Constraints for Probabilistic Logic Programming

Daan Fierens Guy Van den Broeck Maurice Bruynooghe Luc De Raedt
Department of Computer Science, KU Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract

In knowledge representation, one commonly distinguishes *definitions* of predicates from *constraints*. This distinction is also useful for probabilistic programming and statistical relational learning as it explains the key differences between probabilistic programming languages such as ICL, ProbLog and Prism (which are based on definitions) and statistical relational learning languages such as Markov Logic (based on constraints). This motivates us to extend ProbLog with constraints; the resulting cProbLog in a sense unifies ProbLog and Markov Logic and is strictly more expressive than either of them.

1 Introduction

Popular formalisms for representing probability distributions over relational models include probabilistic logic programming languages (PLP) such as PRISM [9], ProbLog [2] and ICL [8], as well as statistical relational modelling languages such as Markov Logic [4]. Inspired by Answer Set Programming (ASP) [1] we explain the differences between these two types of approaches based on the notions of (inductive) definitions and constraints. Basically, the PLP approaches use (inductive) definitions to define predicates while Markov Logic uses first-order logic to define (soft) constraints on possible worlds.

To explain the concept of an inductive definition, consider the example of the transitive closure of a relation in a graph. In logic programming, one would typically use two Horn-clauses to define the transitive closure, one specifying the base case and the other the recursive case; cf. the `reachable` predicate below as the transitive closure of `open_road`. The meaning of this definition is inductive [3]: whenever the condition of the rule is in a partial possible world (i.e., partial Herbrand interpretation), the conclusion part should be added to the possible world as well. In logic programming, this process continues until a fix point (of the well-known T_P operator) is reached and the transitive closure is obtained. It is well-known in the knowledge representation and database literature that it is impossible to define transitive closures in first-order logic [6] .Concretely, while it is of course possible to express in first-order logic that a given relation should be *transitive*, it is not possible to express the *closure*.

PLP is based on logic programming and hence, employs (inductive) definitions to define predicates, while Markov Logic is based on first-order logic, which explains why it is impossible to define transitive closures in Markov Logic. On the other hand, Markov Logic uses first-order logic as soft constraints on the possible worlds, a feature not supported by current PLP approaches.

Motivated by these differences and by ASP (which supports both inductive definitions and constraints), we will now introduce the language cProbLog, an extension of ProbLog, which supports constraints specified in first-order logic. So, cProbLog in a sense unifies the Markov Logic and PLP paradigms and offers the 'best of both worlds'.

2 The cProbLog language

We illustrate cProbLog with a simple example. NIPS workshop participants have to travel to Lake Tahoe via a network of snowy roads. Every road has a prior probability of actually being open. However, the NIPS organizers ensure us that Lake Tahoe will be reachable from at least one of the two airports in the region (Reno or Sacramento). A location Y is defined to be 'reachable' from another location X if there is a path of open roads connecting X to Y, cf. the following cProbLog model.

```
%% 1) Probabilistic facts:
0.4::open_road('Reno','town1').
0.3::open_road('town1','town3').
...
0.5::open_road('town6','Tahoe').

%% 2) Logic programming rules / inductive definition:
reachable(X,Y) :- open_road(X,Y).
reachable(X,Y) :- open_road(X,Z), reachable(Z,Y).

%% 3) Constraints:
reachable('Reno','Tahoe') v reachable('Sacramento','Tahoe').
```

Syntax and informal semantics. Like any cProbLog program, our example program consists of three layers. The first two layers are the same as in the regular ProbLog language [2], the third layer is new in cProbLog.

The *first layer* consists of a set of probabilistic facts. Below, we assume that all probabilistic facts are ground (this is to simplify the presentation; in principle, we can add syntactic sugar for non-ground probabilistic facts as in regular ProbLog). A probabilistic fact specifies the prior probability of the involved atom being true or not: for example, `open_road('Reno','town1')` is true with probability 0.4 and false with probability 0.6. Intuitively, each probabilistic fact can be seen as a 'switch' that is either on or off, with the indicated probability.

The *second layer* consists of a logic program (with the syntactic restriction that no head of a rule in the program unifies with a probabilistic fact). Intuitively, the rules in this program should be read as *(inductive) definitions* which define new predicates in terms of the predicates in the probabilistic facts. In our example, the rules define the concept 'reachable' in terms of the concept 'open road': Y is reachable from X if and only if there is an open road from X to Y, or there is an open road from X to an intermediate location Z and Y is reachable from Z. It is known from the knowledge representation literature [3] that logic programs (and their least Herbrand model semantics) are indeed suitable for expressing this kind of definitional knowledge.

The *third layer* is not present in the regular ProbLog language but is new to cProbLog. This layer consists of a set of First-Order Logic (FOL) formulas. Syntactically, one can write any set of FOL formulas involving the predicates introduced in the previous two layers of the cProbLog program. Intuitively, each FOL formula should be seen as a *constraint* on the possible worlds allowed by the cProbLog program: a world that violates a constraint is considered impossible (and will get zero probability mass). In our example, the constraint says that Lake Tahoe is reachable from Reno or Sacramento (the connective '`v`' stands for disjunction as in FOL, so non-exclusive). Any world in which this constraint does not hold, is ruled out. Such integrity constraints cannot be expressed directly in the regular ProbLog language, though they are often useful for modelling.

The constraints (third layer) and the logic programming rules (second layer) are each suited for expressing a different kind of knowledge. The logic programming rules define new predicates on top of the predicates occurring in the probabilistic facts. This does not alter the set of worlds considered to be possible, it only extends the vocabulary with which these worlds are described. The constraints, on the other hand, allow us to prune the set of worlds, eliminating any worlds that do not satisfy the necessary conditions.

Formal semantics. A cProbLog program defines a probability distribution over possible worlds (in our example, each world is an interpretation of the `open_road` and `reachable` predicates). While this formal semantics can be defined in a self-contained way, space restrictions prevent us from doing so here. Instead, we restrict ourselves to defining the cProbLog semantics *relative* to the

regular ProbLog semantics. For cProbLog programs without constraints, the semantics coincides with that of ProbLog. We now discuss the case of cProbLog programs with constraints.

We use $P(.)$ for the distribution according to cProbLog semantics and $P^*(.)$ for regular ProbLog semantics [7]. Given a cProbLog program π, we use π_{con} to denote the set of all constraints in π and we use π_{reg} to denote the regular ProbLog program obtained by dropping all these constraints from π (so syntactically, $\pi = \pi_{reg} \cup \pi_{con}$). The probability of a world ω given a cProbLog program π, denoted $P_\pi(\omega)$, is $\frac{P^*_{\pi_{reg}}(\omega)}{\sum_{\omega' \models \pi_{con}} P^*_{\pi_{reg}}(\omega')}$ if $\omega \models \pi_{con}$, and is zero otherwise. Here $\omega \models \pi_{con}$ means that ω satisfies all constraints in π. In words: we start from the distribution $P^*_{\pi_{reg}}(\omega)$ as defined by regular ProbLog semantics (ignoring the constraints in π), we set the probability of all worlds that violate a constraint in π to zero, and we re-normalize the resulting distribution. This is equivalent to *conditioning* the distribution $P^*_{\pi_{reg}}(\omega)$ on the evidence that all constraints in π_{con} are true. The point is that conditioning cannot be expressed within the regular ProbLog language (and hence is typically pushed inside the inference process), while it can be expressed in cProbLog.

Operational view. The semantics implies that a cProbLog program specifies a *rejection sampling* process. The sampling process follows the order of the program layers. In the first step, we independently sample a truth value for each probabilistic fact (we 'set the switches'). The result is a world ω_0 described in terms of the probabilistic predicates (e.g., the `open_road` predicate). In the second step, we apply the logic programming rules to determine the truth values of all derived atoms (e.g., the `reachable` atoms) given the truth values of the probabilistic facts. The result is still a single world ω, but one that is now described in terms of a larger vocabulary than ω_0 . In the third step, we retain ω as a sample if it satisfies all constraints, and reject it otherwise. It is this rejection step that can be expressed in cProbLog but not in regular ProbLog.

3 Inference and learning in cProbLog

To do inference and learning in cProbLog, we can extend ProbLog2, the system for the regular ProbLog language. ProbLog2 works in two steps [5]: first it converts the given program (probabilistic facts and rules) to an equivalent weighted Boolean formula, next inference and learning operate on this formula. The inference and learning algorithms do not 'see' the given program directly, only through the formula. Hence, to extend this approach to cProbLog, we only need to adapt the first step such that it takes into account the constraints when building the formula; the inference and learning algorithms are then unaffected.

The most naive way of converting a cProbLog program to an equivalent weighted Boolean formula is very simple: we *separately* build the formula for the probabilistic facts and rules, and the formula for the FOL constraints. For the former, we can use the current ProbLog techniques; for the latter, we can simply apply a FOL grounder to the constraints, yielding an essentially propositional Boolean formula. While this simple approach is in principle sufficient, it might also be possible to *jointly*, rather than separately, convert rules and FOL constraints. The constraints restrict the possible worlds, which could provide information that allows us to prune the encoding of the rules, resulting in a possibly more compact formula. Such optimizations are used in some ASP systems and other related systems [10]. Using them in a probabilistic context is an interesting direction for future research.

4 cProbLog and Markov Logic

Let us now compare cProbLog and Markov Logic as representation languages. We can show that cProbLog strictly subsumes Markov Logic: for every Markov Logic Network (MLN) an equivalent cProbLog program exists, but not vice versa.

From MLNs to cProbLog. Every MLN can be mapped to an equivalent cProbLog program, provided that we introduce some auxiliary predicates in the vocabulary. As a very simple example, consider the MLN consisting of a single formula $a \rightarrow b$ with weight w. The resulting cProbLog program consists of three probabilistic facts `0.5::a`, `0.5::b` and `p::c`, with c an auxiliary atom and with the probability p being $e^w/(e^w + 1)$, and one constraint, $c \leftrightarrow (a \rightarrow b)$. This program is equivalent to the MLN in the sense that, if we marginalize out the auxiliary atom c from the specified

probability distribution, the resulting distribution is identical to that of the MLN. The strategy shown in this simple example can be generalized (introduce one auxiliary atom per weighted formula, etc) and works for any MLN, including non-ground MLNs.

From cProbLog to MLNs. As already mentioned in Section 1, not every cProbLog program can be mapped to an equivalent MLN. This is in particular the case for programs including inductive definitions that define the transitive closure of a given relationship, such as the definition of $reachable$ in our above example. It is well-known in the knowledge representation community that first-order logic is not expressive enough to describe the transitive closure [6]. Hence neither is Markov Logic. This makes Markov Logic unsuitable for certain kinds of applications, like the typical ProbLog applications with probabilistic graphs [2, 7]. On the other hand, the regular ProbLog language is not sufficient when we want to impose certain constraints on the possible worlds. From a representation point of view, cProbLog is in a sense the 'best of both worlds', integrating logic programming and first-order logic.

Acknowledgments

DF and GVdB are supported by are supported by the Research Foundation-Flanders (FWO-Vlaanderen).

References

[1] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[2] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007.

[3] M. Denecker, M. Bruynooghe, and V. W. Marek. Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2(4):623–654, 2001.

[4] P. Domingos, S. Kok, D. Lowd, H. Poon, M. Richardson, and P. Singla. *Probabilistic Inductive Logic Programming - Theory and Applications*, chapter 'Markov Logic'. Lecture Notes in Computer Science. Springer, 2008.

[5] D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and L. De Raedt. Inference in Probabilistic Logic Programs using Weighted CNFs. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*, pages 211–220, 2011.

[6] E. Grädel. On Transitive Closure Logic. In *Proceedings of the 5th Workshop on Computer Science Logic*, volume 626 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 1992.

[7] A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the Implementation of the Probabilistic Logic Programming Language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2010.

[8] D. Poole. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):5–56, 1997.

[9] T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1330–1335, 1997.

[10] J. Wittocx, M. Mariën, and M. Denecker. Grounding FO and FO(ID) with Bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.