Speeding-up ProbLog's Parameter Learning

Francisco H. O. V. de Faria[1], **Arthur C. Gusmão**[1], **Fabio G. Cozman**[1], **Denis D. Mauá**[2]
[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 proposal in the literature (Sato and Kameya 2001). There one has rules as in logic programming, for instance

$$\mathsf{calls}(X,Y) :- \mathsf{alarm}(X), \mathsf{neighbor}(X,Y). \qquad (1)$$

and probabilities such as $\mathbb{P}(\mathsf{neighbor}(X,Y)) = 0.3$, meaning that the probability that any X and Y are neighbors is 0.3.

ProbLog is a freely available package that allows one to manipulate and learn such probabilistic logic programs (Fierens et al. 2014). The package is very friendly; one can easily use it to represent knowledge about deterministic and 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 (Gutmann et al. 2011; Fierens et al. 2014).

However, the EM-variant used by ProbLog is rather slow, *even* when there is no missing data in the input. This is somewhat perplexing for a new user, who may be surprised to find that even small propositional programs demand high computation effort with complete data. In this short paper we analyze this behavior of the EM-variant in ProbLog, and we detect the main reasons for its inefficiency. We propose techniques that lead to orders of magnitude improvements in speed. We demonstrate these gains with experiments. Even though the ideas in this short paper are in essence simple, they will be important in bringing probabilistic logic programming to real applications.

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 $\mathbf{not}$. 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 α, 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 Θ that attain $\max_\Theta L(\Theta)$, where the *log-likelihood* $L(\Theta)$ is the probability $\log \mathbb{P}(D)$ with respect to parameters Θ. 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

$$N_{00} \log(1-\theta_1) + N_{01} \log(1-\theta_1-\theta_2+\theta_1\theta_2) + N_{10} \log \theta_1 + N_{11} \log(\theta_1+\theta_2-\theta_1\theta_2),$$

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 likelihood one finds for example when learning Bayesian networks (Darwiche 2009); however, with some effort we find that the estimates that maximize log-likelihood are

$$\hat{\theta}_1 = \frac{N_{10}}{N_{00}+N_{10}}, \qquad \hat{\theta}_2 = \frac{N_{00}N_{11}-N_{10}N_{01}}{N_{00}N_{11}+N_{00}N_{01}}.$$

In fact, a very large number of rule patterns admit similar exact solutions. In this implementation we have covered all possible patterns for combinations of at most three rules entailing a same predicate. Notice that the log-likelihood expression does not depend on the size of the rules' bodies. And whenever the log-likelihood (for the rules that share the same head) does not admit a closed-form solution, we have found that a fast gradient-based algorithm can quickly find its maximum.

Thus our modified learning algorithm (for complete data) is to maximize likelihood directly, locally maximizing it in closed-form whenever possible, or locally maximizing it numerically with a fast gradient-based routine whenever necessary.

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[1] using 16 propositions, 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.

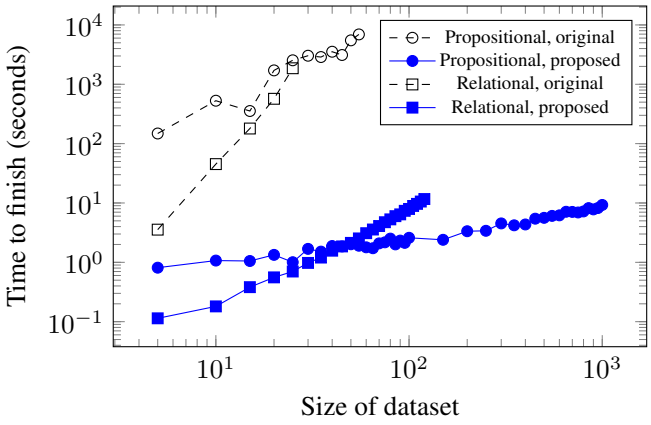

Figure 1: Time to learn parameters from data (note log-scale!). In the propositional case, size of dataset is the number of observations for all atoms; in the relational case, size of dataset is the the number of constants in the program.

The second example is a short relational program consisting of:

$$\theta_1 :: \mathsf{fire}(X).$$
$$\theta_2 :: \mathsf{burglary}(X).$$
$$\theta_3 :: \mathsf{neighbor}(X,Y).$$
$$\theta_4 :: \mathsf{alarm}(X) :- \mathsf{fire}(X).$$
$$\theta_5 :: \mathsf{alarm}(X) :- \mathsf{burglary}(X).$$
$$\theta_6 :: \mathsf{calls}(X,Y) :- \mathsf{neighbor}(X,Y), \mathsf{alarm}(Y).$$

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.

Conclusion

We have shown that ProbLog's parameter learning algorithm can be dramatically improved by resorting to a few insights: first, never introduce unnecessary latent atoms; second, maximize log-likelihood locally in the most efficient manner (closed-form or gradient-based). These insights will be helpful in future work dealing with missing data and with rule learning.

[1] We have obtained the model from the site http://www.machineryspaces.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.

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.