Weight Learning in a Probabilistic Extension of Answer Set Programs

Joohyung Lee and Yi Wang

School of Computing, Informatics and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA
{joolee, ywang485}@asu.edu

Abstract

LP^{MLN} is a probabilistic extension of answer set programs with the weight scheme derived from that of Markov Logic. Previous work has shown how inference in LP^{MLN} can be achieved. In this paper, we present the concept of weight learning in LP^{MLN} and learning algorithms for LP^{MLN} derived from those for Markov Logic. We also present a prototype implementation that uses answer set solvers for learning as well as some example domains that illustrate distinct features of LP^{MLN} learning. Learning in LP^{MLN} is in accordance with the stable model semantics, thereby it learns parameters for probabilistic extensions of knowledge-rich domains where answer set programming has shown to be useful but limited to the deterministic case, such as reachability analysis and reasoning about actions in dynamic domains. We also apply the method to learn the parameters for probabilistic abductive reasoning about actions.

1 Introduction

LP^{MLN} is a probabilistic extension of answer set programs with the weight scheme derived from that of Markov Logic (Richardson and Domingos 2006). The language turns out to be highly expressive to embed several other probabilistic logic languages, such as P-log (Baral, Gelfond, and Rushton 2009), ProbLog (De Raedt, Kimmig, and Toivonen 2007), Markov Logic, and Causal Models (Pearl 2000), as described in (Lee, Meng, and Wang 2015; Lee and Wang 2016; Balai and Gelfond 2016; Lee and Yang 2017). Inference engines for LP^{MLN}, such as LPMLN2ASP, LPMLN2MLN (Lee, Talsania, and Wang 2017), and LPMLN-MODELS (Wang and Zhang 2017), have been developed based on the reduction of LP^{MLN} to answer set programs and Markov Logic.

The weight associated with each LP^{MLN} rule roughly asserts how important the rule is in deriving a stable model. It can be manually specified by the user, which may be okay for a simple program, but a systematic assignment of weights for a complex program could be challenging. A solution would be to learn the weights automatically from the observed data.

With this goal in mind, this paper presents the concept of weight learning in LP^{MLN} and a few learning methods for LP^{MLN} derived from learning in Markov Logic.

Weight learning in LP^{MLN} is to find the weights of the rules in the LP^{MLN} program such that the likelihood of the observed data according to the LP^{MLN} semantics is maximized, which is commonly known as Maximum Likelihood Estimation (MLE) in the practice of machine learning.

In LP^{MLN}, due to the requirement of a stable model, deterministic dependencies are frequent. Poon and Domingos (2006) noted that deterministic dependencies break the support of a probability distribution into disconnected regions, making it difficult to design ergodic Markov chains for Markov Chain Monte Carlo (MCMC) sampling, which motivated them to develop an algorithm called *MC-SAT* that uses a satisfiability solver to find modes for computing conditional probabilities. Thanks to the close relationship between Markov Logic and LP^{MLN}, we could adapt that algorithm to LP^{MLN}, which we call *MC-ASP*. Unlike MC-SAT, algorithm MC-ASP utilizes ASP solvers for performing MCMC sampling, and is based on the penalty-based formulation of LP^{MLN} instead of the reward-based formulation as in Markov Logic.

Learning in LP^{MLN} is in accordance with the stable model semantics, thereby it learns parameters for probabilistic extensions of knowledge-rich domains where answer set programming has shown to be useful but limited to the deterministic case, such as reachability analysis and reasoning about actions in dynamic domains. More interestingly, we demonstrate that the method can also be applied to learn parameters for abductive reasoning about dynamic systems to associate the probability learned from data with each possible reason for the failure.

The paper is organized as follows. Section 2 reviews the language LP^{MLN}, and Section 3 presents the learning framework and a gradient ascent method for the basic case, where a single stable model is given as the training data. Section 4 presents a few extensions of the learning problem and methods, such as allowing multiple stable models as the training data and allowing the training data to be an incomplete interpretation. In addition to the general learning algorithm, Section 5 relates LP^{MLN} learning also to learning in ProbLog and Markov Logic as special cases, which allows for the special cases of LP^{MLN} learning to be computed by existing implementations of ProbLog and Markov Logic. Section 6 introduces a prototype implementation of the general learn-

ing algorithm and demonstrates it with a few example domains where $\mathrm{LP^{MLN}}$ learning is more suitable than other learning methods.

2 Review: Language $\mathrm{LP^{MLN}}$

The original definition of $\mathrm{LP^{MLN}}$ from (Lee and Wang 2016) is based on the concept of a "reward": the more rules are true, the larger weight is assigned to the corresponding stable model as the reward. Alternatively, Lee and Yang [2017] present a reformulation in terms of a "penalty": the more rules are false, the smaller weight is assigned to the corresponding stable model. The advantage of the latter is that it yields a translation of $\mathrm{LP^{MLN}}$ programs that can be readily accepted by ASP solvers, the idea that led to the implementation of $\mathrm{LP^{MLN}}$ using ASP solvers (Lee, Talsania, and Wang 2017). Throughout the paper, we refer to this reformulation as the main definition of $\mathrm{LP^{MLN}}$.

We assume a first-order signature σ that contains no function constants of positive arity, which yields finitely many Herbrand interpretations. An $\mathrm{LP^{MLN}}$ program is a pair $\langle \mathbf{R}, \mathbf{W} \rangle$, where $\mathbf{R}$ is a list of rules $(R_1, \ldots, R_m)$, where each rule has the form

$$A \leftarrow B \wedge N \qquad (1)$$

where A is a disjunction of atoms, B is a conjunction of atoms, and N is a negative formula constructed from atoms using conjunction, disjunction, and negation.[1] We identify rule (1) with formula $B \wedge N \rightarrow A$. The expression $\{A_1\} \leftarrow Body$, where A_1 is an atom, denotes the rule $A_1 \leftarrow Body \wedge \neg\neg A_1$. $\mathbf{W}$ is a list $(w_1, \ldots w_m)$ such that each w_i is a real number or the symbol α that denotes the weight of rule i in $\mathbf{R}$. We can also identify an $\mathrm{LP^{MLN}}$ program with the finite list of weighted rules $\{w_i : R_i \mid i \in \{1, \ldots, m\}\}$. A weighted rule $w : R$ is called *soft* if w is a real number; it is called *hard* if w is α (which denotes infinite weight). Variables range over an Herbrand Universe, which is assumed to be finite so that the ground program is finite. For any $\mathrm{LP^{MLN}}$ program Π, by $gr(\Pi)$ we denote the program obtained from Π by the process of grounding. Each resulting rule with no variables, which we call *ground instance*, receives the same weight as the original rule.

For any $\mathrm{LP^{MLN}}$ program $\Pi = \{w_1 : R_1, \ldots, w_m : R_m\}$ and any interpretation I, expression $n_i(I)$ denotes the number of ground instances of R_i that is false in I, and $\overline{\Pi}$ denotes the set of (unweighted) formulas obtained from Π by dropping the weight of every rule. When Π has no variables, Π_I denotes the set of weighted rules $w : R$ in Π such that $I \models R$.

In general, an $\mathrm{LP^{MLN}}$ program may even have stable models that violate some hard rules, which encode definite knowledge. However, throughout the paper, we restrict attention to $\mathrm{LP^{MLN}}$ programs whose stable models do not violate hard rules. More precisely, given an $\mathrm{LP^{MLN}}$ program

---

[1]For the definition of a negative formula, see (Ferraris, Lee, and Lifschitz 2011).

Π, $\mathrm{SM}[\Pi]$ denotes the set

$\{I \mid I$ is a (deterministic) stable model of $\overline{gr(\Pi)_I}$ that satisfies all hard rules in $gr(\Pi)\}$.

For any interpretation I, its weight $W_\Pi(I)$ and its probability $P_\Pi(I)$ are defined as follows.

$$W_\Pi(I) = \begin{cases} exp\left(-\sum_{w_i:R_i \in \Pi^{\mathrm{soft}}} w_i n_i(I)\right) & \text{if } I \in \mathrm{SM}[\Pi]; \\ 0 & \text{otherwise}, \end{cases}$$

where Π^{soft} consists of all soft rules in Π, and

$$P_\Pi(I) = \frac{W_\Pi(I)}{\sum_{J \in \mathrm{SM}[\Pi]} W_\Pi(J)}.$$

An interpretation I is called a *(probabilistic) stable model* of Π if $P_\Pi(I) \neq 0$. When $\mathrm{SM}[\Pi]$ is non-empty, it turns out that every probabilistic stable model satisfies all hard rules, and the definitions of $W_\Pi(I)$ and $P_\Pi(I)$ above are equivalent to the original definitions (Lee and Wang 2016, Proposition 2).

For any proposition A, the probability of A under Π is defined as $P_\Pi(A) = \sum_{I:I \models A} P_\Pi(I)$.

3 $\mathrm{LP^{MLN}}$ Weight Learning

3.1 General Problem Statement

A parameterized $\mathrm{LP^{MLN}}$ program $\hat{\Pi}$ is defined similarly to an $\mathrm{LP^{MLN}}$ program Π except that non-α weights (i.e., "soft" weights) are replaced with distinct parameters to be learned. By $\hat{\Pi}(\mathbf{w})$, where $\mathbf{w}$ is a list of real numbers whose length is the same as the number of soft rules, we denote the $\mathrm{LP^{MLN}}$ program obtained from $\hat{\Pi}$ by replacing the parameters with $\mathbf{w}$. The weight learning task for a parameterized $\mathrm{LP^{MLN}}$ program is to find the MLE (Maximum likelihood Estimation) of the parameters as in Markov Logic. Formally, given a parameterized $\mathrm{LP^{MLN}}$ program $\hat{\Pi}$ and a ground formula O (often in the form of conjunctions of literals) called *observation* or *training data*, the $\mathrm{LP^{MLN}}$ parameter learning task is to find the values $\mathbf{w}$ of parameters such that the probability of O under the $\mathrm{LP^{MLN}}$ program Π is maximized. In other words, the learning task is to find

$$\underset{\mathbf{w}}{\mathrm{argmax}}\ P_{\hat{\Pi}(\mathbf{w})}(O). \qquad (2)$$

3.2 Gradient Method for Learning Weights From a Complete Stable Model

Same as in Markov Logic, there is no closed form solution for (2) but the gradient ascent method can be applied to find the optimal weights in an iterative manner.

We first compute the gradient. Given a (non-ground) $\mathrm{LP^{MLN}}$ program Π whose $\mathrm{SM}[\Pi]$ is non-empty and given a stable model I of Π, the base-e logarithm of $P_\Pi(I)$, $lnP_\Pi(I)$, is

$$-\sum_{w_i:R_i \in \Pi^{\mathrm{soft}}} w_i n_i(I) - ln \sum_{J \in \mathrm{SM}[\Pi]} exp\left(-\sum_{w_i:R_i \in \Pi^{\mathrm{soft}}} w_i n_i(J)\right).$$

The partial derivative of $lnP_\Pi(I)$ w.r.t. $w_i(\neq \alpha)$ is

$$\frac{\partial lnP_\Pi(I)}{\partial w_i} = -n_i(I) + \frac{\sum\limits_{J\in\text{SM}[\Pi]} exp(-\sum\limits_{w_i:R_i\in\Pi^{\text{soft}}} w_i n_i(J))n_i(J)}{\sum\limits_{K\in\text{SM}[\Pi]} exp(-\sum\limits_{w_i:R_i\in\Pi^{\text{soft}}} w_i n_i(K))}$$

$$= -n_i(I) + \sum_{J\in\text{SM}[\Pi]} \left(\frac{exp(-\sum\limits_{w_i:R_i\in\Pi^{\text{soft}}} w_i n_i(J))}{\sum\limits_{K\in\text{SM}[\Pi]} exp(-\sum\limits_{w_i:R_i\in\Pi^{\text{soft}}} w_i n_i(K))} \right) n_i(J)$$

$$= -n_i(I) + \sum_{J\in\text{SM}[\Pi]} P_\Pi(J)n_i(J) = -n_i(I) + \underset{J\in\text{SM}[\Pi]}{E}[n_i(J)]$$

where $\underset{J\in\text{SM}[\Pi]}{E}[n_i(J)] = \sum\limits_{J\in\text{SM}[\Pi]} P_\Pi(J)n_i(J)$ is the expected number of false ground rules obtained from R_i.

Since the log-likelihood above is a concave function of the weights, any local maximum is a global maximum, and maximizing $P_\Pi(I)$ can be done by the standard gradient ascent method by updating each weight w_i by $w_i + \lambda \cdot (-n_i(I) + \underset{J\in\text{SM}[\Pi]}{E}[n_i(J)])$ until it converges.[2]

However, similar to Markov Logic, computing $\underset{J\in\text{SM}[\Pi]}{E}[n_i(J)]$ is intractable (Richardson and Domingos 2006). In the next section, we turn to an MCMC sampling method to find its approximate value.

3.3 Sampling Method: MC-ASP

The following is an MCMC algorithm for LP^{MLN}, which adapts the algorithm MC-SAT for Markov Logic (Poon and Domingos 2006) by considering the penalty-based reformulation and by using an ASP solver instead of a SAT solver for sampling.

Algorithm 1 MC-ASP

Input: An LP^{MLN} program Π whose soft rules' weights are non-positive and a positive integer N.
Output: Samples $I^1, \ldots, I^N$

1. Choose a (probabilistic) stable model I^0 of Π.

2. Repeat the following for $j = 1, \ldots, N$

 (a) $M \leftarrow \emptyset$;
 (b) For each ground instance of each rule $w_i : R_i \in \Pi^{\text{soft}}$ that is false in I^{j-1}, add the ground instance to M with probability $1 - e^{w_i}$;
 (c) Randomly choose a (probabilistic) stable model I^j of Π that satisfies no rules in M.

When all the weights w_i of soft rules are non-positive, $1 - e^{w_i}$ (at step (b)) is in the range $[0, 1)$ and thus it validly represents a probability. At each iteration, the sample is chosen from stable models of Π, and consequently, it must satisfy all hard rules. For soft rules, the higher its weight, the less likely that it will be included in M, and thus less likely to be not satisfied by the sample generated from M.

---

[2]Note that although any local maximum is a global maximum for the log-likelihood function, there can be multiple combinations of weights that achieve the maximum probability of the training data.

The following theorem states that MC-ASP satisfies the MCMC criteria of ergodicity and detailed balance, which justifies the soundness of the algorithm.

Theorem 1 *The Markov chain generated by MC-ASP satisfies ergodicity and detailed balance.*[3]

Steps 1 and 2(c) of the algorithm require finding a probabilistic stable model of LP^{MLN}, which can be computed by system LPMLN2ASP (Lee, Talsania, and Wang 2017). The system is based on the translation that turns an LP^{MLN} program Π into an ASP program $\text{lpmln2asp}^{\text{pnt}}(\Pi)$. The translation turns each (possibly non-ground) soft rule

$$w_i : \quad Head_i(\mathbf{x}) \leftarrow Body_i(\mathbf{x}) \qquad (3)$$

into [4]

$$\texttt{unsat}(i, w_i, \mathbf{x}) \leftarrow Body_i(\mathbf{x}), \texttt{not } Head_i(\mathbf{x})$$
$$Head_i(\mathbf{x}) \leftarrow Body_i(\mathbf{x}), \texttt{not unsat}(i, w_i, \mathbf{x})$$
$$:\sim \texttt{unsat}(i, w_i, \mathbf{x}). \ [w_i, i, \mathbf{x}]$$

and each hard rule

$$\alpha : \quad Head_i(\mathbf{x}) \leftarrow Body_i(\mathbf{x})$$

into $Head_i(\mathbf{x}) \leftarrow Body_i(\mathbf{x})$. System LPMLN2ASP turns an LP^{MLN} program Π into $\text{lpmln2asp}^{\text{pnt}}(\Pi)$ and calls ASP solver CLINGO to find the stable models of $\text{lpmln2asp}^{\text{pnt}}(\Pi)$, which coincide with the probabilistic stable models of Π. The weight of a stable model can be computed from the weights recorded in $\texttt{unsat}$ atoms that are true in the stable model.

Step 2(c) also requires a uniform sampler for answer sets, which can be computed by XORRO (Gebser et al. 2016).

Algorithm 2 is a weight learning algorithm for LP^{MLN} based on gradient ascent using MC-ASP (Algorithm 1) for collecting samples. Step 2(b) of MC-ASP requires that w_i be non-positive in order for $1 - e^{w_i}$ to represent a probability. Unlike in the Markov Logic setting, converting positive weights into non-positive weights cannot be done in LP^{MLN} simply by replacing $w : F$ with $-w : \neg F$, due to the difference in the FOL and the stable model semantics. Algorithm 2 converts Π into an equivalent program Π^{neg} whose rules' weights are non-positive, before calling MC-ASP. The following theorem justifies the soundness of this method.[5]

Theorem 2 *When* $\text{SM}[\Pi]$ *is not empty, the program* Π^{neg} *specifies the same probability distribution as the program* Π.[6]

---

[3]A Markov chain is *ergodic* if there is a number m such that any state can be reached from any other state in any number of steps greater than or equal to m.

Detailed balance means $P_\Pi(X)Q(X \rightarrow Y) = P_\Pi(Y)Q(Y \rightarrow X)$ for any samples X and Y, where $Q(X \rightarrow Y)$ denotes the probability that the next sample is Y given that the current sample is X.

[4]If $Head_i(\mathbf{x})$ is a disjunction of atoms $a_1(\mathbf{x}) ; \ldots ; a_n(\mathbf{x})$, then not $Head_i(\mathbf{x})$ denotes not $a_1(\mathbf{x}), \ldots,$ not $a_n(\mathbf{x})$.

[5]Note that Π^{neg} is only used in MC-ASP. The output of Algorithm 2 may have positive weights.

[6]Non-emptiness of $\text{SM}[\Pi]$ implies that every probabilistic stable model of Π satisfies all hard rules in Π.

Algorithm 2 Algorithm for learning weights using LPMLN2ASP

Input: Π: A parameterized LP$^{\text{MLN}}$ program in the input language of LPMLN2ASP; O: A stable model represented as a set of constraints (that is, $\leftarrow$ *not* A is in O if a ground atom A is true; $\leftarrow A$ is in O if A is not true); δ: a fixed real number to be used for the terminating condition.
Output: Π with learned weights.
Process:

1. Initialize the weights of soft rules $R_1, \ldots, R_m$ with some initial weights $\mathbf{w}^0$.

2. Repeat the following for $j = 1, \ldots$ until $max\{|w_i^j - w_i^{j-1}| : i = 1, \ldots, m\} < \delta$:

(a) Compute the stable model of $\Pi \cup O$ using LPMLN2ASP (see below); for each soft rule R_i, compute $n_i(O)$ by counting `unsat` atoms whose first argument is i (i is a rule index).

(b) Create Π^{neg} by replacing each soft rule R_i of the form $w :$ $H(\mathbf{x}) \leftarrow B(\mathbf{x})$ in Π where $w > 0$ with

$$0 : H(\mathbf{x}) \leftarrow B(\mathbf{x}),$$
$$\alpha : \texttt{neg}(i, \mathbf{x}) \leftarrow B(\mathbf{x}), \texttt{not } H(\mathbf{x}),$$
$$-w : \leftarrow \texttt{not neg}(i, \mathbf{x}).$$

(c) Run MC-ASP on Π^{neg} to collect a set S of sample stable models.

(d) For each soft rule R_i, approximate $\sum\limits_{J \in \text{SM}[\Pi]} P_\Pi(J) n_i(J)$ with $\sum\limits_{J \in S} n_i(J)/|S|$, where n_i is obtained from counting the number of `unsat` atoms whose first argument is i.

(e) For each $i \in \{1, \ldots, m\}$,
$w_i^{j+1} \leftarrow w_i^j + \lambda \cdot (-n_i(O) + \sum\limits_{J \in S} n_i(J)/|S|)$.

4 Extensions

The base case learning in the previous section assumes that the training data is a single stable model and is a complete interpretation. This section extends the framework in a few ways.

4.1 Learning from Multiple Stable Models

The method described in the previous section allows only one stable model to be used as the training data. Now, suppose we have multiple stable models $I_1, \ldots, I_m$ as the training data. For example, consider the parameterized program $\hat{\Pi}_{coin}$ that describes a coin, which may or may not land in the head when it is flipped,

$$\alpha \; : \; \{flip\}$$
$$w \; : \; head \leftarrow flip$$

(the first rule is a choice rule) and three stable models as the training data: $I_1 = \{flip\}$, $I_2 = \{flip\}$, $I_3 = \{flip, head\}$ (the absence of $head$ in the answer set is understood as landing in tail), indicating that $\{flip, head\}$ has a frequency of $\frac{1}{3}$, and $\{flip\}$ has a frequency of $\frac{2}{3}$. Intuitively, the more we observe the $head$, the larger the weight of the second rule. Clearly, learning w from only one of I_1, I_2, I_3 won't result in a weight that captures all the three stable models: learning

from each of I_1 or I_2 results in the value of w too small for $\{flip, head\}$ to have a frequency of $\frac{1}{3}$ while learning from I_3 results in the value of w too large for $\{flip\}$ to have a frequency of $\frac{2}{3}$.

To utilize the information from multiple stable models, one natural idea is to maximize the joint probability of all the stable models in the training data, which is the product of their probabilities, i.e.,

$$P(I_1, \ldots, I_m) = \prod_{j \in \{1, \ldots, m\}} P_\Pi(I_j).$$

The partial derivative of $lnP(I_1, \ldots, I_m)$ w.r.t. $w_i(\neq \alpha)$ is

$$\frac{\partial lnP(I_1, \ldots, I_m)}{\partial w_i} = \sum_{j \in \{1, \ldots, m\}} \left(-n_i(I_j) + \underset{J \in \text{SM}[\Pi]}{E}[n_i(J)] \right).$$

In other words, the gradient of the log probability is simply the sum of the gradients of the probability of each stable model in the training data. To update Algorithm 2 to reflect this, we simply repeat step 2(a) to compute $n_i(I_k)$ for each $k \in \{1, \ldots, m\}$, and at step 2(e) update w_i as follows:

$$w_i^{j+1} \leftarrow w_i^j + \lambda \cdot \left(-\sum_{k \in \{1, \ldots, m\}} n_i(I_k) + m \cdot \sum_{J \in \text{SM}[\Pi]} P_\Pi(J) n_i(J) \right).$$

Alternatively, learning from multiple stable models can be reduced to learning from a single stable model by introducing one more argument k to every predicate, which represents the index of a stable model in the training data, and rewriting the data to include the index.

Formally, given an LP$^{\text{MLN}}$ program Π and a set of its stable models $I_1, \ldots, I_m$, let Π^m be an LP$^{\text{MLN}}$ program obtained from Π by appending one more argument k to the list of arguments of every predicate that occurs in Π, where k is a schematic variable that ranges over $\{1, \ldots, m\}$. Let

$$I = \bigcup_{i \in \{1, \ldots, m\}} \{p(\mathbf{t}, i) \mid p(\mathbf{t}) \in I_i\}. \tag{4}$$

The following theorem asserts that the weights of the rules in Π that are learned from the multiple stable models $I_1, \ldots, I_m$ are identical to the weights of the rules in Π^m that are learned from the single stable model I that conjoins $\{I_1, \ldots, I_m\}$ as in (4).

Theorem 3 *For any parameterized* LP$^{\text{MLN}}$ *program* $\hat{\Pi}$, *its stable models* $I_1, \ldots, I_m$ *and* I *as defined as in* (4), *we have*

$$\underset{\mathbf{w}}{\text{argmax}} \, P_{\hat{\Pi}^m(\mathbf{w})}(I) = \underset{\mathbf{w}}{\text{argmax}} \prod_{i \in \{1, \ldots, m\}} P_{\hat{\Pi}(\mathbf{w})}(I_i).$$

Example 1 *For the program* $\hat{\Pi}_{coin}$, *to learn from the three stable models* I_1, I_2, *and* I_3 *defined before, we consider the program* $\hat{\Pi}_{coin}^3$

$$\alpha \; : \; \{flip(k)\}.$$
$$w \; : \; head(k) \leftarrow flip(k).$$

($k \in \{1, 2, 3\}$) *and combine* I_1, I_2, I_3 *into one stable model* $I = \{flip(1), flip(2), flip(3), head(3)\}$. *The weight* w *in* $\hat{\Pi}_{coin}^3$ *learned from the single data* I *is identical to the weight* w *in* $\hat{\Pi}_{coin}$ *learned from the three stable models* I_1, I_2, I_3.

4.2 Learning in the Presence of Noisy Data

So far, we assumed that the data $I_1, \ldots, I_m$ are (probabilistic) stable models of the parameterized LP$^{\text{MLN}}$ program. Otherwise, the joint probability would be zero regardless of any weights assigned to the soft rules, and the partial derivative of $lnP(I_1, \ldots, I_m)$ is undefined. However, data gathered from the real world could be noisy, so some data I_i may not necessarily be a stable model. Even then, we still want to learn from the other "correct" instances. We may drop them in the pre-processing to learning but this could be computationally expensive if the data is huge. Alternatively, we may mitigate the influence of the noisy data by introducing so-called "noise atoms" as follows.

Example 2 *Consider again the program* $\hat{\Pi}^m_{coin}$. *Suppose one of the interpretations* I_i *in the training data is* $\{head(i)\}$. *The interpretation is not a stable model of* $\hat{\Pi}^m_{coin}$. *We obtain* $\hat{\Pi}^m_{noisecoin}$ *by modifying* $\hat{\Pi}^m_{coin}$ *to allow for the noisy atom* $n(k)$ *as follows.*

$$
\begin{aligned}
\alpha & : \{flip(k)\}. \\
w & : head(k) \leftarrow flip(k). \\
\alpha & : head(k) \leftarrow n(k). \\
-u & : n(k).
\end{aligned}
$$

Here, u *is a positive number that is "sufficiently" larger than* w. $\{head(i), n(i)\}$ *is a stable model of* $\hat{\Pi}^m_{noisecoin}$, *so that the combined training data* I *is still a stable model, and thus a meaningful weight* w *for* $\hat{\Pi}^m_{noisecoin}$ *can still be learned, given that other "correct" instances* I_j $(j \neq i)$ *dominate in the learning process (as for the noisy example, the corresponding stable model gets a low weight due to the weight assigned to* $n(i)$ *but not 0).*

Furthermore, with the same value of w, *the larger* u *becomes, the closer the probability distribution defined by* $\hat{\Pi}^m_{noisecoin}$ *approximates the one defined by* $\hat{\Pi}^m_{coin}$, *so the value of* w *learned under* $\hat{\Pi}^m_{noisecoin}$ *approximates the value of* w *learned under* $\hat{\Pi}^m_{coin}$ *where the noisy data is dropped.*

4.3 Learning from Incomplete Interpretations

In the previous sections, we assume that the training data is given as a (complete) interpretation, i.e., for each atom it specifies whether it is true or false. In this section, we discuss the general case when the training data is given as a partial interpretation, which omits to specify some atoms to be true or false, or more generally when the training data is in the form of a formula that more than one stable model may satisfy.

Given a non-ground LP$^{\text{MLN}}$ program Π such that SM$[\Pi]$ is not empty and given a ground formula O as the training data, we have

$$
P_\Pi(O) = \frac{\sum_{I \models O, I \in \text{SM}[\Pi]} W_\Pi(I)}{\sum_{J \in \text{SM}[\Pi]} W_\Pi(J)}.
$$

The partial derivative of $lnP_\Pi(O)$ w.r.t. w_i $(\neq \alpha)$ turns out to be

$$
\frac{\partial lnP_\Pi(O)}{\partial w_i} = - \underset{I \models O, I \in \text{SM}[\Pi]}{E}[n_i(I)] + \underset{J \in \text{SM}[\Pi]}{E}[n_i(J)].
$$

It is straightforward to extend Algorithm 2 to reflect the extension. Computing the approximate value of the first term $- \underset{I \models O, I \in \text{SM}[\Pi]}{E}[n_i(I)]$ can be done by sampling on $\Pi^{neg} \cup O$.

5 LP$^{\text{MLN}}$ Weight Learning via Translations to Other Languages

This section considers two fragments of LP$^{\text{MLN}}$, for which the parameter learning task reduces to the same tasks for Markov Logic and ProbLog.

5.1 Tight LP$^{\text{MLN}}$ Program: Reduction to MLN Weight Learning

By Theorem 3 in (Lee and Wang 2016), any tight LP$^{\text{MLN}}$ program can be translated into a Markov Logic Network (MLN) by adding completion formulas (Erdem and Lifschitz 2003) with the weight α. This means that the weight learning for a tight LP$^{\text{MLN}}$ program can be reduced to the weight learning for an MLN.

Given a tight LP$^{\text{MLN}}$ program $\Pi = \langle \mathbf{R}, \mathbf{W} \rangle$ and one (not necessarily complete) interpretation E as the training data, the MLN $Comp(\Pi)$ is obtained by adding completion formulas with weight α to Π.

The following theorem tells us that the weight assignment that maximizes the probability of the training data under LP$^{\text{MLN}}$ programs is identical to the weight assignment that maximizes the probability of the same training data under an MLN $Comp(\Pi)$.

Theorem 4 *Let* L *be the Markov Logic Network* $Comp(\Pi)$ *and let* E *be a ground formula (as the training data). When* SM$[\Pi]$ *is not empty,*

$$
\underset{\mathbf{w}}{\operatorname{argmax}}\, P_{\hat{\Pi}(\mathbf{w})}(E) = \underset{\mathbf{w}}{\operatorname{argmax}}\, P_{\hat{\text{L}}(\mathbf{w})}(E).
$$

($\hat{\text{L}}$ *is a parameterized MLN obtained from* L*.)*

Thus we may learn the weights of a tight LP$^{\text{MLN}}$ program using the existing implementations of Markov Logic, such as ALCHEMY and TUFFY.

5.2 Coherent LP$^{\text{MLN}}$ Program: Reduction to Parameter Learning in ProbLog

For another special class of LP$^{\text{MLN}}$ programs, weight learning can be reduced to weight learning in ProbLog (Fierens et al. 2013).

We say an LP$^{\text{MLN}}$ program Π is *simple* if all soft rules in Π are of the form

$$
w : A
$$

where A is an atom, and no atoms occurring in the soft rules occur in the head of a hard rule.

We say a simple LP$^{\text{MLN}}$ program Π is *k-coherent* $(k > 0)$ if, for any truth assignment to atoms that occur in Π^{soft}, there are exactly k probabilistic stable models of Π that satisfies the truth assignment. We also apply the notion of k-coherency when Π is parameterized.

Without loss of generality, we assume that no atom occurs more than once in Π^{soft}. (If one atom A occurs in multiple rules $w_1 : A, \ldots, w_n : A$, these rules can be combined

into $w_1 + \cdots + w_n : A$.) A k-coherent LP$^{\text{MLN}}$ program Π can thus be identified with the tuple $\langle PF, \Pi^{\text{hard}}, \mathbf{w} \rangle$, where $PF = (pf_1, \ldots, pf_m)$ is a list of (possibly non-ground) atoms that occur as soft rules in Π, Π^{hard} is a set of hard rules in Π, and $\mathbf{w} = (w_1, \ldots, w_m)$ is the list of soft rule's weights, where w_i is the weight of pf_i.

A ProbLog program can be viewed as a tuple $\langle PF, \mathbf{R}, \mathbf{pr} \rangle$ where PF is a list of atoms called *probabilistic facts*, $\mathbf{R}$ is a set of rules such that no atom that occurs in PF occurs in the head of any rule in $\mathbf{R}$, and $\mathbf{pr}$ is a list $(p_1, \ldots, p_{|PF|})$, where each p_i is the probability of probabilistic atom $pf_i \in PF$. A *parameterized* ProbLog program is similarly defined, where $\mathbf{pr}$ is a list of parameters to be learned.

Given a list of probabilities $\mathbf{pr} = (p_1, \ldots, p_n)$, we construct a list of weights $\mathbf{w^{pr}} = (w_1, \ldots, w_n)$ as follows:

$$w_i = ln(\frac{p_i}{1 - p_i}) \qquad (5)$$

for $i \in \{1, \ldots n\}$.

The following theorem asserts that weight learning on a 1-coherent LP$^{\text{MLN}}$ program can be done by weight learning on its corresponding ProbLog program.

Theorem 5 *For any 1-coherent parameterized* LP$^{\text{MLN}}$ *program* $\langle PF, P, \mathbf{w} \rangle$ *and any interpretation* T *(as the training data), we have*

$$\mathbf{w} = \underset{\mathbf{w}}{\operatorname{argmax}} \, P_{\langle PF, P, \mathbf{w} \rangle}(T)$$

if and only if

$$\mathbf{w} = \mathbf{w^{pr}} \text{ and } \mathbf{pr} = \underset{\mathbf{pr}}{\operatorname{argmax}} \, P_{\langle PF, P, \mathbf{pr} \rangle}(T).$$

According to the theorem, to learn the weights of a 1-coherent LP$^{\text{MLN}}$ program, we can simply construct the corresponding ProbLog program, perform ProbLog weight learning, and then turn the learned probabilities into LP$^{\text{MLN}}$ weights according to (5).

In (Lee and Wang 2018), k-coherent programs are shown to be useful for describing dynamic domains. Intuitively, each probabilistic choice leads to the same number of histories. For such a k-coherent LP$^{\text{MLN}}$ program, weight learning given a complete interpretation as the training data can be done by simply counting true and false ground instances of soft atomic facts in the given interpretation.

For an interpretation I and $c_i \in PF$, let $m_i(I)$ and $n_i(I)$ be the numbers of ground instances of c_i that is true in I and false in I, respectively.

Theorem 6 *For any k-coherent parameterized* LP$^{\text{MLN}}$ *program* $\langle PF, \Pi^{\text{hard}}, \mathbf{w} \rangle$, *and any (complete) interpretation* I *(as the training data), we have*

$$\underset{\mathbf{w}}{\operatorname{argmax}} \, P_{\langle PF, \Pi^{\text{hard}}, \mathbf{w} \rangle}(I; \mathbf{w}) = \Big(ln \frac{m_1(I)}{n_1(I)}, \ldots, ln \frac{m_{|PF|}(I)}{n_{|PF|}(I)} \Big).$$

6 Implementation and Examples

We implemented Algorithm 2 and its extensions described above using CLINGO, LPMLN2ASP, and a near-uniform answer set sampler XORRO . The implementation LPMLN-LEARN is available at https://github.com/

ywng485/lpmln-learning together with a manual and some examples.

In this section, we show how the implementation allows for learning weights in LP$^{\text{MLN}}$ from the data enabling learning parameters in knowledge-rich domains.

For all the experiments in this section, δ is set to be 0.001. λ is fixed to 0.1 and 50 samples are generated for each call of MC-ASP. The parameters for XORRO are manually tuned to achieve the best performance for each specific example.

6.1 Learning Certainty Degrees of Hypotheses

The LP$^{\text{MLN}}$ weight learning algorithm can be used to learn the certainty degree of a hypothesis from the data. For example, consider a person A carrying a certain virus contacting a group of people. The virus spreads among them as people contact each other. We use the following ASP facts to specify that A carries the virus and how people contacted each other:

```
carries_virus("A").
contact("A", "B"). contact("B", "C"). ...
```

Consider two hypotheses that a person carrying the virus may cause him to have a certain disease, and the virus may spread by contact. The hypotheses can be represented in the input language of LPMLN-LEARN by the following rules, where w(1) and w(2) are parameters to be learned:

```
@w(1) has_disease(X) :- carries_virus(X).
@w(2) carries_virus(Y) :- contact(X, Y),
                          carries_virus(X).
```

The parameterized LP$^{\text{MLN}}$ program consists of these two rules and the facts about contact relation. The training data specifies whether each person carries the virus and has the disease, for example:

```
:- not carries_virus("E").   :- carries_virus("H").
...
:- not has_disease("A").     :- has_disease("H").
```

The learned weights tell us how certain the data support the hypotheses. Note that the program models the transitive closure of the carries_virus relation, which is not properly done if the program is viewed as an MLN.[7] Learning under the MLN semantics results in weights that associate unreasonably high probabilities to people carrying virus even if they were not contacted by people with virus.

For example, consider the following graph
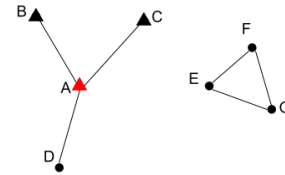
where A is the person who initially carries the virus, triangle-shaped nodes represent people who carry virus in the evidence, and the edges denote the contact relation. The

---

[7]That is, identifying the rule $H \leftarrow B$ with a formula in first-order logic $B \rightarrow H$.

cluster consisting of E, F, and G has no contact with the cluster consisting of A, B, C, and D. The following table shows the probability of each person carrying the virus, which is derived from the weights learned in accordance with Markov Logic and LP^{MLN}, respectively. We use ALCHEMY for the weight learning in Markov Logic.

Person	MLN	LP^{MLN}	carries_virus (ground truth)
B	0.823968	0.6226904833	Y
C	0.813969	0.6226904833	Y
D	0.818968	0.6226904833	N
E	0.688981	0	N
F	0.680982	0	N
G	0.680982	0	N

As can be seen from the table, under MLN, each of E, F, G has a high probability of carrying the virus, which is unintuitive.

6.2 Learning Probabilistic Graphs from Reachability

Consider an (unstable) communication network such as the one in Figure 1, where each node represents a signal station that sends and receives signals. A station may fail, making it impossible for signals to go through the station. The following LP^{MLN} rules define the connectivity between two stations X and Y in session T.

```
connected(X,Y,T) :- edge(X,Y), not fail(X,T),
                                not fail(Y,T).
connected(X,Y,T) :- connected(X,Z,T), connected(Z,Y,T).
```

A specific network can be defined by specifying edge relations, such as edge(1,2). Suppose we have data showing the connectivity between stations in several sessions. Based on the data, we could make decisions such as which path is most reliable to send a signal between the two stations. Under the LP^{MLN} framework, this can be done by learning the weights representing the failure rate of each station. For the network in Figure 1, we write the following rules whose weights $w(i)$ are to be learned:

```
@w(1) fail(1, T).    ...    @w(10) fail(10, T).
```
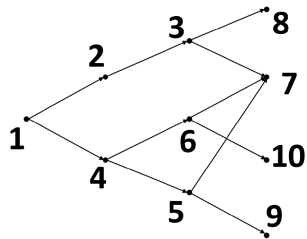

Figure 1: Example Communication Network

Here T is the auxiliary argument to allow learning from multiple training examples, as described in Section 4.1. The training example contains constraints either :- not connected(X,Y) for known connected stations

X and Y or :- connected(X,Y) for known disconnected stations X and Y. Since the training data is incomplete in specifying the connectivity between the stations, we use the extension of Algorithm 2 described in Section 4.3. The failure rates of the stations can be obtained from the learned weights as $\frac{e^{w(i)}}{e^0+e^{w(i)}}$.
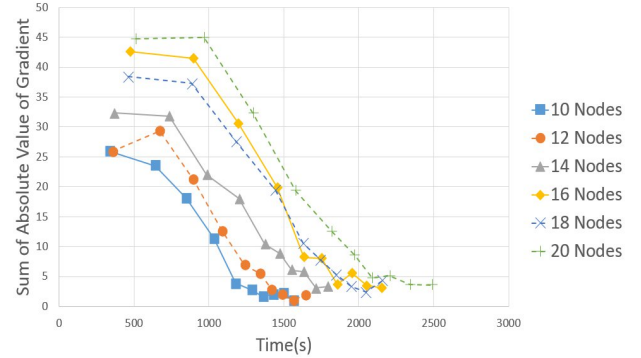

Figure 2: Convergence Behavior of Failure Rate Learning

We execute learning on graphs with $10, 12, \ldots, 18, 20$ nodes, where the graph with 10 nodes is shown in Figure 1. We add $1, 2, \ldots, 5$ layers of 2 nodes between Node 1 and Node 2, 4 to obtain the other graphs, where there is an edge between every node in one layer and every node in the previous and next layer. Figure 2 shows the convergence behavior over time in terms of the sum of the absolute values of gradients of all weights. Running time is mostly spent by the uniform sampler for answer sets. The experiments are performed on a machine with 4 Intel(R) Core(TM) i5-2400 CPU with OS Ubuntu 14.04.5 LTS and 8 GB memory.

Figure 2 shows that convergence takes longer as the number of nodes increases, which is not surprising. Note that the current implementation is not very efficient. Even for graphs with $10 - 20$ nodes, it takes $1500 - 2000$ seconds to obtain a reasonable convergence. The computation bottleneck lies in the uniform sampler used in Step 2(c) of Algorithm 1 whereas creating Π_{neg} and turning LP^{MLN} programs into ASP programs are done instantly. The uniform sampler that we use, XORRO, follows Algorithm 2 in (Gomes, Sabharwal, and Selman 2007). It uses a fixed number of random XOR constraints to prune out a subset of stable models, and randomly select one remaining stable model to return. The process of solving for all stable models after applying XOR constraints can be very time-consuming.

In this example, it is essential that the samples are generated by an ASP solver because information about node failing needs to be correctly derived from the connectivity, which involves reasoning about the transitive closure.

As Theorem 5 indicates, this weight learning task can alternatively be done through ProbLog weight learning. We use PROBLOG,[8] an implementation of ProbLog. The performance of PROBLOG on weight learning depends on the tightness of the input program. We observed that for many tight

---

[8] https://dtai.cs.kuleuven.be/problog/

programs, PROBLOG appears to have better scalability than our prototype LPMLN-LEARN. However, PROBLOG system does not show a consistent performance on non-tight programs, such as the encoding of the network example above, possibly due to the fact that it has to convert the input program into weighted Boolean formulas, which is expensive for non-tight programs.[9] We can identify many graph instances of the network failure example where our prototype system outperforms PROBLOG, as the density of the graph gets higher. For example, consider the graph in Figure 1. With the nodes fixed, as we add more edges to make the graph denser, we eventually hit a point when PROBLOG does not return a result within a reasonable time limit. Below is the statistics of several instances.

# Edges	LPMLN-LEARN	PROBLOG	PROBLOG (with modified program)
10	351.237s	2.565s	0.846s
14	476.656s	2.854s	0.833s
15	740.656s	> 20 min	0.957s
20	484.348s	> 20 min	76.143s
40	304.407s	> 20 min	26.642s

The input files to PROBLOG consist of two parts: edge lists and the part that defines the node failure rates and connectivity. The latter is different for the second column and the third column in the table. For the second column it is the same as the input to LPMLN-LEARN:

```
t(_)::fail(1).    ...    t(_)::fail(10).

connected(X, Y):- edge(X, Y), not fail(X), not fail(Y).
connected(X, Y):- connected(X, Z), connected(Z, Y).
```

For the third column, we rewrite the rules to make the Boolean formula conversion easier for PROBLOG. The input program is:[10]

```
t(_)::fail(1).    ...    t(_)::fail(10).

aux(X, Y) :- edge(X, Y), not fail(X), not fail(Y).
connected(X, Y) :- aux(X, Y).
connected(X, Y) :- connected(X, Z), aux(Z, Y).
```

Although all graph instances have some cycles in the graph, the difference between the instance with 14 edges and 15 edges is the addition of one cycle. Even with the slight change in the graph, the performance of PROBLOG becomes significantly slower.

6.3 Learning Parameters for Abductive Reasoning about Actions

One of the successful applications of answer set programming is modeling dynamic domains. LP^{MLN} can be used for extending the modeling to allow uncertainty. A high-level action language $p\mathcal{BC}+$ is defined as a shorthand notation for LP^{MLN} (Lee and Wang 2018). The language allows

---

[9]The difference appears to be analogous to the different approaches to handling non-tight programs by answer set solvers, e.g., the translation-based approach such as ASSAT and CMODELS and the native approach such as CLINGO.

[10]This was suggested by Angelika Kimmig (personal communication)

for probabilistic diagnoses in action domains: given the action description and the histories where an abnormal behavior occurs, how to find the reason for the failure? There, the probabilities are specified by the user. This can be enhanced by learning the probability of the failure from the example histories using LPMLN-LEARN.[11] In this section, we show how LP^{MLN} weight learning can be used for learning parameters for abductive reasoning in action domains. Due to the self-containment of the paper, instead of showing $p\mathcal{BC}+$ descriptions, we show its counterpart in LP^{MLN}.

Consider the robot domain described in (Iwan 2002): a robot located in a building with 2 rooms r1 and r2 and a book that can be picked up. The robot can move to rooms, pick up the book, and put down the book. Sometimes actions may fail: the robot may fail to enter the room, may fail to pick up the book, and may drop the book when it has the book. The domain can be modeled using answer set programs, e.g., (Lifschitz and Turner 1999). We illustrate how such a description can be enhanced to allow abnormalities, and how the LP^{MLN} weight learning method can learn the probabilities of the abnormalities given a set of actions and their effects.

We introduce the predicate $Ab(i)$ to represent that some abnormality occurred at step i, and the predicate $Ab(AbnormalityName, i)$ to represent that a specific abnormality $AbnormalityName$ occurred at step i. The occurrences of specific abnormalities are controlled by probabilistic fact atoms and their preconditions. For example,

$$w_1 : Pf_1(i)$$
$$\alpha : Ab(EnterFailed, i) \leftarrow Pf_1(i), Ab(i).$$

defines that the abnormality $EnterFailed$ occurs with probability $\frac{e^{w_1}}{e^{w_1}+1}$ (controlled by the weighted atomic fact $Pf_1(i)$, which is introduced to represent the probability of the occurrence of $EnterFailed$) at time step i if there is some abnormality at time step i. Similarly we have

$$w_2 : Pf_2(i)$$
$$\alpha : Ab(DropBook, i) \leftarrow Pf_2(i), Ab(i).$$
$$w_3 : Pf_3(i)$$
$$\alpha : Ab(PickupFailed, i) \leftarrow Pf_3(i), Ab(i).$$

When we describe the effect of actions, we need to specify "no abnormality" as part of the precondition of the effect: The location of the robot changes to room r if it goes to room r unless abnormality $EnterFailed$ occurs:

$$\alpha : LocRobot(r, i+1) \leftarrow Goto(r, i), not\ Ab(EnterFailed, i).$$

The location of the book is the same as the location of the robot if the robot has the book:

$$\alpha : LocBook(r, i) \leftarrow LocRobot(r, i), HasBook(T, i).$$

---

[11]ProbLog could not be used in place of LP^{MLN} here because it has the requirement that every total choice leads to exactly one well founded model, and consequently does not support choice rules, which has been used in the formalization of the robot example in this section.

The robot has the book if it is at the same location as the book and it picks up the book, unless abnormality *PickupFailed* occurs:

$\alpha : HasBook(\text{TRUE}, i+1) \leftarrow PickupBook(\text{TRUE}, i),$
$\quad LocRobot(r, i), LocBook(r, i), not\ Ab(PickupFailed, i).$

The robot loses the book if it puts down the book:

$\alpha : HasBook(\text{FALSE}, i+1) \leftarrow PutdownBook(\text{TRUE}, i).$

The robot loses the book if abnormality *DropBook* occurs:

$\alpha : HasBook(\text{FALSE}, i+1) \leftarrow Ab(DropBook, i).$

The commonsense law of inertia for each fluent is specified by the following hard rules:

$\alpha : \{LocRobot(r, i+1)\} \leftarrow LocRobot(r, i), astep(i).$
$\alpha : \{LocBook(r, i+1)\} \leftarrow LocBook(r, i), astep(i).$
$\alpha : \{HasBook(b, i+1)\} \leftarrow HasBook(b, i), astep(i).$

For the lack of space, we skip the rules specifying the uniqueness and existence of fluents and actions, rules specifying that no two actions can occur at the same timestep, and rules specifying that the initial state and actions are exogenous.

We add the hard rule

$$\alpha : Ab(i) \leftarrow astep(i)$$

to enable abnormalities for each timestep i.

To use multiple action histories as the training data, we use the method from Section 4.1 and introduce an extra argument to every predicate, that represents the action history ID.

We then provide a list of 12 transitions as the training data. For example, the first transition (ID =1) tells us that the robot performed `goto` action to room `r2`, which failed.

```
:- not loc_robot("r1",0,1). :- not loc_book("r2",0,1).
:- not hasBook("f",0,1).    :- not goto("r2",0,1).
:- not loc_robot("r1",1,1).
```

Among the training data, `enter_failed` occurred 1 time out of 4 attempts, `pickup_failed` occurred 2 times out of 4 attempts, and `drop_book` occurred 1 time out of 4 attempts. The transitions are partially observed data in the sense that they specify only some of the fluents and actions; other facts about fluents, actions and abnormalities have to be inferred.

Note that this program is $(|A| + 1)$-coherent, where $|A|$ is the number of actions (i.e., *Goto*, *PickupBook* and *DropBook*) and 1 is for no actions. We execute gradient ascent learning with 50 learning iterations and 50 sampling iterations for each learning iteration. The weights learned are

```
Rule 1: -1.084    Rule 2: -1.064    Rule 3: -0.068
```

The probability of each abnormality can be computed from the weights as follows:

$$P(\texttt{enter\_failed}) = \frac{exp(-1.084)}{exp(-1.084) + 1} \approx 0.253$$

$$P(\texttt{drop\_book}) = \frac{exp(-1.064)}{exp(-1.064) + 1} \approx 0.257$$

$$P(\texttt{pickup\_failed}) = \frac{exp(-0.068)}{exp(-0.068) + 1} \approx 0.483$$

The learned weights of `pf` atoms indicate the probability of the action failure when some abnormal situation `ab(I, ID)` happens. This allows us to perform probabilistic diagnostic reasoning in which parameters are learned from the histories of actions. For example, suppose the robot and the book were initially at `r1`. The robot executed the following actions to deliver the book from `r1` to `r2`: pick up the book; go to `r2`; put down the book. However, after the execution, it observes that the book is not at `r2`. What was the problem?

Executing system LPMLN2ASP on this encoding tells us that the most probable reason is that the robot fails at picking up the book. However, if we add that the robot itself is also not at `r2`, then LPMLN2ASP computes the most probable stable model to be the one that has the robot failed at entering `r2`.

7 Conclusion

The work presented relates answer set programming to *learning from data*, which has been under-explored, with some exceptions like (Law, Russo, and Broda 2014; Nickles 2016). Via LP^{MLN}, learning methods developed for Markov Logic can be adapted to find the weights of rules under the stable model semantics, utilizing answer set solvers for performing MCMC sampling. Rooted in the stable model semantics, LP^{MLN} learning is useful for learning parameters for programs modeling knowledge-rich domains. Unlike MC-SAT for Markov Logic, MC-ASP allows us to infer the missing part of the data guided by the stable model semantics. Overall, the work paves a way for a knowledge representation formalism to embrace machine learning methods.

The current LP^{MLN} learning implementation is a prototype with the computational bottleneck in the uniform sampler, which is used as a blackbox. Unlike the work in machine learning, sampling has not been much considered in the context of answer set programming, and even the existing sampler we adopted was not designed for iterative calls as required by the MCMC sampling method. This is where we believe a significant performance increase can be gained. Using the idea such as constrained sampling (Meel et al. 2016) may enhance the solution quality and the scalability of the implementation, which is left for future work.

PrASP (Nickles and Mileo 2014) is related to LP^{MLN} in the sense that it is also a probabilistic extension of ASP. Weight learning in PrASP is very similar to weight learning in LP^{MLN}: a variation of gradient ascent is used to update the weights so that the weights converge to a value that maximizes the probability of the training data. In PrASP setting, it is a problem that the gradient of the probability of the training data cannot be expressed in a closed form, and is thus hard to compute. The way how PrASP solves this problem is to approximate the gradient by taking the difference between the probability of training data with current weight and with current weight slightly incremented. The

probability of training data, given fixed weights, is computed with inference algorithms, which typically involve sampling methods.

In this paper, we only considered LP$^{\text{MLN}}$ weight learning with the basic gradient ascent method. There are several advanced weight learning techniques and sophisticated problem settings used for MLN weight learning that can possibly be adapted to LP$^{\text{MLN}}$. For example, (Lowd and Domingos 2007) discussed some enhancement to the basic gradient ascent, (Khot et al. 2011) proposed a method for learning the structure and the weights simultaneously, and (Mittal and Singh 2016) discussed how to automatically identify clusters of ground instances of a rule and learn different weight for each of these clusters.

References

Balai, E., and Gelfond, M. 2016. On the relationship between P-log and LP$^{\text{MLN}}$. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 915–921.

Baral, C.; Gelfond, M.; and Rushton, J. N. 2009. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming* 9(1):57–144.

De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI*, volume 7, 2462–2467.

Erdem, E., and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming* 3:499–518.

Ferraris, P.; Lee, J.; and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence* 175:236–263.

Fierens, D.; Van den Broeck, G.; Renkens, J.; Shterionov, D.; Gutmann, B.; Thon, I.; Janssens, G.; and De Raedt, L. 2013. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*,15(3), 358-401. doi:10.1017/S1471068414000076

Gebser, M.; Schaub, T.; Marius, S.; and Thiele, S. 2016. xorro: Near uniform sampling of answer sets by means of XOR. https://potassco.org/labs/2016/09/20/xorro.html.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2007. Near-uniform sampling of combinatorial spaces using XOR constraints. In Schölkopf, B.; Platt, J. C.; and Hoffman, T., eds., *Advances in Neural Information Processing Systems 19*. MIT Press. 481–488.

Iwan, G. 2002. History-based diagnosis templates in the framework of the situation calculus. *AI Communications* 15(1):31–45.

Khot, T.; Natarajan, S.; Kersting, K.; and Shavlik, J. 2011. Learning Markov Logic Networks via functional gradient boosting. In *2011 11th IEEE International Conference on Data Mining*, 320–329.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive learning of answer set programs. In *Logics in Artificial Intelligence*. Springer. 311–325.

Lee, J., and Wang, Y. 2016. Weighted rules under the stable model semantics. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 145–154.

Lee, J., and Wang, Y. 2018. A probabilistic extension of action language $\mathcal{BC}+$. *Theory and Practice of Logic Programming*,18(3-4), 607-622. doi:10.1017/S1471068418000303

Lee, J., and Yang, Z. 2017. LPMLN, weak constraints, and P-log. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1170–1177.

Lee, J.; Meng, Y.; and Wang, Y. 2015. Markov logic style weighted rules under the stable model semantics. In Technical Communications of the 31st International Conference on Logic Programming.

Lee, J.; Talsania, S.; and Wang, Y. 2017. Computing LPMLN using ASP and MLN solvers. *Theory and Practice of Logic Programming*, 17(5-6), 942-960. doi:10.1017/S1471068417000400

Lifschitz, V., and Turner, H. 1999. Representing transition systems by logic programs. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 92–106.

Lowd, D., and Domingos, P. 2007. Efficient weight learning for markov logic networks. In *European Conference on Principles of Data Mining and Knowledge Discovery*, 200–211. Springer.

Meel, K. S.; Vardi, M. Y.; Chakraborty, S.; Fremont, D. J.; Seshia, S. A.; Fried, D.; Ivrii, A.; and Malik, S. 2016. Constrained sampling and counting: Universal hashing meets sat solving. In *AAAI Workshop: Beyond NP*.

Mittal, H., and Singh, S. S. 2016. Fine grained weight learning in markov logic networks. In *International Workshop on Statistical Relational AI*.

Nickles, M., and Mileo, A. 2014. Probabilistic inductive logic programming based on answer set programming. In *15th International Workshop on Non-Monotonic Reasoning (NMR 2014)*.

Nickles, M. 2016. A tool for probabilistic reasoning based on logic programming and first-order theories under stable model semantics. In *European Conference on Logics in Artificial Intelligence (JELIA)*, 369–384.

Pearl, J. 2000. *Causality: models, reasoning and inference*, volume 29. Cambridge Univ Press.

Poon, H., and Domingos, P. 2006. Sound and efficient inference with probabilistic and deterministic dependencies. In *AAAI*, volume 6, 458–463.

Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62(1-2):107–136.

Wang, B., and Zhang, Z. 2017. A parallel LPMLN solver: Primary report. In Working Notes of the Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP).