UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Explainable Artificial Intelligence

Prediction in Data-Driven System

Edoardo Sarri

7173337

Settembre 2025

Indice

Elenco delle figure

1 Introduzione

Questo progetto è stato svolto durante la laurea magistrale in Ingegneria Informatica per il corso di *Explainable Artificial Intelligence*.

L'idea di *Prediction in Data-Driven System* nasce dalla situazione in cui all'interno di un'azienda abbiamo un sistema di cui non è noto il modello, cioè non conosciamo i task che lo compongono e le loro caratteristiche, ma abbiamo solo i dati prodotti dal logging. In questo scenario, se abbiamo la necessità di valutare quale task potrebbe portare a dei problemi, e quindi quale dovrebbe essere modificato o studiato più approfonditamente, non possiamo fare un'analisi statica sul modello ma possiamo solo considerare i dati.

Vorremmo avere un metodo che, dato uno storico di file di log, possa prevedere se una traccia di cui non conosciamo l'esito (e.g., la traccia di log attuale che il sistema ci fornisce) avrà un fallimento e da quale task questo fallimento sarà causato.

1.1 Scenario real-time

Nel nostro contesto specifico i dati sono prodotti da un sistema real-time, cioè un sistema i cui task devono terminare entro una definita deadline. In questo contesto il fallimento di una traccia sarà quindi dovuto al superamento della deadline di un task.

Essendo in uno scenario simulato e non avendo un sistema reale, come prima cosa ci è servito un generatore di dati che ci fornisse il dataset su cui applicare le nostre idee. Per generare i dati è stato quindi usato un altro mio progetto che ci fornisce un simulatore per lo scheduling di task real-time. Tale simulatore, presente nella repo del mio GitHub real-time-scheduling-simulator, è in grado di generare un dataset di tracce dato un modello di taskset e un algoritmo di scheduling.

1.2 ILP System

Per imparare la motivazione del possibile fallimento di una traccia è stato usato un ILP (Inductive Logic Programming) system.

Questo ci permette di utilizzare la logica per definire una groundtruth, e dei predicati per spiegare il fallimento di una traccia.

Il tool utilizzato per eseguire il nostro studio è stato Numsynth. Si tratta di un'estensione di Popper che permette anche il ragionamento numerico. È stato deciso di utilizzare questo sistema per due motivi: essendo basato su Popper, ha dietro un sistema noto e solido; può eseguire valutazioni su valori numerici, cosa fondamentale visto

che i file di log sono basati su tempi in cui accadono eventi; utilizza il Learning From Failure (LFE) che permette, nonostante la complessità del ragionamento numerico, di limitare lo spazio di ricerca della soluzione.

Numsynth permette di specificare dei predicati che sono particolari e che ci potrebbero essere utili nel seguito:

- `direction(predicato,(dir1,dir2,...))`:
  Permette di specificare quali valori devono essere noti e quali saranno restituiti dal predicato. Le direzioni possibili sono: *in* definisce che il valore deve essere noto, cioè già introdotto in precedenza nella regola; *out* definisce che il valore sarà restituito dal predicato.

- `max_vars(n)`:
  Definisce il vincolo sulla complessità delle regole, specificando il numero massimo di variabili che una regola può contenere.

- `max_body(n)`:
  Definisce il vincolo sulla complessità delle regole, specificando il numero massimo di letterali che il corpo di una regola può avere.

- `magic_value_type(type)`:
  Definisce il tipo di valori che il sistema può inventare. Solitamente infatti i valori (anche quelli numerici) che compaiono nella soluzione sono solo quelli presenti nella background knowledge.

- `bounds(predicato,arg_pos,(min, max))`:
  Definisce i limiti in cui ricercare un valore numerico. Serve per diminuire lo spazio di ricerca e la complessità del training. Di default questo valore è 100.

2 Analisi

In questo capitolo analizzeremo il ragionamento che ci ha portato all'implementazione dei nostri esperimenti.

In particolar modo ci concentraremo sulle decisioni che sono state prese riguardo al representation language, che in Numsyth è contenuto nel file *bias.pl*. Una volta che abbiamo deciso come vogliamo che il sistema ci spieghi il fatto su cui facciamo inferenza, cioè definito lo spazio di ricerca, diventa semplice definire la background knowledge (file *bk.pl*) e le etichette degli esempi di training (file *exs.pl*).

2.1 Gestione delle tracce

Un punto cruciale è stato definire come il sistema deve gestire gli stessi task in tracce diverse. Deve infatti essere consapevole che due task in tracce diverse che hanno lo stesso ID rappresentano la stessa implementazione con le stesse caratteristiche, ma non deve produrre una regola usando informazioni provenienti da tracce diverse.

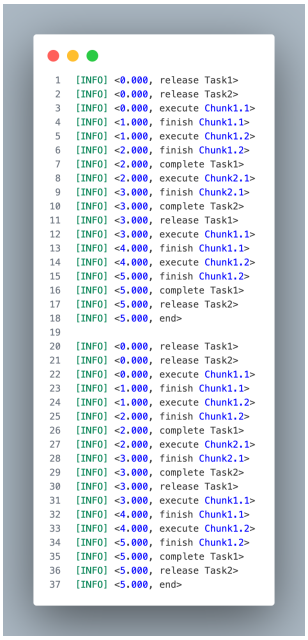
Prendiamo come esempio la Figura 2.1; questa è portata solo come esempio e ha poco senso in uno scenario reale, visto che le due tracce sono identiche. Quello che vogliamo è che il sistema non unisca le informazioni relative a quello che succede al *Task1* nella prima traccia con quello che succede allo stesso task nella seconda traccia. È però importante che sappia che il *Task1* è sempre lo stesso in entrambe le tracce, anche se non conosce le caratteristiche esplicite che il modello del sistema potrebbe fornirci.

A livello implementativo questo si traduce aumentando di un'unità l'arietà di ogni predicato, in modo da includere la traccia in cui l'evento è stato generato. Se ad esempio l'arietà di *release* sarebbe 2, cioè se le sue variabili sono tempo e task, questa diventa 3, tempo, task e traccia.

2.2 Gestione del tempo

Un altro aspetto che ha richiesto attenzione è la gestione del tempo.

Siccome Numsynth gestisce valori numerici sia di tipo *int* che *float* [1], come prima idea potrebbe venire di utilizzare il tipo *float* e prendere il tempo direttamente dal file di logging. Questo però comporterebbe avere una complessità maggiore e un tempo di training più lungo, vista la complessità nella gestione dei tipi in virgola mobile.



```
[INFO] <0.000, release Task1>
[INFO] <0.000, release Task2>
[INFO] <0.000, execute Chunk1.1>
[INFO] <1.000, finish Chunk1.1>
[INFO] <1.000, execute Chunk1.2>
[INFO] <2.000, finish Chunk1.2>
[INFO] <2.000, complete Task1>
[INFO] <2.000, execute Chunk2.1>
[INFO] <3.000, finish Chunk2.1>
[INFO] <3.000, complete Task2>
[INFO] <3.000, release Task1>
[INFO] <3.000, execute Chunk1.1>
[INFO] <4.000, finish Chunk1.1>
[INFO] <4.000, execute Chunk1.2>
[INFO] <5.000, finish Chunk1.2>
[INFO] <5.000, complete Task1>
[INFO] <5.000, release Task2>
[INFO] <5.000, end>

[INFO] <0.000, release Task1>
[INFO] <0.000, release Task2>
[INFO] <0.000, execute Chunk1.1>
[INFO] <1.000, finish Chunk1.1>
[INFO] <1.000, execute Chunk1.2>
[INFO] <2.000, finish Chunk1.2>
[INFO] <2.000, complete Task1>
[INFO] <2.000, execute Chunk2.1>
[INFO] <3.000, finish Chunk2.1>
[INFO] <3.000, complete Task2>
[INFO] <3.000, release Task1>
[INFO] <3.000, execute Chunk1.1>
[INFO] <4.000, finish Chunk1.1>
[INFO] <4.000, execute Chunk1.2>
[INFO] <5.000, finish Chunk1.2>
[INFO] <5.000, complete Task1>
[INFO] <5.000, release Task2>
[INFO] <5.000, end>
```

Figura 2.1: Trace management example.

Per semplificare l'addestramento è stato quindi deciso di rappresentare il tempo utilizzando *int*. Siccome il sistema che genera le tracce mostra i tempi in millisecondi con tre cifre decimali, come si può vedere dalla Figura 2.1, si è moltiplicato per 10^3 il tempo mostrato. In questo modo è possibile usare tipi interi, ma si deve essere a conoscenza che il tempo va considerato appunto in microsecondi.

2.3 Gestione dei task

Il nostro obiettivo è quello di fare inferenza su una traccia non vista non solo per predire il suo fallimento, ma anche per spiegarne il motivo, cioè per capire quale task è il responsabile di aver superato la sua deadline.

Con questa premessa diventa importante spiegare il fallimento di una traccia non solo sulla base degli eventi che definiscono la traccia, ma anche sulla base del comportamento dei task e dei chunk.

In questo modo non solo otteniamo una spiegazione sensata per i nostri requisiti, ma riduciamo anche la complessità del problema riducendo lo spazio di ricerca.

A livello implementativo, quanto detto sopra si traduce definendo i predicati che esplicitano gli eventi in modo che contengano anche l'ID del task e del chunk che lo ha generato.

2.4 Scelta predicati

Negli esperimenti descritti nel Capitolo 3 sono stati utilizzati diversi predicati. Questi, in un sistema di ILP, sono gli elementi base per costruire la regola.

I primi predicati usati sono quelli che compaiono nel file di log. Questi infatti li potremmo definire come predicati banali, in quanto se compaiono nel logging sicuramente sono di rilievo.

Un predicato non banale che è stato fondamentale negli esperimenti, e in particolare in quelli della Sezione 3.6, è relativo al tempo di esecuzione dei vari chunk. Questo non compare direttamente nel file di log, ma tramite uno script che elabora l'inizio e la fine dell'esecuzione di un chunk è possibile ricavarlo.

Un predicato invece che non è stato considerato per la mancanza del modello è la differenza tra il completamento di un task e la sua deadline. Non essendo la deadline un'informazione inclusa nel log, per ricavarla servirebbe il modello del taskset, cosa che però non abbiamo. Per simulare questo valore potremmo usare la differenza tra il tempo in cui un task completa e il suo prossimo rilascio: questo ha senso solo se il task è puramente periodico, dove deadline relativa e periodo coincidono; se invece il task non è puramente periodico allora potrebbe avere un periodo molto lungo (e quindi essere rilasciato di rado) ma avere una deadline molto breve (e quindi richiedere un'esecuzione immediata).

3 Esperimenti

Dopo aver eseguito una prima fase di analisi nel Capitolo 2, adesso andiamo a descrivere gli esperimenti e i risultati a cui questi portano. Di rilievo sarà la descrizione del representation bias.

Quando si esegue un esperimento è necessario, oltre ad avere a disposizione il file di log *trace.log*, definire solo il representation bias. È stato infatti pensato uno script, che dovrà essere modificato in base all'esperimento e trovarsi nella cartella, che legge il file di log e definisce i file tipici di Popper (e quindi di Numsynth) *bk.pl* e *exs.pl* necessarie per il sistema.

Una volta che abbiamo definito una cartella per l'esperimento contenente un file *bias.pl* e uno di log *trace.log*, possiamo lanciare l'esperimento dalla root directory con `./numsynth.sh <folder-name>`.

3.1 Primo approccio

Il primo approccio tentato è stato il più semplice possibile: abbiamo inserito all'interno del representation bias solo quei predicati che rappresentano ciò che i log del simulatore di taskset mostra. Questi nello specifico sono: rilascio e completamento di un task; inizio e fine dell'esecuzione di un chunk. Nella cartella *my-experiments* questo esperimento è start.

In questo modo stiamo chiedendo al sistema di spiegarci il fallimento o il successo della traccia su cui facciamo inferenza solamente tramite questi predicati. Sarà poi Numsynth a cercare di trovare delle relazioni, anche numeriche, tra chunk e task per coprire il maggior numero di esempi positivi e non coprire nessuno di quelli positivi.

3.1.1 Modello

Ogni traccia in questo esperimento ha una durata di massimo 100ms e sono state generate 100 tracce, di cui 72 hanno terminato con successo e 28 fallendo.

Il modello usato per questo esperimento, essendo il primo, è stato semplice e forse banale. È stato usato Rate Monotonic (RM) come algoritmo di scheduling e il taskset ha queste caratteristiche:

- Task1: periodo 3ms; deadline relativa 3ms; chunk1 con execution time campionato da un ConstantSampler con parametro 1ms; chunk2 con execution time campionato da un UniformSampler con parametri 0.5ms e 1.1.

- Task2: periodo 5ms; deadline relativa 5ms; chunk1 con execution time campionato da un ConstantSampler con parametro 1ms.

3.1.2 Risultati

I risultati, che possiamo osservare nel file result.txt, mostrano la semplicità del modello scelto per questo esperimento.

- Il tempo di addestramento è di circa 14s.
- La precision e la recall sono entrambe massime e pari a 1. Questo significa che sono stati coperti tutti gli esempi negativi e nessuno di quelli positivi.
- La soluzione è composta da quattro letterali: `failure(A):-execute(A,D,B,F), finish(A,C,B,F),mult(C,99,D).`

La soluzione che il sistema ha trovato ci porta a dei ragionamenti che ne evidenziano le problematiche.

In primo luogo la generalizzazione è molto bassa. Il sistema ha trovato una regola che è il massimo dei True Positive (TP) e il massimo dei True Negative (TN); probabilmente su log più complessi questa regola non riuscirà a prevedere il fallimento della traccia in modo corretto.

Dalla regola appresa possiamo vedere come il sistema classifica una traccia come corretta in base alla sua lunghezza. Nello specifico infatti la traccia ha successo se esiste un chunk *BF* che esegue al tempo D e termina al tempo C, dove $D = C \cdot 100$. Siccome 100 è il massimo valore di una simulazione (i.e., il tempo massimo in una traccia), se una traccia fallisce allora sicuramente non arriva al tempo 100.

Per confermare quanto osservato è stato fatto l'esperimento start250, identico al precedente tranne che per la lunghezza delle tracce che è stata portata fino a 250ms. Tra le varie informazioni del file result.txt possiamo osservare quanto previsto: la regola imparata è `failure(A):- finish(A,B,E,D),execute(A,C,E,D),mult(B,246,C)`, che ci conferma quanto ipotizzato.

3.2 Inversione dei positivi e negativi

Per cercare di impedire al sistema di utilizzare la lunghezza delle tracce come criterio per valutare la correttezza di una traccia, sono stati invertiti gli esempi positivi e negativi, lasciando tutto il resto come descritto nella Sezione 3.1.1, ed è stato così prodotto l'esperimento posNeg-inversion.

In questo modo il sistema dovrà cercare di spiegare come mai una traccia fallisce e non perché ha successo; non dovrebbe quindi usare la lunghezza della traccia co-

me parametro fondamentale, visto che il fallimento potrebbe avvenire all'inizio della traccia, ma anche un attimo prima del tempo massimo della simulazione.

3.2.1 Risultati

Il risultato di questo esperimento, come si può vedere nel file result.txt, mostra come i predicati descritti fino a questo momento nel representation bias (i.e., il file *bias.pl*) non sono sufficienti. Infatti il sistema non riesce a trovare una soluzione che abbia una precisione e una recall accettabili.

Inoltre, come si spiega nel paper [1], per ogni task di ricerca è predisposto un timer di 600s che permette al sistema di entrare in un loop. Questo timer in questo caso viene superato e la ricerca si arresta durante la ricerca di una soluzione con cinque letterali.

3.3 Without multiplications

Per cercare di evitare che il sistema utilizzi le moltiplicazioni per spiegare il fallimento o il successo di una traccia, gli è stata levata la possibilità di utilizzare le moltiplicazioni nell'esperimento wo-mult.

Il modello utilizzato è sempre quello del primo esperimento descritto nella Sezione 3.1.1. L'unica modifica che è stata portata è quella di eliminare le moltiplicazioni dal representation bias.

3.3.1 Risultati

La prima cosa interessante è notare come il sistema non abbia trovato una soluzione di dimensione quattro, cosa che invece succede nell'omonimo esperimento con la moltiplicazione come predicato utilizzabile.

La seconda cosa che è chiara è che il sistema non riesce a gestire l'elevata ricorsione e quindi la ricerca della soluzione fallisce.

Il risultato finale, che possiamo osservare nel file result.txt, ci fornisce due spunti molto importanti:

- Senza moltiplicazioni non si si riesce a spiegare il fallimento con una regola che abbia sia un'alta precision che un'alta recall.
- Senza predicati aggiuntivi, l'unico modo di spiegare il successo di una traccia in modo efficace (i.e., con un'alta precision e recall) è tramite la sua lunghezza.

3.4 Magic Popper

Come abbiamo detto nella Sezione 1.2, Numsynth estende Popper con i predicati numerici. Oltre a questo permette anche di utilizzare le caratteristiche di MagicPopper [2], cioè permette di far si che in una soluzione non compaiano solamente valori presenti nella background knowledge, ma anche valori inventati.

Da questa idea nasce l'esperimento magic. La configurazione è esattamente la stessa di quello senza la moltiplicazione presentato nella Sezione 3.3; nel representation bias è stato aggiunto `magic_value_type(int)` che permette a Numsynth di inventare valori per il tipo intero, cioè per il tempo.

3.4.1 Risultati

I risultati ottenuti sono esattamente quelli attesi. La regola trovata è `failure(A):-release(A,95000,D),release(A,0,D)` e ci mostra quanto il sistema si basi sulla lunghezza delle regole per classificarle.

Essa infatti ci dice che, perchè una traccia abbia successo, deve esistere un task *D* che viene rilasciato sia al tempo 0 che al tempo 95000, che praticamente è quasi il tempo massimo che una traccia può durare (100ms).

Ovviamente questo non è quanto vogliamo e ci dice ancora di più che i predicati attuali non sono sufficienti.

3.5 Descrittività predicati

Adesso analiziamo quegli esperimenti che realizzano l'idea descritta nella Sezione 2.3: vogliamo spiegare il fallimento di una traccia correlando l'evento al task che ha provocato la deadline miss.

Il modello utilizzato è ancora una volta quello base descritto nel primo esperimento nella Sezione 3.1.

Per lavorare sui predicati, è stato modificato lo script che genera i file *bk.pl* e *exs.pl*. Il risultato adesso è che per ogni predicato `body_pred(event,3)` o `body_pred(event,4)` sono stati generati tanti predicati `body_pred_taskID(event,2)` o `body_pred_taskID_chunkID(event,2)` quanti sono i task e chunk, dove le due variabili del predicato sono l'ID della traccia e il tempo in cui l'evento è stato generato.

3.5.1 Successo

Nell'esperimento event-task-chunk si è cercato di spiegare il motivo del successo di una traccia usando i predicati che descrivono sia l'evento che il chunk (e il relativo task) che lo ha generato.

I risultati sono stati deludenti per un motivo molto semplice: aggiungere il task che ha generato l'evento non impedisce al sistema di utilizzare la lunghezza della traccia per spiegare il suo successo. In realtà il risultato ottenuto è forse peggiore: la regola `failure(A):- mult(B,99,C),finish_1_1(A,B),release_1(A,C)`, che ha precision e recall massime, attribuisce il successo di una futura traccia su cui faremo inferenza solamente al Task1, considerando il suo rilascio e la fine dell'esecuzione di uno dei suoi chunk.

3.5.2 Fallimento

Nell'esperimento event-task-inversion si è invece cercato di spiegare il fallimento di una traccia usando i predicati che descrivono sia l'evento che il task che lo ha generato.

Il risultato non è soddisfacente: se osserviamo il file result.txt vediamo chiaramente che la recall della soluzione è molto bassa, e questo vuol dire che il numero di esempi non coperti dalla regola è molto basso. Questo ci testimonia il fatto che con i predicati attuali non riusciamo a spiegare il fallimento di una traccia.

3.6 Tempi di esecuzione dei chunk

Fino a questo momento abbiamo capito che il sistema cerca di spiegare i successi tramite la lunghezza delle tracce e che con i predicati attuali non riusciamo a ottenere una spiegazione che abbia una buona copertura degli esempi.

Da ora in poi eseguiremo gli esperimenti senza moltiplicazioni: non ci sono motivi infatti per cui il fallimento (ma anche il successo) di una traccia debba dipendere dal prodotto di tempi.

Inoltre non considereremo più il successo di una traccia, ma cercheremo solo di spiegarne il fallimento. Per fare questo dobbiamo aggiungere dei predicati che il sistema possa usare.

3.6.1 Modello

L'obiettivo in questo momento è cercare di capire se il sistema impara una regola corretta: ci serve un modello di taskset di cui sappiamo a priori il motivo del fallimento. Il taskset considerato ha le seguenti caratteristiche:

- Task1: periodo e deadline di 2ms. Il primo chunk esegue sempre per 0.5ms, il secondo campiona 0.5ms con probabilità del 95%, 1ms con probabilità complementare.
- Task2: periodo e deadline di 4ms. L'unico chunk ha tempo di esecuzione di 2ms.

La normale esecuzione del taskset, cioè quella che non porta a fallimento è mostrata in Figura 3.1. In questo modo è chiaro che il taskset fallisce, a causa del Task2, quando

il secondo chunk del primo task campiona un tempo di esecuzione di 1ms invece che di 0.5ms.

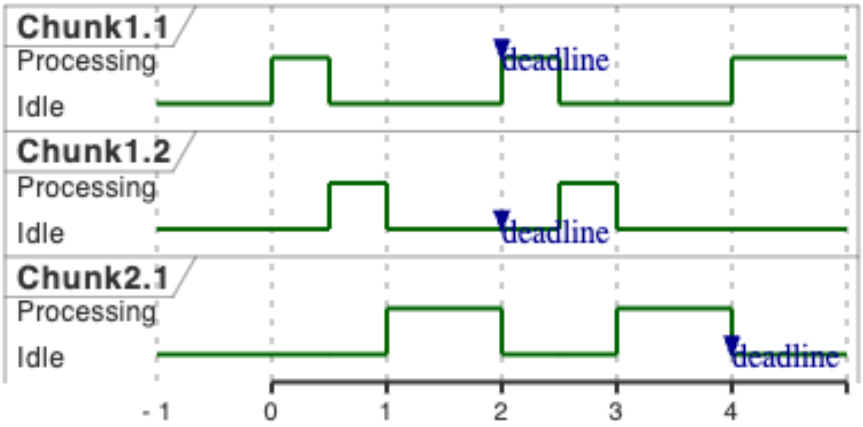


Figura 3.1: Normal execution of taskset.

3.6.2 Predicati

Per questo esperimento, oltre ad eliminare le moltiplicazioni come abbiamo detto sopra, sono stati aggiunti i predicati che modellano il tempo di esecuzione di ogni chunk. Questo dovrebbe permettere al sistema di capire come il tempo di esecuzione è responsabile del fallimento di una traccia.

Per implementare questi predicati non è stato toccato il file di log ma è stato modificato lo script che genera *bias.pl*, *bk.pl* e *exs.pl*.

3.6.3 Risultati

Per capire se quanto supposto funzionasse, in un primo momento sono state considerate solo 10 tracce, di cui 9 fallivano e 1 aveva successo. L'esperimento executionTime-10traces ha dato esattamente la soluzione che ci aspettavamo: `failure(A):-executionTime_1_2(A,C),geq(C,501)`: questo ci dice esattamente che se il tempo di esecuzione del Chunk1.2 (i.e., il secondo del primo task) campiona un valore superiore di 0.5ms allora la traccia fallisce.

3.6.4 Complessità

La nostra idea funziona, ma dobbiamo testarla su un dataset di tracce maggiore.

Nell'esperimento executionTime-100traces sono state testate 100 tracce, di cui 18 falliscono e 82 hanno successo. In questo caso il sistema trova la soluzione `failure(A):-executionTime_1_2(A,C),geq(C,501).`, che è ancora esattamente quella attesa.

Se però usiamo lo stesso numero di tracce, ma il numero di fallimenti, e quindi di esempi da spiegare, aumenta, allora il timeout scatta e non troviamo nessuna so-

luzione. Queste è infatti quello successo nell'esperimento executionTime-100traces-morefailures, dove sono le tracce che falliscono sono 27.

L'ultimo esperimento ci fa invece capire che il sistema fallisce quando il numero di tracce da spiegare è troppo elevato. Nell'esperimento executionTime-1000traces sono state usate 1000 tracce, ma di cui solo 14 falliscono. In questo caso la soluzione `failure(A):- executionTime_1_2(A,C),geq(C,1000)` è esattamente quella attesa; il valore non è più 501 come negli esperimento con meno esempi, ma comunque corretta e giusta.

3.6.5 Isolamento predicati corretti

Vogliamo vedere adesso se il sistema riesce a isolare i predicati che davvero spiegano un fallimento anche se ne sono presenti altri non necessari. È stato aggiunto il predicato sulla preemption che un task può subire, esplicito nel file di log ma non incluso fino a questo momento.

Il risultato dell'esperimento preemption mostra ancora `failure(A):-geq(C,501),executionTime_1_2(A,C)`, esattamente la soluzione corretta.

3.7 Motivazione ignota

A questo punto possiamo fare l'esperimento unknown, cioè l'esperimento di cui non conosciamo a priori la motivazione del fallimento delle tracce.

3.7.1 Definizione

Il modello scelto è semplice, in modo che possiamo valutare la soluzione del sistema. Il modello di taskset ha la seguente struttura:

- Task1: periodo e deadline di 3ms. Il primo chunk esegue sempre per 1ms, il secondo campiona 0.5ms con probabilità del 99.9% e 0.7ms con probabilità complementare.
- Task2: periodo e deadline di 5ms. L'unico chunk ha tempo di esecuzione di 2ms.

Seguendo quanto appreso dagli esperimenti precedenti riguardo alla complessità che Numsynth può gestire, abbiamo generato 100 tracce ognuna della durata massima di 100ms. Solamente circa l'1% di tracce termina con un fallimento, cioè abbiamo 8 tracce da spiegare.

3.7.2 Risultati

Come possiamo osservare da file di risultati, il sistema non riesce a trovare una soluzione. Più nello specifico va in overflow a causa di una ricorsione troppo elevata, fermandosi a regole di lunghezza cinque.

La motivazione più plausibile è che, come già osservato per gli esperimenti precedenti, i predicati che al momento sono stati forniti non siano sufficienti.

4 Risultati

In questo capitolo verranno riportati e riassunti i risultati descritti negli esperimenti del Capitolo 3. L'obiettivo è capire brevemente quali sono stati gli approcci tentati e i problemi riscontrati durante lo svolgimento del lavoro.

4.1 Lunghezza tracce

In un primo momento sono stati inseriti all'interno del representation language, oltre ai predicati numerici di base che Numsynth ci mette a disposizione, solamente i predicati che compaiono nei log (i.e., *release*, *complete*, *execute* e *finish*). In questo modo abbiamo visto, tramite gli esperimenti descritti dalla Sezione 3.1 alla Sezione 3.5, che il sistema tenta di spiegare il successo di una traccia solamente tramite la sua lunghezza.

Anche tentando un approccio diverso come spiegare il fallimento invece che il successo, come fatto nella Sezione 3.2, il risultato non è cambiato.

Questo testimonia il fatto che i soli predicati usati non sono sufficienti al sistema per ottenere un risultato interessante. Infatti eliminando la possibilità di spiegare il successo tramite la moltiplicazione, cioè tramite il predicato *mult*, il sistema non restituisce nessuna regola che abbia una buona accuracy.

Questo risultato non cambia se, come fatto per gli esperimenti nelle Sezioni 3.5, introduciamo predicati che descrivono l'evento e l'ID del task e del chunk che hanno generato, lasciando solo la traccia e il tempo come variabile.

4.2 Tempi di esecuzione dei chunk

Utilizzando un taskset di cui conosciamo il motivo del fallimento, siamo riusciti a ottenere i risultati sperati introducendo un predicato che era fondamentale perché il sistema arrivasse alla soluzione corretta. Il predicato in questione è quello relativo al tempo di esecuzione dei chunk di ogni task.

Una volta arrivati alla soluzione corretta su questo taskset, è stata valutata la gestione della complessità del sistema. Abbiamo visto che Numsynth ha difficoltà quando il numero di tracce aumenta, ma soprattutto soffre quando aumenta il numero di tracce da spiegare.

4.3 Isolamento dei predicati corretti

Nell'esperimento descritto nella Sezione 3.6.5 abbiamo visto che introducendo il predicato *preempt*, che indica quando un task subisce preemption da uno a priorità maggiore, il sistema riesce a trovare correttamente la soluzione, capendo comunque che quel predicato non è necessario.

Per arrivare a questo risultato non era necessario l'aggiunta di questo predicato in realtà, visto che anche negli esperimenti precedenti il sistema aveva ignorato altri predicati, come ad esempio il tempo di inizio e fine dell'esecuzione di task.

4.4 Taskset complesso da spiegare

Abbiamo fatto un esperimento dove la spiegazione del fallimento delle tracce non era banale da capire. Nella Sezione 3.7 abbiamo notato come il sistema non riesca a trovare una soluzione.

5 Conclusioni

Concludiamo questa relazione sul progetto con delle brevi conclusioni sulle conseguenze che il sistema Numsynth ha mostrato nei vari esperimenti descritti nel Capitolo 3 e riassunti nel Capitolo 4.

Numsynth permette di introdurre il ragionamento numerico in un ILP system. Questo nelle situazioni dove il tempo è fondamentale per apprendere qualche relazione, è una cosa fondamentale. Un altro vantaggio è quello che, quando impara le regole corrette, lo fa isolando bene i predicati che sono effettivamente necessari alla spiegazione, senza includere in essa predicati che possono essere evitati.

Il sistema è estremamente sensibile alla presenza dei corretti predicati. La sensazione è che riesca a imparare buone regole se e solo se sono stati inclusi nel representation language i predicati che spiegano effettivamente i fallimenti delle tracce. Se questi predicati non ci sono, o ne mancano anche solo alcuni, il sistema non riesce a trovare una regola che spieghi, anche solo parzialmente, il problema. Se per tracce semplici e brevi questi predicati possono essere trovati a mano, quando abbiamo molti eventi e molte combinazioni, allora il problema può diventare non risolvibile.

Un ulteriore limite di Numsynth, forse anche maggiore del precedente, è la complessità. Il sistema non riesce ad arrivare alla soluzione se il numero di tracce è troppo elevato. Inoltre è ancora più sensibile al numero di tracce positive, cioè al numero di tracce che deve spiegare: anche se il numero di tracce totali rimane lo stesso, aumentando quelle che falliscono si può non arrivare alla soluzione.

Bibliografia

[1] Céline Hocquette. *NumSynth-AAAI23*. 2023. URL: https://github.com/celinehocquette/numsynth-aaai23 (visitato il giorno 08/09/2025).

[2] Céline Hocquette e Andrew Cropper. «Learning programs with magic values». In: *Machine Learning* 112.5 (2023), pp. 1551–1595.

[3] Logic and Learning Lab. *Popper: an Inductive Logic Programming system*. 2020. URL: https://github.com/logic-and-learning-lab/Popper (visitato il giorno 08/09/2025).

[4] Jan Wielemaker. *SWI-Prolog Source Documentation*. URL: https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/pldoc.html%27) (visitato il giorno 18/09/2025).