Probabilistic Logic Programming

Felix Weitkämper
LMU München

October 19, 2021

For most of its history, artificial intelligence was intimately connected with the art of automated reasoning. In a classical system of symbolic artificial intelligence, knowledge is encoded by experts as facts and rules in a suitable formal language. Using those, new knowledge can be derived by reasoning. Facts and rules could be highly sophisticated, involving intricate relations between different entities. Paradigmatic for this approach are expert systems and other artificial intelligence applications based on *logic programming*, and in particular on the language Prolog and its dialects. While this leads to a highly expressive way of representing knowledge, a major limitation soon emerged: The world is indeed complex, but it is also uncertain, and general rules hardly hold without exception.

Although probability theory and statistics are well-developed, time-honoured parts of science, incorporating probabilistic reasoning into artificial intelligence was long considered out of the question, since it would be impossible to specify all conceivable correlations and dependencies between factors. However, with the introduction of Bayesian Networks in the late 1980s, transparent independence assumption suddenly made this viable. Since then, statistical machine learning has developed into a large area of artificial intelligence. In the process, the emphasis shifted from reasoning about complex relationships to learning the dependencies between simple properties.

However, the world has not become any less interconnected since then. Therefore, since the mid-1990s, the new field of STAtistical Relational Artficial Intelligence (StarAI) aims to bring those paradigms together. This can be approached from either direction: Either, one can expand logic programming to incorporate probabilities, or one can extend graphical models to incorporate relational information.

In this short review, we will discuss the former approach to StarAI, *probabilistic logic programming*. This provides the backdrop to the first column on *What's hot in statistical relational AI* in this same issue, which focuses on recent developments presented at this year's International Conference for Logic Programming.

The key idea behind probabilistic logic programming is the *distribution semantics*, introduced by T. Sato in 1995. The distribution semantics neatly divides a probabilistic logic program into a simple list of probabilistic facts and

an ordinary logic program which takes those probabilistic facts as input to compute more complex predicates. More precisely, a probabilistic logic program consists of a list of assertions, each annotated with a probability between 0 and 1, and a logic program whose extensional vocabulary coincides with the signature of those facts. We illustrate this idea with a common toy example from the literature:

Example. The probabilistic logic program *Smokers and Friends* consists of the probabilistic facts

```
0.2 :: befriends(X,Y).
0.5 :: influences(X,Y).
0.3 :: stress(X).
```

and the rules

```
friends(X) :- befriends(X,Y).
friends(X) :- befriends(Y,X).
smokes(X)  :- stress(X).
smokes(X)  :- friends(X,Y), smokes(Y), influences(Y,X).
```

The semantics of this program reads as follows: For every domain entity (referred to as a person in the following), there is a 30% chance that this person is stressed (that is, `stress(person)` is true). Similarly, for every pair of persons, there is a 20% chance that one person befriends the other, and a 50% chance that one person influences the other. All these random choices are made independently of one another. After these choices have been made, the rules of the program are brought to bear. In this case, the binary relation `friends` is evaluated as the symmetric closure of `befriends`, and the smokes relation is computed as follows: Firstly, every stressed individual smokes. Then, `smokes` predicate is spread recursively along friends influencing each other. Once this process has been completed and no more smokers can be added this way, the program terminates.

On the first glance, this formalism seems very restrictive, since it confines probability entirely to independent choices on factual assertions. However, the ingenuity of the distribution semantics is that probabilistic rules can be expressed by simply adding additional auxiliary predicates:

Example. The probabilistic rule

```
0.5 :: smokes(X) :- friends(X,Y), smokes(Y).
```

is equivalent to the clauses

```
smokes(X)  :- friends(X,Y), smokes(Y), influences(Y,X).
  0.5 :: influences(X,Y).
```

of the program in the first example.

In fact, it has been shown that if only ground (propositional) clauses are considered, acyclic probabilistic logic programs have the same expressiveness as Bayesian networks, while general probabilistic logic programs go beyond that by adding recursion.

However, the real strength of probabilistic logic programs (and StarAI in general) lies in the ability to specify rules and facts with variables, and therefore to specify a particular probability distribution for any domain under consideration. From this viewpoint the probabilistic logic program associates with every domain of people a probability distribution over possible L-structures on this domain, where L contains all the predicates mentioned in the program.

This framework can be extended to consider as input not just plain domains but structures in an extensional vocabulary $E \subseteq L$; for instance, the network of friendships in the examples here could be taken as input rather than generated probabilistically. In that case, every input structure M gives rise under a probabilistic logic program to a probability distribution over all possible L-structures extending M. For a more detailed contemporary treatment of probabilistic logic programming under the distribution semantics, see the comprehensive texbook by F. Riguzzi (2020:Foundations of Probabilistic Logic Programming, River Publishers).

The tasks for any probabilistic logic program system can be divided into two main headers: inference and learning.

Among inference tasks, the most prominent are marginal inference, where the probabilities of a certain property are computed, and maximum-a-posteriori (MAP) inference, where the most likely configuration to arise from a probabilistic logic program is computed. In the example above, a marginal query could compute the probability of a certain individual smoking. MAP inference could answer the question: What is the most likely smoking behaviour of this group of individuals given some observations?

Unfortunately, the naive approach to grounding, that is, substituting the domain individuals into the probabilistic logic program, and then performing inference on the ground program does not scale well to large datasets. Therefore, current research is much concerned with lifted inference, which is performed to varying extent on the level of the original probabilistic logic program (with variables) rather than on the large grounded program.

The problem of learning probabilistic logic programs from data is posed in different settings. In parameter learning, the rules of the program are fixed and an optimal set of probabilities for those rules are sought. In structure learning, the rules themselves are not given either and have to be found by the learning algorithm. Structure learning of probabilistic logic programs is generally considered a difficult problem, since it subsumes and significantly extends the problem of learning deterministic logic programs, known as Inductive Logic Programming. Indeed, early structure learning algorithms approached structure-learning in two parts, applying first an Inductive Logic Programming system to learn the rules and then a parameter learning algorithm for the probabilities. However, better results have recently been obtaied by interleaving the two parts.

I would like to close by highlighting two particularly active and well-maintained systems, both of which have an easy-to-use online interface and tutorial.

ProbLog 2, an implementation of the ProbLog language maintained by the KU Leuven group available at https://dtai.cs.kuleuven.be/problog/, is a powerful Python-based system with many features. Its underlying language, ProbLog, has been designed particularly with usability and straightforward syntax in mind, and it has been used extensively in bioinformatics. A variety of academic and real-world applications are listed at
https://dtai.cs.kuleuven.be/problog/applications.html.

cplint, a SWI-Prolog-based system supporting various languages and maintained by the University of Ferrara, is available at https://cplint.ml.unife.it/. It is particularly rich in configurations and supported algorithms, including state-of-the-art methods for structure learning and lifted inference (neither of which are supported by ProbLog 2 at present). On the other hand, it is not quite as straightforward to use as ProbLog 2.