Speeding-up ProbLog’s Parameter Learning

Francisco H. O. V. de Faria¹, Arthur C. Gusmão¹, Fabio G. Cozman¹, Denis D. Mauá²
¹ Escola Politécnica, ² Instituto de Matemática e Estatística
Universidade de São Paulo, São Paulo, Brazil

Abstract

ProbLog is a state-of-art combination of logic programming and probabilities; in particular ProbLog offers parameter learning through a variant of the EM algorithm. However, the resulting learning algorithm is rather slow, even when the data are complete. In this short paper we offer some insights that lead to orders of magnitude improvements in ProbLog’s parameter learning speed with complete data.

Introduction

There are many ways to combine logical expressions and probabilities (Getoor and Taskar 2007; Raedt et al. 2010). Sato’s distribution semantics is perhaps the most popular probabilistic statements. To do parameter learning (that is, to learn probability values for a given program), ProbLog implements a variant of the EM algorithm that resorts to BDD diagrams so as to speed inference whenever possible (Fierens et al. 2014). The package is very friendly; one can easily use it to represent knowledge about deterministic and probabilistic rules/facts, and where there is an edge from each grounded subgoal to the corresponding grounded head. A program is acyclic if its dependency graph is acyclic.

A very short review

We consider the following syntax, entirely taken from the ProbLog package as described by Fierens et al. (2014). A rule is written as \( h := b_1, \ldots, b_n \), where \( h \) is an atom, called the head, and each \( b_i \) is an atom perhaps preceded by not. Each \( b_i \) is a subgoal and the left hand side is the body. A rule without a body, written \( h \), is a fact. Rules and facts can be grounded by replacing logical variables by constants. The dependency graph of a program is a graph where the nodes are the grounded atoms, and where there is an edge from each grounded subgoal to the corresponding grounded head. A program is acyclic if its dependency graph is acyclic.

A probabilistic fact, denoted by \( \alpha :: h \), consists of a number \( \alpha \), here assumed to be a rational in \([0, 1]\), and an atom \( h \). A probabilistic fact may contain logical variables, in which case it is interpreted as the set of grounded probabilistic facts produced by replacing logical variables by constants in every possible way. Additionally, ProbLog allow for probabilistic rules, written as \( \theta :: h := b_1, \ldots, b_n \), and interpreted as a pair consisting of a probabilistic fact \( \theta :: x \) and a rule \( h := b_1, \ldots, b_n, x \), where \( x \) is an auxiliary atom (with the same logical variables as \( h \)) that is not present anywhere else in the program.

Suppose we have a set of probabilistic rules/facts, but we do not know the values of the probabilities. We can use a dataset \( D \) to learn those parameters; we assume this is done by choosing parameters \( \Theta \) that attain \( \max_\Theta L(\Theta) \), where the log-likelihood \( L(\Theta) \) is the probability \( \log \Pr(D) \) with respect to parameters \( \Theta \). When \( D \) has some missing data, one popular way to maximize log-likelihood is to resort of the EM algorithm: here one iterates between inference and maximization of the expected log-likelihood. EM typically requires computing the probability of each random variable together with the missing variables that affect it (that is, the variable and its “parents”) (Darwiche 2009).

Parameter learning is done by ProbLog as follows. Any probabilistic rule is written as a pair consisting of a fresh auxiliary probabilistic fact and a deterministic rule. This guarantees that every probability is associated with an atom that has no parents in the dependency graph: both the inference and the maximization steps then become rather elementary (Fierens et al. 2014).
A better algorithm

A point to note is that, even when the data are complete, the auxiliary facts introduced to handle probabilistic rules are missing. Thus ProbLog must run the EM-style algorithm even when the input \( D \) is complete. One can see the consequences of this in Figure 1 even for small datasets, even for propositional acyclic programs, learning takes too long.

Our solution is not to insert an auxiliary (latent) atom for each probabilistic rule. Instead, we must write down the log-likelihood and maximize it directly; the main insight is that, for many rule patterns, this maximization can be done in closed-form. Consider an example. Suppose we have two propositional rules with the same head, say

\[
\theta_1 :: h, \quad \text{and} \quad \theta_2 :: h :: b,
\]

and a complete dataset with \( N \) observations of \((h, b)\). The log-likelihood (restricted to this head atom) is

\[
N_{i0} \log(1 - \theta_1) + N_{i1} \log(1 - \theta_1 - \theta_2 + \theta_1 \theta_2) + N_{10} \log(\theta_1) + N_{11} \log(\theta_1 + \theta_2 - \theta_1 \theta_2),
\]

where \( N_{ij} \) is the number of times the configuration \((h = i, b = j)\) (taking 1 to mean true and 0 to mean false). This is apparently much more complex than the usual log-likelihood (restricted to this head atom) is

\[
\log(1 - \theta) + \log(1 - \theta + \theta^2),
\]

in closed-form. Consider an example. Suppose we have two

\[
\begin{aligned}
\theta_1 &:: \text{fire}(X), \\
\theta_2 &:: \text{burglary}(X), \\
\theta_3 &:: \text{neighbor}(X,Y), \\
\theta_4 &:: \text{alarm}(X) :: \text{fire}(X), \\
\theta_5 &:: \text{alarm}(X) :: \text{burglary}(X), \\
\theta_6 &:: \text{calls}(X,Y) :: \text{neighbor}(X,Y), \text{alarm}(Y).
\end{aligned}
\]

17 probabilistic rules and 7 probabilistic facts. This is a relatively small program, yet the original ProbLog algorithm requires significant computer time, as can be seen in Figure 1.

We should note that our algorithm found similar values for the log-likelihood in all cases; that is, by introducing auxiliary atoms, ProbLog makes the maximization harder without making it more effective. For a propositional dataset with 50 observations, ProbLog reaches a log-likelihood of \(-176.65\) in 5498.06 seconds, while our algorithm reaches a log-likelihood of \(-176.60\) in 2.02 seconds.

The second example is a short relational program consisting of:

\[
\begin{aligned}
\theta_1 &:: \text{fire}(X), \\
\theta_2 &:: \text{burglary}(X), \\
\theta_3 &:: \text{neighbor}(X,Y), \\
\theta_4 &:: \text{alarm}(X) :: \text{fire}(X), \\
\theta_5 &:: \text{alarm}(X) :: \text{burglary}(X), \\
\theta_6 &:: \text{calls}(X,Y) :: \text{neighbor}(X,Y), \text{alarm}(Y).
\end{aligned}
\]

Suppose we have \( N \) constants, each one of them denoting a person in some city. Figure 1 compares the computational effort spent by the original ProbLog and our algorithm (\( N \) corresponds to the dataset size in the propositional case). Our algorithm outperforms the original ProbLog algorithm in computer time, reaching similar log-likelihood values. For a relational dataset with 25 constants, ProbLog reaches a log-likelihood of \(-523.11\) in 1863.03 seconds, while our algorithm reaches a log-likelihood of \(-523.11\) in 0.69 seconds.

**Experiments**

We have implemented the techniques discussed in the previous section, by modifying ProbLog’s parameter learning code. To demonstrate that the techniques are indeed effective, we present here two experiments; they are necessarily small because the original ProbLog algorithm cannot handle large models, and we want to compare our results with that previous algorithm. All of our tests were run in identical processors at Amazon Web Services.

So, consider first an acyclic propositional program that encodes the energy plant of a ship\footnote{We have obtained the model from the site \url{http://www.machineriespaces.com/emergency-power-supply.html}; this is an “almost” deterministic system in the sense that several relations are Boolean, together with sources of random noise.} using 16 propositions,
Acknowledgement
The first author is supported by a scholarship from Toshiba Corporation. The second author is supported by a scholarship from CNPq. The third and fourth authors are partially supported by CNPq. This work was partly supported by the São Paulo Research Foundation (FAPESP) grant 2016/01055-1 and the CNPq grants 303920/2016-5 and 420669/2016-7; also by São Paulo Research Foundation (FAPESP) grant 2016/18841-0 and CNPq grant 308433/2014-9; finally by FAPESP 2015/21880-4.

References
[Darwiche 2009] Darwiche, A. 2009. Modeling and Reasoning with Bayesian Networks. Cambridge University Press.
[Fierens et al. 2014] Fierens, D.; Van den Broeck, G.; Renkens, J.; Shrerionov, D.; Gutmann, B.; Janssens, G.; and de Raedt, L. 2014. Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory and Practice of Logic Programming 15(3):358–401.
[Getoor and Taskar 2007] Getoor, L., and Taskar, B. 2007. Introduction to Statistical Relational Learning. MIT Press.
[Gutmann et al. 2011] Gutmann, B.; Thon, I.; and De Raedt, L. 2011. Learning the Parameters of Probabilistic Logic Programs from Interpretations. Berlin, Heidelberg: Springer Berlin Heidelberg. 581–596.
[Raedt et al. 2010] Raedt, L. D.; Frasconi, P.; Kersting, K.; and Muggleton, S. 2010. Probabilistic Inductive Logic Programming. Springer.
[Sato and Kameya 2001] Sato, T., and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. Journal of Artificial Intelligence Research 15:391–454.