella forma $A : -B_1, ..., B_n$, dove A è la testa della clausola e B_i è il corpo della clausola.
- **Fatti**: sono clausole con il corpo vuoto.
- **Clausole ground**: sono clausole senza variabili.

La sussunzione

Una proprietà del linguaggio logico fondamentale in ILP è la sussunzione:
Si dice che date due clausole c_1 *e*
c_2*, allora* c_1 $\theta-$*sussume* c_2 *se e solo se esiste una sostituzione* θ *tale che* $c_1\theta \subseteq c_2$.
c_1 è una generalizzazione di (sussume) c_2 e
c_2 è specializzazione di (è sussunto da) c_1 sotto sussunzione.
Applicando la sostituzione θ ad un termine, atomo o clausola
si può ottenere un'istanza di termine, atomo o clausola
in cui ogni occorrenza di una variabile V_i è istantaneamente
sostituita da un termine t_i.

Universo di Herbrand in ILP

In logica matematica, per ogni linguaggio formale, come il linguaggio logico del primo ordine, con un insieme di termini, individuiamo:

- **Universo di Herbrand**: l'insieme dei termini
- **Base di Herbrand**: l'insieme di tutte le atomiche ground che possono essere composte formando predicati dai termini dell'universo di Herbrand.
- **Interpretazione di Herbrand**: sottoinsieme della base di Herbrand.
- **Modello di Herbrand**: è un insieme di clausole modello. Una clausola è un modello di una interpretazione di Herbrand se e solo se la sussunzione delle variabili del body è contenuta nell'interpretazione stessa e la testa è un elemento dell'interpretazione.

La semantica

La semantica è l'impostazione generale per ILP che definisce come sono composti gli insiemi fondamentali, ovvero B, E e H.
Abbiamo 2 tipi di semantica:

- la semantica **monotona** (semantica normale)
- la semantica **non-monotona**

La semantica monotona

Nella semantica monotona B è l'insieme della conoscenza posseduta a priori, e H è l'insieme delle clausole ipotesi, mentre l'insieme degli esempi è così definita $E = E^+ \cup E^-$ ciò significa che contiene sia esempi positivi che negativi.
Le condizioni da rispettare per la semantica monotona sono:

- **Condizione di soddisfacibilità a priori:** $B \cup E^- \not\models \square$
- **Condizione di soddisfacibilità a posteriori:** $B \cup H \cup E^- \not\models \square$
- **Condizione necessaria a priori:** $B \not\models E^+$
- **Condizione sufficiente a posteriori:** $B \cup H \models E^+$

Definizione formale delle condizioni della semantica monotona

Le condizioni da rispettare per la semantica monotona sono:

- **Condizione di soddisfacibilità a priori:** $\forall e^- \in E^-$ *è falsa nel* $\mathcal{M}^+(B)$
- **Condizione di soddisfacibilità a posteriori:** $\forall e^- \in E^-$ *è falsa nel* $\mathcal{M}^+(B \cup H)$
- **Condizione necessaria a priori:** $\exists e^+ \in E^+$ *falsa nel* $\mathcal{M}^+(B)$
- **Condizione sufficiente a posteriori:** $\forall e^+ \in E^+$ *è vero nel* $\mathcal{M}^+(B \cup H)$

La semantica non monotona

Nella semantica non-monotona l'insieme della conoscenza B è un insieme di clausole definite e l'insieme delle ipotesi H è un insieme di clausole generali, ma l'insieme degli esempi E è vuoto.
Ma quindi senza E^+ e E^- come funziona ILP?
E^+ e E^- vengono solo ridefinite come:

- $E^+ \subseteq B$
- $E^- = \forall e \notin B$

Questo dipende dall'interpretazione a **mondo chiuso**, ovvero tutto ciò che non fa parte della conoscenza di background è considerato falso.

Definizione formale delle condizioni della semantica non-monotona

Le condizioni da rispettare per la semantica non-monotona vengono ridefinite come segue:

- **Validità**: $\forall h \in H$ *è vera in* $\mathcal{M}^+(B)$
- **Completezza**: *se un clausola generale* g *è vera in* $\mathcal{M}^+(B)$ allora $H \models g$
- **Minimalità**: $\nexists G \subseteq H$ *che sia valido e completo*

Relazioni di generalità

Due relazioni sono particolarmente importanti quando parliamo di ILP e algoritmi ILP:

- Generalizzazione
- Specializzazione

Possiamo quindi affermare quanto segue:
Un'ipotesi G è generale e S è se e solo se $G \models S$ ma $S \not\models G$, ovvero G implica semanticamente S ma non vale il contrario.
S è detta anche più specifica di G.

Proprietà del linguaggio conseguenza della relazione di generalità

Con la relazione di generalità ed alla sussunzione possiamo definire diverse nuove proprietà all'interno del linguaggio, ma le più importanti per l'apprendimento sono:

- **Equivalenza**: si usa per determinare che, semanticamente, il valore di una clausola è uguale ad un altra. Quando accade ciò è vera sia $c_1\theta$ -sussume c_2 e quindi $c_1 \models c_2$, sia $c_2\theta$ -sussume c_1 quindi $c_2 \models c_1$, e se una è implicazione dell'altra allora sono equivalenti.
- **Riduzione**: Quello delle equivalenze è in realtà un problema in ILP. Viene introdotta la riduzione r come un sottoinsieme minimo di letterali di c tale per cui r e c siano equivalenti. Il sistema finale dovrebbe contenere solo clausole ridotte.
- **Reticolo**: L'insieme delle clausole ridotte forma un reticolo, questo significa che ogni coppia di clausole fa parte di un reticolo, quindi ogni coppia di clausole ha un limite minimo superiore (lub) e un limite inferiore massimo (glb).

Datalog

Datalog è un paradigma che serve a definire relazioni tramite regole.
Datalog è un sottoinsieme dei programmi Prolog, ciò significa che impone delle restrizioni ai normali programmi logici, ovvero:

- nessun termine complesso come argomento dei predicati
- accetta solo clausole di Horn
- restrizioni sull'utilizzo delle negazioni

Inoltre, Datalog implementa programmi **dichiarativi**, di conseguenza:

- riordinare le clausole non cambia il programma
- garantisce la terminazione

Ma ciò comporta una **eccessiva riduzione** nella **espressività del linguaggio**.
In compenso permette di ridefinire i problemi come problemi di **soddisfacibilità**.

Higher Order

Higher Order è un particolare paradigma che permette a delle funzioni di accettare come parametri altre funzioni e/o restituire funzioni come risultato.
I vantaggi di Higher Order sono:

- una migliore rappresentazione dei dati, grazie alla maggiore espressività.
- utilizzo del lambda calcolus, i predicati possono essere usati come termini.
- riduzione della complessità.
- aumento della accuratezza.

Esempio di Higher Order:
$f(A, B) :- map(A, B, f1)$.
$f1(A, B) :- reverse(A, C), tail(C, D), reverse(B, D)$.

Probabilistic ILP

ILP modella fatti e dati relazionali, ma non può gestire incertezza.
Probabilistic ILP (PILP) nasce proprio per gestire l'incertezza estendendo la normale semantica con impostazioni probabilistiche.
Le basi probabilistiche introdotte sono:

- $\mathcal{X}$ è una variabile casuale del dominio, $domain(\mathcal{X}) = \{x_1, x_2, ..., x_n\}$
- $P(\mathcal{X})$ è la distribuzione di probabilità
- $P(x_1, x_2, ..., x_n)$ con $n > 1$ è chiamata *distribuzione di probabilità congiunta*
- la *probabilità condizionata* è definita tramite la *legge di Bayes* ovvero $P(X|\,Y) = \frac{P(Y|X)P(X)}{P(Y)}$, che in caso ci sia una variabile indipendente, diventa $P(X,\,Y|Z) = P(X|Z)P(\,Y|Z)$

Probabilistic ILP

L'introduzione della probabilità diventa centrale, quindi:

- Le clausole sono annotate con informazioni probabilistiche, come le probabilità condizionali, diventando clausole di Horn probabilistiche
- Le variabili possono essere quelle della logica dei predicati o quelle probabilistiche, che rappresentano rispettivamente l'incertezza logica e quella probabilistica
- Ogni volta che viene composto un nuovo dato ne viene calcolata la probabilità
- La relazione di copertura diventa un calcolo di probabilità condizionata ovvero $cover(e, H \cup B) = P(e|H, B)$

Il problema di ILP viene ridefinito con la nuova relazione di copertura, e diventa:
Dato un linguaggio logico-probabilistico $\mathcal{L}_{\mathcal{H}}$ e un insieme E di esempi su un linguaggio $\mathcal{L}_{\mathcal{E}}$, trovare l'ipotesi H in $\mathcal{L}_{\mathcal{H}}$ che massimizza $P(e|H, B)$. Sotto l'ipotesi di variabili distribuite in modo identico e indipendente, ciò risulta nella massimizzazione di

$$P(e|H, B) = \prod_{e \in E} P(e|H^*, B) = \prod_{e \in E} covers(e, H^* \cup B)$$

Declarative Bias

I sistemi ILP fanno uso di linguaggi logico con un infinito potenziale espressivo, pertanto la definizione dei **bias** è una parte fondamentale in ILP.
Una delle ragioni dell'importanza dei bias è proprio regolare la "TROPPA" conoscenza.
La "troppa" conoscenza è dovuta a ridondanze ed equivalenze.
Provoca grossi problemi come grave deterioramento delle performance o un apprendimento errato.
I bias sono fondamentalmente delle **restrizioni** sul linguaggio.
Anche se teoricamente possono essere infiniti sono difficili da inserire in un sistema perché anche solo uno è in grado di stravolgere il risultato.
Esistono 2 tipi di bias

- bias sintattico
- bias semantico

I bias sintattici

I bias sintattici definiscono l'insieme delle ipotesi ben formate agendo sulla formula delle clausole.
In precedenza i sistemi ILP impiegavano un bias sintattico implicito, adesso è gestito in modo attivo.
Il vantaggio di questo formalismo generale è che il bias linguistico può essere disaccoppiato da particolari implementazioni ILP, di conseguenza, diventa un vero e proprio parametro del sistema.
In questo tipo di sistema basta cambiare pochi parametri per rendere il linguaggio molto più espressivo.

I bias semantici

I bias semantici sono restrizioni poste sul comportamento delle ipotesi indotte.
Spesso si utilizzano per fornire indicazione per efficientare i programmi logici tramite *predicate invention*
Esistono due principali tipi di bias semantici la dichiarazione di modalità e le metaregole

I bias semantici - Dichiarazione di modalità

Le dichiarazioni di modalità forniscono istruzioni per l'utilizzo di una clausola ed è espressa come:
$mode(recall, predicate(m_1, ..., m_n))$
recall indica quante volte un predicato può essere richiamato in una clausola
$predicate(m_1, ..., m_n)$ indica come si usa un predicato e gli argomenti di cui ha bisogno.
Per gli argomenti si può specificare anche se gli argomenti sono di *input, output* o *ground* usando i simboli $+, -, \#$

I bias semantici - Dichiarazione di modalità

L'utilizzo della dichiarazione di modalità assicura 2 vantaggi

1. se ci troviamo in un contesto di apprendimento a predicato singolo e i predicati di sfondo e le loro modalità sono specificati correttamente, il learner può garantire la terminazione e conformità.
2. il learner può ottimizzare la sua ricerca quando risponde alle query, infatti, definito le modalità, al learner basta esplorare una volta lo spazio delle ipotesi per la soluzione.

I bias semantici - meta-regole

Le meta-regole definiscono una struttura ammissibile dei predicati indotti con l'obiettivo di ridurre lo spazio delle ipotesi.
Le meta-regole sono clausole di ordine superiore che introducono il concetto di variabili di secondo ordine.
Una meta-regola è definita come una clausola logica, per esempio:

$$X(A, C) := Y(A, B), Z(B, C)$$

Dove X, Y, Z sono le variabili di secondo ordine, sostituibili dai predicati
A, B, C sono variabili del primo ordine

I bias semantici - meta-regole

Differentemente dalle dichiarazioni di modalità le meta-regole sono affermazioni logiche, sono molto potenti e hanno un più ampio campo di applicazione.
Nonostante ciò sono pochi gli algoritmi per individuare le migliori meta-regole

Recursion (Ricorsione)

La ricorsione è una caratteristica fondamentale in un linguaggio logico.
Consente la produzione di una infinita quantità di clausole tramite un programma finito, ad esempio:
$f(A, B) : -tail(A, C), empty(C), head(A, B).$
$f(A, B) : -tail(A, C), f(C, B).$
Questo procedimento genera liste a prescindere dalla lunghezza tramite passo base e passo induttivo.
Ricorsione è usata per definire predicati complessi tramite l'utilizzo di predicati semplici.
Per un sistema ILP è molto difficile generalizzare una ricorsione.
Sono necessari vari esempi che comprendano il caso base e vari casi induttivi.

Predicate Invention

ILP usa, solitamente, background fatti a mano da esperti, che nonostante ciò potrebbero mancare di espressività.
L'obiettivo dell'invenzione dei predicati (PI) è che un sistema ILP inventi automaticamente nuovi simboli predicati ausiliari.
I vantaggi del PI sono una riduzione della duplicazione del codice e miglioramento della leggibilità e dell'espressività.
Il problema è la difficoltà di insegnare al sistema quando e come generare un predicato inventato.

Predicate Invention

Anche se è necessario un criterio per decidere se PI serve, non è facile da definire.
Solitamente viene impiegato in contesti in cui serve **riformulare** predicati con altri che migliorano espressività e leggibilità, o fare **Bias Shift** ovvero specializzare un'ipotesi troppo generica e renderla coerente con gli esempi.
Per generare nuovi predicati PI necessita della definizione di dichiarazione di modalità, che avviene in 2 modi:

- definizione di bias appositi
- definizione di meta-regole

I sistemi che più lo utilizzano sono sistemi ASP e MIL.
Più in generale, tutti i sistemi che utilizzano abduzione per la generalizzazione sono in grado di inventare nuovi predicati.

Introduzione

Il problema principale degli ILP è la ricerca efficiente di ipotesi valide in un ampio spazio di ipotesi.
I primi metodi di ricerca di ILP sono stati il metodo **top-down** e il **bottom-up**, e successivamente un metodo ottenuto dalla **loro combinazione**.
Sono per lo più algoritmi basati sulla definizione di generalità che gestiscono attraverso le θ-sostituzioni Più recentemente sono stati introdotti il metodo MIL e ASP, che si basano sulla definizione dei bias semantici.

Metodo Top-down

Gli approcci top-down partono da un'ipotesi generale e poi la specializzano.
Questo tipo di sistemi opera con algoritmi *divide et impera*
Vantaggi: i sistemi top-down sono in grado di apprendere programmi ricorsivi
Svantaggi: creano troppe ipotesi e non è detto che coprano gli esempi.

Metodo Top-down

Il processo i specializzazione si basa sulla ricerca di clausole sempre più specifiche che coprano il maggior numero possibile di esempi positivi escludendo i negativi.
L'approccio top-down è utile se si ha una buona conoscenza del dominio, se la knowledge base è troppo generica le regole non saranno abbastanza accurate e utili.
Un esempio di questo approccio è la **bottom-clause propositionalisation** il cui processo consiste nella conversione esplicita di un insieme di dati relazionali in un insieme di dati proposizionali.

Metodo Bottom-Up

Gli approcci bottom-up adottano una strategia di costruzione delle clausole generalizzando gli esempi di addestramento.
Generalmente utilizzano dei reticoli basati su sostituzioni-θ per trovare la generalizzazione meno generale.
Vantaggi: tipicamente molto veloci
Svantaggi: ipotesi inutilmente lunghe, difficoltà con ipotesi ricorsive o troppi predicati contemporaneamente, non supportano facilmente la *predicate invention*
Le due tecniche peculiari di questo approccio sono la **relative-least-general generalization** o **RLGG** e l'**inverse-entailment** o **IE**.

Metodo Bottom-Up

La **relative-least-general generalization**, come previsto dalla ricerca bottom up, parte da una clausola più specifica del background e per arrivare a una molto più generale.
La clausola generale RLGG sarà in grado di dimostrare le clausole ground tramite operazione di θ-sussunzione.
Gli esempi positivi vengono utilizzati per la ricerca delle ipotesi nello spazio delle ipotesi.
Gli esempi negativi servono per ridurre lo spazio delle ipotesi e introdurre vincoli nelle ipotesi
Per la risoluzione delle RLGG viene introdotto l'operatore V. Facendo riferimento ad un reticolo la soluzione è quella alla base della V mentre le due clausole su cui è calcolato sono indicate dalle braccia.
Quest'operatore permette di semplificare le teorie scartando le clausole ridondanti.

Metodo Bottom-Up

L'**inverse-entailment** è una forma di generalizzazione che ha come obiettivo la creazione di regole che possono indurre sia conoscenza di background che esempi positivi.
IE si basa sull'assunto che data una ipotesi e della conoscenza di background da cui è possibile implicare l'esempio, se l'esempio viene negato sarà possibile implicarlo tramite conoscenza di background e l'ipotesi negata.
IE funziona solo se background, esempi e ipotesi utilizzano un unico modello di dichiarazione.

Metodo misto

L'approccio misto è stato introdotto da Muggleton con la creazione di PROGOL.
L'approccio misto si basa sull'utilizzo di metodi top-down e bottom-up combinati.
Nell'algoritmo proposto, PROGOL, le ipotesi vengono create da algoritmi bottom-up (specific-to-general), successivamente tramite algoritmi come A^* le ipotesi sono migliorate utilizzando algoritmi top-down (general-to-specific).
Questo tipo di approccio nasce per migliorare le criticità dei due approcci usati singolarmente.

MIL

MIL agisce ad un livello di astrazione più alto rispetto alle metodologie top-down e bottom-up viste finora, il **meta-livello**.
Il problema di ricerca in MIL viene codificato come un problema di ricerca dichiarativa. Vengono definite meta-regole in un meta-programma che viene utilizzato per descrivere come le ipotesi dovranno essere formate per poi lasciare ad un risolutore standard il compito di comporre le ipotesi in automatico.
Vantaggi: possono apprendere facilmente programmi ricorsivi e ottimali (le meta-regole favoriscono l'utilizzo di PI), non sono vincolati al solo utilizzo di Prolog.
Svantaggi: scalano male su domini complessi o su clausole troppo lunghe.

MIL - Meta-regole

Le meta-regole sono definite come:

$$metarule(name, Subs, (Atom : Body))$$

Una meta-regola è molto simile ad una clausola ordinaria ma comprende elementi che identificano variabili di ordine superiore.
Avere una meta-regola significa introdurre un livello di astrazione maggiore nel programma.

ASP

L'**Answer Set Programming** (**ASP**) comprende una famiglia di approcci basata su impostazione non-monotona e per programmi logici dichiarativi.
Un normale sistema ILP non è abbastanza dichiarativo, basti pensare che riordinando le clausole è possibile modificare l'efficienza, la terminazione o anche la correttezza del programma logico. Ma non succede in MIL e ASP.
ASP non è vincolato ad un unico modello di stabile e definito, ne supporta uno, nessuno e molti.
Ciò permette ad ASP di supportare costrutti linguistici come disiunzioni nella testa delle clausole, regole di scelta o vincoli sia deboli che forti.
Vantaggi: è portato per ragionamenti di senso comune, è molto efficace per la risoluzione di problemi complessi.
Svantaggi: può essere dispendioso a livello di risorse, la definizione dei vencoli è fondamentale e difficile quanto quella dei bias semantici.

ASP

Degli esempi di clausole peculiari di ASP sono:
$0\{in(V0, V1)\}1 := edge(V0, V1).$
Questa è una regola di scelta, che significa che un atomo può essere vero. In questo esempio, la regola indica che può esistere un bordo tra il vertice V1 e V0.
$:= not\ reach(V0), node(V0).$
$:= V1! = V2, in(V0, V1), in(V0, V2).$
Queste due regole sono vincoli rigidi, impongono essenzialmente vincoli di integrità.
Il primo vincolo rigido afferma che è impossibile avere un nodo non raggiungibile.
Il secondo vincolo rigido stabilisce che è impossibile avere un vertice con due spigoli in entrata da nodi distinti.

ASP

Gli approcci ASP seguono all'incirca tutti lo stesso procedimento:

- definizione dei bias e generazione dell'insieme di regole dello scheletro
- vengono generate delle abduzioni associate alle scheletro
- ciò che è stato generato viene messo in un risolutore ASP

Il risolutore generalizza le regole delle scheletro secondo i bias definiti dall'utente, in modo che non vengano generate regole equivalenti. Le regole ordinate vengono inserite in un subset R^o.

ASP

Distinguiamo due approcci di apprendimento: l'approccio coraggioso e l'approccio prudente.
Gli approcci coraggiosi generano un programma tale che almeno un answer set copra tutti gli esempi.
Gli approcci prudenti generano un programma tale che tutti gli esempi siano coperti in tutti gli answer set.

FOIL

FOIL è un algoritmo supervisionato, con approccio Top-Down.
L'algoritmo è una generalizzazione degli algoritmi *sequential-covering* e *learn-one-rule*.
L'algoritmo si divide quindi in due parti riconoscibili in due cicli innestati:

- il ciclo esterno aggiunge regole finché ci sono esempi positivi da dimostrare
- il ciclo interno aggiunge i migliori letterali alla regola fino a che non avrà escluso tutti gli esempi negativi

FOIL funziona solo con background composto di clausole ground.

FOIL

Per capire se il letterale aggiunto è l'opzione migliore viene calcolato tramite una gain function che quantifica il guadagno confrontando le coperture degli esempi prima e dopo l'aggiunta della clausola.
Per avere maggiore attendibilità il guadagno viene calcolato usando clausole ground di E^+
L'algoritmo FOIL permette di apprendere teorie ricorsive.
FOIL risolve il problema dell'ordine generalizzando un metodo di ordinamento sia per le clausole nel programma che per i predicati nelle clausole e ciò previene loop e duplicati.
Essendo essenzialmente un algoritmo greedy, potrebbe creare troppe regole o regole eccessivamente lunghe.

GOLEM

GOLEM è il principale approccio bottom-up.
GOLEM utilizza il calcolo delle RLGG per generalizzare coppie di clausole in Background, valuta quelle che coprono più esempi positivi e riduce le clausole utilizzando gli esempi negativi.
Per questo GOLEM ha bisogno che esempi e background contengano solo atomi ground.

GOLEM

Il problema delle RLGG è che possono essere infinite, possono generare clausole enormi e, non permette teorie ricorsive o complesse.
Per risolvere questi problemi GOLEM fa uso di restrizioni, ovvero:

- *generative clauses*: ogni variabile della testa deve apparire nel corpo.
- *ij-determinacy*: tutti i letterali nel corpo della RLGG possono contenere al massimo j input, e le variabili possono avere una profondità di al massimo i.
- *extentinal background knowledge*: background contiene solo atomi ground.
- *h-easy model*: se il background non contiene solo atomi ground, si hanno a disposizione h iterazioni per esplicitare la conoscenza implicita.

Un limite di GOLEM è che non è in grado di riconoscere clausole ricorsive, spesso anche avendo molti esempi

GOLEM

L'algoritmo di GOLEM è di per sé abbastanza semplice, e prevede:

1. Per costruire C_e, il corpo dell'esempio viene riempito con tutti gli atomi presenti nella conoscenza di base e poi vengono filtrati i termini del corpo ritenuti logicamente superflui, che non fanno parte della catena tra variabili di input e output, o dimostrano esempi negativi.
2. Selezionare coppie s in modo casuale dagli esempi positivi non coperti
3. Calcolare RLGG delle varie coppie e scegliere quella che copre più esempi positivi, S
4. Finché il numero di esempi positivi coperti aumenta
   1. selezionare le clausole degli esempi positivi non coperti una alla volta
   2. calcolare la RLGG della clausola selezionata con S
   3. trovare la migliore e sostituire S
5. Eliminare letterali inutili tramite riduzione funzionale e esempi negativi
6. Ripetere dal passo 2, finché tutti gli esempi positivi sono coperti

PROGOL

PROGOL è un particolare implementazione di sistema ILP che utilizza sia una metodo bottom-up che uno top-down per la ricerca delle ipotesi.
I metodi top-down (in combinazione con IE) è utilizzata per comporre clausole generali.
I metodi bottom-up per la costruzione di clausole con la massima compressione.
Utilizza l'*inverse entailment* (IE) per la costruzione delle ipotesi.
Una peculiarità di PROGOL è che necessita della dichiarazione di modalità, che consistono semplicemente nell'imporre vincoli sintattici sugli atomi di testa e del corpo delle clausole ipotizzate.

PROGOL

I sistemi ILP imparano tramite deduzione, ovvero devono dimostrare che dato un background B, l'ipotesi H copre l'esempio e se e solo se è dimostrata la relazione $B \wedge H \models e$.
Nella logica induttiva si parte avendo a disposizione il background B e l'esempio e e l'obiettivo è indurre l'ipotesi H.
L'inverse entailment si basa sull'osservazione che $B \wedge H \models e$ è equivalente a $B \wedge \neg e \models \neg H$.
Questa relazione permette di calcolare l'ipotesi H che coprirà l'esempio rispetto a B.

PROGOL

L'algoritmo principale di PROGOL è di per sé abbastanza semplice:

1. selezionare un esempio positivo e che combaci con la testa del concetto target
2. costruire la *mostSpecificClause* utilizzando le dichiarazioni di modalità e tipo specificate dall'utente
3. trovare la miglior generalizzazione per la *mostSpecificClause*
4. ripetere le operazioni finché non verranno coperti tutti gli esempi positivi.

PROGOL

É facile intuire che il fulcro di PROGOL si concentra attorno alla costruzione della *mostSpecificClause*. Questa avviene:

1. Inizializzare la testa della clausola con la testa della clausola di esempio e ”variabilizzata”
2. Inizializzare l'insieme V con tutte le coppie (variabili di input, termine di e corrispondente)
3. Finché non si raggiunge la lunghezza massima della clausola
   1. Trova tutte le variabilizzazioni con variabili di V per l'input, conformi alla restrizione di tipo e trova tutte le sostituzioni che sostituiscono tutte le variabili con termini che sono stati precedentemente associati a queste variabili.
   2. tentare di provare ogni letterale risultante con un massimo di h passi di risoluzione
   3. Se è stata provata la teoria la *mostSpecificClause* viene sostituita e i termini sono sostituiti con variabili e salvati in V

PROGOL

Molto importante in PROGOL è l'euristica per la compressione delle clausole.
L'euristica garantisce che la ricerca trovi la clausola migliore.
la formula dell'euristica è $score = p(n + c + h)$ dove
p è il numero di esempi positivi coperti
n è il numero di esempi negativi coperti
c è la lunghezza della clausola
h è una stima del numero di letterali mancanti alla massima compressione

PROGOL

Come per GOLEM le clausole create non possono essere ricorsive ma sono molto potenti e complesse non essendo vincolate a contenere solo i letterali della testa.
Le performance di PROGOL sono paragonabili a quelle di FOIL, ma l'algoritmo best-first utilizzato da PROGOL diventa costoso quando la lunghezza del corpo della regola diventa troppo lungo.
PROGOL non può creare nuovi predicati a meno che non vengano specificati con la dichiarazione di modalità.
PROGOL ha inoltre il problema che può definire due diverse *modeb* che non sono dimostrabili a priori nella conoscenza di base.

Metagol

Metagol è un sistema ILP basato su MIL.
Metagol è sostanzialmente un meta-interprete per Prolog.
MIL permette di supportare l'apprendimento di definizioni ricorsive e di sistemi puramente simbolici.
MIL introduce anche l'apprendimento automatico di una sotto-definizione, consentendo la decomposizione di un predicato complesso in una gerarchia di parti riutilizzabili. Questo è possibile grazie all'invenzione dei predicati.
Per utilizzare il meta-interprete, Metagol richiede delle meta-regole in input.
Le meta-regole sono simili alle dichiarazioni di modalità di PROGOL, e sono sostanzialmente dei bias sintattici.
Le meta-regole sono molto simili alle normali clausole, possono presentare variabili higher order.
Per supportarle Metagol introduce un nuovo livello di astrazione.

Metagol

Una meta-regola è definita come:

$$metarule(Nome, Subs, (Testa :- Corpo))$$

dove banalmente *Nome* è il nome della meta-regola,
Subs indica le variabili di ordine superiore,
Testa indica gli atomi che compongono la testa della regola
Corpo indica gli atomi che compongono il corpo della regola

Metagol

L'algoritmo di Metagol fa da meta-interprete Prolog.
Per prima cosa, tramite Prolog, cerca di dedurre gli atomi positivi.
Se Progol non riesce a dimostrare l'atomo con il background knowledge e le regole indotte fino a quel momento, allora ne vengono indotte altre.
Per indurre una nuova regola, Metagol esegue una sostituzione parziale delle meta-regola, sostituisce una variabile di ordine superiore in testa con il predicato dell'esempio lasciando invariati gli argomenti.
Successivamente cerca di dimostrare la regola sostituendo tutte le variabili di ordine superiore nel corpo, con i letterali provenienti dalla conoscenza di base e dalle regole indotte.

Metagol

Se ancora non si riesce a dimostrare l'esempio, allora viene utilizzata la predicate invention.
L'astrazione permette di utilizzare le variabili di ordine superiore delle meta-regola per risolvere ricorsivamente il problema, scomponendolo in sotto-problemi che vengono risolti uno alla volta nello stesso modo.
Quando una regola viene indotta si controlla se è coerente.
Le regole coerenti vengono aggiunte al background.
Se la meta-regola utilizzata non produce ed esegue un nuovo ramo nell'SDL-tree.

Metagol

Le tecniche induttive sfruttate da Metagol seguono una logica completamente diversa dagli altri sistemi ILP.
La prima differenza è l'assenza di vincoli dovuti al problema della generalità e la mancanza di veri bias.
I bias per descrivere la composizione delle regole sono sostituiti da meta-regole che sono più versatili e potenti.
Le meta-regole permettono anche di indurre regole ricorsive e di inventare predicati per necessità
La logica di Metagol dà la possibilità di creare ragionamenti più complessi.
Per contro Metagol è limitato all'induzione di regole composte da termini con un'arità massima di 2 e un corpo di dimensioni non superiori a 3.
La scelta di un insieme collaudato di meta-regole per l'induzione, pur essendo più semplice della definizione di bias linguistici, rimane un problema da risolvere.

ILASP

ILASP utilizza il paradigma ASP per indurre nuove regole.
ASP è un paradigma dichiarativo, ciò permette di imparare specifiche logiche di un programma.
I programmi appresi da ILASP possono includere altri tipi di regole che non sono disponibili in Prolog, come le regole di scelta e i vincoli hard e weak.
ILASP permette la combinazione di framework coraggioso e cauto di ASP, il che lo rende perfetto per apprendere l'intera classe dei problemi ASP.
ILASP inoltre ammette contesti non-monotoni.
ILASP ha diverse versioni con le quali cerca di sopperire le mancanze delle precedenti, attualmente è alla versione ILASP 3.

ILASP

ILASP, come altri sistemi ILP, parte da una conoscenza di background B, esempi E e le dichiarazioni dei bias.
ILASP parte dal presupposto che, tramite la dichiarazione di modalità, è possibile costruire un insieme finito di regole, dal quale sarà possibile ricavare i programmi ASP.
Nonostante ILASP abbia dinamiche differenti, background e bias hanno la stessa funzione che hanno in altri sistemi ILP, ma per gli esempi il discorso è diverso.
Mentre altri sistemi imparano a coprire ed escludere esempi positivi e negativi, in ILASP, questo approccio non funziona per via del contesto non monotonico, infatti le nuove regole potrebbero ”annullare” la copertura degli esempi precedentemente coperti.

ILASP

Essendo l'output di ASP un insieme di insiemi di risposte, ILASP apprende da esempi positivi e negativi di interpretazioni parziali, dalle quali ricava regole e vicoli.
Questi esempi sono sufficienti per apprendere qualsiasi programma ASP composto da regole normali, regole di scelta e vincoli rigidi, ma non sono sufficienti per apprendere i vincoli deboli, che richiedono una specificazione più precisa.
I vincoli deboli non sono altro che indicazioni che creano un ordine di preferenze sugli insiemi di risposta.

ILASP

ILASP è uno dei pochi sistemi ILP che tiene conto di esempi rumorosi.
Per gestire gli esempi non coperti o etichettati in modo errato, ILASP assegna a ciascun esempio una penalità, che rappresenta il costo per la mancata copertura dell'esempio.
La ricerca del programma ottimale minimizza la lunghezza del programma e la somma delle penalità degli esempi non coperti.
ILASP è anche in grado di costruire un sottoinsieme di esempi rilevanti per apprendere il programma corretto, riducendo la dimensione dell'insieme di esempi a pochi esempi rappresentativi di una classe di clausole.

ILASP

L'algoritmo di ILASP si compone di tre fasi principali:

1. pre-processing: mappa il compito di apprendimento in ingresso in un programma ASP;
2. learning: il programma ASP viene risolto da un motore ASP;
3. post-processing: estrae il programma appreso.

ILASP

L'algoritmo di ILASP è composto come segue:
L'insieme degli esempi non coperti, *Uncov*, si inizializza con tutti gli esempi.
Finché *Uncov* contiene elementi:

- Vengono individuati gli *esempi rilevanti*
- Gli *esempi rilevanti* vengono utilizzati per ricavare i vincoli per il risolutore ASP, i *coverage constraint*
- I vincoli vengono controllati, *implication check*. L'implication check controlla quali altri esempi sarebbero garantiti come scoperti se i vincoli di copertura non fossero soddisfatti
- Gli esempi rilevanti e i vincoli di copertura vengono usati come input per il risolutore ASP, che trova il programma ottimale in una esecuzione.
- I programma ottimale viene memorizzato in H, mentre in *Uncov* vengono memorizzati eventuali esempi non coperti.

ILASP

ILASP è in grado di gestire informazioni incomplete e incerte e apprendere da dati parziali o rumorosi, il che lo rende adatto ad applicazioni reali.
ILASP può incorporare la conoscenza di base nel processo di apprendimento il che permette di ridurre la quantità di dati necessari per l'apprendimento e migliorare l'accuratezza dei modelli appresi.
Il limite principale di ILASP è la sua complessità computazionale.
Un altro limite è la sua dipendenza dalle rappresentazioni logiche, che non lo rendono versatile come una NN.