



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

1	Introduzione	6
1.1	Scenario	6
1.2	Numsynth	6
2	Analisi	8
2.1	Gestione delle tracce	8
2.2	Gestione del tempo	8
2.3	Gestione dei task	9
3	Esperimenti	10
3.1	Primo approccio	10
3.1.1	Modello	10
3.1.2	Risultati	11
3.2	Inversione dei positivi e negativi	11
3.2.1	Risultati	12
3.3	Without multiplications	12
3.3.1	Risultati	12
3.4	Magic Popper	13
3.4.1	Risultati	13
3.5	Predicati event-task	13
3.5.1	Modello e predicati	13
4	Risultati	14
4.1	Lunghezza tracce	14

Elenco delle figure

2.1	Trace management example.	9
-----	-----------------------------------	---

Elenco delle tabelle

Listings

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, non possiamo fare un'analisi statica sul modello, ma possiamo solo considerare i dati.

1.1 Scenario

Nel nostro contesto specifico volevamo analizzare i dati prodotti da un sistema real-time, cioè un sistema i cui task devono dare garanzia di terminare entro una definita e nota deadline. In questo scenario il fallimento di una traccia sarà quindi dovuto al superamento della deadline di un task che ancora non ha terminato uno dei suoi chunk.

Essendo in uno scenario simulato, come prima cosa ci è servito un generatore di dati che ci fornisse il dataset su cui applicare le nostre idee. Questo progetto è in qualche modo collegato a un altro svolto in precedenza che ci fornisce un simulatore per lo scheduling di task real-time. Tale simulatore, presenza in 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 Numsynth

Il sistema 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 solito; 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; *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.

2 Analisi

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

In particolar modo ci concentreremo sulle decisioni che sono state prese riguardo al representation language. 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 e le etichette degli esempi di training.

2.1 Gestione delle tracce

Un punto cruciale è stato definire come il sistema deve gestire 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.

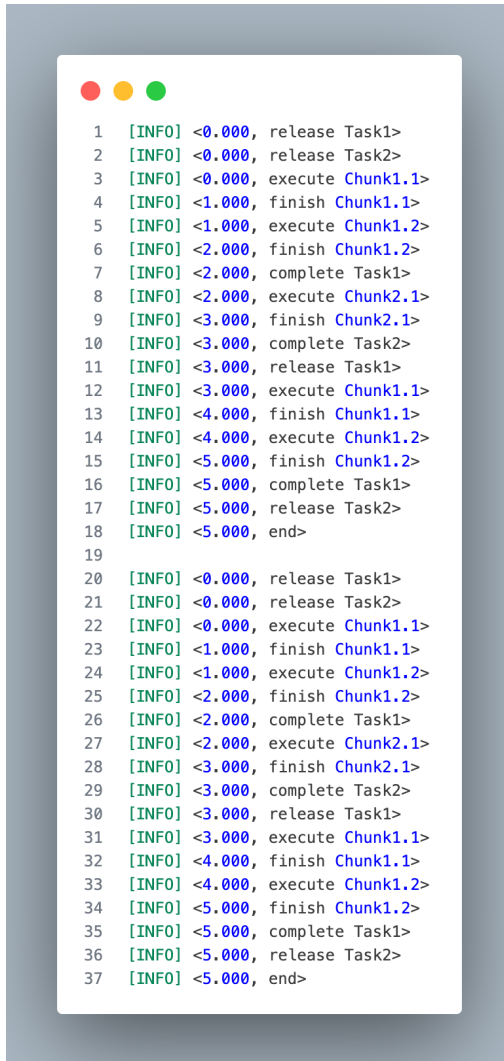
A livello implementativo questo si traduce aumentando di un'unità l'arietà di ogni predicato, in modo da includere la traccia che ha generato quell'evento. Se ad esempio l'arietà di *release* sarebbe 2, tempo e task1, questa diventa 3, tempo, task e traccia.

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 un tempo di training più lungo, vista la complessità nella gestione dei tipi in virgola mobile.

Per semplificare l'addestramento è stato quindi deciso di rappresentare il tempo utilizzando *int*.



```
1 [INFO] <0.000, release Task1>
2 [INFO] <0.000, release Task2>
3 [INFO] <0.000, execute Chunk1.1>
4 [INFO] <1.000, finish Chunk1.1>
5 [INFO] <1.000, execute Chunk1.2>
6 [INFO] <2.000, finish Chunk1.2>
7 [INFO] <2.000, complete Task1>
8 [INFO] <2.000, execute Chunk2.1>
9 [INFO] <3.000, finish Chunk2.1>
10 [INFO] <3.000, complete Task2>
11 [INFO] <3.000, release Task1>
12 [INFO] <3.000, execute Chunk1.1>
13 [INFO] <4.000, finish Chunk1.1>
14 [INFO] <4.000, execute Chunk1.2>
15 [INFO] <5.000, finish Chunk1.2>
16 [INFO] <5.000, complete Task1>
17 [INFO] <5.000, release Task2>
18 [INFO] <5.000, end>
19
20 [INFO] <0.000, release Task1>
21 [INFO] <0.000, release Task2>
22 [INFO] <0.000, execute Chunk1.1>
23 [INFO] <1.000, finish Chunk1.1>
24 [INFO] <1.000, execute Chunk1.2>
25 [INFO] <2.000, finish Chunk1.2>
26 [INFO] <2.000, complete Task1>
27 [INFO] <2.000, execute Chunk2.1>
28 [INFO] <3.000, finish Chunk2.1>
29 [INFO] <3.000, complete Task2>
30 [INFO] <3.000, release Task1>
31 [INFO] <3.000, execute Chunk1.1>
32 [INFO] <4.000, finish Chunk1.1>
33 [INFO] <4.000, execute Chunk1.2>
34 [INFO] <5.000, finish Chunk1.2>
35 [INFO] <5.000, complete Task1>
36 [INFO] <5.000, release Task2>
37 [INFO] <5.000, end>
```

Figura 2.1: Trace management example.

Siccome il sistema che genera le tracce mostra i tempi in millisecondi con tre cifre decimali, 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 considerando appunto in microsecondi.

2.3 Gestione dei task

Il nostro obiettivo è quello di fare inferenza su una traccia non solo per predire il suo fallimento, ma anche di spiegarne il motivo, cioè di 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 di task. Per questo motivo sono stati eseguiti degli esperimenti che

utilizzano predicati che descrivono sia l'evento che il task che lo ha generato.

Definire predicati che non descrivono solo l'evento ma anche il task permette inoltre di ridurre l'arietà degli stessi. In questo modo la complessità del problema diminuisce e lo spazio in cui ricercare una soluzione diventa più piccolo.

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.

Volevamo ottenere un taskset che fosse facile da comprendere. Una traccia dovrebbe avere sempre successo tranne quando il secondo chunk del primo task (i.e., chunk1.2) campio un valore superiore a 1. Essendo sotto RM, in questo caso al task2 non rimarrebbe abbastanza tempo per eseguire entro la propria deadline.

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 come 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 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 sì 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 Predicati event-task

Come abbiamo capito dagli esperimenti precedenti, usare solo i predicati che rappresentano gli eventi non è sufficiente a spiegare il successo o il fallimento di una traccia. Per questo motivo in questo momento seguiamo l'idea data nella Sezione 2.3: vorremmo spiegare il fallimento di una traccia correlandolo con il task che lo ha provocato.

3.5.1 Modello e predicati

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 3.4, 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.

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).