Designing of Nonmonotonic Inductive Logic Programming Systems

Chongbing Liu

October 24, 2005
Outlines

■ Basic Algorithms and Properties

■ Sequential Learning Algorithm

■ Parallelization
Basic Algorithms and Properties

Necessary conditions

Learn from a single positive example

Learn from a single negative example

Learn from a set of examples

General properties
Necessary Conditions

Given

\(B\): a program, \(H\): a rule, \(E\): a ground literal.

Proposition

\[B \cup \{H\} \models E \text{ and } B \models H \implies B \models E \quad (i)\]

From (i), we can prove

\[B \not\models E \text{ and } B \cup \{H\} \models E \implies B \not\models H \quad (ii) \quad (E: \text{ positive example})\]

\[B \models E \text{ and } B \cup \{H\} \not\models E \implies B \not\models H \quad (iii) \quad (E: \text{ negative example})\]

Trivial hypothesis

- Let \(M^+ = M \cup \{l \mid l \notin M \text{ and } l \in \mathcal{HB}\}\), where \(M\) is the stable model of \(B\) and \(\mathcal{HB}\) is the Herbrand model of \(B\).

- \(B \not\models H\) implies \(M^+ \not\models H\).

- Let \(\Gamma = \{K \in M^+ \mid K \text{ is relevant to } E \text{ and is involved in } B \cup \{E\}\}\).

- Since \(M^+ \models \Gamma\), we have \(M^+ \not\models r_0\) where \(r_0 = \leftarrow \Gamma\).

- Integrity constraint \(r_0 = \leftarrow \Gamma\) is a trivial valid (ground) candidate of \(H\).
Learning from a single positive example

algorithm: Learn-single-pos

Input: a categorical program $B$, a ground atom $E$ (positive)

Output: a rule $R$

1. compute the answer set $M$ of $B$ and its expansion set $M^+$;
2. construct the integrity constraint $\leftarrow \Gamma$ from $M^+$;
3. produce a rule $E \leftarrow \Gamma'$ by shifting $\text{not } E$ in $\Gamma$;
4. generate a general rule $R$ where $R\theta = (L \leftarrow \Gamma')$ for some $\theta$. 
Learning from a single positive example

Illustration

Given:

\[ B = \{ \text{bird}(X) \leftarrow \text{penguin}(X). \text{bird}(\text{tweety}). \text{penguin}(\text{polly}). \} \]

\[ E = \{ \oplus \text{flies}(\text{tweety}). \} \]

Note: \[ B \models \text{not flies}(\text{tweety}). \]

Steps:

1. compute answer set \( M \) of \( B \) and expansion set \( M^+ \):
   - stable model of \( B \) (same as that of \( B \) not \( E \)):
     \[ M = \{ \text{bird}(\text{tweety}). \text{bird}(\text{polly}). \text{penguin}(\text{polly}). \} \]
   - expansion of \( M \):
     \[ M^+ = \{ \text{bird}(\text{tweety}). \text{bird}(\text{polly}). \text{penguin}(\text{polly}). \text{not penguin}(\text{tweety}). \text{not flies}(\text{tweety}). \text{not flies}(\text{polly}). \} \]

2. construct the integrity constraint \( \Gamma \) from \( M^+ \):
   \[ \text{← bird(\text{tweety}), not penguin(\text{tweety}), not flies(\text{tweety}).} \]

3. produce a rule \( E \leftarrow \Gamma' \) by shifting not \( E \) in \( \Gamma \):
   \[ r_0 = \text{flies(\text{tweety}) \leftarrow bird(\text{tweety}), not penguin(\text{tweety}).} \]

4. generate a general rule :
   \[ H = \text{flies}(X) \leftarrow \text{bird}(X), \text{not penguin}(X). \]
   (simplified as \( ab(x) \leftarrow \text{penguin}(x). \)
Learning from a single positive example

Properties

$B$: categorical program,

$E$: positive example,

$R$: learned rule by algorithm LearnSingle-pos

Properties:

- $B \not\models R$.

- $\text{pred}(\text{head}(R)) = \text{pred}(E)$.

- If $R$ is negative-cycle-free and its head predicate appears nowhere in $B$, then $B \cup \{R\}$ is also categorical.

- If $R$ is negative-cycle-free and its head predicate appears nowhere in $B$, then $B \cup \{R\} \models E$. 
Learning from a single negative example

algorithm: Learn-single-neg

Input:
a categorical program $B$, a ground atom $E$ (negative example),
a target predicate $K(\ldots)$ on which $pred(E)$ strongly and negatively depends in $B$.

Output: a rule $R$

1. compute the answer set $M$ of $B$ and its expansion set $M^+$;
2. construct the integrity constraint $\leftarrow \Gamma$ from $M^+$;
3. produce the rule $K(\ldots) \leftarrow \Gamma'$ by shifting $not\ K(\ldots)$ in $\Gamma$;
4. obtain $\Gamma''$ by dropping from $\Gamma'$ every literal $l$ whose predicate $pred(l)$ strongly and negatively depends on $K(\ldots)$ in $B$.
5. generate a general rule $R$ from $K(\ldots) \leftarrow \Gamma''$ such that $R\theta = K(\ldots) \leftarrow \Gamma''$ for some $\theta$. 
Learning from a single negative example

Illustration

Given:

\[ B : \quad \text{flies}(x) \leftarrow \text{bird}(x), \not \text{ab}(x), \]
\[ \text{bird}(x) \leftarrow \text{penguin}(x), \]
\[ \text{bird(tweety)}, \]
\[ \text{penguin}(polly). \]

\[ E : \quad \not \text{flies}(polly). \]

target predicate: \textit{ab}

Note: \( B \models \text{flies}(polly). \)

Steps:

1. compute answer set \( M \) of \( B \) and expansion set \( M^+ \):
   ... omitted ...

2. construct the integrity constraint \( \leftarrow \Gamma \) from \( M^+ \):
   \[ \leftarrow \text{bird}(polly), \text{penguin}(polly), \text{flies}(polly), \not \text{ab}(polly). \]

3. produce the rule \( K(\ldots) \leftarrow \Gamma' \) by shifting \( \not K(\ldots) \) in \( \Gamma \):
   \[ \text{ab}(polly) \leftarrow \text{bird}(polly), \text{penguin}(polly), \text{flies}(polly). \]

4. dropping from \( \Gamma' \) every literal \( l \) whose predicate \( \text{pred}(l) \) strongly and netagively depends on predicate \textit{ab}:
   \[ \text{ab}(polly) \leftarrow \text{bird}(polly), \text{penguin}(polly). \]

5. generate a general rule \( H \):
   \[ \text{ab}(x) \leftarrow \text{bird}(x), \text{penguin}(x). \]
   (simplified as \( \text{ab}(x) \leftarrow \text{penguin}(x) \).

Note: Now since \( \text{ab}(polly) \) is \textit{true}, \( \not \text{ab}(penguin) \) is \textit{false}. Therefore, the newly learned theory prevents the first rule in \( B \) from deriving \text{flies}(polly).
Learning from a single negative example

Properties

$B$: categorical program,

$E$: negative example,

$K$: target predicate,

$R$: learned rule by algorithm Learnsingle-pos

Properties:

- $B \not\models R$.

- $\text{pred}(\text{head}(R)) \neq \text{pred}(E)$, instead, $\text{pred}(\text{head}(R)) = K$.

- if $R$ is negative-cycle-free, then $B \cup \{R\}$ is not necessarily categorical.

- if $B \cup \{R\} \models R$ and $B \cup \{R\}$ is consistent, then $B \cup \{R\} \not\models E$. 

Learning from a set of examples

all positive examples

1. Let $B$ be a categorical program, and $R_i$ is a rule learned from $B$ and a positive example $E_i$, $1 \leq i \leq n$.

If each $R_i$ is negativ-cycle-free and $\text{pred}(E_i)$ appears nowhere in $B$, then $B \cup \{R_1, \ldots, R_n\} \models E_i$.

2. Let $B$ be a categorical program, $E_1$ and $E_2$ be positive examples such that $\text{pred}(E_1)$ and $\text{pred}(E_2)$ appear nowhere in $B$.

Suppose rule $R_1$ learned from $B$ and $E_1$ is negative-cycle-free, and rule $R_2$ learned from $B \cup \{R_1\}$ and $E_2$ is negative-cycle-free.

Then $B \cup \{R_1, R_2\} \models E_i (i = 1, 2)$. (monotonicity)
Learning from a set of examples

all negative examples

... ... omitted ... ...
Learning from a set of examples

mixed set of positive and negative examples

1. may not necessarily produce a solution which satisfies both positive and negative examples.

2. in incremental learning mode, the order in which the examples are taken, does matter. (obvious in multiple-predicate learning, less obvious in single-predicate learning)
General Properties

- Both positive and negative examples may lead to new rules learned.

- Based on answer set semantics, so have both abductive and inductive nature.

- Example-driven learning, therefore bottom-up search in general.

- **(Induction in noncategorical programs)** Suppose program $B$ has answer sets $S_1, \ldots, S_n$, and rule $R_i$ is obtained by algorithm Learn-single-pos using $B$ and a same positive example $E$. If each $R_i$ is negative-cycle-free and $\text{pred}(E)$ appears nowhere in $B$, then $B \cup \{R_1, \ldots, R_n\} \models E$.

- No modifications to the rules as background knowledge. But the result of induction often has the same effect as modifying rules in a program, given appropriate program transformation techniques. For instance, let $B = \{p \leftarrow q, r. \quad q.\}$ and $E = p$. Then algorithm Learn-single-pos will learn a rule $p \leftarrow q, \text{not } r$.

  However, this rule and the first rule in $B$ can be merged as $p \leftarrow q$, which is equivalent to the rule obtained by dropping $r$ from the first rule in $B$.

- Since the learned theory may contain a lot of redundancies, it seems that we really need some robust program transformation procedures.

- This feature allows the batch learning systems to incorporate some prior knowledge, which was not allowed in traditional batch learning.

- Batch learning is preferred to incremental learning, since it leads to less redundant theories.
Sequential Learning Algorithms

Incremental Learning

Initialize \( \Sigma \) to \( \{ \square \} \) or some prior knowledge

repeat

read the next (positive or negative) example

while \( \Sigma \) is not correct w.r.t. the examples read so far

if \( \exists e^- \) s.t. \( \Sigma \models e^- \)

learn a rule from \( \Sigma \) and \( e^- \) using Learn-single-neg

add the learned rule to \( \Sigma \)

if \( \exists e^+ \) s.t. \( \Sigma \not\models e^+ \)

learn a rule from \( \Sigma \) and \( e^+ \) using Learn-single-pos

add the learned rule to \( \Sigma \)

simplify \( \Sigma \)

until no examples left to read.
Sequential Learning Algorithms

Batch Learning

Initialize $\Sigma$ to $\{\Box\}$ or some prior knowledge

while there are positive examples uncovered by $\Sigma$

    learn a rule $R$ from $\Sigma$ and a randomly selected $e^+$
    find a best consistent rule $R'$ between $\Box$ and $R$
        using algorithm Learn-single-pos
    remove positive examples covered by $R'$
    add the rule $R'$ to $\Sigma$
    simplify $\Sigma$

while there are negative examples

    learn a rule $R$ from $\Sigma$ and a randomly selected $e^-$
        using algorithm Learn-single-neg
    remove $e^-$
    add the rule $R$ to $\Sigma$
    simplify $\Sigma$
Parallel Learning Algorithms

master processor:

- Initialize $\Sigma$ to $\{\square\}$ or some prior knowledge
- partition the positive examples to $p$ processors
- replicate all negative examples to all processors
- broadcast $\Sigma$ to all the worker processors
- collect learned $\Sigma_i$ from processor $i$
- merge $\Sigma_i$’s and simplify them into a new $\Sigma$

worker processor $i$:

- receive its partition of positive examples
- and all the negative examples
- learn a theory $\Sigma_i$ sequentially
- send $\Sigma_i$ to the master