Predicate Invention in Rule Learning with Popper use case - When What is Known is not Enough

David M. Cerna

June 26^{th} 2024

Czech Academy of Sciences

Inductive Logic Programming (ILP)

- ILP is a form of symbolic machine learning.
- Introduced in the early 90s (Muggleton, 1991).
- **Goal:** Form an explanatory hypothesis using:
  1) Positive and negative **evidence**
  2) Provided **background knowledge**

Background Knowledge (BK)		Evidence
$\texttt{mother}(a, b).$	$\texttt{father}(g, b).$	$\texttt{grandparent}(a, d)^{+}.$
$\texttt{mother}(a, c).$	$\texttt{father}(g, c).$	$\texttt{grandparent}(g, d)^{+}.$
$\texttt{mother}(b, d).$	$\texttt{father}(f, d).$	$\texttt{grandparent}(a, h)^{+}.$
$\texttt{mother}(e, f).$	$\texttt{father}(c, h).$	$\texttt{grandparent}(g, h)^{+}.$
		$\texttt{grandparent}(a, e)^{-}.$

- $\texttt{mother}(a, b)$ denotes a is a mother of b.

Grandparent Example

- From *mother/2* and *father/2* we learn a **logic program** for

  X is a grandparent of Y

- One solution is

$$\texttt{grandparent}(X, Y)\text{:-}\texttt{mother}(X, Z), \texttt{mother}(Z, Y)$$
$$\texttt{grandparent}(X, Y)\text{:-}\texttt{mother}(X, Z), \texttt{father}(Z, Y)$$
$$\texttt{grandparent}(X, Y)\text{:-}\texttt{father}(X, Z), \texttt{mother}(Z, Y)$$
$$\texttt{grandparent}(X, Y)\text{:-}\texttt{father}(X, Z), \texttt{father}(Z, Y)$$

- A popular learning paradigm is Learning from Entailment:

$$BK, \mathbf{H} \models E^{+} \qquad\qquad BK, \mathbf{H} \not\models E^{-}$$

- **Goal:** Find **H**.

A "Better" Grandparent

BK	E^+	E^-
$\mathtt{mom}(a,b).\ \mathtt{dad}(e,b).$	$\mathrm{gp}(a,d).\ \mathrm{gp}(e,d).$	$\mathrm{gp}(a,b).\ \mathrm{gp}(b,c).$
$\mathtt{mom}(a,c).\ \mathtt{dad}(e,c).$	$\mathrm{gp}(a,f).\ \cancel{\mathrm{gp}(e,f)}.$	$\mathrm{gp}(c,f).\ \mathrm{gp}(d,f).$
$\mathtt{mom}(b,d).\ \mathtt{dad}(c,f).$		

Learning is less brittle with the parent predicate

$\mathrm{gp}(X,Y)\text{:-}\mathtt{mom}(X,C),\mathtt{mom}(C,Y)$
$\mathrm{gp}(X,Y)\text{:-}\mathtt{mom}(X,C),\mathtt{dad}(C,Y)$
$\mathrm{gp}(X,Y)\text{:-}\mathtt{dad}(X,C),\mathtt{mom}(C,Y)$
$\cancel{\mathrm{gp}(X,Y)\text{:-}\mathtt{dad}(X,C),\mathtt{dad}(C,Y)}$

$\mathrm{gp}(X,Y)\text{:-}\mathbf{p}(X,C),\mathbf{p}(C,Y)$
$\mathbf{p}(X,Y)\text{:-}\mathtt{mom}(X,Y)$
$\mathbf{p}(X,Y)\text{:-}\mathtt{dad}(X,Y)$

What is Predicate Invention (PI)?

- When learning a model (logic program) for a task, introduction of a concept which is neither:
  - given as part of the background knowledge, nor
  - the solution to the task itself.

$$qsort([],[]).$$

$$qsort([H|T],S) \text{ :- } part(H,T,L1,L2), \mathbf{qsort(L1,S1)}, \mathbf{qsort(L2,S2)}, append(S1,[H|L2],S).$$

- In *"The Appropriateness of PI as Bias Shift..."*, I. Stahl suggest that recursive PI is **Essential** to learning.

What is Predicate Invention (PI)?

- When learning a model (logic program) for a task, introduction of a concept which is neither:
  - given as part of the background knowledge, nor
  - the solution to the task itself.

$$
\begin{aligned}
&qsort([], []).\\
&qsort([H|T], S) \text{ :- } part(H, T, L1, L2), \mathbf{qsort(L1,S1)},\\
&\qquad \mathbf{qsort(L2,S2)}, append(S1, [H|L2], S).\\
&part(\_, [], [], []).\\
&part(P, [H|T1], [H|T2], L) \text{ :- } X \leq P, part(P, T1, T2, L).\\
&part(P, [H|T1], L, [H|T2]) \text{ :- } X > P, part(P, T1, L, T2).
\end{aligned}
$$

- In *"The Appropriateness of PI as Bias Shift..."*, I. Stahl suggest that recursive PI is **Essential** to learning.

Other Types of Predicate Invention

- **Compressive PI**

  Reduces program size
  grandparent using parent
- **Bias Expansion PI**

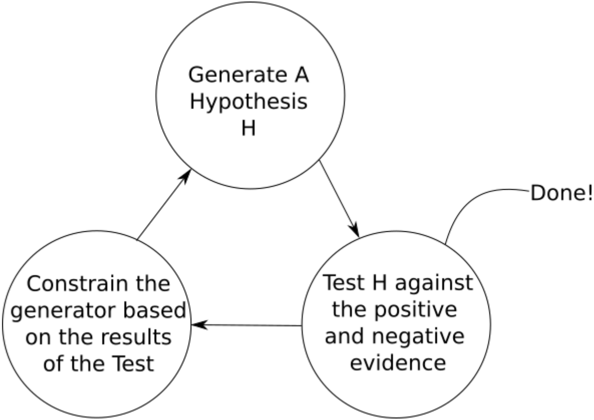
  New operations beyond the bias specified in the task
  defining repeated and useful predicate sequences
- **Negative PI**

  Provides the complement of bias expansion PI
  universal search over a finite domain
- **Higher-Order PI**

  Provides input for higher-order definitions
  Mix of all the above

Popper

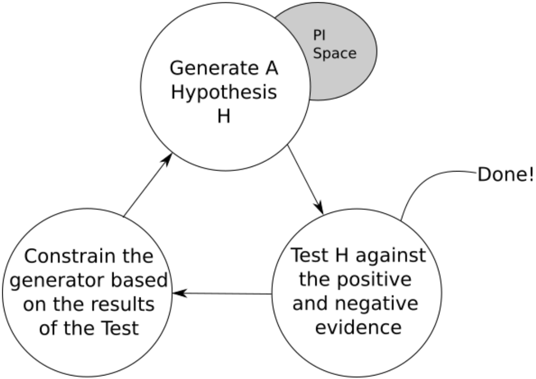
- You heard a lot about Popper last seminar from Celine:
- **Recap:**



- What changes when PI is involved?

Popper with PI

- The main difference is a larger search space.



- Within this larger search space **multiple representations** of the same program exists,
- Some **much smaller** than others.

Popper With Higher-order Definitions?

- Similar to PI, but invented predicates used for HO definitions.

$$\begin{aligned} &\mathtt{map}(P, [], []). \\ &\mathtt{map}([H1|T1], [H2|T2], P) \text{:-} P(H1, H2), \mathtt{map}(T1, T2). \end{aligned}$$

$$\begin{aligned} &\mathtt{fold}(\_, X, [], X). \\ &\mathtt{fold}(P, Acc, [H|T], Y) \text{:-} P(Acc, H, W), \mathtt{fold}(P, W, T, Y). \end{aligned}$$

$$\begin{aligned} &\mathtt{any}(P, [H|\_], B) \text{:-} P(H, B). \\ &\mathtt{any}(P, [\_|T], B) \text{:-} \mathtt{any}(P, T, B). \end{aligned}$$

- Additional bias used to learn solutions for tasks.

Introducing Higher-order Definitions

- Higher-order definitions, larger space $\Rightarrow$ smaller programs:

$$\texttt{map}(P, [], []).$$
$$\texttt{map}([H1|T1], [H2|T2], P)\text{:-}P(H1, H2), \texttt{map}(T1, T2).$$

- First-order dropLast:

$$\begin{aligned}\texttt{dropLast}(A, B)\text{:-}\ & \texttt{empty}(A), \texttt{empty}(B).\\ \texttt{dropLast}(A, B)\text{:-}\ & \texttt{con}(A, B, C), \texttt{reverse}(C, E),\\ & \texttt{tail}(E, F), \texttt{reverse}(F, G),\\ & \texttt{con}(B, G, H), \texttt{dropLast}(D, H).\end{aligned}$$

- Higher-order dropLast:

$$\begin{aligned}\texttt{dropLast}(A, B)\text{:-}\ & \texttt{map}(inv, A, B).\\ \texttt{inv}(A, B)\text{:-}\ & \texttt{reverse}(A, C), \texttt{tail}(C, D),\\ & \texttt{reverse}(D, B).\end{aligned}$$

Interesting Examples

- Is B a subtree of A:

$$\begin{aligned} \texttt{isSubTree}(A,B)\text{:- } & \texttt{eq}(A,B). \\ \texttt{isSubTree}(A,B)\text{:- } & \texttt{children}(A,C),\ \texttt{any}(p,C,B). \\ \texttt{p}(A,B)\text{:- } & \texttt{isSubTree}(A,B). \end{aligned}$$

- Is list B the second half of list A:

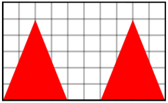
$$\begin{aligned} \texttt{lastHalf}(A,B)\text{:- } & \texttt{reverse}(A,C), \texttt{caseList}(p,q,C,B). \\ \texttt{p}(A)\text{:- } & \texttt{empty}(A). \\ \texttt{q}(A,B,C)\text{:- } & \texttt{front}(B,E), \texttt{caseList}(p,q,E,D), \\ & \texttt{app}(D,A,C). \end{aligned}$$

- **Problem:** many programs are semantically equivalent.
- **Open:** how to efficiently learn such programs?

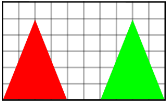
Motivating Negative PI

Consider the following game

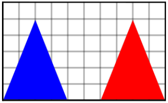
- A **teacher** chooses examples which follow a **hidden rule**.



E^{+}



E^{-}

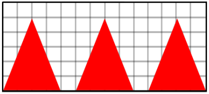


E^{-}

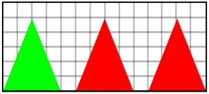
- The **students**, based on the examples, **guess** possible rules.
- For example, *"there are two red cones"*

Motivating Negative PI

- What if the **teacher** provided additional examples?



E^+



E^-

- *"[there are two red cones] or [there are three red cones]"*?
- What if the **teacher** provided examples **with up to n cones**?

Motivating Negative PI

- Obviously, we want to generalize beyond case enumeration.

  "All cones are red"

- Most **ILP** approaches struggle to learn such rules.
- Often the *hypothesis space* is restricted to
  - Datalog,
  - Datalog$^+$, or
  - Definite programs

Motivating Negative PI

- None are capable of encoding the required **universal** search.

$$f(S) \text{ :- } scene(S), \textbf{not } inv_1(S).$$
$$inv_1(S) \text{ :- } cone(S, P), \textbf{not } red(P).$$

- that is **“there does not exists a cone which is not red.”**
- How does one learn such *Stratified Logic Programs*?

Beyond Definite Clause Subsumption

Definition

A clause C_1 **subsumes** a clause C_2 if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$.

Definition

A *definite program* P subsumes a definite program Q ($P \preceq_\theta Q$) if and only if $\forall r_2 \in Q, \exists r_1 \in P$ such that r_1 **subsumes** r_2.

Theorem (**Entailment property**)

Let P and Q be definite programs s.t. $P \preceq_\theta Q$. Then $P \models Q$.

- Does not hold for **Stratified Programs**.

Beyond Definite Clause Subsumption

$$P = \left\{ \begin{array}{r} a. \\ f \text{ :- } \textbf{not } inv_1. \\ inv_1 \text{ :- } b. \\ inv_1 \text{ :- } a. \end{array} \right\} \qquad Q = \left\{ \begin{array}{r} a. \\ f \text{ :- } \textbf{not } inv_1. \\ inv_1 \text{ :- } b. \end{array} \right\}$$

$P \preceq_\theta Q$ and $Q \models f$ but $P \not\models f$. (P more specialised)

$$P' = \left\{ \begin{array}{r} a. \\ f \text{ :- } \textbf{not } inv_1. \\ inv_1 \text{ :- } a, b. \end{array} \right\} \qquad Q' = \left\{ \begin{array}{r} a. \\ f \text{ :- } \textbf{not } inv_1. \\ inv_1 \text{ :- } a. \end{array} \right\}$$

$Q' \preceq_\theta P'$ and $P' \models f$ but $Q' \not\models f$. (P' more general)

Generality and speciality swap.

Polar Fragment

- **Stratified Program:**

  Strict order on negated occurances of head predicate symbols.

  $\Longrightarrow$ **No Recursion** through negation.
  $\Longrightarrow$ A head predicate symbol may occur both **negated and non-negated**. (Breaks subsumption)
- We label head symbols as Positive, Negative, or Neither.
- Depends on use of negation relative to symbol occurance.

Definition (**Polar program**)

A stratified program P is polar if and only if every head predicate symbol in P is **exclusively positive or negative**.

Polar Subsumption

- Rules of P with positive (negative) head symbols are positive, that is in P^+ (negative, that is in P^-).

$$
\begin{aligned}
r_1 : &\quad \mathit{unconnected}(A,B) \text{ :- } \textbf{not } \mathit{inv}_1(A,B)\\
r_2 : &\quad \mathit{inv}_1(A,B) \text{ :- } \mathit{edge}(A,B)\\
r_3 : &\quad \mathit{inv}_1(A,B) \text{ :- } \mathit{edge}(A,C), \mathit{inv}_1(C,B)
\end{aligned}
$$

- For *unconnected/2*, $P^+ = \{r_1\}$ and $P^- = \{r_2, r_3\}$.

Definition (Polar subsumption)

Let P and Q be polar programs. Then P *polar subsumes* Q ($P \preccurlyeq_{\diamond} Q$) iff $P^+ \preceq_\theta Q^+$ and $Q^- \preceq_\theta P^-$.

Theorem (Entailment property)

Let P and Q be polar programs: $P \preccurlyeq_{\diamond} Q \Longrightarrow P \models Q$.

Learning from Failures with Negation (LFFN)

- An extension of **Learning from Failures** (A. Cropper & R. Morel, 2020) to polar programs
- Takes as input *positive* (E^+) and *negative* (E^-) examples, *background* (B), polar programs ($\mathcal{H}$), and constraints (C).

Definition (LFFN solution)

Given $(E^+, E^-, B, \mathcal{H}, C)$, an $H \in \mathcal{H}_C$ is a **solution** if H is *complete* $(\forall e \in E^+,\ B \cup H \models e)$ and *consistent* $(\forall e \in E^-,\ B \cup H \not\models e)$.

Constraint Soundness: Let $(E^+, E^-, B, \mathcal{H}, C)$ and $H_1, H_2 \in \mathcal{H}_C$:

Generalisation

If H_1 is inconsistent and $H_2 \preccurlyeq_\diamond H_1$, then H_2 is not a solution.

Specialisation

If H_1 is incomplete and $H_1 \preccurlyeq_\diamond H_2$, then H_2 is not a solution.

NOPI

- *NOPI* Learns **Polar Programs** and
- prunes the search space using **non-monotonic constraints**.

H$_1$:

$$r_1: \quad f(S) \text{ :- } scene(S), \textbf{not } inv_1(S)$$
$$r_2: \quad inv_1(A, B) \text{ :- } cone(S, A)$$

H$_2$:

$$r_1: \quad f(S) \text{ :- } scene(S), \textbf{not } inv_1(S)$$
$$r_2: \quad inv_1(A, B) \text{ :- } cone(S, A)$$
$$r_3: \quad inv_1(A, B) \text{ :- } contact(S, A, \_), red(A), \textbf{not } blue(A)$$

- Observe, $H_1 \preccurlyeq_{\diamond} H_2$. If H_1 is **incomplete**, then H_2 is pruned by a **polar specialisation constraint**.

Example Programs

- Is B an independent set of A:

$$ind(A,B) \text{ :- } \textbf{not } inv1(B,A).$$
$$inv1(A,B) \text{ :- } member(C,A), edge(B,D,C), member(D,A).$$

- Is B a dominating set of A:

$$\begin{aligned} dominating(A,B) &\text{ :- } \textbf{not } inv1(A,B). \\ inv1(A,B) &\text{ :- } node(A,C), \textbf{not } member(C,B), \\ &\qquad \textbf{not } inv2(A,B,C). \\ inv2(A,B,C) &\text{ :- } edge(A,C,D), member(D,B). \end{aligned}$$

- **English:** For all nodes C in the graph A that are not members B there exists a node D in B such that there is an edge from C to D.

Future Work

- There are many open problems concerning PI:
  - **Obstructive**: When PI is unnessary it makes learning harder
  - **Twins**: There are many ways to write the same program
  - **Efficiency**: are some uses of PI completely useless
  - **modularity**: sperating PI from the main program
  - **Domain Specificity**: What problems need which type of PI