ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Il Ragionamento Logico come Forma di Apprendimento: Sviluppo di Un Framework per ILP

Tesi di laurea in
Sistemi Autonomi

Relatore
Prof. Andrea Omicini

Candidato
Giovanni Maria Speciale

Correlatori
Prof.ssa Roberta Calegari
Dott. Giovanni Ciatto

Prima Sessione di Laurea
Anno Accademico 2020-2021

Sommario

Questa tesi si focalizza sulla programmazione logica induttiva (ILP) e, in particolare, in una sua ricontestualizzazione all'interno di ecosistemi tecnologici per lo sviluppo di applicativi di intelligenza artificiale (AI) moderni.

ILP è un paradigma per l'apprendimento automatico: sulla base di una conoscenza del dominio e una serie di esempi (positivi e negativi) nell'ambito di interesse – rappresentati utilizzando la logica –, un sistema ILP riesce a derivare un programma logico che generalizza tutti gli esempi positivi e nessuno degli esempi negativi. Il suo innesto all'interno di ecosistemi di intelligenza artificiale simbolica può portare allo sviluppo di nuovi scenari applicativi in cui la logica induttiva può diventare il ponte tra mondo simbolico e quello sub-simbolico.

In particolare, la tesi ha un duplice obiettivo. In primo luogo, una sistematizzazione dello stato dell'arte al fine di evidenziare gli approcci e le tecniche ILP esistenti ed eventuali tecnologie correlate. In secondo luogo, la progettazione e realizzazione di un modulo ILP all'interno di un ecosistema tecnologico per AI simbolica – 2P-Kt – fornendo una prima implementazione fruibile in contesti pervasivi—quali quelli richiesti dalle moderne applicazioni di AI. Detto modulo andrà a supportare i principali algoritmi classici per ILP – Golem, Progol e Metagol – tramite una tecnologia multi-paradigma, consentendo l'utilizzo di tecniche induttive sia come applicazione che come libreria riusabile.

A me stesso, ai miei sacrifici e alla mia tenacia che mi hanno permesso di raggiungere ogni obiettivo nella vita fino ad oggi.

Ringraziamenti

L'intero percorso di studi, e la realizzazione di questo elaborato, ha caratterizzato in me un periodo di profondo apprendimento, non solo a livello scientifico, ma anche personale. Pertanto, vorrei spendere due parole di ringraziamento nei confronti di tutte le persone che mi hanno sostenuto e aiutato durante questo periodo.

Prima di tutti, vorrei ringraziare il mio relatore, il Prof. A. Omicini, ed i rispettivi correlatori, la Prof.ssa R. Calegari e il Dott. G. Ciatto, per il loro indispensabile supporto e la loro preziosa disponibilità. Mi hanno fornito tutti gli strumenti necessari per intraprendere la strada giusta e portare a compimento la mia tesi, accrescendo le mie competenze.

Un sentito ringraziamento è rivolto alla mia famiglia, in particolar modo a mia madre, pilastro solido sul quale poter sempre fare affidamento. È anche grazie ai vostri consigli e al vostro sostegno durante i momenti di sconforto se sono riuscito a terminare il percorso universitario.

Un ringraziamento particolare è rivolto alla mia fidanzata Carmela per avermi trasmesso in ogni istante il suo amore e la sua immensa forza. Grazie per tutto il tempo che mi hai dedicato, grazie per la straordinaria pazienza dimostrata. Grazie per la tua immancabile presenza.

Infine, vorrei ringraziare tutti i colleghi con i quali ho intrapreso il percorso di studi magistrale presso il Campus di Cesena. La loro fantastica collaborazione ha giovato al miglioramento delle mie capacità e mi ha sempre proiettato in avanti positivamente. Degli amici più che dei semplici colleghi di studio.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

Nell'*Artificial Intelligence (AI)* il *Commonsense Reasoning*, o ragionamento basato sul buon senso, è la capacità di presumere il significato di determinate azioni che si sono verificate o che potranno verificarsi in futuro. Tale capacità, se abbinata ad un sistema tecnologico, è in grado di trarre giudizi e conclusioni similmente al ragionamento dell'essere umano. Lo studio del commonsense reasoning è oggi diviso in due macro approcci: *approcci knowledge-based* ed *approcci Machine Learning*. La programmazione logica si è rivelata un potente paradigma per concettualizzare il ragionamento comune e, per tale ragione, ha trovato un forte impiego come area di ricerca promettente nel campo dell'intelligenza artificiale. Lo scopo della ricerca è quello di ricreare un ragionatore automatico che, premesse delle regole logiche e conoscendo il risultato, generi delle ipotesi sensate a riguardo. Questo paradigma è oggi conosciuto come *Programmazione Logica Induttiva (ILP)*. Essendo ILP un paradigma che adotta un approccio basato sulla conoscenza, tenta di allineare il concetto di "pensiero umano" con il concetto di "pensiero macchina". Tuttavia, non è ancora stabilito che l'adozione della logica, nel contesto dell'IA, sia la scelta migliore per l'apprendimento. Quest'ultima osservazione viene rafforzata dalla presenza di differenti approcci che sfruttano il paradigma di ILP per la realizzazione del ragionatore automatizzato; la vasta eterogeneità di sistemi dunque, non consente la definizione di un approccio univoco basato sulla logica che sfrutti a pieno le potenzialità del paradigma.

Diversi approcci e tecniche alla base dello stato dell'arte, come evidenziato nel Capitolo 2, sono caratterizzate da differenti vantaggi e svantaggi che descrivono le caratteristiche del passo d'induzione all'interno della propria area d'esecuzione, ma non in modo universale. Inoltre, alcune tecniche sono presentate attraverso descrizioni puramente teoriche in articoli di ricerca, altre forniscono esempi teorici privi di riscontri pratici; fatta eccezione per qualche algoritmo più recente che offre una base di esempi pratici sull'induzione sfruttando esclusivamente un ragionatore logico come Prolog o ASP, è ancora oggi difficile trovare esempi di elaborazione

degli algoritmi alla base dell'arte che evidenzino e mostrino le reali differenze tra le tecniche e le vere e proprie caratteristiche dell'apprendimento per induzione logica. Il paradigma di ILP possiede le potenzialità per offrire al mondo dell'intelligenza artificiale un nuovo meccanismo di apprendimento che si distanzi dall'affermato ML; un meccanismo che goda di semplicità di utilizzo, facilità di comprensione da parte dell'essere umano e indipendenza da strutture dati dispendiose, in termini di tempo e spazio.

Obiettivi della Tesi. La tesi è realizzata con un duplice scopo.

Il primo è quello di dare una chiara e definita panoramica sulle diverse tecniche di induzione logica per permettere una maggiore comprensione del paradigma.

Il secondo è la realizzazione di un framework, attraverso il linguaggio multi-paradigma Kotlin, che consentirà di:

- indurre delle ipotesi basandosi su tre dei principali algoritmi alla base del paradigma dell'induzione;
- permettere l'esecuzione di uno o più algoritmi in modo da evidenziare i differenti percorsi di apprendimento e le diverse ipotesi indotte;
- per ciascun algoritmo, offrire la possibilità di eseguire induzioni diverse in sequenza.

Struttura della Tesi. La tesi è organizzata come segue.

Nel Capitolo 2, viene riassunto e descritto lo stato dell'arte di ILP, i suoi obiettivi e le sue componenti tecniche. Rappresenta la sezione in cui viene descritta ed approfondita la cronologia storica e le basi dell'Inductive Logic Programming, dagli inizi fino ad oggi.

Nel Capitolo 3 viene descritta la tecnologia 2P-Kt utilizzata per la realizzazione del framework e le tipologie di induttore implementate, in Golem, Progole Metagol, con i principali aspetti tecnici che le caratterizzano. Viene messa a fuoco la progettazione di ciascun sistema in termini strutturali ed algoritmici ed evidenziate le differenza tra essi.

Nel Capitolo 4 vengono evidenziati alcuni dettagli implementativi ai fini delle funzionalità e della coerenza dell'induzione, come l'utilizzo di particolari Tipi dalla gerarchia di 2P-Kt, la realizzazione di più data class ad-hoc e l'implementazione dei comandi dell'interfaccia a riga di comando, utili all'induzione.

Nel Capitolo 5 viene descritto l'orientamento di testing adottato e le principali classi di test sfruttate, la descrizione di coverage da un punto di vista progettuale ed induttivo e la descrizione dell'interfaccia a linea di comando per l'esecuzione dell'intero sistema.

Infine nel Capitolo 6 le conclusioni su quanto realizzato ed i possibili sviluppi futuri.

Capitolo 2

Stato dell'Arte

2.1 ML e ILP a confronto

L'induzione logica è un processo di inferenza dal particolare al generale. Nel ragionamento logico per induzione si conoscono le premesse e le conclusioni di un discorso e, a partire da queste, si formulano ipotesi sulle regole d'implicazione [23]. Ciò che viene generato da un ragionamento induttivo è un risultato incerto, non assoluto, in quanto non è detto che la regola indotta dal ragionamento sia effettivamente applicabile all'intero universo.

Quindi, in definitiva, un ragionamento logico per induzione si basa sull'osservazione di premesse o di fatti veri per ottenere ipotesi sulle relazioni d'implicazioni, o su regole, probabilmente vere. Da quanto appena osservato, ILP è vista come una forma innovativa di machine learning, rappresentata da una raccolta di tecniche per costruire programmi logici da esempi. I programmi logici costruiti comprendono tutti gli esempi positivi ma non comportano nessuno degli esempi negativi.

Quindi, da una prospettiva di ML, lo scopo è quello di mappare ogni esempio ad una valutazione della sua verità o falsità in base agli assiomi forniti al sistema.

ILP ha tre interessanti caratteristiche:

1. il programma appreso è una struttura simbolica esplicita che può essere ispezionata, compresa e verificata;

2. essere straordinariamente efficiente in termini di dati, in grado di generalizzare bene da una piccola manciata di esempi;

3. supportare l'apprendimento continuo e trasferibile, fornendo un modo economico di immagazzinare le conoscenze apprese (conoscenze di base).

ILP dunque, è diverso dalla maggior parte degli approcci machine learning. La maggior parte di questi, come decision trees, support vector machines (SVM) e neural netowrks, si basano sull'inferenza statistica. Al contrario, ILP si basa sull'inferenza logica e spesso su tecniche di ragionamento automatizzato e rappresentazione della conoscenza [3].

La programmazione logica induttiva estende la teoria e la pratica della logica computazionale sfruttando l'induzione piuttosto che la deduzione come modalità di inferenza di base. Inoltre può superare i due principali limiti delle tradizionali tecniche di ML:

- utilizzo di un formalismo limitato per la rappresentazione della conoscenza;
- difficoltà nell'utilizzo di conoscenze di base nel processo di apprendimento.

Tuttavia, un punto di forza dei più utilizzati approcci ML, come le reti neurali, è che, a differenza dei sistemi ILP, sono resistenti al rumore e all'ambiguità. Pertanto, i sistemi ILP tradizionali, dovrebbero essere re-implementati.

I sistemi basati su reti neurali non costruiscono una rappresentazione simbolica esplicita di un programma; apprendono una procedura implicita, distribuita nei pesi della rete, che produce i risultati desiderati.

Due caratteristiche interessanti:

1. tolleranza al rumore;
2. l'apprendimento può iniziare con input di pixel grezzi e non elaborati.

Due principali svantaggi:

1. la procedura implicita non è ispezionabile o leggibile dall'uomo;
2. le prestazioni diminuiscono bruscamente quando i dati del test sono significativamente più grandi dei dati di addestramento.

2.2 Costruzione di un sistema ILP

La costruzione di un sistema ILP richiede diverse scelte, che fanno parte del pregiudizio linguistico. Un bias induttivo è essenziale per l'apprendimento trattabile e tutti gli approcci di ML impongono un bias induttivo (Mitchell, 1997) [7]. Le scelte possono essere classificate come:

- *Impostazione dell'apprendimento*: come rappresentare esempi; la maggior parte dei sistemi impara da insiemi di fatti;

- *Linguaggio di rappresentazione*: come rappresentare la BK e le ipotesi; la maggioranza dei sistemi ILP apprende programmi first-order o higher-order;
- *Bias linguistico*: come definire lo spazio delle ipotesi;
- *Metodo di ricerca*: come cercare lo spazio delle ipotesi.

La semantica dei programmi logici si basa sui concetti di Herbrand universe, Herbrand base e Herbrand interpretation. Tutti e tre i concetti si basano su un dato vocabolario contenente tutte le costanti, le funzioni e i simboli dei predicati di un programma. In base a come viene espresso il vocabolario si distinguono due principali semantiche: la semantica normale e la semantica non-monotonica.

2.2.1 Semantica Normale

Parliamo di semantica normale quando viene utilizzata un'impostazione generale per ILP e viene consentito che esempi, teoria di base e ipotesi siano espressi da qualsiasi formula logica ben definita. Lo scopo di tale semantica è quello di trovare un'ipotesi tale che valgano le condizioni di:

- *Sufficienza* o *Completezza* per gli esempi positivi;
- *Soddisfacibilità* o *Consistenza* per gli esempi negativi.

Il caso speciale della semantica definita, in cui l'evidenza è limitata a fatti veri e falsi, sarà chiamata impostazione di esempio ed è l'impostazione principale di ILP, utilizzata dalla maggior parte dei sistemi.

2.2.2 Semantica Non-Monotonica

Nell'impostazione non-monotonica, la teoria di fondo è un insieme di clausole definite, l'evidenza è vuota e le ipotesi sono insiemi di clausole generali esprimibili usando lo stesso alfabeto della teoria. La ragione per cui l'evidenza è vuota è che l'evidenza positiva è considerata parte della teoria di base mentre l'evidenza negativa è derivata implicitamente. L'impostazione non-monotonica ipotizza solo le proprietà che sono presenti nel database. Pertanto, realizza l'induzione per deduzione; questo produce generalizzazioni al di là delle osservazioni. Di conseguenza, le proprietà derivate da tale impostazione sono più conservative di quelle derivate nell'impostazione normale.

Lo scopo di tale semantica è quello di trovare un'ipotesi che rispetti i requisiti di:

- *Validità*: assicura che tutte le clausole appartenenti a un'ipotesi rispettino la conoscenza di base (BK), dunque affermino la verità delle proprietà dei dati;
- *Completezza*: afferma che tutte le informazioni valide nella conoscenza di base devono essere codificate nell'ipotesi. Questo requisito dovrebbe essere in relazione a un dato bias sintattico, di cui parleremo più avanti, che determina l'insieme di ipotesi ben formate;
- *Minimalità*: mira a derivare ipotesi non ridondanti.

Le differenze tra le due impostazioni sono legate al presupposto del mondo chiuso. Intuitivamente, questo mostra che le soluzioni ai problemi nell'impostazione normale, in cui viene applicata l'assunzione del mondo chiuso, sono valide anche nell'impostazione non monotonica.

2.2.3 Pregiudizio linguistico: Bias

Tra le diverse problematiche che si possono incontrare nei sistemi ILP, certamente una di queste, è la presenza di troppa conoscenza di base. La ridondanza di clausole porta ad un'esponenziale crescita di BK provocando una grave inefficienza durante la ricerca o un apprendimento errato delle ipotesi. Affrontare il problema di vaste BK è stato oggetto di ricerche insufficienti. Lo spazio delle ipotesi contiene tutti i possibili programmi che possono essere costruiti nel linguaggio di rappresentazione scelto pertanto, se non limitato, è infinito; è quindi importante restringerlo per rendere fattibile la ricerca.

Il modo principale per limitare lo spazio delle ipotesi è imporre un bias induttivo (Mitchell, 1997).

Il bias più comune è un pregiudizio linguistico che impone restrizioni sulle ipotesi, come limitare il numero di variabili, lettere e clausole in un'ipotesi. Queste restrizioni possono essere classificate come:

- *bias sintattico*: restrizioni sulla forma delle clausole in un'ipotesi;
- *bias semantico*: restrizioni sul comportamento delle ipotesi indotte.

Esistono diversi modi per realizzare un pregiudizio linguistico ma l'attenzione sarà rivolta ai due più diffusi: dichiarazioni di modalità [27] e meta-rules [12].

Dichiarazione di modalità

Le *dichiarazioni di modalità* indicano quali simboli predittivi possono apparire in una clausola, quanto spesso e anche i loro tipi di argomento. Una dichiarazione di

modalità ha la forma:

$$mode(recall, predicate(m_1, m_2, \ldots, m_n)).$$

nella quale:

- *recall*: indica il numero massimo di volte in cui una dichiarazione di modalità può essere utilizzata in una clausola;
- *predicate(m_1,m_2,...,m_n)*: indica il simbolo del predicato che può apparire nella testa o nel body di una clausola e il tipo di argomenti che richiede `(+input, -output, #ground)`.

Diversi sistemi ILP utilizzano le dichiarazioni di modalità in modi leggermente diversi. Progol, ad esempio, usa dichiarazioni di modalità con tipi di argomenti `+/-` perché inducono programmi Prolog, dove l'ordine dei letterali in una clausola è importante. Al contrario, ILASP induce programmi ASP che non utilizzano argomenti `+/-` in quanto l'ordine dei letterali in una clausola non ha importanza.

Meta-Rules

Le *meta-rules* sono clausole Horn del secondo ordine che definiscono la struttura dei programmi apprendibili che a loro volta definiscono lo spazio delle ipotesi. Una meta-rule è nella forma:

$$P(A, B) \text{ :- } Q(A, C), R(C, B).$$

nella quale:

- *P,Q* ed *R*: indicano variabili del secondo ordine che possono essere associate a simboli predicati;
- *A,B* e *C*: indicano variabili del primo ordine che possono essere associate a simboli costanti.

A differenza di altre forme di bias nell'ILP, come i modi o le grammatiche, le meta-rules sono esse stesse affermazioni logiche che consentono di ragionare su di esse.

Metagol, ad esempio, usa meta-rules per ridurre lo spazio delle ipotesi da generare. Tuttavia, resta ancora una sfida la decisione di meta-rules da utilizzare.

2.3 Metodi di ricerca

Come anticipato, la costruzione di un sistema ILP conduce a dover fare delle scelte; una tra queste è cercare in modo efficiente un ampio spazio di ipotesi. In letteratura esistono quattro principali metodi di ricerca, descritti di seguito.

Top-Down

Metodo utilizzato da sistemi ILP meno recenti, parte da un'ipotesi generale per poi specializzarla; sistemi che sfruttano tale metodo possono apprendere programmi ricorsivi ma possono generare molte ipotesi inefficienti.

Un esempio di sistema ILP che utilizza questo approccio è FOIL, (First-Order Inductive Learning).

Bottom-Up

Metodo utilizzato da sistemi ILP meno recenti, inizia con esempi e li generalizza; sistemi che sfruttano tale metodo sono tipicamente veloci ma non supportano facilmente ipotesi ricorsive e l'invenzione di predicati.

L'algoritmo Golem, ad esempio, generalizza coppie di esempi sulla base di una generalizzazione minima relativa.

Meta-Level

Metodo utilizzato da sistemi ILP più recenti, formula il problema di apprendimento come un problema di ricerca dichiarativa; possono apprendere più facilmente programmi ricorsivi e ottimali, ma possono avere difficoltà a ridimensionare problemi con domini non banali e programmi con clausole di grandi dimensioni.

Metagol ad esempio, utilizza la definizione di meta-regole per provare la validità delle ipotesi in input.

Conflict-Driven

Nuovo metodo utilizzato da poche categorie di sistemi ILP che promuove l'apprendimento attraverso la definizione di vincoli, dai quali stabilisce se una determinata ipotesi copre i vincoli definiti, permettendo una costruzione incrementale del programma [18].

ILASP ad esempio, utilizza un ciclo di selezione e vincolo basato su un contro esempio; ad ogni iterazione, utilizza un risolutore ASP per trovare la migliore ipotesi possibile. Se l'ipotesi non copre uno degli esempi, ne trova il motivo e ne genera i vincoli in modo da aggiungerli al programma di meta-livello per guidare la ricerca successiva.

2.4 Tecniche alla base di ILP

Partendo da una base filosofica, definita da Socrate e successivamente sfruttata da Aristotele, il *passo d'induzione* viene definito come "il processo che parte da fatti

particolari per condurre a fatti universali". Tale definizione è in contrapposizione con il generale percorso che adotta il ragionamento umano; esso infatti sfrutta un processo *deduttivo* caratterizzato dall'ottenimento di fatti particolari partendo da una base universale. Tuttavia per Socrate, né i sensi per via induttiva né la razionalità per via deduttiva, conducono ad una garanzia di verità universale e pertanto afferma che l'induzione può essere considerata soltanto come processo di partenza a cui applicare l'intelletto per poi sfociare in una consequenzialità logica.

Le tecniche per la generazione delle ipotesi trovano fondamento da una moderna definizione d'induzione non più vista solo come passo iniziale di un ragionamento logico ma categorizzata come *inferenza ampliativa probabilistica*. Dunque i risultati, che possono essere ottenuti per induzione, assumono un grado di correttezza e validità legato ad un valore di probabilità tra 0 e 1, rappresentando un trade-off nella scoperta di nuovi fatti partendo da una conoscenza precedentemente appresa.

2.4.1 Relative-Least-General Generalization (RLGG)

La tecnica relative-least-general generalization proposta da Plotkin, si basa sulla generazione di una clausola meno generale di una sussunzione, a partire da una conoscenza di base. Lo scopo quindi di tale tecnica è quello di trovare un'unica clausola tale che, aggiunta alla BK, riesca a dimostrare più clausole contemporaneamente una sola volta. Dunque generalizza clausole logiche del primo ordine, *First Order Logic (FOL)*), tramite una ricerca Bottom-Up, dei soli esempi positivi e utilizza quelli negativi per ridurre la dimensione dello spazio delle ipotesi da ricercare e per introdurre vincoli, in modo da eliminare letterali inutili al fine della ricerca [22].

Nella pratica, attuare una *relativizzazione*, consiste nel creare una clausola in Forma Normale, ad esempio:

$$h_1(a,b) \vee \neg b_1(a,d) \vee \neg b_2(e,b) \vee \ldots \vee \neg b_n(d,c).$$

nella quale:

- h_1: indica il letterale ground dell'esempio;
- b_1,b_2 e b_n: indicano i letterali contenuti nella background knowledge;
- a,b,c,d ed e: indicano i rispettivi termini ground in ogni letterale.

Applicato il relativismo, si procede costruendo la *generalizzazione meno generale* accoppiando ogni letterale del body con ogni letterale del body di un secondo esempio relativizzato, come approfondito in 2.5.1; infine vengono sostituiti i termini ground dei letterali con delle variabili per ottenere la generalizzazione della

clausola, poi convertita in Forma di Horn, ad esempio:

$$h_1(X,Y) \text{ :- } b_1(Y,X) \wedge \ldots \wedge b_n(X,Y).$$

Formalmente, la RLGG gode di tre importanti proprietà:

- Riduzione logica: un letterale logicamente ridondante in una clausola, rispetto alla conoscenza di base, può essere rimosso;
- Riduzione funzionale: ciascuna variabile presente nella testa della clausola di output è direttamente calcolata dalle variabili di input da variabili collegate alle variabili di input;
- Utilizzo di esempi negativi: rimuove i letterali dal RLGG generato purché non venga coperto nessun esempio negativo.

Tuttavia, presenta tre problemi rilevanti:

- Modello infinito di ground BK: un modello di conoscenza di base infinito può generare un RLGG infinito, anche se ridotto, provocando inefficienza;
- Dimensione del RLGG: anche se il modello di conoscenza di base è finito, la dimensione del RLGG può crescere esponenzialmente in quanto è costruito in base al numero di esempi;
- Multiple clausole di ipotesi: concetti target con più clausole che li dimostrano non possono essere appresi in quanto il RLGG di un insieme di esempi corrisponde ad una singola clausola.

2.4.2 Inverse-Entailment (IE)

L'Inverse-Entailment è una relazione di generalità nella programmazione logica induttiva. Più in dettaglio, quando si impara da un'implicazione usando una teoria di base, l'ipotesi da apprendere copre un esempio relativo alla teoria di base; lo scopo dunque è indurre una regola che insieme alla conoscenza di base comporti l'esempio. L'implicazione inversa si basa sull'osservazione che se l'esempio è implicato dalle ipotesi e dalla background knowledge è equivalente affermare che lo stesso esempio negato è implicato dalla stessa base di conoscenza e dalla negazione della regola [25, 29]. In particolare, tale tecnica sfrutta delle dichiarazioni di modalità, introdotte in 2.2.3, per derivare la clausola più specifica nel linguaggio che implica un dato esempio. Per fare ciò esegue una ricerca guidata dalla compressione sulle clausole che includono la clausola più specifica.

Il principale problema che caratterizza l'utilizzo di IE è la presenza di più modelli di dichiarazione. In questo caso, la tecnica è inapplicabile al programma; necessita di un rilassamento delle condizioni per ulteriori estensioni.

2.4.3 Meta-Interpretative Learning (MIL)

La recente tecnica di Meta-Interpretative Learning, rispetto alle tecniche precedentemente descritte, supporta l'apprendimento di definizioni ricorsive e l'invenzione di predicati. Un altro aspetto innovativo di MIL consiste nell'introduzione automatica di sotto definizioni nel momento in cui vengono apprese nuove definizioni di predicato, consentendone la scomposizione in una gerarchia di parti riutilizzabili [11, 16].

Normalmente, un meta-interprete ricava una dimostrazione recuperando ripetutamente le clausole Prolog del primo ordine le cui teste si uniscono con un dato obiettivo. Inoltre, l'interprete recupera le meta-regole higher-order le cui teste si uniscono con l'obiettivo e salva le meta-sostituzioni risultanti per formare un programma. Il meta-interprete viene fornito dall'utente con meta-regole, che sono quindi espressioni higher-order che descrivono le forme delle clausole consentite nei programmi ipotizzati. L'interprete prova gli esempi e, per ogni prova di successo, salva le sostituzioni per le variabili esistenzialmente quantificate trovate nelle meta-regole associate. Quando queste sostituzioni vengono applicate alle meta-regole, danno luogo a un programma definito del primo ordine che rappresenta una generalizzazione induttiva degli esempi.

2.4.4 Bottom-Clause Propositionalization (BCP)

In generale, la *proposizionalizzazione* è un processo di esplicita conversione di un dataset relazionale in un dataset proposizionale (tabella attributo-valore). I dati di input possono essere esempi rappresentati da termini strutturati, diversi predicati nella logica del primo ordine o diverse tabelle in un database relazionale. L'output è una rappresentazione del valore dell'attributo in una singola tabella, in cui ogni esempio corrisponde a una riga ed è descritto dai suoi valori per un insieme fisso di attributi. I nuovi attributi sono detti *funzionalità* per sottolineare la loro costruzione a partire da attributi originali. Lo scopo del processo è di pre-elaborare i dati relazionali per successive analisi da parte di learner del tipo valore-attributo.

In particolare, la *bottom-clause propositionalization*, rappresenta una nuova tecnica di proposizionalizzazione logic-oriented che consiste nel generare bottom clause, per ogni esempio del primo ordine ed utilizzare l'insieme di tutti i letterali nel body delle bottom clause come possibili caratteristiche; in termini di proposizionalizzazione, come possibili colonne della tabella attributo-valore [15]. Diverse sono le ragioni che portano all'utilizzo di tale tecnica; tra esse, le più importanti sono quella di ridurre la complessità del programma e di accelerare il tempo dell'apprendimento complessivo.

2.5 Sistemi ILP

In questa sezione vengono descritti i principali algoritmi alla base dello stato dell'arte, uno per ogni tecnica precedentemente proposta, evidenziandone i singoli step e le principali mancanze.

2.5.1 Algoritmi alla base di ILP

Golem

Golem è un algoritmo per l'induzione logica, sviluppato da S. H. Muggleton e Feng, che si basa sulla tecnica del RLGG sfruttando un metodo di ricerca Bottom-Up. L'algoritmo Golem risolve il problema della dimensione dello spazio delle ipotesi, introdotto dall'utilizzo della tecnica rlgg, impiegando delle restrizioni di ricerca in modo da non permettere l'aumento esponenziale delle dimensioni [2].

Più dettagliatamente, l'algoritmo è suddiviso in quattro fasi:

1. seleziona casualmente un numero di coppie di esempi positivi;
2. per ogni coppia costruisce una spazio di ipotesi rlgg e seleziona lo spazio che copre il maggior numero di esempi positivi;
3. mentre il numero di esempi coperti aumenta:
   (a) seleziona un numero di esempi positivi non coperti fino a quel momento;
   (b) trova l'esempio migliore tra questi in modo che lo spazio copra più esempi più positivi;
   (c) definisce lo spazio rlgg come l'unione tra lo spazio migliore trovato fino a quel momento e l'esempio migliore;
4. sfrutta gli esempi negativi per ridurre la dimensione dello spazio generato.

Ripete tutti i passi precedenti fino a coprire tutti gli esempi positivi.

Progol

Progolè l'implementazione sviluppata da S. H. Muggleton per l'induzione logica che combina i metodi di ricerca Top-Down e Bottom-Up, applicando la tecnica di IE [24, 26].

L'algoritmo utilizzato in Progol è composto da tre fasi:

1. seleziona una clausola dalla conoscenza di base la cui testa corrisponde al concetto target;

2. definendo un livello di profondità, costruisce la clausola più specifica che rispetta determinate restrizioni di tipo e modalità specificate dall'utente e generalizza la clausola selezionata. Più dettagliatamente:

   (a) inizializza la testa della clausola specifica con una testa variabilizzata della clausola iniziale;

   (b) attraverso un loop, fino alla profondità definita, trova tutte le variabilizzazioni della clausola, utilizzando variabili che rispettano le restrizioni dell'utente;

   (c) trova tutte le sostituzioni che rimpiazzano le variabili con i termini precedentemente associati;

   (d) tenta ogni possibile prova per le sostituzioni fatte entro un finito numero di step;

   (e) aggiunge i letterali provati alla clausola specifica e sostituisce i termini con le variabili; infine aggiunge le nuove variabili all'insieme delle variabili che rispettano quel target.

3. trova la migliore generalizzazione tra la clausole specifiche costruite utilizzando una ricerca best-first, cioè espandendo sempre la clausola migliore che si avvicina al target.

Ritorna al punto 1 fino a coprire tutti gli esempi positivi; la ricerca si ferma quando è garantito che non esiste una clausola migliore con una lunghezza inferiore a quella trovata fino a quel momento.

Il principale problema degli algoritmi impiegati in Golem e Progol è la loro incapacità di apprendere definizioni ricorsive e di gestire l'invenzione dei predicati. Per la gestione di tali problemi, è stato sviluppato il sistema descritto di seguito.

Metagol

Metagol è un sistema per l'induzione logica che si basa sulla tecnica MIL e sfrutta un metodo di ricerca dello spazio delle ipotesi Meta-Level.

L'algoritmo su cui si basa Metagol è composto da tre fasi:

1. seleziona un esempio positivo da generalizzare; se non esiste, s'interrompe;

2. tenta di provare l'esempio, sfruttando tre diverse possibilità:

   - la BK definita;
   - una clausola già indotta;
   - unificando l'atomo dell'esempio con la testa di una meta-rule e dimostrando il corpo della meta-rule attraverso il metodo di meta-interpretazione;

3. una volta provati tutti gli esempi positivi, confronta l'ipotesi finale con gli esempi negativi. Se essa non comporta alcun esempio negativo l'algoritmo si ferma altrimenti ritorna al punto 2.

Seppur apparentemente efficiente, Metagol, in presenza di piccole BK e/o predicati con arità maggiore di due, risulta inadatto provocando un difficile apprendimento e inoltre, non è in grado di gestire esempi poco chiari e fatica ad apprendere programmi di grandi dimensioni.

CILP++

CILP++ è un sistema open-source che estende C-IL^{2}P *(Connectionist Inductive Learning and Logic Programming)*, un sistema neurale-simbolico, che costruisce una rete neurale artificiale ricorsiva utilizzando conoscenze di base composte da clausole proposizionali per apprendere dalla logica del primo ordine, utilizzando la tecnica BCP.

In dettaglio, l'algoritmo di apprendimento del sistema CILP++, è composto da tre fasi:

1. fase di applicazione BPC; converte ogni letterale target in un vettore numerico utilizzabile da una ANN come input attraverso due passaggi:

   (a) trasformazione di ogni esempio in una bottom clause, generando nuovi vettori numerici;

   (b) mappatura degli esempi nella tabella attributo-valore;

2. fase di costruzione; creazione di una rete iniziale per l'addestramento mappando ogni letterale del body su un neurone di input e ogni letterale della testa su un neurone di output. Dopo questa fase solo le bottom clause generate da esempi positivi potranno essere sfruttate come BK;

3. fase di training; utilizzo di:

   - backpropagation, per l'addestramento standard;
   - un metodo di convalida incrociata incorporato, per misurare l'errore di generalizzazione durante ogni epoca;
   - un'opzione di arresto anticipato, per interrompere l'addestramento quando una misura di errore inizia ad aumentare. BCP può generare reti di grandi dimensioni pertanto un'interruzione precoce può rivelarsi molto efficace per evitare l'overfitting.

L'utilizzo di BPC permette a CILP++ di ottenere maggiore accuratezza e maggiore velocità nell'ottenere risultati. Inoltre, l'utilizzo della selezione delle funzionalità nella fase di training, consente di ridurre drasticamente la dimensione della rete a discapito di una piccola perdita di precisione e talvolta, di un aumento dei tempi di training.

ILASP

ILASP, *Inductive Learning of Answer Set Programs*, è un nuovo framework che differisce dai tradizionali sistemi che sfruttano un metodo di ricerca basato su Meta-Level, come Metagol o l'algoritmo di Popper.

Esso, è un particolare sistema per l'apprendimento induttivo che delega l'attività di apprendimento ad un solver ASP invocato in modo incrementale su un programma che cresce durante l'apprendimento e non su un programma fisso [19]. Le prime versioni del framework, ILASP1 e ILASP2, hanno approcciato l'induzione basandosi su un puro metodo di ricerca di meta livello; l'ultima versione, ILASP3, ha spostato il sistema verso un nuovo modello di ricerca, detto *Conflict-driven*, che costruisce sistemi in modo iterativo a partire da vincoli definiti sullo spazio della soluzione.

Nel dettaglio, ad ogni iterazione:

1. il solver trova un'ipotesi che soddisfa i correnti vincoli definiti;

2. cerca un conflitto tale che l'ipotesi non rappresenta una soluzione induttiva;

3. dal risultato del passo precedente:

    - se il conflitto non esiste, restituisce l'ipotesi che risulta dunque una soluzione induttiva valida;
    - altrimenti crea un nuovo vincolo, a partire dal conflitto trovato, a supporto di successivi programmi. Questo processo è chiamato *analisi del conflitto*.

Il principale svantaggio di ILASP3 consiste nei vincoli; questi possono essere estremamente grandi e costosi da calcolare, specialmente all'aumentare dello spazio del programma. Questo accade poiché i vincoli sono condizioni sufficienti e necessarie per la copertura dell'esempio. ILASP4, nuova estensione del framework ancora in fase di sviluppo, cerca di rendere i vincoli soltanto necessari per la copertura degli esempi; lo scopo dunque è quello di garantire meno complessità a ciascuna iterazione e generare vincoli molto più piccoli, a discapito di un maggior numero di iterazioni provocate da una ridondanza degli esempi trovati.

2.5.2 Sistemi ILP a confronto

Dalla definizione dei precedenti algoritmi, risulta quindi evidente come gli approcci e i metodi utilizzati, nel corso degli anni, hanno subito un notevole cambiamento [8].

Vecchi sistemi ILP, come Golem e Progol, sfruttano una conoscenza di base costruita esclusivamente sulla logica del primo ordine e, metodi di ricerca come Bottom-Up e Top-Down, risultano efficaci soltanto al loro scopo ma non indicati per un approccio evolutivo. Non prevedono nessun tipo di gestione per definizioni ricorsive e invenzione di nuovi predicati durante l'apprendimento, sfruttano framework come Prolog, che eseguono un unico programma per la costruzione dell'apprendimento e prevedono limitati meccanismi per la riduzione dello spazio d'ipotesi da indurre.

Nuovi sistemi ILP invece, come Metagol, CILP++ e ILASP, sfruttano una conoscenza di base che può essere costruita da un set di risposte [20] o da logiche higher-order [10] ed utilizzano metodi di ricerca, come Meta-Level o il recente Conflict-driven, che permettono una maggiore evoluzione dei programmi. Inoltre permettono l'apprendimento di programmi ricorsivi e l'invenzione di predicati generando ottimalità del processo e sfruttano framework che favoriscono la riduzione dello spazio di ipotesi indotto, come ASP e NN, e riducono il tempo totale d'apprendimento.

2.6 Reti Neurali e ILP

Nuovi sforzi di ricerca, verso un'intelligenza artificiale spiegabile, *XAI*, mirano a mitigare il problema dell'opacità dei dati e perseguono l'obiettivo finale di costruire sistemi intelligenti comprensibili, responsabili e affidabili, sebbene ancora con una lunga strada da percorrere. Gli approcci simbolici o basati sulla logica, e sub-simbolici o statistici di apprendimento automatico, sono in qualche modo complementari tra loro; mentre i primi sono comprensibili dall'uomo e parsimoniosi in termini di dati, i secondi sono intrinsecamente opachi e desiderosi di dati. Di conseguenza, diventa fondamentale capire in che misura una combinazione di approcci simbolici e sub-simbolici possa contribuire a rendere la IA più ispezionabile, interpretabile o spiegabile [4].

Più precisamente l'ibridazione del modello sta attualmente guidando la ricerca a partire da due schemi principali, *Model Integration* e *Model Composition*, con lo scopo di costruire sistemi che perseguano proprietà fondamentali per XAI come trasferibilità, accessibilità, spiegabilità, correttezza ed affidabilità.

2.6.1 Model Integration

A questa categoria, oltre al sistema CILP++ discusso in 2.5.1, appartengono i sistemi nei quali l'ibridazione tra gli approcci simbolici e sub-simbolici avviene attraverso una "fusione" in un modello unico che include le caratteristiche di entrambi [32]. Di seguito, vengono suggeriti i sistemi più recenti basati su ILP, aperti a future rivisitazioni ed implementazioni.

Logic Tensor Networks (LTN), 2017

Le LTN integrano l'apprendimento basato su reti tensoriali con una logica first-order many-valued basata sul ragionamento. A partire da dati disponibili sotto forma di vettori a valori reali, i vincoli logici e le relazioni che si applicano a certi sottoinsiemi dei vettori, possono essere specificati in modo compatto utilizzando FOL. Le formule logiche sono usate per costruire una funzione di perdita che mira ad addestrare una rete in grado di approssimare il valore di verità, compreso nell'intervallo [0,1], delle formule date come input. Questo avviene cercando la migliore rappresentazione possibile per i costrutti simbolici in uno spazio vettoriale, come messa a terra di atomi, funzioni o predicati, in modo che la soddisfacibilità della rete sia il più vicino possibile a 1 sul set di dati di test. La rete risultante è in grado di apprendere dagli esempi reali correttamente etichettati, ma mantiene l'impronta logica data nella fase di formazione [1].

Questi sistemi presentano diversi vantaggi interessanti:

- *Trasferibilità*: supportano la previsione di nuovi fatti dai dati;
- *Affidabilità*: riducono fortemente il rischio che le reti neurali apprendano comportamenti inattesi o indesiderati dai dati;
- *Correttezza ed Equità*: offrono agli sviluppatori la possibilità di esprimere vincoli per prevenire l'apprendimento di comportamenti scorretti;
- *Apertura*: sono dotati di un framework basato su Python, ben documentato e mantenuto attivamente.

Differentiable Inductive Logic Programming (∂ILP), 2017

∂ILP è una re-implementazione di ILP in un'architettura differenziabile end-to-end che combina i vantaggi di ILP con i vantaggi dei sistemi basati su reti neurali [14]. L'idea principale è imitare la deduzione logica su clausole definite tramite una rete neurale utilizzando il concatenamento in avanti, invece del concatenamento all'indietro. Viene sfruttata una semantica continua, che mappa gli atomi nell'intervallo [0,1], e dei pesi continui per determinare una distribuzione di probabilità

sulle clausole; implementa dunque una deduzione differenziabile su valori continui. La rete risultante viene quindi addestrata per ridurre al minimo l'entropia incrociata rispetto agli esempi positivi e negativi, implementando una forma continua di induzione.

Peculiarità di questi sistemi sono:

- *Correttezza*: raggiungono la spiegabilità per costruzione, sfruttando le logiche come tecnica di vincolo;
- *Ambiguità*: gestiscono dati ambigui e confusi;
- *XAI*: possono essere sfruttati per perseguire trasferibilità, informatività ed accessibilità.

Seppur presenti alcuni esperimenti su tali sistemi, non sono ancora disponibili implementazioni usabili ed affermate.

Lyrics, 2019

Lyrics è un'estensione di LTN che fornisce un linguaggio di input dichiarativo in grado di definire conoscenze di base FOL arbitrarie. I predicati e le funzioni della conoscenza possono essere associati a qualsiasi grafo computazionale e le formule vengono convertite in un insieme di vincoli a valori reali, che partecipano al problema di ottimizzazione generale [21]. Grazie alla sua dichiaratività può combinare più reti neurali realizzando un unico grafo computazionale che viene quindi ottimizzato rispetto ai dati disponibili, consentendo di apprendere i pesi sotto i vincoli imposti dalla conoscenza pregressa. I principali vantaggi del framework sono:

- apprendimento supervisionato: apprendimento da dati corretti, permesso dalla conoscenza di base;
- classificazione collettiva: classificazione combinata attraverso la correlazione tra l'oggetto target ed i valori corretti precedentemente osservati;
- text chunking: classificazione basata non solo sulle caratteristiche dei dati ma anche dal contesto in cui si trovano; in dati testuali, per ogni parola riconosce semanticamente il ruolo logico nella frase.

Del framework sono presenti script in Python stabili ed esempi di utilizzo. Tuttavia, il codice sorgente non prevede alcuna documentazione ed implementazione ufficiale.

2.6.2 Model Composition

A questa categoria appartengono i sistemi in cui l'ibridazione del modello avviene lasciando in blocchi separati gli approcci simbolici e sub-simbolici ma sfruttati congiuntamente per produrre un sistema intelligente e spiegabile. Possono essere identificate due grandi linee di ricerca:

- *Estrazione di conoscenza simbolica*: tecniche la cui idea chiave è estrarre una rappresentazione simbolica da un sistema preaddestrato;
- *Iniezione di grafi di conoscenza*: tecniche la cui idea chiave è incorporare componenti di un'ontologia in spazi vettoriali continui, per consentire alle reti neurali di accettare un determinato tipo di informazioni strutturate come input e sfruttare la sua conoscenza di base per eseguire ordinarie attività di apprendimento automatico.

Estrazione di Conoscenza Simbolica

Le tecniche di estrazione possono essere descritte in base a diverse dimensioni ortogonali, come:

- struttura della conoscenza simbolica che estraggono (regole e alberi decisionali, ecc.);
- tipo di vincoli che sfruttano per il processo decisionale (vincoli lineari, regole M-di-N, ecc.);
- tipo di predittori sub-simbolici gestiti (reti neurali, macchine a vettori di supporto, ecc.).

Quando la conoscenza simbolica estratta è rappresentata da semplici regole nella forma:

$$\begin{gathered}\text{if } condition_1 \text{ then } outcome_1\\ \text{if } condition_2 \text{ then } outcome_2\\ \dots\\ \text{if } condition_n \text{ then } outcome_n,\end{gathered}$$

si parla di estrazione di *regole decisionali* in cui ogni condizione può essere una congiunzione o disgiunzione di predicati booleani, vincoli lineari o M-di-N regole sugli attributi dei dati, utilizzati per addestrare il predittore sub-simbolico.

Gli approcci basati sull'estrazione di regole possono essere a loro volta categorizzati in:

- *Pedagogici*: sono basati sul concetto di *black box*; permettono l'estrazione da classificatori arbitrari, trascurando la struttura della rete sottostante (es. RxREN, ALPA, ecc.);

- *Decomposizionali*: al contrario, sono strettamente legati alla struttura della rete (KT, RX, ecc.).

Entrambi gli approcci tentano di perseguire gli obiettivi di XAI elencati in 2.6 tuttavia, trascurano le proprietà di accuratezza, coerenza ed interpretabilità in quanto l'elenco delle regole estratto potrebbe non riflettere perfettamente le intuizioni di quello originale e può facilmente deteriorarsi a causa della quantità di regole.

Invece, quando la conoscenza simbolica estratta è rappresentata da una natura gerarchica, si parla di estrazione di *alberi decisionali ordinari* i cui nodi sono rappresentati da regole costituite da una congiunzione o disgiunzione di predicati booleani, vincoli lineari o regole M-di-N sugli attributi dei dati utilizzati per addestrare il predittore sub-simbolico. Anche le tecniche basate su alberi decisionali possono essere categorizzate in approcci:

- *Pedagogici*: estraggono alberi decisionali da classificatori arbitrari, fornendo libertà nel considerare la struttura interna della rete;

- *Decomposizionali*: estraggono alberi decisionali da classificatori specifici.

Entrambi gli approcci anche in questo caso, tentano di perseguire gli obiettivi di XAI; tuttavia, a causa della natura gerarchica, all'aumentare della dimensionalità dell'albero trascurano la proprietà di interpretabilità.

Iniezione di Grafi di Conoscenza

Le tecniche di iniezione si concentrano sull'introduzione di un *grafo della conoscenza (KG)* la cui idea chiave è incorporare componenti di un'ontologia in spazi vettoriali continui, per consentire alle reti neurali di accettare in input tipi di informazioni strutturate e sfruttare la conoscenza di base per eseguire attività di apprendimento automatico. Provvedono dunque all'iniezione di conoscenza simbolica in modelli sub-simbolici [31].

La maggioranza delle tecniche disponibili opera misurando la plausibilità dei fatti sfruttando alcune funzioni di punteggio e, affinché ogni incorporamento possa essere appreso, dev'essere compatibile solo all'interno del singolo fatto. Ciononostante, questa procedura potrebbe rendere l'incorporamento non sufficientemente predittivo per le attività a valle.

Diverse possono essere le motivazioni per impiegare approcci orientati all'iniezione di conoscenza simbolica:

- completamento di KG: approcci che applicano delle regole di exploit per perfezionare l'incorporazione finalizzata al corretto completamento di grafi della conoscenza; riducono notevolmente lo spazio della soluzione e migliorano in modo significativo l'accuratezza dell'inferenza (ad esempio *RESCAL + TRESCAL, 2015 e INS, 2015*);
- riduzione della perdita globale: approcci che incorporano simultaneamente fatti di KG e regole logiche in un quadro unificato con lo scopo di ridurre al minimo una perdita globale su formule atomiche e complesse (ad esempio *LLE: Low-rank Logic Embeddings, 2015 e KALE, 2016*);
- regolarizzazione della rete: approcci in grado di iniettare conoscenza generale e relazioni ontologiche in una rete neurale profonda, sfruttando una conoscenza di dettaglio come regolarizzatore per la rete (ad esempio *OSCAR, 2019*).

Tutti gli approcci di iniezione perseguono gli obiettivi di XAI e sfruttano la conoscenza di base per prevenire la presenza di dati distorti. Solamente INS e LLE includono del software pubblico ma non aggiornato recentemente.

Infine, queste tecniche condividono l'inconveniente comune di dover istanziare le regole quantificate universalmente in regole di base prima di apprendere i loro modelli (*procedura di messa a terra*); questo può essere inefficiente in termini di tempo e spazio, soprattutto in scenari di Big Data o di complessità delle regole.

2.7 Logic Programming

La programmazione logica è un paradigma di programmazione per la rappresentazione e la manipolazione della conoscenza, attraverso la logica formale e l'inferenza. Le principali famiglie di linguaggi logici includono Prolog, ASP e Datalog e, tutte queste, utilizzano una rappresentazione *Clausal Form*, un sottoinsieme di FOL in *Normal Form*, per scrivere delle regole sotto forma di clausole.

2.7.1 Prolog

Prolog, *PROgramming in LOGic*, è un linguaggio di programmazione dichiarativo basato sulla programmazione logica in *Horn Form* [17]. Realizzato per creare un linguaggio che consenta l'espressione della logica invece di istruzioni specifiche su una macchina, è oggi comunemente usato nelle applicazioni di intelligenza artificiale, analisi del linguaggio naturale, robotica e molti altri.

In Prolog, la ricerca consiste nell'esplorazione di un albero attraverso il quale si cerca di provare delle regole date dall'utente, fornendo come risultato una o più risposte attraverso l'eccellente meccanismo di backtrack, equivalente alla *depth-first*

search nella teoria dei grafi. Tale procedura è meglio conosciuta come *Principio di Risoluzione* [28] e può essere applicata ad una qualsiasi teoria fornita dall'utente che utilizza il programma logico.

Teorie

Prolog è un linguaggio colloquiale; dunque, l'utente e il programma hanno una "conversazione" che avviene per mezzo di una Teoria, ovvero un alfabeto di simboli costituito da regole di inferenza e assiomi. L'alfabeto è caratterizzato da differenti classi di simboli, utilizzate per la rappresentazione di:

- fatti: oggetti del dominio della teoria;
- regole: possibili relazioni tra diversi oggetti;
- obiettivi: target da dimostrare nel linguaggio dell'alfabeto.

Clausole

In generale, tutte le relazioni espresse in una teoria sono rappresentate da clausole, ovvero regole scritte nella forma:

```
Head :- Body
```

in cui, in *Horn Form*:

- `Head`: indica la testa, parte sinistra della regola, contenente al massimo un oggetto della teoria;
- `Body`: indica il corpo, parte destra della regola, contenente nessun oggetto o le relazioni tra più oggetti del dominio, Tabella 2.1.
- `:-` : indica il simbolo d'implicazione utilizzato dal linguaggio.

Relazione	Logic Form	Prolog Form	Operatore	Syntax
Congiunzione	$\wedge$	,	AND	... :- b1, b2
Disgiunzione	$\vee$	;	OR	... :- b1; b2

Tabella 2.1: Relazioni tra Oggetti Logici

Attraverso tale definizione, è possibile esprimere che la testa è conseguenza logica del corpo, permettendo di realizzare nel dettaglio ragionamenti logici deduttivi. Solitamente, gli obiettivi sono clausole senza testa per le quali il programma logico cerca di trovare una dimostrazione:

- se le clausole sono composte da simboli variabili, il programma tenta di trovare una risposta corretta per "sostituzione", cercando gli oggetti corretti nella teoria che, sostituiti alle variabili, dimostrano il target.
- se le clausole sono composte da simboli costanti, il programma controlla l'esistenza dei rispettivi oggetti o delle regole che li dimostrano.

2.7.2 2P-Kt: tuProlog-Kotlin

In questo paragrafo viene definita una recente tecnologia costituente il fulcro nella progettazione e nell'implementazione del modulo descritto nei successivi capitoli. Pertanto, viene introdotto il framework alla base della tecnologia, la struttura e le potenzialità di quest'ultima.

tuProlog: 2P

tuProlog è un framework open source che consente lo sviluppo in un linguaggio di programmazione che adotta il paradigma di programmazione logica basata sul calcolo dei predicati. È leggero in quanto il suo nucleo contiene solo gli elementi essenziali di Prolog, in modo da lasciare ulteriore implementazione nelle librerie e, di conseguenza, totale libertà di personalizzazione agli sviluppatori. È un framework ad-hoc per lo sviluppo di sistemi rule-based ed è perfettamente integrato con la programmazione funzionale e quella orientata agli oggetti in modo da poter applicare uno sviluppo multi-paradigma [13].

2P-Kt

2P-Kt è una rivisitazione multi piattaforma di 2P ed è basato su Kotlin. Mira a diventare un ecosistema aperto per l'intelligenza artificiale simbolica. Per questo motivo è costituito da una serie di moduli interdipendenti volti a supportare la manipolazione ed il ragionamento simbolico in modo estensibile e flessibile [5, 6]. 2P-Kt dunque, si concentra sul supporto della rappresentazione della conoscenza e del ragionamento attraverso la programmazione logica, presentando un'architettura modulare volta ad incoraggiare l'estensione ed il supporto verso differenti sistemi di AI simbolici, come i sistemi ASP ed altri. Inoltre, 2P-Kt è puramente scritto in Kotlin fornendo all'implementazione:

- supporto multi piattaforma: al momento limitato a JS, Android e JVM;
- utilizzo di una libreria minimale.

Nella programmazione logica, dati e conoscenza sono rappresentati attraverso termini logici, ovvero strutture dati ad albero prive di semantica predefinita. In

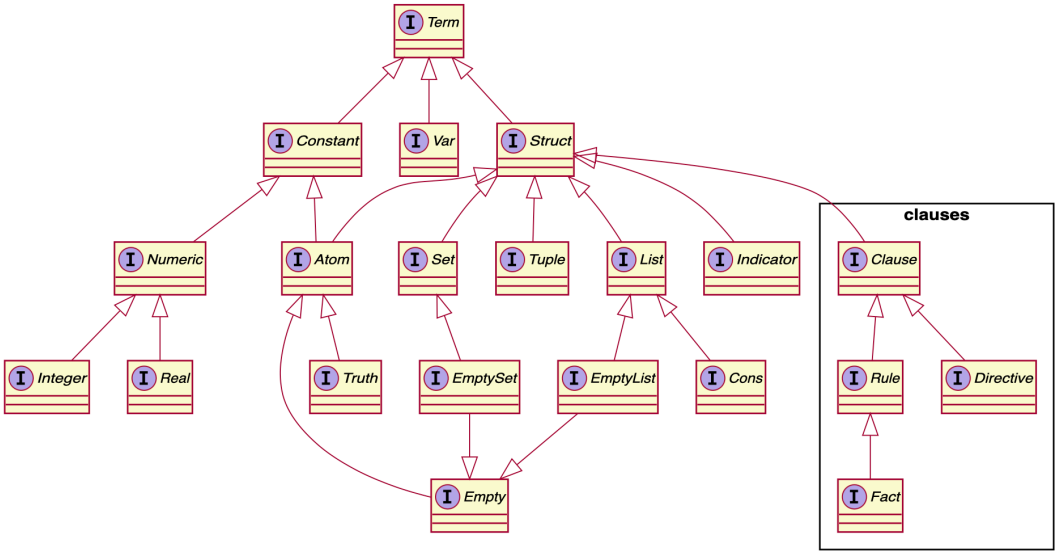

Figura 2.1: Gerarchia dei termini logici in 2P-Kt

2P-Kt, la gerarchia dei termini logici può essere riassunta dalla Figura 2.1 che rappresenta la base da cui si è partiti per la progettazione del framework, descritta nel Capitolo 3. Ogni algoritmo è progettato per l'utilizzo di questa gerarchia di termini, supportando la stessa estensione promossa da 2P-Kt verso differenti sistemi di intelligenza artificiale.

Capitolo 3

Design

In questo capitolo viene descritta la progettazione del framework, mettendo in luce le tipologie di requisiti richiesti, le tecnologie alla base dello sviluppo, le architetture delle singole componenti e le loro principali funzionalità, con lo scopo di effettuare il passo d'induzione sfruttando un linguaggio multi-paradigma (Object Oriented Programming e Functional Programming) partendo da una conoscenza in input fornita dall'utente, caratterizzata da regole e fatti logici.

3.1 Requisiti

3.1.1 Requisiti di Business

I requisiti di business rappresentano le caratteristiche desiderate del prodotto finale che devono essere soddisfatte.

Il framework nella tesi prevede la realizzazione di un sistema in grado di svolgere l'apprendimento per mezzo della programmazione logica induttiva. È composto da:

- *Sistema d'induzione*: parte automatizzata, caratterizzata dalla componente algoritmica, per effettuare l'operazione d'induzione.
- *Command Line Interface*: parte gestionale, caratterizzata da un interfaccia a linea di comando, per interagire con il sistema d'induzione.

3.1.2 Requisiti Utente

I requisiti utente rappresentano le azioni che un qualsiasi utente può effettuare utilizzando il framework:

- deve poter scegliere uno o più algoritmi tra quelli sviluppati;

- deve poter svolgere l'induzione con gli algoritmi scelti;
- deve poter rieseguire la stessa induzione;
- deve poter eseguire nuove induzioni con l'algoritmo selezionato.

3.1.3 Requisiti Funzionali

I requisiti funzionali rappresentano le funzionalità del sistema e come esso reagisce a stimoli esterni.

Induzione

I requisiti funzionali per quanto concerne l'operazione d'induzione sono:

- ogni induzione viene svolta sugli algoritmi selezionati all'avvio del programma;
- ogni algoritmo scelto svolge l'induzione a partire dai parametri inseriti dall'utente inizialmente;
- ogni induzione deve poter ritornare un risultato se l'inserimento dei parametri è avvenuto correttamente;
- i parametri accettati in ingresso devono poter essere teorie logiche in Prolog;
- ogni algoritmo, finita l'induzione, può richiedere nuovi parametri per nuove induzioni.

CLI

I requisiti funzionali per quanto concerne la CLI:

- scelta dell'algoritmo: *(i)* Golem, *(ii)* Progol, e *(iii)* Metagol possono essere selezionati più algoritmi alla volta;
- devono essere inizialmente definiti i parametri corretti in relazione a ciascun algoritmo d'induzione scelto;
- devono poter essere inseriti nuovi parametri per effettuare nuove induzioni con lo stesso algoritmo;
- possibilità di rieseguire, eseguirne una nuova o terminare l'induzione con uno specifico algoritmo;
- visualizzazione dell'algoritmo appena eseguito;
- visualizzazione delle ipotesi indotte.

3.1.4 Requisiti Non Funzionali

I requisiti non funzionali rappresentano le proprietà del sistema che bisogna cercare di soddisfare a pieno.

- *Usabilità*: il sistema deve essere utilizzabile facilmente dagli utenti, anche ripetutamente;
- *Scalabilità*: il sistema deve essere aperto a future migliorie implementative in ogni sua sottoparte;
- *Modularità*: il sistema deve essere composto da diversi sotto moduli in modo da poter facilitare l'intervento sul singolo componente e non sull'intero progetto;
- *Efficienza*: il sistema deve rispondere in modo efficiente ed esaustivo ad ogni richiesta effettuata dagli utenti.

3.1.5 Requisiti Implementativi

Per lo sviluppo del sistema, si è adottata una metodologia di programmazione multi-paradigma che ha portato ai seguenti requisiti implementativi:

- *Kotlin*: il sistema è realizzato interamente attraverso il linguaggio multi-piattaforma Kotlin;
- *2P-Kt*: utilizzo della libreria "tuprolog" per la manipolazione della logica del sistema in chiave multi-piattaforma;
- *CliKt*: utilizzo della libreria "CliKt" per la realizzazione dell'interfaccia utente a riga di comando;
- *TDD - Test Driven Development*: come impone una metodologia ben organizzata, è stato adottato il TDD per una migliore integrazione e continuo testing dello sviluppo.

3.2 Design Architetturale

Nel pattern software architetturale scelto, sono preponderanti la User Interface e il comportamento; il modello della progettazione invece, è già ben definito dallo stato dell'arte.

La natura quasi interamente algoritmica del sistema prevede certamente la figura del Model come motore logico alla base dell'esecuzione, che definisce il comportamento del sistema. Non è prevista una vera e propria View; è stata realizzata una semplice interfaccia a linea di comando per guidare l'utente alle differenti induzioni.

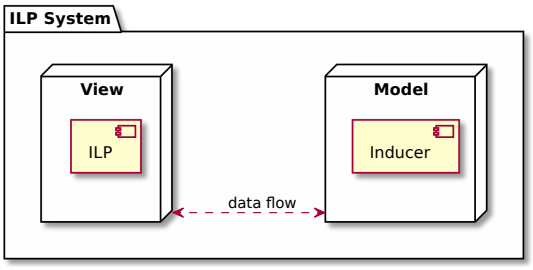

Figura 3.1: Architettura del Sistema

Per quanto asserito, non è stato necessario introdurre la figura del Controller per eseguire la funzione di intermediazione, come evidenziato dalla Figura 3.1.

3.2.1 Model

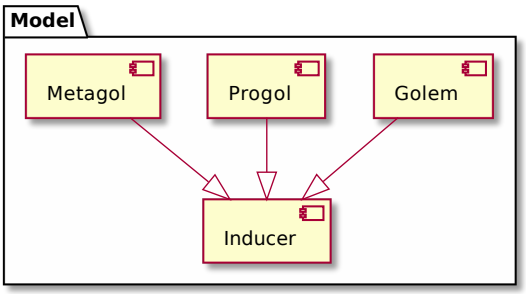

Figura 3.2: Architettura del Model

La componente Model, rappresentata nella Figura 3.2, è l'entità cardine del sistema. Essa si occupa di eseguire l'induzione sfruttando alcuni parametri prestabiliti, definiti dall'utente. In particolare, adotta il principio di modularità scomponendo la figura dell'induttore in tre sotto induttori, ciascuno indipendente dall'altro, massimizzando le relazioni intra-modulo (*coesione*) e minimizzando quelle inter-modulo (*accoppiamento*):

- Golem: componente che si occupa di eseguire l'induzione sfruttando la tecnica relative least general generalization; restituisce l'ipotesi indotta;

- Progol: componente che si occupa di eseguire l'induzione sfruttando la tecnica dell'inverse entailment; restituisce l'ipotesi indotta;

- Metagol: componente che si occupa di eseguire l'induzione sfruttando la tecnica del meta-interpretative learning; restituisce l'ipotesi indotta più eventuali predicati inventati.

Ogni modulo dunque, è limitato dalla semantica dell'algoritmo che lo racchiude ed è progettato per sfruttare una *coesione funzionale*, ovvero svolgere l'unica azione richiesta, l'induzione. Inoltre, ogni modulo è perfettamente estendibile ad altri sistemi ed aperto a nuove rivisitazioni.

3.2.2 View

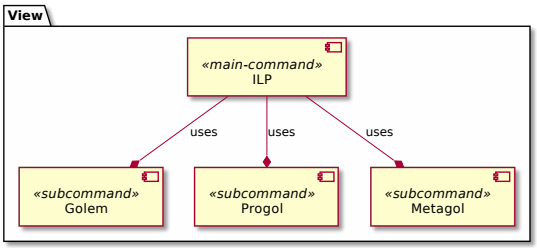

Figura 3.3: Architettura della View

La componente View, rappresentata nella Figura 3.3, è l'entità che mette a disposizione un'interfaccia a riga di comando, generata dal componente ILP. Essa consente all'utente di interagire con il Model a run-time, per consentire multiple induzioni con l'algoritmo scelto. Similmente alla componente precedente, la View è composta da una gerarchia modulare composta da sottocomandi, uno per ogni algoritmo d'induzione; in particolare:

- Golem: sotto comando dell'interfaccia utente che avvia l'algoritmo Golem;

- Progol: sotto comando dell'interfaccia utente che avvia l'algoritmo Progol;

- Metagol: sotto comando dell'interfaccia utente che avvia l'algoritmo Metagol;

Ciascuna induzione fornisce le ipotesi indotte direttamente a riga di comando e lascia la possibilità all'utente di ripartire con una nuova induzione.

3.3 Design di Dettaglio

3.3.1 Inducer

Il ruolo dell'induttore è quello di processare una forma di conoscenza in input, sotto forma di fatti logici, o regole logiche nella forma di Horn, al fine di indurre uno spazio delle ipotesi che sia coerente e plausibile sulla base di esempi di apprendimento appartenenti alla conoscenza fornita. Come si evince dalla Figura 3.4, nell'elaborato sono stati implementati tre diverse tipologie di induttore che, seppur aventi lo stesso obiettivo, ciascuno dei quali presenta sostanziali differenze dagli altri sia da un punto di vista tecnico che pratico.

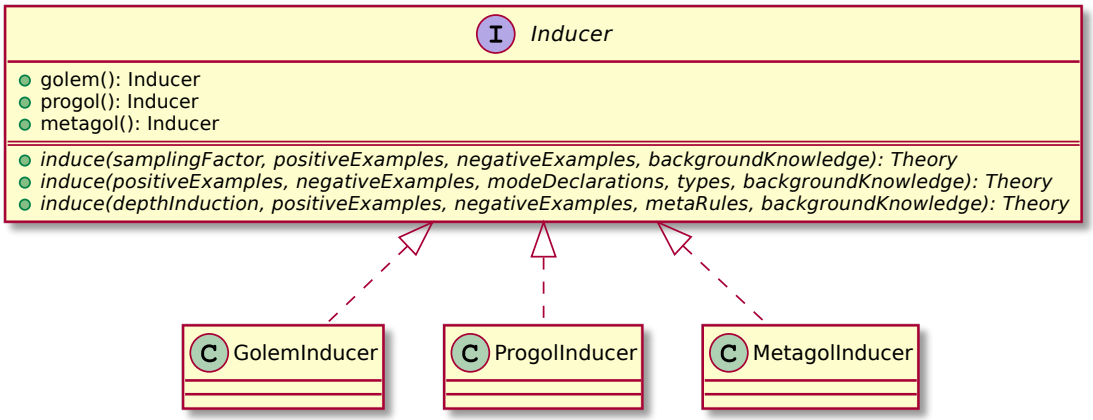

Figura 3.4: Schema Architetturale Inducer

L'interfaccia `Inducer`, il core del framework, è dotata di un unico companion object che richiama rispettivamente i metodi factory associati alle classi di ogni algoritmo di induzione:

- `golem()`: `GolemInducer()`: metodo factory e classe per l'induttore Golem;
- `progol()`: `ProgolInducer()`: metodo factory e classe per l'induttore Progol;
- `metagol()`: `MetagolInducer()`: metodo factory e classe per l'induttore Metagol.

Ogni implementazione esegue l'overload dell'unico metodo pubblico `induce()` presente nell'interfaccia, con differenti parametri in base alla tipologia d'induttore. Gli unici parametri in comune sono rappresentati da:

- `positiveExamples`: insieme di fatti positivi da cui iniziare l'apprendimento induttivo;

- `negativeExamples`: insieme di fatti negativi da cui vincolare l'apprendimento induttivo;

- `backgroundKnowledge`: teoria di base, composta da regole logiche, per la dimostrazione di ogni esempio, sia positivo che negativo.

3.3.2 Golem Inducer

L'induttore Golem è un sistema che esegue l'apprendimento induttivo attraverso il metodo di ricerca Bottom-Up e la tecnica di costruzione delle ipotesi *relative least general generalization (rlgg)*.

L'algoritmo Golem tradizionale, presentato nel Capitolo 2, opera su clausole in *Normal Form* applicando, dunque, la disgiunzione tra la testa ed ogni letterale del corpo negato. In questa riformulazione multi-paradigma di Golem invece, le clausole sono direttamente espresse e trattate nella *Horn Form*, ovvero clausole composte da un solo letterale nella testa e la congiunzione di `n` letterali nel corpo, con `n > 0`.

Struttura

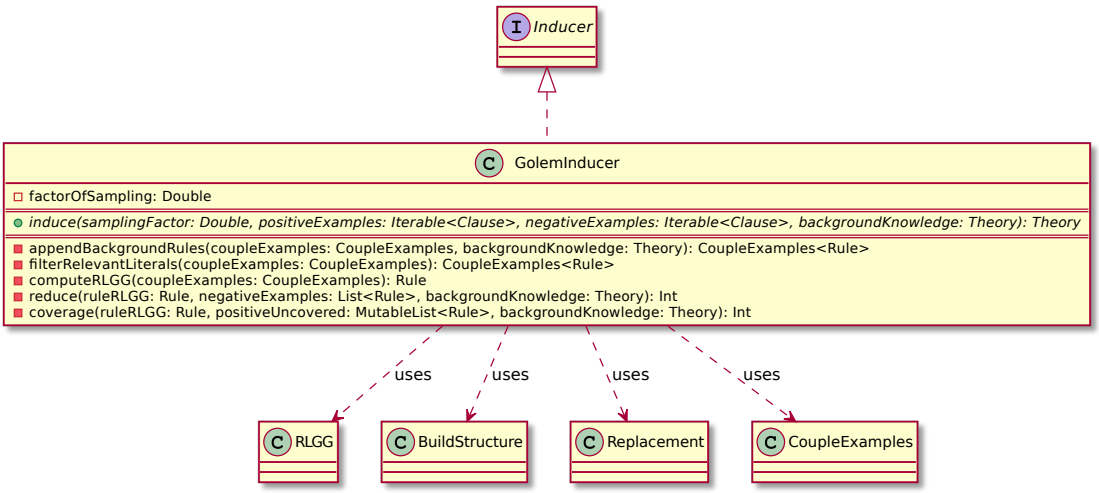

Figura 3.5: Schema Architetturale dell'Induttore Golem

È uno dei primi sistemi sviluppati in letteratura ed è caratterizzato dalla sua complessa costruzione delle ipotesi. Inoltre, limita fortemente l'induzione a particolari tipi di clausole; sono ammesse all'induzione le sole clausole che hanno nel corpo soltanto i letterali aventi per argomento i soli argomenti della testa, ad

esempio:

$$head(X, Y) \text{ :- } bodyTerms(X, Y).$$

dove $bodyTerms(X, Y)$ rappresenta l'insieme delle congiunzioni tra i letterali del corpo aventi come argomenti i soli X e Y.

La particolarità di Golem, ma al contempo il suo limite, è l'applicazione della tecnica rlgg che consiste nel rendere *relativo (R)* ogni esempio positivo in input; costruisce dunque una regola avente per testa il letterale dell'esempio e per corpo la conoscenza di base completa. La *generalizzazione minima (LGG)* è applicata ad una coppia di esempi relativizzati; combina ogni letterale del primo esempio con ogni letterale del secondo e, sostituendo gli stessi termini ground con la stessa variabile, ottiene la generalizzazione della regola.

Golem è caratterizzato dall'unico metodo pubblico `induce()` avente come soli parametri quelli comuni già definiti. Pertanto non adotta nessuna tipologia di bias sintattico; sfrutta esclusivamente un campo privato `factorOfSampling`, inizializzato a `samplingFactor`, come rate di campionamento per la formazione delle coppie di esempi positivi da cui indurre l'ipotesi finale, legando strettamente ogni esempio alla base di conoscenza.

Dettaglio

Premessa la possibilità di apprendere una sola regola alla volta, la condizione necessaria è che tutti gli esempi positivi abbiano lo stesso funtore, quindi abbiano lo stesso nome del predicato da apprendere.

L'algoritmo presente può essere riassunto dal Listato 3.1:

Listato 3.1: Algoritmo Golem

```
while (uncoveredPositives.isNotEmpty()) {
  val bestRlgg: Rule = couples
    .map { appendBackgroundRules(it, background) }
    .map { filterRelevantLiterals(it) }
    .map { computeRLGG(it) }
    .filter { reduce(it, negativeExamples, background) == 0 }
    .maxByOrNull { coverage(it, uncoveredPositives, background) }
}
```

Dunque ad ogni coppia di esempi:

1. attraverso il metodo `appendBackgroundRules()`, viene riempito il body di ogni esempio con tutte le regole presenti nella teoria di base;

2. attraverso il metodo `filterRelevantLiterals()`, vengono filtrati i termini del body di ogni clausola di esempio che non contengono gli argomenti della

testa dell'altra clausola della coppia; ad esempio la seguente coppia, verrà filtrata in tal modo:

$$example_1(a,b) \texttt{ :- } b_1(b,a), b_2(b), \cancel{b_3(b,d)}, \cancel{b_4(d)}, \cancel{b_5(d,a)}.$$

$$example_2(a,d) \texttt{ :- } \cancel{b_1(b,a)}, \cancel{b_2(b)}, \cancel{b_3(b,d)}, b_4(d), b_5(d,a).$$

dove $example_1$ ed $example_2$ sono stati definiti diversi per esplicitare i due esempi nella coppia ma contengono lo stesso funtore.

3. attraverso il metodo `computeRLGG()` viene calcolata la rlgg della coppia ed applicata la generalizzazione ad ogni argomento uguale, per ottenere una regola generale finale; dunque:

   (a) viene inizialmente calcolata la lgg della testa combinando le due teste della coppia di clausole; considerando l'esempio precedente già filtrato, la lgg sarà:

   $$lgg(example_1(a,b), example_2(a,d)) = example((a,a),(b,d)) =$$

   $$= example(X,Y), \text{ con } X = (a,a) \text{ ed } Y = (b,d);$$

   (b) successivamente viene calcolata la lgg del corpo combinando ogni termine del corpo della prima clausola di esempio con ogni termine del corpo della seconda; considerando l'esempio precedente e continuando dalla lgg della testa, le lgg del corpo saranno:

   $$lgg(b_1(b,a), b_1(d,a)) = b_1((b,d),(a,a)) = b_1(Y,X),$$

   $$\text{con } X = (a,a) \text{ e } Y = (b,d);$$

   $$lgg(b_4(b), b_4(d)) = b_4((b,d)) = b_4(Y),$$

   $$\text{con } Y = (b,d);$$

   (c) viene costruita la clausola generale dalle lgg calcolate;

   $$example(X,Y) \texttt{ :- } b_1(Y,X), b_4(Y)$$

4. attraverso il metodo `reduce()` vengono rimosse tutte le rlgg che provano almeno un esempio negativo; se tutte coprono almeno un esempio negativo, l'algoritmo termina con un eccezione;

5. attraverso il metodo `coverage()` viene provata la copertura degli esempi positivi non ancora provati per ogni rlgg rimasta e, successivamente, viene selezionata la rlgg che copre il maggior numero di esempi positivi rimasti. Se nessuna li dimostra, l'algoritmo termina con un'eccezione.

6. il `while loop` indica la ripetizione dei cinque passi precedenti fintanto che tutti gli esempi positivi non sono dimostrati.

Infine, in output può essere restituita:

- l'ipotesi indotta nella forma di Horn generalizzata, se l'induzione è andata a buon fine;
- l'eccezione `ImpossibleInductionException` che indica la non avvenuta induzione, causata da due possibili fattori:
  - rate di campionamento troppo basso: si invita l'utente a riprovare l'induzione con un rate più alto;
  - conoscenza di base insufficiente o esempi errati: non è possibile indurre un'ipotesi ragionevole da una background incompleta o da esempi con semantica differente.

La tecnica d'induzione sfruttata in questo algoritmo non permette l'apprendimento di ipotesi complesse, come le regole ricorsive, e limita fortemente l'induzione a regole semplici, aventi nel corpo solo termini con argomenti presenti nella testa. Inoltre, la costruzione del reticolo lgg può crescere in modo esponenziale all'aumentare della conoscenza di base fornita in input, rallentando notevolmente l'induzione senza condurre al risultato atteso.

3.3.3 Progol Inducer

L'induttore Progol è un sistema che esegue l'apprendimento induttivo attraverso entrambi i metodi di ricerca Bottom-Up e Top-Down e sfrutta la tecnica dell'*inverse entailment (IE)* per la costruzione delle ipotesi. In particolare, l'uso di tale tecnica implica la definizione di dichiarazioni di modalità che guidano la costruzione delle regole indotte.

Struttura

Similmente al sistema Golem, Progol rientra tra i primi algoritmi realizzati nella programmazione induttiva ma, a differenza del precedente, consente l'apprendimento di clausole più complesse non composte soltanto da letterali nel corpo contenenti i soli argomenti del letterale della testa; ad esempio:

$$head(X, Y) \text{ :- } bodyTerms(X, Y, Z).$$

dove $bodyTerms(X, Y, Z)$ rappresenta l'insieme delle congiunzioni tra i letterali del corpo aventi come argomenti non i soli X e Y della testa, ma anche un ipotetico Z calcolato per induzione.

La particolarità di Progol quindi, consiste nell'applicazione del metodo Top-Down per la costruzione di clausole generali, dette *most specific clause*, pari al numero di dichiarazioni di modalità della testa in input, ovvero regole aventi per testa il letterale dell'esempio e per corpo i soli letterali della conoscenza di base che contengono tra i loro argomenti gli argomenti della testa. In secondo luogo, applica il metodo Bottom-Up per la costruzione di *clausole con massima compressione*, ovvero regole derivanti dalle precedenti alle quali vengono rimossi i letterali del corpo non conformi all'ipotesi da indurre e, dunque, effettuando sostituzioni dal generale al particolare per provare i fatti indotti.

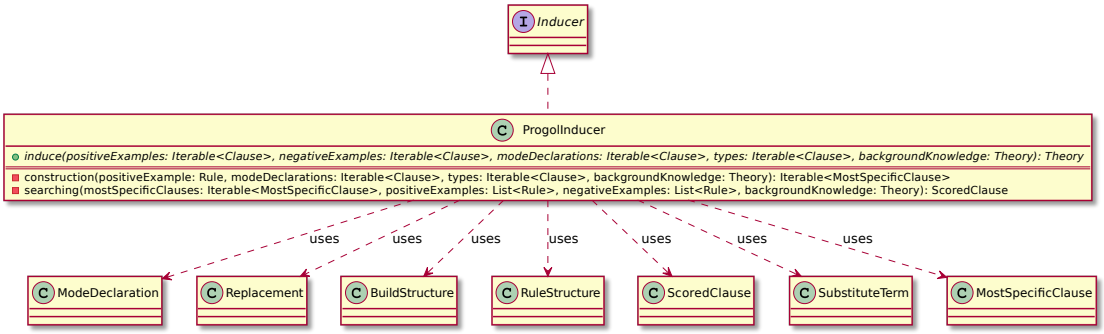

Figura 3.6: Schema Architetturale dell'Induttore Progol

Anche l'induttore Progol, implementando l'interfaccia Inducer, è caratterizzato dall'unico metodo pubblico `induce()` avente come parametri in input quelli comuni già definiti insieme a delle dichiarazioni di modalità, come bias, e ad un insieme di tipi, che definiscono le tipologie di argomenti applicabili ad ogni dichiarazione definita dall'utente. Per semplicità sintattica, nella sezione successiva, le dichiarazioni di modalità vengono indicate dalle sigle `modeh` e `modeb`; la prima per una dichiarazione del letterale della testa mentre la seconda per una dichiarazione di un letterale del corpo.

Dettaglio

Così come per l'algoritmo Golem, anche in Progol è possibile apprendere una sola regola per volta, dunque tutti gli esempi positivi per cui indurre un'ipotesi devono necessariamente avere lo stesso funtore. Inoltre questo algoritmo sfrutta la definizione di *argomenti di input* per le modeh e le modeb, ovvero argomenti che permettono di selezionare direttamente i letterali corretti al fine di non costruire regole inconsistenti. Ad esempio:

$$modeh(recall, functorH(+type, -type))$$

$$modeb(recall, functorB_1(+type, -type))$$

$$modeb(recall, functorB_2(-type, +type))$$

il `+type` di `functorH` e tutti `-type` dei $\texttt{functorB}_n$ verranno aggiunti agli argomenti di input in sequenza con lo scopo di utilizzarli al posto dei `+type` nelle singole modeb.

L'algoritmo presente può essere riassunto dal Listato 3.2:

Listato 3.2: Algoritmo Progol

```
val clausesInduced = mutableListOf<ScoredClause>
for (positive in positivesRule)
  if (notAlreadyExistInBackground(positive, background)) {
    val mostSpecificClauses = construction(positive,
       modeDeclarations, types, background)
    val clauseWithMaxCompression = searching(mostSpecificClauses,
       positivesRule, negativesRule, background)
    clausesInduced.add(clauseWithMaxCompression)
  }
clausesInduced.maxByOrNull { it.score }?.rule
```

Dunque, per ogni esempio positivo:

1. attraverso il metodo `notAlreadyExistInBackground()`, viene controllato che la conoscenza di base in input non contenga già almeno una regola che abbia lo stesso funtore dell'esempio; tale check non permetterà l'induzione di regole già esistenti e dunque previene la costruzione di uno spazio di ipotesi ridondante;

2. se non esiste nessuna regola come l'esempio corrente, attraverso il metodo `construction()`, vengono create tante most specific clause quante sono le modeh con lo stesso funtore dell'esempio. Per ogni modeh:

   (a) viene rimpiazzata la testa con il letterale esempio;

   (b) viene costruito il body attraverso la congiunzione dei letterali della conoscenza di base che rimpiazzano ogni modeb, purché abbiano:

   - stesso funtore della modeb;
   - almeno un argomento appartenente agli argomenti di input fino a quel momento trovati.

   Inoltre, la costruzione dei letterali del corpo, è guidata dal numero di presenze espresse in ogni modeb dall'intero `recall`. Intuitivamente, ogni modeb verrà ripetuta in base al numero di recall indicato;

3. attraverso il metodo `searching()`, viene ricercata la clausola con massima compressione, ovvero la clausola che rispetta la dimensione del body prevista dalla somma dei recall di ciascuna modeb utilizzata. La ricerca della clausola

avviene provando, in modo incrementale, ciascun letterale del corpo di ciascuna most specific clause ed infine selezionata quella avente score maggiore, dove:

$$score = p - (n + c + h)$$

nella quale

- p: numero di esempi positivi coperti;
- n: numero di esempi negativi coperti;
- c: lunghezza del body corrente;
- h: numero di letterali mancanti alla massima compressione.

Le ricerche effettuate su regole che non contengono tutte le variabili della testa o che hanno una dimensione del body diversa rispetto a quella indicata dalla massima compressione, attribuiscono alle regole un valore di score pari a `-20`. Questa valutazione permette di scartare in anticipo gran parte delle regole che risultano incomplete o non attinenti al target da apprendere;

4. infine, due possibili risultati:

   - se la ricerca trova almeno una clausola con massima compressione tra tutti gli esempi positivi, verrà selezionata ancora una volta la clausola con score maggiore o l'unica trovata;
   - altrimenti viene lanciata l'eccezione `InductionImpossibleException` per errata definizione delle mode o per mancanza di sufficiente base di conoscenza.

A differenza di Golem, la tecnica d'induzione sfruttata in questo algoritmo permette l'apprendimento di regole più complesse, composte da letterali nel corpo aventi non solo argomenti della testa. Tuttavia, non permette l'apprendimento di regole ricorsive o l'invenzione di nuovi predicati per induzione; inoltre, la selezione della clausola indotta per mezzo di uno score calcolato, non sancisce l'esatta correttezza dell'induzione anche se ogni clausola, durante la ricerca, viene provata contro tutti gli esempi, sia positivi che negativi.

Ciò che invece accomuna i due algoritmi è l'utilizzo di una conoscenza di base composta da fatti univoci, comuni per ogni possibile apprendimento. Dato un semplice caso di studio, in cui si suppone che l'utente voglia apprendere la seguente regola:

```
son(X,Y) :- male(X), parent(Y,X).
```

la presenza del fatto `parent` nella BK è essenziale ai fini dell'induzione e non può essere sostituito dai fatti più specifici `father`/`mother` in quanto viene a mancare la generalità dell'espressione genitoriale. Inoltre:

- in Golem, sarebbe impossibile per definizione calcolare lgg tra due predicati diversi, se entrambi presenti;
- in Progol, andrebbero definite due modeb differenti che non sono a priori dimostrabili nella BK.

Per tale ragione, l'induzione nei due algoritmi può portare ad ipotesi composte da termini diversi ma applicabili per ogni possibile induzione.

3.3.4 Metagol Inducer

L'induttore Metagol è un sistema che esegue l'apprendimento induttivo attraverso il metodo di ricerca Meta-Level in cui l'induzione alla ricerca di un'ipotesi è delegata ad un interprete di meta-livello, che adotta la tecnica del *meta-interpretative learning (MIL)*. In particolare, l'adozione del meta-interprete consiste nel recuperare ripetutamente clausole del primo ordine le cui teste sono unite con un determinato obiettivo definito dall'adozione di meta-regole che guidano la prova della ricerca.

Struttura

A differenza dei precedenti, l'algoritmo Metagol è tra i più recenti sistemi puramente simbolici realizzati; attraverso la definizione di meta-regole, è in grado di apprendere clausole complesse e soprattutto ricorsive, consentendo dunque una più complessa costruzione delle ipotesi.

La caratteristica fondamentale dunque è la definizione di tali regole che portano l'induzione ad un livello più alto di astrazione, in cui non sono presenti soltanto sostituzioni del *primo ordine*, ovvero sostituzioni riguardanti gli argomenti dei termini, ma vengono introdotte le così dette sostituzioni del *secondo ordine*, ovvero sostituzioni riguardanti i funtori dei termini. In tal modo, la struttura delle clausole indotte viene forzata da queste regole che guidano il passo d'induzione verso ipotesi corrette. Altra peculiarità dell'algoritmo consiste nella realizzazione della *predicate invention* [30, 9], ovvero la costruzione di predicati ad-hoc creati durante l'induzione per due principali ragioni:

- *Bias Shift*: eliminazione della ridondanza di termini;
- *Recursive Invention*: definizione di clausole higher-order.

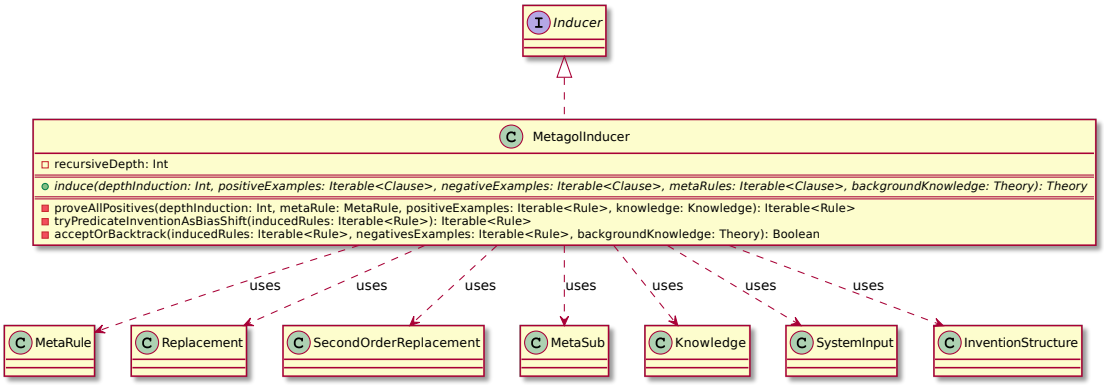

Figura 3.7: Schema Architetturale dell'Induttore Metagol

Anche l'induttore Metagol, è caratterizzato dall'unico metodo pubblico `induce()` avente come parametri in input quelli comuni già definiti insieme a delle meta-regole, come bias di costruzione. Inoltre, sfrutta il campo privato `recursiveDepth`, inizializzato a `depthInduction`, per limitare la profondità dell'induzione, nel caso di ricorsioni troppo complesse.

Dettaglio

Premessa la possibilità di apprendere una sola regola per volta, tutti gli esempi positivi per cui indurre un'ipotesi devono necessariamente avere lo stesso funtore. Inoltre, questo algoritmo sfrutta per l'induzione la definizione di meta-regole generalizzate contenenti la struttura della testa e la struttura di ogni predicato presente nel body. Nell'esempio seguente:

$$metarule(name, [X, Y], [X, A, B], [Y, A, B])$$

- *name*: nome della meta-regola; nel caso dell'esempio si tratta della meta-regola *identità*;
- $[X, Y]$: corrispondono alle variabili del secondo ordine presenti nella meta-regola;
- $[X, A, B]$: corrisponde al termine della testa con A e B variabili del primo ordine;
- $[Y, A, B]$: corrisponde all'unico termine del body con A e B variabili del primo ordine, in questo caso uguali alla testa.

Questa definizione dunque, corrisponde alla clausola generale:

$$X(A, B) \text{ :- } Y(A, B)$$

che consente l'induzione di clausole nelle quali la testa è identica al body.

La definizione delle meta-regole è assegnata all'utente e solitamente le più comuni in input, oltre l'esempio sopra visto, sono rappresentate dalla Tabella 3.1:

Name	Meta Rule	Second Orders	First Orders
inverse	X(A, B) :- Y(B, A).	X, Y	A, B
preCond	X(A, B) :- Y(A), Z(A, B)).	X, Y, Z	A, B
postCond	X(A, B) :- Y(A, B), Z(B)).	X, Y, Z	A, B
chain	X(A, B) :- Y(A, C), Z(C, B)).	X, Y, Z	A, B, C
tailrec	X(A, B) :- Y(A, C), X(C, B)).	X, Y	A, B, C

Tabella 3.1: Meta Regole Comuni

L'algoritmo presente può essere riassunto dal Listato 3.3:

Listato 3.3: Algoritmo Metagol

```
for (metaRule in metaRules) {
  var inducedRules = proveAllPositives(depthInduction, metaRule,
      positivesFact, baseKnowledge)
  inducedRules = tryPredicateInventionAsBiasShift(inducedRules)

  if (acceptOrBacktrack(inducedRules, negativesFact,
      backgroundKnowledge)) {
    return Theory.of(inducedRules)
  } // else backtrack to next meta rule
}
```

Fatte le dovute premesse, per ogni meta-regola:

1. attraverso il metodo `proveAllPositives()`, vengono provati tutti gli esempi positivi con la meta-regola corrente; per ogni positivo:
   (a) cerca di provare l'esempio con la conoscenza di base attuale o con clausole già indotte;
   (b) se non ha successo, prova l'esempio con la meta-regola corrente:
      i. rimpiazza la testa della meta-regola con l'esempio, realizzando una *meta sostituzione parziale* avente come prima variabile del secondo ordine il predicato dell'esempio, e come prime variabili del primo ordine gli argomenti dello stesso;
      ii. continua dalla meta sostituzione creata cercando la prova di ogni singolo termine del body con la conoscenza di base; se non esistono

termini che rispettano la struttura della meta-regola, fino a quel momento rimpiazzata, prova ricorsivamente ad inventare un predicato, tornando indietro al punto (a) trattando come esempio il termine corrente per il quale non si è trovato il match. Esegue tale procedura fino a rimpiazzare ogni termine del body, realizzando una *meta sostituzione completa*, o fino ad una profondità definita dal campo `recursiveDepth`, nel caso di sostituzione fallita.

(c) ritorna le ipotesi indotte dall'esempio, o una lista vuota in caso di mancata induzione.

2. provati tutti gli esempi, attraverso il metodo `tryPredicateInventionAsBiasShift()` prova ad inventare nuovi predicati sostituendo tutti i termini diversi, con le stesse variabili del primo ordine, in modo da evitare la duplicazione di regole semanticamente uguali. Ad esempio, alle seguenti regole:

$$inducedRule_1(A, B) \texttt{ :- } b_1(A), b_2(A, B)$$

$$inducedRule_1(A, B) \texttt{ :- } b_3(A), b_2(A, B)$$

viene applicata la seguente predicate invention:

$$inducedRule_1(A, B) \texttt{ :- } inducedRule_2(A), b_2(A, B)$$

$$inducedRule_2(A, B) \texttt{ :- } b_1(A)$$
$$inducedRule_2(A, B) \texttt{ :- } b_3(A)$$

Ritorna dunque le regole indotte modificate dalla predicate invention o quelle originali, nel caso non sia stato inventato nessun predicato.

3. infine, attraverso il metodo `acceptOrBacktrack()`:

   - ritorna in output le ipotesi indotte se nessun esempio negativo è coperto da esse;
   - altrimenti torna indietro al punto 1, selezionando la prossima meta-regola. Se nessuna meta-regola riesce ad indurre uno spazio di ipotesi, l'induzione fallisce con l'eccezione `InductionImpossibleException`.

Contrariamente ai precedenti algoritmi, la tecnica d'induzione sfruttata da Metagol realizza una vera e propria evoluzione nell'apprendimento delle regole logiche. La possibilità di indurre regole ricorsive e di inventare predicati su necessità, apre a possibilità d'induzione più vaste e, di conseguenza, a ragionamenti più complessi. Inoltre, non è vincolato dal problema della generalità come i precedenti; l'invenzione dei predicati e la definizione di strutture generali per le clausole con le

meta-regole, permette l'utilizzo di una conoscenza di base composta da fatti non necessariamente applicabili a tutte le induzioni.

Tuttavia, seppur un passo avanti rispetto a Golem e Progol, è ancora limitato all'induzione di regole composte da termini con arità massima pari a 2 ed una dimensione del body non superiore a 3. Infine, la scelta di un set collaudato di meta-regole per l'induzione, rimane un problema a cui trovare soluzione.

3.4 Interazione

Il framework è di natura puramente algoritmica, pertanto l'interazione con l'utente non viene messa in evidenza, poiché debolmente presente. Il contributo del framework è mostrare come avviene l'induzione e non come l'utente interagisce con il sistema affinché questa avvenga. Ciononostante, esiste una minima interazione che permette all'utente di interagire a run-time con il sistema.

Nella progettazione di tale processo, centrali per lo sviluppo sono stati i principi di:

- *Usabilità*: misura lo sforzo cognitivo dell'utente nell'utilizzare il sistema;
- *Efficacia*: misura il gap tra il risultato ottenuto e quello sperato;
- *Efficienza*: misura il rendimento e la capacità costante di risposta del sistema.

Dalla Figura 3.8 l'user, avviata l'induzione con un qualsiasi induttore, resta in attesa di ricevere l'output e, una volta ottenuto, può scegliere se cessare l'interazione con il sistema o continuare con ulteriori induzioni.

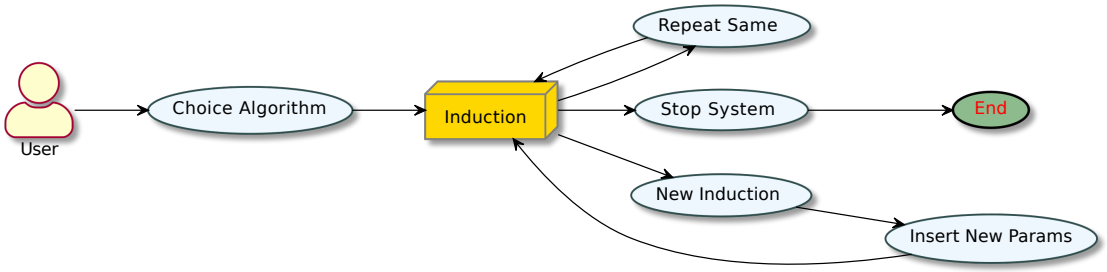

Figura 3.8: Interazione Utente

Seppur la componente *user interface (UI)* è presente in forma minima e concettualmente semplice, l'interazione sistema-utente fornisce una discreta ed intuitiva *user experience (UX)*:

- nessuna interruzione durante l'induzione;
- svolgimento di multiple induzioni senza rieseguire tutte le volte il sistema;
- l'output ottenuto è rappresentato da clausole *Horn Form* totalmente *human-readable*.

Capitolo 4

Implementazione

In questo capitolo vengono descritte le principali implementazioni sulle quali è basato lo sviluppo del framework, focalizzandosi sui principali *Tipi* utilizzati e sulla descrizione e le funzionalità delle *Data Class* realizzate.

4.1 Tipi: Termini Logici

I termini logici, in 2P-Kt, sono rappresentati dal tipo base `Term`, radice della gerarchia di termini come mostra la Figura 2.1. La caratteristica fondamentale dei tipi `Term` è che essi sono strutture dati *immutabili*, pertanto tutti i loro sottotipi non hanno proprietà e metodi pubblici; questo favorisce non solo l'implementazione, ma ne consente un controllo accurato, in quanto una proprietà verificata in precedenza su un termine resterà sempre la stessa. Al fine dell'implementazione del framework, i principali sottotipi utilizzati sono definiti di seguito.

4.1.1 Variabili

Una *variabile*, in 2P-Kt indicata dal tipo `Var`, è un tipo di termine che funge da segnalatore di un altro termine; in altre parole, identifica la posizione nella quale un termine dovrà essere inserito. Sono rappresentate nella forma:

$$\_nameVariable\_intSuffix$$

definita dall'espressione regolare

$$[\_A-Z](\_A-Za-z0-9)*$$

e possono essere istanziate da due metodi statici:

- `Var.of(name:String)`: crea una nuova istanza di Var che abbia nome uguale a `name` ed il suffisso è scelto non deterministicamente da 2P-Kt;

- `Var.anonymous()`: crea una nuova istanza di Var che abbia nome uguale a `"_"` ed il suffisso è scelto come il precedente. Una variabile senza nome è dunque detta *anonima*.

Due variabili possono essere definite uguali se hanno *nome completo* uguale, ovvero quando hanno lo stesso nome e lo stesso suffisso. È importante sottolineare infatti che due nuove istanze, che abbiano lo stesso nome, creano due variabili diverse; ad esempio:

```
Var.of("X") = X_0 e Var.of("X") = X_1
```

Nel caso in cui sia necessario il riutilizzo della stessa Var, quest'ultima dev'essere assegnata ad una variabile affinché resti la stessa:

```
val x:  Var = Var.of("X")
```

Il riutilizzo delle variabili è stato sfruttato durante la generalizzazione delle ipotesi per assegnare la stessa variabile agli argomenti di termini diversi.

4.1.2 Strutture

Una *struttura*, in 2P-Kt indicata dal tipo `Struct`, è un tipo di termine composto da altri termini. È caratterizzata da un *funtore*, una stringa che indica il nome della struttura, e da *un'arità*, un numero che indica da quanti termini è composta; nel caso specifico, una struttura con arità uguale a 0 è detta *atomo*, rappresentata dal sottotipo `Atom`, ed ha la caratteristica di avere un valore pari al suo funtore. Dunque, una struttura può essere istanziata nel seguente modo:

```
Struct.of(functor, termsList)
```

nella quale:

- `functor`: stringa rappresentante il nome della struttura;
- `termsList`: lista di termini contenuti nella struttura.

Ad esempio, supponendo `example(a,b)`, `a` e `b` tre termini, possono essere istanziati come segue:

```
example(a,b) = Struct.of("example", a, b)

a = Struct.of("a")

b = Struct.of("b")
```

A seconda dell'interpretazione che viene data ai termini contenuti nella struttura, possono essere definite differenti strutture, dette `Collections`. La struttura classica utilizzata nell'implementazione del framework è definita dal sottotipo `Clause`.

4.1.3 Clausole

Una clausola, in 2P-Kt indicata dal tipo `Clause`, è un sottotipo di Struct utilizzato per la rappresentazione delle clausole di Horn, ovvero clausole aventi un termine come testa ed un termine come body.

In base alla presenza o all'assenza della testa e del corpo, una clausola può essere categorizzata in:

- *Regola*, `Rule` in 2P-Kt: clausola caratterizzata da un termine nella testa e da un Term nel corpo. Se il termine del corpo della regola assume valore `true`, allora la regola viene detta *Fatto*, `Fact` in 2P-Kt;

$$Rule)\ Head\ \texttt{:-}\ Body \qquad Fact)\ Head\ \texttt{:-}\ true$$

- *Direttiva*, `Directive` in 2P-Kt: clausola caratterizzata dalla testa nulla e da un Term nel corpo.

$$Directive)\ \ \texttt{:-}\ Body$$

In generale, le tipologie di clausole evidenziate possono essere anch'esse istanziate attraverso il metodo statico `of`:

```
of(head:  Struct, varargs body:  Term)
```

in cui, la keyword `varargs`, indica la possibilità di costruire il body attraverso una congiunzione di Term. Dunque, per ogni sottotipo:

- *Clausole e Regole*:

```
Clause.of(head?, body)    Rule.of(head, body)
```

  il "?" indica il possibile valore `null` della testa in caso di direttive;

- *Fatti e Direttive*:

```
Fact.of(head)    Directive.of(body))
```

4.1.4 Teorie

Una teoria, in 2P-Kt indicata dal tipo `Theory`, è un sottotipo di `Iterable` utilizzato per la rappresentazione di tutta la conoscenza di base in input e dello spazio di ipotesi indotto. Fornisce, dunque, una gestione di alto livello delle clausole intesa come un'interfaccia attraverso la quale è possibile accedere ad una conoscenza logica di base. Ogni teoria è composta da un insieme di clausole che ne caratterizzano la semantica e, come per i precedenti tipi, è possibile sfruttare diversi metodi factory per realizzarne un'istanza. Similmente alle Clause Collections possono essere mutabili o immutabili e possono essere implementate come:

- *indexed*: godono di maggiore velocità di look-up a discapito di una maggiore occupazione dello spazio.
- *list-based*: godono di una velocità di look-up ridotta ma preservano l'ordine delle clausole ed offrono maggiore efficienza in termini di memoria.

In base alla presenza o all'assenza di clausole al suo interno, essa può essere istanziata in differenti modi:

- `Theory.of(varargs Clause)`: per istanziare una teoria non vuota, contenente le clausole date. In particolare:
  - `Theory.indexedOf(varargs Clause)`: per istanziare una teoria non vuota indicizzata, contenente le clausole date;
  - `Theory.listedOf(varargs Clause)`: per istanziare una teoria non vuota supportata da una lista, contenente le clausole date;
- `Theory.empty()`: per istanziare una teoria vuota. In particolare:
  - `Theory.emptyIndexed()`: per istanziare una teoria vuota supportata da una struttura dati indicizzata;
  - `Theory.emptyListed()`: per istanziare una teoria vuota supportata da una struttura di tipo lista;

Infine, le principali operazioni eseguibili su una teoria sono:

- `assertA(Clause)` o `assertZ(Clause)`: aggiungono rispettivamente la clausola data all'inizio o alla fine;
- `retract(Clause)`: elimina la clausola data dalla teoria;

4.2 Data Classes

Uno dei vantaggi nello sfruttare un linguaggio multi paradigma come *Kotlin* per l'implementazione, è quello di permettere la definizione di *Data Classes* derivanti direttamente dalla programmazione funzionale. Una data class è un modo semplice e comodo di definire una classe, in poche righe di codice, che ha il ruolo di contenitore di dati, senza che essa abbia gli appositi getter e setter per ogni proprietà che la caratterizza; il compilatore deriva automaticamente i precedenti metodi da tutte le proprietà dichiarate nel costruttore. Inoltre, possono estendere altre classi senza eseguire l'override dei metodi, tuttavia, non è permesso fornire un'implementazione esplicita dei metodi del sopratipo.

Affinché una data class sia utilizzabile, è necessario che:

- abbia almeno una parametro nel costruttore;
- non sia astratta, aperta, interna o sealed;
- tutti i parametri definiti debbano essere contrassegnati come `val` o `var`, in base alla mutabilità della proprietà.

La realizzazione di data class per lo sviluppo del framework ha trovato ispirazione dalla continua necessità di utilizzare coppie/triple di valori con lo scopo di memorizzare lo stato di avanzamento delle sostituzioni `Var/Term` durante l'induzione. Successivamente, il loro utilizzo è diventato imprescindibile al fine di rendere più semplice lo sviluppo ed ottenere un codice più chiaro, fornendo nomi più significativi alle proprietà rispetto agli standard `Pair/Triple`.

4.2.1 Golem: Data Classes

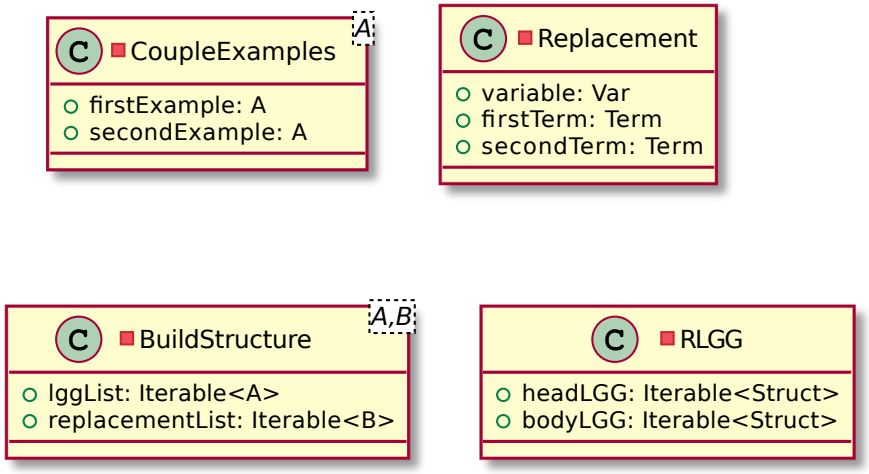

Figura 4.1: Data Classes: Induttore Golem

CoupleExamples

La data class `CoupleExamples` rappresenta la classe creata all'inizio dell'induzione per la realizzazione delle coppie di esempi dalle quali apprendere lo spazio delle ipotesi. La peculiarità della classe consiste nell'utilizzo del *generic A* che indica il tipo degli esempi nella coppia (Fatti, Regole o altri).

Replacement

La data class `Replacement` rappresenta, come suggerisce il nome, la classe per la memorizzazione di una sostituzione `Var/(Term,Term)` avvenuta durante l'induzione. Dunque, il campo `variable` rappresenta la variabile da inserire ogni qualvolta l'induttore incontra una coppia di termini definita da `firstTerm` e `secondTerm` durante il calcolo delle lgg.

BuildStructure

La data class `BuildStructure` rappresenta la classe utilizzata durante la costruzione della rlgg tra due esempi. Attraverso l'utilizzo dei *generics* tiene memoria della corrente lista delle lgg calcolate, `lggList`, e delle sostituzioni avvenute fino a quel momento, `replacementList`.

RLGG

La data class `RLGG` è stata realizzata per maggiore leggibilità del codice. Essa indica la rlgg computata, formata dalle lgg tra le teste degli esempi, `headLGG`, e le lgg tra i termini del body degli esempi, `bodyLGG`.

4.2.2 Progol: Data Classes

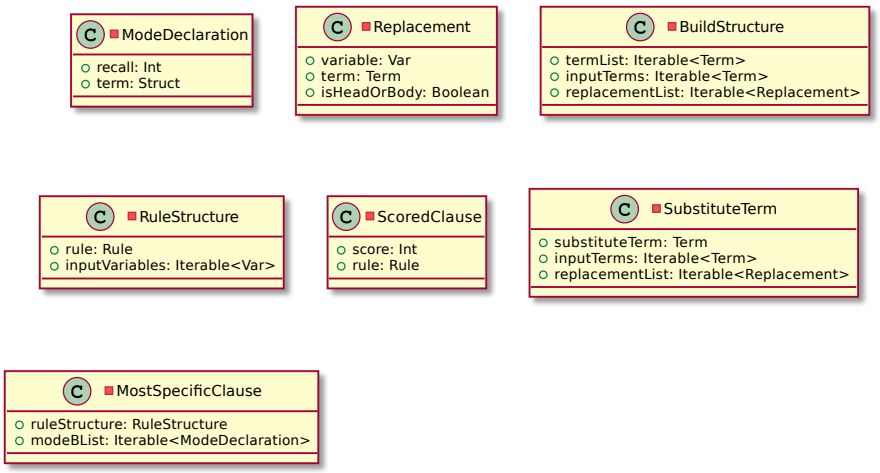

Figura 4.2: Data Classes: Induttore Progol

ModeDeclaration

La data class `ModeDeclaration`, rappresenta la traduzione sintattica delle modeh e delle modeb ottenute in input dal sistema. In particolare, il campo `recall` indica il numero di volte che può essere utilizzata la modalità mentre, il campo `term` indica la sua struttura.

Replacement

La data class `Replacement` ha la stessa semantica della medesima data class in Golem; tuttavia, in questo caso, rappresenta la memorizzazione di una sostituzione `Var/Term` avvenuta durante l'induzione, a seconda che essa riguardi un termine della testa o del body. Dunque il campo `variable` rappresenta la variabile da inserire ogni qualvolta l'induttore incontra il termine definito da `term` e, nel campo booleano `isHeadOrBody`, viene indicato se la sostituzione riguarda la testa, `true`, o il corpo, `false`.

BuildStructure

Similmente alla classe Replacement, la data class `BuildStructure` ha la stessa semantica della medesima data class in Golem. Ciò che differisce sono i campi `termList` e `inputTerms`; il primo indica la corrente lista di termini sostituti, il secondo indica la lista corrente di quelli per cui trovare una sostituzione. La necessità di definire i due campi nasce dalla modalità di costruzione dell'ipotesi di Progol. Il campo `replacementList` invece, rappresenta l'insieme corrente di sostituzioni avvenute.

RuleStructure

La data class `RuleStructure` rappresenta la struttura della regola indotta ritornata dalla costruzione. È composta dal campo `rule` che indica la regola e dal campo `inputVariables` contenente le sole variabili presenti nella testa della regola.

ScoredClause

La data class `ScoredClause`, come RLGG per Golem, è stata realizzata per pure ragioni di leggibilità. Rappresenta la clausola alla quale è stato assegnato un punteggio ed è formata dal campo `rule` e dal campo `score`.

SubstituteTerm

La data class `SubstituteTerm`, rappresenta la costruzione del singolo termine sostituto ritornato da una sostituzione. È formata dagli stessi campi della clas-

se BuildStructure ad eccezione di `substituteTerm` che indica il singolo termine sostituto anziché la lista dei sostituti.

MostSpecificClause

La data class `MostSpecificClause`, come per ScoredClause, è stata realizzata per la maggiore leggibilità del codice e rappresenta la clausola ritornata dopo la costruzione. È formata dal campo `ruleStructure`, che indica la struttura della regola costruita, e dal campo `modeBList` che contiene la lista delle modeb utilizzate durante la costruzione.

4.2.3 Metagol: Data Classes

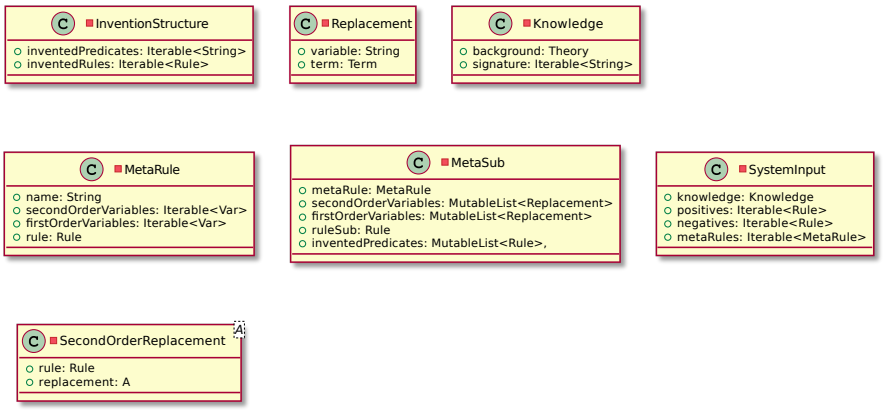

Figura 4.3: Data Classes: Induttore Metagol

InventionStructure

La data class `InventionStructure` rappresenta la struttura per la quale realizzare l'invenzione dei predicati. È formata dal campo `inventedPredicates`, che indica la lista dei funtori da rimpiazzare con un nuovo predicato, e dal campo `inventedRules`, che indica la lista dei nuovi predicati.

Replacement

Similmente alle precedenti, la data class `Replacement` rappresenta una sostituzione. Ogni Replacement è carattarizzato dal campo `variable`, stringa che indica la variabile utilizzata, e il campo `term`, indicante il termine associato alla variabile. In Metagol, un replacement può essere riferito ad una sostituzione del primo o del secondo ordine.

Knowledge

La data class `Knowledge` rappresenta una più articolata rappresentazione della conoscenza di base. Quest'ultima non è soltanto caratterizzata dalla teoria di base, il campo `background`, ma ad essa viene aggiunto il campo `signature` rappresentante l'insieme dei funtori definiti nella conoscenza di base.

MetaRule

Come per le ModeDeclaration in Progol, la data class `MetaRule` rappresenta la traduzione sintattica delle meta-rules ottenute in input dal sistema. In particolare, il campo `name` indica il nome della meta-regola, i campi `secondOrderVariables` e `firstOrderVariables` indicano rispettivamente le variabili del primo e del secondo ordine presenti nella meta-regola, infine il campo `rule` rappresenta la regola formale.

MetaSub

La data class `MetaSub` rappresenta lo stato corrente delle sostituzioni avvenute nella MetaRule. Definisce la meta-regola di riferimento attraverso il campo `metaRule`, le attuali sostituzioni del primo e del secondo ordine attraverso i campi `secondOrderVariables` e `firstOrderVariables`, l'attuale regola rimpiazzata con il campo `ruleSub` e gli eventuali predicati inventati, scaturiti dalle sostituzioni, con il campo `inventedPredicates`.

SystemInput

La data class `SystemInput` rappresenta una compatta formattazione dell'input del sistema trasformando ogni parametro in una struttura più comoda. È caratterizzata dalla conoscenza di base `knowledge`, dagli esempi positivi/negativi `positives` e `negatives` e dalle meta-regole `metaRules`.

SecondOrderReplacement

La data class `SecondOrderReplacement` rappresenta il caso specifico di sostituzione del secondo ordine. È infatti caratterizzata dall'attuale regola sostituita `rule` e dalla/e sostituzione/i, sfruttando l'utilizzo del *generic* nel campo `replacement`.

4.3 Sottocomandi CLI

In Kotlin, lo scopo di utilizzare una *classe astratta* è quello di poterne realizzare delle derivazioni, definendo al suo interno metodi e proprietà, tutti o in parte,

anch'essi astratti. Questo permette d'implementare il concetto di *polimorfismo*, uno dei punti cardine della programmazione orientata agli oggetti, che consente di assegnare comportamenti differenti ad istanze di classi differenti derivanti tutte dalla stessa classe astratta.

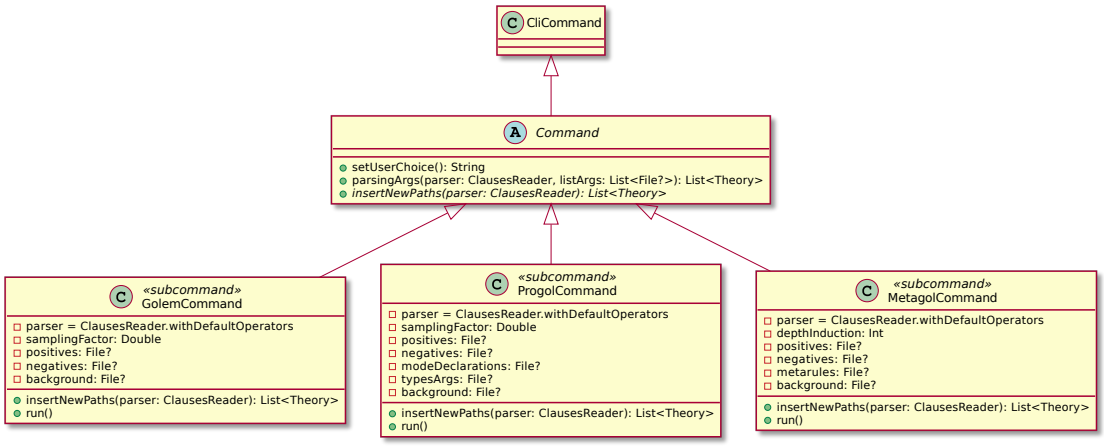

Figura 4.4: Gerarchia Sottocomandi CLI

Come raffigurato nella Figura 4.4, nella CLI del framework realizzato, ogni algoritmo è eseguibile attraverso uno specifico sottocomando dell'interfaccia utente, implementato attraverso una classe che estende la classe astratta `Command`. Quest'ultima classe, a sua volta, estende la classe principale della libreria CliKt, `CliktCommand`, che permette la definizione del comando.

In questo contesto, il polimorfismo è applicato per definire:

1. il differente inserimento dei parametri per le nuove induzioni, come suggerito dall'override del metodo astratto `insertNewPaths()`;

2. la differente esecuzione dell'operazione d'induzione, come suggerito dall'override del metodo `run()`.

Capitolo 5

Validazione

5.1 Testing

Il software testing indica un'indagine sulla qualità del prodotto o del servizio. È l'attività in cui un sistema o un componente viene eseguito in determinate condizioni, i risultati vengono osservati e viene effettuata una valutazione di alcuni aspetti del sistema o del componente.

In tale contesto, la fase di testing è stata rivolta all'osservazione dei singoli processi d'induzione, a partire da induzioni più semplici fino ad alcune più complesse, rappresentanti il funzionamento cardine dell'intero sistema. L'orientamento adottato è stato basato su:

- la libreria standard *Test* di Kotlin, per l'asserzione della correttezza delle ipotesi indotte da ciascun induttore, per garantire il corretto passo d'induzione;
- il pattern *TDD*, per lo sviluppo di codice guidato dai test;
- la proprietà di *Quality of Service (QoS)*, per focalizzare i test come parte integrante del processo di sviluppo.

L'utilizzo del pattern TDD consente un processo di scrittura del codice basato sul cosiddetto *Red-Green-Refactor Cycle* in cui, partendo da un test che fallisce, si scrive il codice affinché esso abbia successo per poi passare alla fase di refactoring. Una suite di testing pulita e comprensibile può essere ottenuta garantendo sei importanti caratteristiche:

- *Leggibilità*: ogni test dev'essere facilmente comprensibile senza confondere il lettore sullo scopo del test stesso;
- *F.I.R.S.T.*: acronimo che sta ad indicare le restanti cinque caratteristiche:

- *Fast*: la velocità è fondamentale; un test lento porta lo sviluppatore a lanciarlo raramente, non permettendo il controllo di eventuali errori;
- *Indipendent*: ogni test deve poter essere lanciato in modo indipendente dai suoi predecessori o successori; una forte dipendenza tra test provoca fallimenti a catena di cui risulta complesso comprenderne gli errori;
- *Repeatable*: ogni test dev'essere deterministico, per evitare la presenza di falsi positivi o falsi negativi che ne compromettono la correttezza;
- *Self-Validating*: ogni test dev'essere auto-validante; non deve dipendere dall'azione dello sviluppatore per asserire la sua correttezza;
- *Timely*: la velocità di scrittura di ciascun test deve poter essere minima; test complessi da realizzare portano il programmatore a scriverli in modo errato o non scriverli del tutto.

Un linguaggio moderno come Kotlin, possiede caratteristiche che permettono di aumentare al massimo la leggibilità delle suite, tra le quali la possibilità di utilizzare un linguaggio naturale nella nomenclatura dei test.

I test sono stati orientati al solo dominio delle *"family-relations"* ma sono estendibili a qualsiasi altro dominio, purché i termini non siano caratterizzati da argomenti sotto forma di liste del tipo `example([H|T])`. Le seguenti classi di test: (i) `TestGolemInducer()`, (ii) `TestProgolInducer()`, (iii) `TestMetagolInducer()`, contengono simili test d'induzione per evidenziare le potenzialità di ciascun induttore. In particolare viene posto l'accento sui test principali in ordine di complessità, utili al confronto tra le diverse induzioni:

- `testInduceSon` e `testInduceDaughter`: testano le semplici relazioni di parentela `son`/`daughter` e, in riferimento all'osservazione fatta in 3.3.3, gli algoritmi Golem e Progol generano le ipotesi:

```
son(X,Y) :- male(X), parent(Y,X).
daughter(X,Y) :- female(X), parent(Y,X).
```

  mentre Metagol genera l'insieme di ipotesi:

```
son(X,Y) :- male(X), son1(Y,X).
son1(X,Y) :- father(X,Y); mother(X,Y).
daughter(X,Y) :- female(X), daughter1(Y,X).
daughter1(X,Y) :- father(X,Y); mother(X,Y).
```

- `testInduceGrandparent`: testa la relazione `grandparent`; a questo livello di complessità è già possibile notare le differenti potenzialità d'induzione; nello specifico:

  - Golem non riesce ad indurre nessuna ipotesi per quanto detto nel Capitolo 3;
  - Progol genera un'ipotesi del tipo:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

  - Metagol invece, genera lo spazio di ipotesi:

```
grandparent(X,Y) :- grandparent1(X,Z), grandparent1(Z,Y).
grandparent1(X,Y) :- father(X,Y); mother(X,Y).
```

- `testInduceGreatGrandparent`: testa la relazione `ggrandparent`; a questo livello di complessità soltanto Metagol è in grado di produrre uno spazio delle ipotesi corretto:

```
ggrandparent(X,Y) :- ggrandparent1(X,Z), ggrandparent11(Z,Y).
ggrandparent11(X,Y) :- ggrandparent1(X,Z), ggrandparent1(Z,Y).
ggrandparent1(X,Y) :- father(X,Y); mother(X,Y).
```

- `testInduceRecurisiveAncestor`: testa la relazione di parentela più generale `ancestor`; anche in questo caso, soltanto Metagol genera uno spazio di ipotesi consistente:

```
ancestor(X,Y) :- ancestor1(X,Y).
ancestor(X,Y) :- ancestor1(X,Z), ancestor(Z,Y).
ancestor1(X,Y) :- father(X,Y); mother(X,Y).
```

- `testIdempotence`: testa la proprietà per la quale la molteplice applicazione di funzioni uguali produce sempre lo stesso risultato. In particolare, in Metagol, è previsto un ulteriore test per provare l'idempotenza su induzioni ricorsive: `testRecursiveIdempotence`;

- `testTwoTargetInSequence`: testa la possibilità di eseguire due differenti induzioni in sequenza.

Per il parsing dei file in input forniti dall'utente e per il target matching delle corrette ipotesi indotte, è stata sfruttata la libreria *it.unibo.tuprolog* messa a disposizione da 2P-Kt; in particolare:

- *dsl-theory*: per la realizzazione delle ipotesi target per le quali viene verificato il match con le ipotesi indotte dagli algoritmi;

- *parser-theory*: per la realizzazione delle teorie Prolog che simulano i file.pl forniti in input dall'utente.

5.2 Coverage

La *code coverage* è solitamente definita come la percentuale di codice eseguito dai test rispetto alla totale base di codice; il suo ruolo dunque è quello di identificare parti non testate dell'applicazione. Tuttavia, il testing mostra la presenza di errori ma non la loro assenza (Dijkstra, 1972), per cui lo scopo non è quello di ottenere una percentuale di coverage del 100%, ma di testare le parti corrette del sistema. Le tecniche di coverage impiegate durante lo sviluppo, suggerite dall'IDE sfruttato (IntelliJ), sono di seguito evidenziate:

- *Statement Coverage*: tecnica utilizzata per il calcolo del numero di istruzioni nel codice sorgente che sono state eseguite e per derivare lo scenario di esecuzione dei test sulla base dei parametri d'esecuzione inseriti;
- *Branch Coverage*: tecnica utilizzata per testare ogni risultato di un'istruzione o di un loop in modo che ogni loro condizione decisionale venga eseguita almeno una volta; aiuta a misurare frazioni di segmenti di codice indipendenti e a scoprire sezioni senza rami.

Coverage				
Class Name	Class %	Method %	Line %	Branch %
GolemInducer	100%	94,4%	97,2%	100%
ProgolInducer	100%	94,1%	96,3%	100%
MetagolInducer	100%	85,9%	94,2%	100%

Tabella 5.1: % Statement e Branch Coverage

Nella Tabella 5.1:

- *Statement (Class, Method, Line)*: mostrano le percentuali di coverage inerenti alle principali classi del Framework, i loro metodi e le linee di codice che le compongono;
- *Branch*: mostra la percentuale di coverage inerente ad ogni branch implementato.

Come si evince dallo statement coverage, la copertura delle classi stesse è massima, a differenza della copertura dei metodi e delle linee di codice. Sulla base di parametri in input corretti, il valore delle percentuali indica che per il passo d'induzione di ciascun algoritmo, non dev'essere eseguito tutto il codice prodotto ma soltanto le parti inerenti al suo corretto funzionamento. La percentuale di branch coverage del 100% invece, rappresenta la gestione di ogni possibile ramo di ciascun

branch. Tali percentuali guidano lo sviluppatore ad una corretta supervisione del sistema durante le fasi di sviluppo.

Infine, dal punto di vista semantico dell'induzione, ottenere un buon livello di coverage indica il soddisfacimento di due importanti proprietà:

- *Soundness*: la proprietà di correttezza è garantita quando le ipotesi indotte dal passo d'induzione dimostrano tutti gli esempi positivi forniti dal sistema;
- *Consistency*: la proprietà di consistenza invece è garantita quando le ipotesi indotte dal passo d'induzione dimostrano nessuno degli esempi negativi forniti dal sistema.

Soddisfare entrambe le proprietà garantisce un'induzione di successo, sulla base della definizione di una conoscenza pregressa e degli obiettivi da raggiungere.

5.3 CliKt: Kotlin CLI

CliKt è una libreria Kotlin utilizzata per la scrittura di interfacce a riga di comando semplici ed intuitive. È dunque progettata per semplificare tale processo di scrittura e consentirne una personalizzazione avanzata quando necessario. CliKt ha cinque caratteristiche fondamentali:

- consente la nidificazione arbitraria di comandi;
- offre l'utilizzo di parametri type-safe;
- supporta una grande varietà di stili di interfaccia a riga di comando;
- supporta pacchetti multi piattaforma per JVM, NodeJS, Linux, Windows e MacOS nativi;
- compatibile a partire dalla versione di Gradle 6.0.

Per il sistema sviluppato, è stata realizzata un'interfaccia che, forniti in input le teorie richieste dagli induttori, in `file.pl`, stampa in console l'induzione eseguita. L'interfaccia è dotata di tre sottocomandi, uno per ogni induttore, in modo da permettere all'utente di scegliere l'algoritmo con cui eseguire l'induzione ed i corretti parametri di input che necessitano.

L'esecuzione del sistema può avvenire in due modi:

1. run ad algoritmo singolo:
    - Golem:

```
ilp golem -s SAMPLING_FACTOR -p PATH -n PATH -b PATH
```

- Progol:

```
ilp progol -p PATH -n PATH -d PATH -t PATH -b PATH
```

- Metagol:

```
ilp metagol -s DEPTH_INDUCTION -p PATH -n PATH -m PATH -b
PATH
```

2. run ad algoritmi multipli: concatenando due o tre dei run sopra descritti, per evidenziare la differente induzione tra algoritmi, soprattutto tra Golem/Progol e Metagol.

in cui:

- `ilp`: comando principale per avviare l'interfaccia;
- `golem`, `progol`, `metagol`: sottocomandi, per avviare l'induzione con uno specifico algoritmo;
- `-s`, `-p`, `-n`, `-d`, `-t`, `-m`, `-b`: nomi delle proprietà per richiamare le tipologie di argomenti da fornire a linea di comando:
  - `-s`: precede il fattore di campionamento (double) degli esempi in Golem e la profondità dell'induzione ricorsiva (int) in Metagol;
  - `-p`: precede il path del file degli esempi positivi;
  - `-n`: precede il path del file degli esempi negativi;
  - `-d`: precede il path del file delle dichiarazioni di modalità;
  - `-t`: precede il path del file dei tipi di termini;
  - `-m`: precede il path del file delle meta-regole;
  - `-b`: precede il path del file della background knowledge.

Alla fine dell'induzione di ciascun algoritmo, l'interfaccia pone all'utente tre possibilità di scelta:

1. rieseguire la stessa induzione;
2. eseguire una nuova induzione, invitando l'utente ad inserire i nuovi path dei file necessari all'algoritmo;
3. terminare l'algoritmo:
   - se l'esecuzione è ad algoritmo singolo, il programma termina;

- se l'esecuzione è ad algoritmo multiplo, il programma provvede all'esecuzione del prossimo algoritmo definito inizialmente e riporta l'utente al punto 1.

Questo consente all'utente di poter svolgere diverse induzioni senza aprire e chiudere ripetutamente il programma.

Capitolo 6

Conclusioni

Nell'ultimo decennio, molti articoli scientifici hanno fornito un forte contributo ad ILP, lasciando margini di forte crescita per la ricerca futura. Si è partiti dalla realizzazione di semplici algoritmi, il cui scopo è l'induzione di semplici regole, fino ad algoritmi più complessi che permettono induzioni più articolate, sfruttando l'invenzione di predicati e l'uso della logica higher-order. Le numerose ricerche sul campo hanno portato ILP ad essere considerato come un possibile meccanismo di apprendimento nell'AI; si pensi all'integrazione o alla composizione tra sistemi simbolici e sub simbolici che cercano di trarre i vantaggi di entrambe le tipologie di sistemi e, allo stesso tempo, cercano di minimizzare i loro svantaggi. Tuttavia, l'apprendimento induttivo tramite la programmazione logica non presenta ancora forte stabilità come l'affermato machine learning classico. La possibilità di utilizzare differenti semantiche e bias sintattici per la manipolazione della logica, porta ad un obiettivo non ancora ben definito, sul quale possono ancora essere fatte notevoli considerazioni.

Lo sviluppo di questo elaborato propone una rivisitazione di quando fatto negli anni precedenti, cercando di arricchire ulteriormente ciò che è già stato provato e darne una maggiore chiarezza. Inoltre, cerca di porre l'evoluzione di ILP su un piano differente; pone una prospettiva evolutiva incentrata su più sistemi e non soltanto a forme e sistemi standard di apprendimento. L'utilizzo di un framework multi-piattaforma come 2P-Kt porta l'induzione logica ad un più alto livello di astrazione, non radicato alla sola programmazione logica. La implementazione dei tre principali algoritmi in questo ambito, Golem, Progol e Metagol, in chiave multi-piattaforma, offre un significativo cambiamento nel modo di osservare e manipolare il ragionamento logico, utilizzando una tecnologia alla portata di tutti.

Il contributo della tesi dunque, accresce in primo luogo l'intelligibilità, la semplicità e la coerenza della base di conoscenza nel dominio dell'induzione logica e, inoltre, offre una base implementativa differente ed evolutiva totalmente estendibile e sopratutto usabile.

Nonostante quanto detto, preesistono ancora molti limiti che in futuro devono essere affrontati e che il sistema implementato ancora non copre.

6.1 Sviluppi Futuri

Il framework realizzato nella tesi, nonostante offra un'implementazione usabile in ambito ILP in grado di far conoscere le sue caratteristiche principali e rimarcare le sostanziali differenze tra i principali algoritmi, presenta grandi margini di miglioramento in diverse sezioni implementative:

- possibilità di estendere il framework ad altri algoritmi d'induzione per raggiungere una maggiore eterogeneità nella copertura di sistemi differenti;
- affiancamento del sistema ad una metodologia sub-simbolica per la riduzione dei tempi di apprendimento.
- miglioramento del processo di testing ed introduzione di analisi delle performance su algoritmi diversi, come conseguenza dei punti precedenti;
- incremento delle percentuali di coverage aggiungendo più reazioni del sistema sulla base di input e comportamenti differenti;
- miglioramento dell'interfaccia utente per una migliore UX, non solo limitata a riga di comando;
- estensione dell'implementazione degli algoritmi realizzati a diversi domini d'induzione, come la manipolazione di liste negli argomenti dei termini;

In conclusione, ulteriori sviluppi futuri ed interessanti miglioramenti possono scaturire dall'aumento di conoscenza nella ricerca, che ogni giorno incrementa le sue potenzialità.

Bibliografia

[1] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic tensor networks, 2020.

[2] Svetla Boytcheva and Zdravko Markov. An algorithm for inducing least generalization under relative implication. In Susan M. Haller and Gene Simmons, editors, *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, May 14-16, 2002, Pensacola Beach, Florida, USA*, pages 322–326. AAAI Press, 2002.

[3] Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38(11):65–70, 1995.

[4] Roberta Calegari, Giovanni Ciatto, and Andrea Omicini. On the integration of symbolic and sub-symbolic techniques for XAI: A survey. *Intelligenza Artificiale*, 14(1):7–32, 2020.

[5] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. Lazy stream manipulation in prolog via backtracking: The case of 2p-kt. In Wolfgang Faber, Gerhard Friedrich, Martin Gebser, and Michael Morak, editors, *Logics in Artificial Intelligence - 17th European Conference, JELIA 2021, Virtual Event, May 17-20, 2021, Proceedings*, volume 12678 of *Lecture Notes in Computer Science*, pages 407–420. Springer, 2021.

[6] Giovanni Ciatto, Roberta Calegari, Enrico Siboni, Enrico Denti, and Andrea Omicini. 2p-kt: logic programming with objects & functions in kotlin. In Roberta Calegari, Giovanni Ciatto, Enrico Denti, Andrea Omicini, and Giovanni Sartor, editors, *Proceedings of the Workshop on 21st Workshop "From Objects to Agents", Bologna, Italy, September 14-16, 2020*, volume 2706 of *CEUR Workshop Proceedings*, pages 219–236. CEUR-WS.org, 2020.

[7] Andrew Cropper and Sebastijan Dumancic. Inductive logic programming at 30: a new introduction, 2020.

[8] Andrew Cropper, Sebastijan Dumancic, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30, 2021.

[9] Andrew Cropper and Rolf Morel. Predicate invention by learning from failures, 2021.

[10] Andrew Cropper, Rolf Morel, and Stephen H. Muggleton. Learning higher-order logic programs, 2019.

[11] Andrew Cropper and Sophie Tourret. Derivation reduction of metarules in meta-interpretive learning. In Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese, editors, *Inductive Logic Programming - 28th International Conference, ILP 2018, Ferrara, Italy, September 2-4, 2018, Proceedings*, volume 11105 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2018.

[12] Andrew Cropper and Sophie Tourret. Logical reduction of metarules. *Mach. Learn.*, 109(7):1323–1369, 2020.

[13] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm java-prolog integration in tuprolog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[14] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018.

[15] Manoel V. M. França, Gerson Zaverucha, and Artur S. d'Avila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Mach. Learn.*, 94(1):81–104, 2014.

[16] Céline Hocquette and Stephen H. Muggleton. Complete bottom-up predicate invention in meta-interpretive learning. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 2312–2318. ijcai.org, 2020.

[17] Robert A. Kowalski. *Logic for problem solving*, volume 7 of *The computer science library : Artificial intelligence series*. North-Holland, 1979.

[18] Mark Law. Conflict-driven inductive logic programming, 2021.

[19] Mark Law, Alessandra Russo, and Krysia Broda. The ILASP system for inductive learning of answer set programs, 2020.

[20] Nicola Leone and Francesco Ricca. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In Wolfgang Faber and Adrian Paschke, editors, *Reasoning Web. Web Logic Rules - 11th*

International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures, volume 9203 of *Lecture Notes in Computer Science*, pages 308–326. Springer, 2015.

[21] Giuseppe Marra, Francesco Giannini, Michelangelo Diligenti, and Marco Gori. LYRICS: A general interface layer to integrate logic inference and deep learning. In Ulf Brefeld, Élisa Fromont, Andreas Hotho, Arno J. Knobbe, Marloes H. Maathuis, and Céline Robardet, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part II*, volume 11907 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2019.

[22] Fumio Mizoguchi and Hayato Ohwada. Constrained relative least general generalization for inducing constraint logic programs. *New Gener. Comput.*, 13(3&4):335–368, 1995.

[23] Stephen Muggleton. Inductive logic programming. *New Gener. Comput.*, 8(4):295–318, 1991.

[24] Stephen Muggleton. Inverse entailment and progol. *New Gener. Comput.*, 13(3&4):245–286, 1995.

[25] Stephen Muggleton. Completing inverse entailment. In David Page, editor, *Inductive Logic Programming, 8th International Workshop, ILP-98, Madison, Wisconsin, USA, July 22-24, 1998, Proceedings*, volume 1446 of *Lecture Notes in Computer Science*, pages 245–249. Springer, 1998.

[26] Stephen Muggleton and Christopher H. Bryant. Theory completion using inverse entailment. In James Cussens and Alan M. Frisch, editors, *Inductive Logic Programming, 10th International Conference, ILP 2000, London, UK, July 24-27, 2000, Proceedings*, volume 1866 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2000.

[27] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[28] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[29] Chiaki Sakama. Inverse entailment in nonmonotonic logic programs. In James Cussens and Alan M. Frisch, editors, *Inductive Logic Programming, 10th International Conference, ILP 2000, London, UK, July 24-27, 2000, Proceedings*, volume 1866 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2000.

[30] Ute Schmid, Christina Zeller, Tarek R. Besold, Alireza Tamaddoni-Nezhad, and Stephen Muggleton. How does predicate invention affect human comprehensibility? In James Cussens and Alessandra Russo, editors, *Inductive Logic Programming - 26th International Conference, ILP 2016, London, UK, September 4-6, 2016, Revised Selected Papers*, volume 10326 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2016.

[31] Sheng Wang, Xiaoyin Chen, and Shengwu Xiong. Rule injection-based generative adversarial imitation learning for knowledge graph reasoning. In Kamal Karlapalem, Hong Cheng, Naren Ramakrishnan, R. K. Agrawal, P. Krishna Reddy, Jaideep Srivastava, and Tanmoy Chakraborty, editors, *Advances in Knowledge Discovery and Data Mining - 25th Pacific-Asia Conference, PAKDD 2021, Virtual Event, May 11-14, 2021, Proceedings, Part III*, volume 12714 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 2021.

[32] Zhun Yang. Extending answer set programs with neural networks. In Francesco Ricca, Alessandra Russo, Sergio Greco, Nicola Leone, Alexander Artikis, Gerhard Friedrich, Paul Fodor, Angelika Kimmig, Francesca A. Lisi, Marco Maratea, Alessandra Mileo, and Fabrizio Riguzzi, editors, *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020*, volume 325 of *EPTCS*, pages 313–322, 2020.