Inference and Learning with Model Uncertainty in Probabilistic Logic Programs

Victor Verreet,[1,2] **Vincent Derkinderen,**[1,2] **Pedro Zuidberg Dos Martires,**[1,2] **Luc De Raedt** [1,2,3]

[1] Department of Computer Science, KU Leuven, Belgium
[2] Leuven.AI - KU Leuven Institute for AI, Belgium
[3] Center for Applied Autonomous Systems, Örebro University, Sweden
{victor.verreet, vincent.derkinderen, pedro.zudo, luc.deraedt}@kuleuven.be

Abstract

An issue that has so far received only limited attention in probabilistic logic programming (PLP) is the modeling of so-called *epistemic uncertainty*, the uncertainty about the model itself. Accurately quantifying this model uncertainty is paramount to robust inference, learning and ultimately decision making. We introduce BetaProbLog, a PLP language that can model epistemic uncertainty. BetaProbLog has sound semantics, an effective inference algorithm that combines Monte Carlo techniques with knowledge compilation, and a parameter learning algorithm. We empirically outperform state-of-the-art methods on probabilistic inference tasks in second-order Bayesian networks, digit classification and discriminative learning in the presence of epistemic uncertainty.

1 Introduction

Uncertainty is often described as being either aleatoric or epistemic in nature (Hüllermeier and Waegeman 2021). The former is the intrinsic uncertainty that a probabilistic model brings and that cannot be reduced with further observations of the world. For example, the outcome of a coin toss is intrinsically uncertain and consequently the model of the toss will be probabilistic. In contrast, epistemic uncertainty, also called model uncertainty, stems from the lack of knowledge about the true underlying model and can be reduced with more observations. Explicitly modeling both types of uncertainty is important for gauging the uncertainty of learned probabilistic models and allows for reasoning over the robustness of subsequent predictions. It allows the user to avoid risky decisions when a learner is uncertain about its results or to perform probabilistic inference even when the probabilistic model is not exactly known.

Modeling epistemic uncertainty explicitly is studied in the fields of credal sets (CS) (Levi 1983) and subjective logic (SL) (Jøsang 2016). CS models the epistemic uncertainty by reasoning over convex sets of distributions, whereas SL uses beliefs. Both fields have issues that we address by introducing BetaProbLog, an extension of the probabilistic logic programming (PLP) language ProbLog. The first issue is that the field of SL often uses approximating operators. Cerutti et al. (2019) introduce SLProbLog, whose semantics rely on such approximations. As a result the semantics are not exactly defined. In contrast, BetaProbLog has a well-defined semantics rooted in probability theory. Second, although convex sets of distributions can suffice to make more informed decisions, CS lacks the ability to calculate expectation values. With so-called *second-order queries*, BetaProbLog can calculate any expectation value, offering a vast range of query types. Third, we provide an easy to interpret Monte Carlo inference algorithm for BetaProbLog based on knowledge compilation (Darwiche and Marquis 2002) combined with parallelized tensor operations, implemented with PyTorch (Paszke et al. 2019). This leads to an inference algorithm that is more than an order of magnitude faster than the one proposed by Cerutti et al. (2019). Fourth, we tackle (discriminative) parameter learning with epistemic uncertainty. To our knowledge, learning in second-order networks has so far been limited to networks of only two nodes (Kaplan et al. 2020). Our algorithm can handle networks of any size, by combining stochastic gradient descent with sampling and the reparametrization trick (Kingma and Welling 2014). We empirically evaluate our work on probabilistic inference tasks in second-order Bayesian networks, digit classification, and by performing parameter learning in the presence of epistemic uncertainty.

Consider the example of tossing two coins with unknown probabilities. We know that the queried probability of a single heads is in $[0.5, 0.6]$ and of two heads is in $[0.3, 0.4]$. From this information we want to learn a distribution over the coin probabilities. We can use BetaProbLog, SL or CS. However, SL can make a considerable approximation error in this case. CS returns intervals for the probabilities, but not their full distribution. With Bayesian learning probabilities are learned from data counts instead of target intervals, making it unfit for this type of problem. Using a loss that penalizes probabilities outside the target interval, BetaProbLog can learn the full distribution, allowing more queries to be answered, such as finding moments or quantiles. This helps users make decisions when it is important to know whether some probability lies below a given threshold.

2 Background

2.1 Inference in ProbLog

ProbLog is a PLP language that allows users to compactly write down complex relational probabilistic models while

abstracting away the intricacies of performing inference. The primary inference task supported by ProbLog is computing conditional probabilities $P(q \mid e)$. Since $P(q \mid e) = P(q, e)/P(e)$, we will focus on computing marginal probabilities and omit evidence for the sake of brevity. We now briefly delineate ProbLog's semantics and inference mechanism.

Example 1 *A simple ProbLog program comprised of probabilistic facts (lines* 1 *to* 3*) and logical rules (lines* 5 *and* 6*) is shown below.*

```
1  0.6::burglary.
2  0.2::earthquake.
3  0.5::alarm_on.
4
5  alarm :- alarm_on, burglary.
6  alarm :- alarm_on, earthquake.
```

A *probabilistic fact* is composed of an atom and the probability that the atom is true. In the example, the `burglary` atom is true with probability 0.6. A *literal* is an atom a or its negation $\neg a$.

A *logical rule* `h :- b` is composed of a head `h` and a body `b`. The head is a single atom, for example `alarm`, while the body consists of one or more literals. The meaning of a logical rule is that when the body is true, then the head can be derived to be true as well. In the example, `alarm` is true iff `alarm_on` is true and either `burglary` or `earthquake` is true. We assume without loss of generality that an atom only occurs as a probabilistic fact or as the head of rule, but not both. The former are called *probabilistic atoms*, the latter *derived atoms*.

A *model* M of a program is a truth assignment to all atoms in the program such that the probabilistic atoms are assigned either true or false, and the truth assignment of each derived atom further follows from the logical rules in the program. We will represent a model M as a set of literals where for each atom a: either $a \in M$ or $\neg a \in M$. For example, $M_1 = \{$`burglary`, $\neg$`earthquake`, `alarm_on`, `alarm`$\}$ is one model of the example program. For a more formal explanation of the two-valued well-founded model semantics utilised by ProbLog, we refer to Fierens et al. (2015). The probability of a model is defined as the product of the probabilities associated with the probabilistic atoms. For example, the model M_1 has a probability of $0.6 \cdot (1-0.2) \cdot 0.5 = 0.24$.

Given a ProbLog program and a query to that program, inference in ProbLog consists of four steps: grounding, conversion, compilation and finally evaluation. The grounding step first turns a ProbLog program and query `q` into a relevant ground program, thereby eliminating all logic variables. Note that the program in Example 1 is already ground. The relevant ground program for `q` is then converted into a weighted propositional theory ψ_q, capturing all of its two-valued well founded models (Fierens et al. 2015). The probability of atom `q` being satisfied is given by Sato's distribution semantics (Sato 1995):

$$P(q) = \sum_{M \models \psi_q} \prod_{l \in M} w(l) \tag{1}$$

where $w(l)$ maps a literal l to its weight, which is a probability. The sum runs over all models M of the program such that ψ_q evaluates to true.

Example 2 *To compute the probability of* `alarm` *being true in Example 1, the ProbLog program is converted into a propositional weighted formula* ψ_q *taking the form* $a \leftrightarrow on \wedge (b \vee e)$ *with weights* $w = \{b \mapsto 0.6, \neg b \mapsto 0.4, \ldots, a \mapsto 1, \neg a \mapsto 0\}$*. Due to* $w(\neg a) = 0$ *all models where* `alarm` *is false are ignored and only those where it is true are considered.*

The sum-product computation in Equation 1 is often referred to as the *weighted model counting* (WMC) problem – a #P-hard problem (Darwiche 2009). ProbLog follows a two step approach to perform WMC: a knowledge compilation step that is followed by an evaluation step, cf. (Chavira and Darwiche 2008). In the compilation step, ψ_q is first compiled into a data structure that then allows for poly-time query answering (evaluation).

Separating the WMC into a compilation step and an evaluation step is especially useful when the compiled structure can be re-used multiple times. The #P-hard compilation is performed in an offline step, which then allows repeated poly-time online evaluations using different weight functions. An iterative learning approach will naturally require many such evaluations.

The structures resulting from the knowledge compilation step are often referred to as decision diagrams, with binary decision diagrams (Bryant 1986) probably being the most prominent representative. Decision diagrams are trivially converted into arithmetic circuits by replacing the literals with their associated weight and replacing the $\wedge$ and $\vee$ connectives with multiplication and addition operations respectively. The transformed decision diagrams are called arithmetic circuits (Darwiche 2000).

Definition 1 *An arithmetic circuit (AC) is a rooted directed acyclic graph where the leafs are associated with a weight, and the internal nodes are associated with either a sum- or a product operation.*

An AC also has specific properties required to accurately represent the WMC problem, for more details we refer to (Darwiche and Marquis 2002) and (Kimmig, Van den Broeck, and De Raedt 2017). For now, one can consider an AC as a type of computational graph representing the computations required to answer probabilistic queries in time linear in the circuit size.

2.2 Beta Distributed Bayesian Networks

The beta distribution is a univariate distribution supported in the interval $[0, 1]$ and has two parameters α and β. A beta distributed random variable $X \sim \text{Beta}(\alpha, \beta)$ often represents the probability of the outcome of a Bernoulli experiment, for example a coin flip. Its density is given by

$$p_{\text{Beta}}(X; \alpha, \beta) = \frac{X^{\alpha}(1-X)^{\beta}}{\int z^{\alpha}(1-z)^{\beta}\,\mathrm{d}z} \tag{2}$$

When using a beta distribution to model uncertainty, the parameters α and β are used to represent the epistemic uncertainty. Lower values of α and β correspond to high uncertainty. The probability X on the other hand is an intrinsic aleatoric uncertainty in the probabilistic model. An extension of the beta distribution for multivalued discrete variables is called the Dirichlet distribution. For the sake of

brevity, we will focus on beta distributions, but all our results also apply to Dirichlet distributions.

A beta distributed Bayesian network is a directed acyclic graph in which every node represents a Boolean random variable. The probability of every variable is beta distributed with parameters that depend on the value of the variable's parents in the network. These networks are special cases of second-order Bayesian networks as discussed by Kaplan and Ivanovska (2018). The beta parameters are used to model epistemic uncertainty and the beta distributed variable itself to model the network's aleatoric uncertainty. In our experimental evaluation we will use beta distributed Bayesian networks for benchmarking inference and learning.

2.3 Distributions and Credal Sets

Another common way to model epistemic uncertainty is via credal sets. A credal set is a convex set of probability distributions over one or more random variables (Mauá et al. 2014). For a binary variable X, a credal set is specified by a subinterval S of $[0, 1]$ wherein the probability p of X being true lies. A credal set can also be viewed as the support of the distribution over p. Reasoning about credal sets of random variables is then equivalent to reasoning about the support of the distributions over the probabilities p associated with each variable.

Credal sets provide reasoning over robust boundaries, making them especially useful for robust decision making applications. However, they do not specify any other details about the probability distributions beyond the support, and can therefore not accurately model complex distribution shapes such as skewness. Furthermore, it is not possible to use them to calculate expectation values. This is in contrast to beta distributions, which naturally allow for the calculation of any function's expectation value – regardless of the distribution shape. Comparing credal sets and beta distributions is difficult, as one aims to reason over boundaries while the other aims to reason over the entire distribution. Whether one should use distributions or credal sets depends on the modeling goals and whether outlier behaviour or expected behaviour is more important. It is also possible to combine both approaches using beta distributions that are cut off outside a given interval. We can compare both methods by assuming a credal set $[a, b]$ represents the uniform distribution Uniform(a, b). By matching the mean and standard deviation of the unifrom distribution and the beta distribution, we map a credal interval $[a, b]$ to a beta distribution with mean $(a + b)/2$ and standard deviation $(b - a)/\sqrt{12}$.

3 BetaProbLog

BetaProbLog is an extension of ProbLog, a probabilistic logic language which in turn builds upon Prolog (Fierens et al. 2015). The main addition is the ability to distribute probabilities of facts according to beta distributions, allowing the user to model epistemic uncertainty in the program.

3.1 Semantics

Definition 2 (BetaProbLog) *A BetaProbLog program consists of three disjoint sets $\mathcal{F}$, $\mathcal{B}$, and $\mathcal{R}$:*

1. *a set of probabilistic facts $\mathcal{F}$. Given a probabilistic fact p_f`::f`, the atom f has probability p_f of being true in a world of the program.*
2. *a set of beta facts $\mathcal{B}$. A fact `beta(`α_f`,`β_f`)::f` defines a random variable $X_f \sim$ Beta(α_f, β_f) representing the probability of the atom f.*
3. *a set of logic rules $\mathcal{R}$ of the form `h :-` b_1`,...,`b_n. Whenever the body literals $b_1, \ldots, b_n$ are true in a world, the head atom `h` has to be true in order for the world to be a model of the program.*

Note that in the absence of beta facts ($\mathcal{B} = \emptyset$) BetaProbLog reduces to ProbLog.

Example 3 *A BetaProbLog program with two beta facts, one probabilistic fact and two clauses is shown below.*

```
1 beta(40, 160)::burglary.
2 beta(10, 90)::earthquake.
3 0.8::alarm_on.
4
5 alarm :- alarm_on, burglary.
6 alarm :- alarm_on, earthquake.
```

If we include beta facts in the program, the probability of a query atom `q` is defined using the same semantics as in Equation 1, except that the weight of a beta fact f will now be a random variable X_f, which represents the probability of that fact. Therefore, the probability of the query atom, which is now a sum-product of random variables, will itself be a random variable. We will call this random variable X_q, representing the probability of `q`. In general we will use the symbol X to denote a random variable. We define X_q as

$$X_q = \sum_{M \models \psi_q} \prod_{l \in M} w(l) \tag{3}$$

The symbol ψ_q is a propositional theory such as in Equation 1 and the weight function returns a real number $w(f) = p_f$ if f is a non-negated probabilistic fact and a random variable $w(f) = X_f$ if f is a non-negated beta fact. For negated facts we use $w(\neg f) = 1 - w(f)$. Note that the negation $\neg f$ of a beta fact f with $X_f \sim$ Beta(α_f, β_f) is distributed according to $(1 - X_f) \sim$ Beta(β_f, α_f).

Alongside a BetaProbLog program, the user specifies a second-order query. To answer this query, it is necessary to obtain the distribution over the probability X_q of the query atom. This task is analogous to distributional inference in second-order Bayesian networks as done by Kaplan and Ivanovska (2018).

Definition 3 (Second-Order Query) *For a BetaProbLog program, a second-order query $Q = (q, g)$ has two components: a query atom `q` present in the program and a measurable function $g : [0, 1] \to \mathbb{R}$ operating on the probability X_q of atom `q` being satisfied. The answer $A(Q)$ to the query Q is the expected value, denoted with $\mathbb{E}$, of $g(X_q)$ with regard to the probability distribution $p(\boldsymbol{X}) \coloneqq \prod_{i=1}^{|\mathcal{B}|} p(X_{f_i})$.*

$$A(Q) = \mathbb{E}_{p(\boldsymbol{X})}[g(X_q)] = \int g(X_q) \left(\prod_{i=1}^{|\mathcal{B}|} p(X_{f_i}) \right) \mathrm{d}\boldsymbol{X} \tag{4}$$

The differential $\mathrm{d}\boldsymbol{X}$ is shorthand for $\mathrm{d}\boldsymbol{X} = \prod_{i=1}^{|\mathcal{B}|} \mathrm{d}X_{f_i}$. Note, the dependence of X_q on X_{f_i} is given by Equation 3.

We will from now on omit the subscript in the expectation value unless stated otherwise.

Example 4 (BetaProbLog Query) *Consider the program in Example 3. To compute the probability $P(X_{alarm} < 0.3)$, we can use the second order query $Q = (alarm, g)$ with an indicator $g(X_q) = [1$ if $X_q < 0.3$ else $0]$.*

Nested Expectation Values In Equation 4 we are computing two expectations, one expectation over the probability distribution $p(\boldsymbol{X})$ and one somewhat hidden expectation X_q. To see this, we rewrite Equation 3 as an expectation:

$$X_q = \sum_{M \models \psi_q} \prod_{l \in M} w(l) = \sum_{M \in \mathcal{I}(\mathbb{P})} \mathbb{1}_{M \models \psi_q} \underbrace{\prod_{l \in M} w(l)}_{=: p(M)} = \mathbb{E}_{p(M)}[\mathbb{1}_{M \models \psi_q}] \tag{5}$$

In the equation above $\mathbb{1}$ is an indicator, $\mathcal{I}(\mathbb{P})$ denotes all the possible models of the program $\mathbb{P}$ and $p(M) := \prod_{l \in M} w(l)$ is the probability of model M. Note that $p(M)$ and, consequently, the expectation in the last line, depend on the random variables in X_f with f a beta fact. This means that Equation 4 effectively defines a nested query, i.e. an expectation value of an expectation value. We can regard BetaProbLog as a specific case of nested probabilistic programs (Mantadelis and Janssens 2011; Rainforth 2018).

Conditional Queries BetaProbLog also supports conditional second-order queries $Q{=}(q, e, g)$, where q is a queried atom and e the evidence. Note that we discuss evidence over atoms in the program and not on beta distributed probabilities. To clarify, e can be evidence of `burglary` being true, but not evidence of `burglary` taking a specific probability p. A conditional second-order query is defined as:

$$A(Q) = \mathbb{E}_{p(\boldsymbol{X})}[g(X_{q\,|\,e})] = \int g(X_{q\,|\,e}) p(\boldsymbol{X}) \,\mathrm{d}\boldsymbol{X} \tag{6}$$

where $X_{q\,|\,e} = X_{q \wedge e}/X_e$ and $X_{q \wedge e}$ is a random variable representing the probability of $q \wedge e$ being true. Effectively the second-order query computes the expected probability of a conditioned probabilistic logic program with respect to the probability distribution $p(\boldsymbol{X})$.

3.2 Special Cases of BetaProbLog

Below, we list a couple of particular choices of second-order queries to illustrate the strength of BetaProbLog and its relation to other work.

Mean Values When calculating the mean value of X_q by choosing the identity for the query function $g(X_q) = X_q$, the query answer is given by

$$A(Q) = \mathbb{E}[g(X_q)] = \mathbb{E}[X_q] = \mathbb{E}\left[\sum_{M \models \psi_q} \prod_{l \in M} w(l)\right] = \sum_{M \models \psi_q} \prod_{l \in M} \mathbb{E}[w(l)] \tag{7}$$

where we have filled in Equation 3 for X_q and used the fact that an expectation value can be pushed inside a sum and a product of independent random variables in the last equality. The BetaProbLog query thus reduces to a regular ProbLog inference problem, because $\mathbb{E}[w(l)]$ is a real number in the unit interval.

Example 5 (BetaProbLog Query Evaluation) *Consider the program*

```
0.4::a.
beta(3,7)::b.
c :- a, b.
c :- \+a, \+b.
```

with the query $Q = (c, I)$ where I is the identity function. The models of this program satisfying c are $\{a, b, c\}$ and $\{\neg a, \neg b, c\}$. These have weights $0.4 \cdot X_b$ and $0.6 \cdot (1 - X_b)$ respectively. The probability X_c is by Equation 3 equal to

$$X_c = 0.4 \cdot X_b + 0.6 \cdot (1 - X_b)$$

and is thus a random variable. The query answer is its expectation value

$$\begin{aligned} A(Q) &= \mathbb{E}[g(X_c)] = \mathbb{E}[X_c] = \mathbb{E}[0.4 \cdot X_b + 0.6 \cdot (1 - X_b)] \\ &= 0.4 \cdot \mathbb{E}[X_b] + 0.6 \cdot (1 - \mathbb{E}[X_b]) \\ &= 0.4 \cdot 0.3 + 0.6 \cdot (1 - 0.3) = 0.54 \end{aligned}$$

where the expectation $\mathbb{E}[X_b] = 3/(3 + 7)$ is the mean of the beta distribution Beta$(3, 7)$. The query answer is thus obtained by replacing every beta fact by a probabilistic fact with probability given by the mean of the beta distribution.

Moments Higher moments of the distribution can be calculated with $g(X_q) = X_q^k$ for the kth moment. This subsequently allows the calculation of other statistics such as the variance or skewness of the distribution.

Subsets It is possible to calculate the probability that X_q lies in a measurable subset $S \subseteq [0, 1]$ by choosing

$$g(X_q) = \begin{cases} 1 & \text{if } X_q \in S \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

in the second-order query. In particular we can consider interval subsets $S = [a, b] \subseteq [0, 1]$. In this case, second-order inference becomes an instance of weighted model integration (Belle, Passerini, and Van den Broeck 2015).

3.3 Comparison to SLProbLog

While BetaProbLog and SLProbLog are syntactically similar, their semantics are different. SLProbLog supports both SL beliefs and beta distributions, which were, however, shown to be equivalent repreentations of epistemic uncertainty (Jøsang 2016). Furthermore, SLProbLog uses algebraic ProbLog (Kimmig, Van den Broeck, and De Raedt 2011), which requires commutative addition and multiplication operators to define consistent semantics. However, the operators used by Cerutti et al. (2019) are only approximately commutative and therefore not well suited to define the semantics. It is not guaranteed that two different ACs produce the same result, even when they encode the same theory ψ_q and use the same weight function w. Furthermore, the operators approximate sums and products of beta variables with new beta variables. This means that the query distribution is also approximated by a beta distribution, which in some cases is an unfaithful approximation.

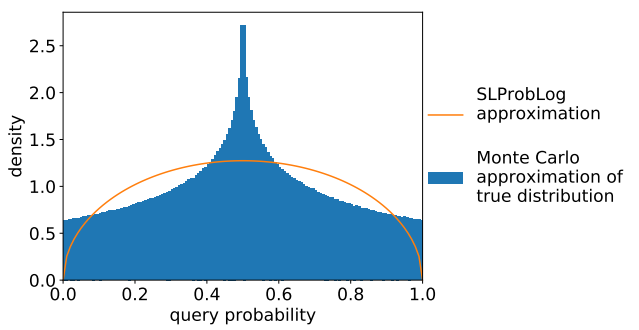

Figure 1: Mismatch between the moment matching beta density and actual density of the query probability. X_c is the actual density.

Example 6 (Beta Mismatch) *The query probability in the program below is not beta distributed, as Figure 1 shows.*

```
1  beta(0.5,0.5)::a.
2  beta(0.5,0.5)::b.
3  c :- a, not(b).
4  c :- b, not(a).
5  query(c).
```

In blue is a histogram of the true density, obtained by sampling many times and in orange is the beta density obtained by moment matching. The peak in the middle and the convexity of the density is entirely missed.

4 Inference and Learning

4.1 General Inference Approach

ProbLog performs probabilistic inference by converting a queried program into a weighted propositional theory and solving a WMC task on a compiled circuit (cf. Section 2.1). BetaProbLog also follows this compile and evaluate paradigm. A BetaProbLog program with query atom $\mathtt{q}$ is compiled into an arithmetic circuit Γ_q, a compact representation of the function

$$\Gamma_q : (X_{f_1}, ..., X_{f_{|\mathcal{B}|}}) \mapsto X_q = \sum_{M \models \psi_q} \prod_{l \in M} w(l) \quad (9)$$

The circuit Γ_q maps the probabilities of the $|\mathcal{B}|$ beta facts $X_{f_1}, \ldots, X_{f_{|\mathcal{B}|}}$ onto the probability X_q of the query in a computationally efficient fashion. Recall that $w(f) = X_f$ if f is a beta fact. Any probability of a probabilistic fact f is implicitly present in the circuit function via the weight $w(f) = p_f$. We omit them in the function arguments as they are constant. The complexity of compiling a ground probabilistic logic program to a circuit is #P-hard and depends only on the logic structure. Consequently, the compilation step is identical for ProbLog, BetaProbLog and SLProbLog.

Example 7 (Circuit) *The BetaProbLog program of Example 3 can be compiled into the arithmetic circuit of Figure 2. The circuit represents* Γ_a *as* $w(a) = 1$ *and* $w(\neg a) = 0$.

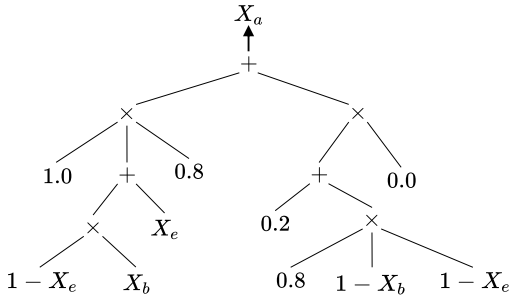

Figure 2: Arithmetic circuit for the program in Example 3.

The answer $A(Q)$ to query $Q = (q, g)$ is given by

$$A(Q) = \mathbb{E}[g(\Gamma_q(X_{f_1}, ..., X_{f_{|\mathcal{B}|}}))] \quad (10)$$
$$= \int g(\Gamma_q(X_{f_1}, ..., X_{f_{|\mathcal{B}|}})) \left(\prod_{i=1}^{|\mathcal{B}|} p(X_{f_i}; \alpha_{f_i}, \beta_{f_i}) \right) \mathrm{d}\boldsymbol{X}$$

Again we use the shorthand $\mathrm{d}\boldsymbol{X} = \prod_{i=1}^{|\mathcal{B}|} \mathrm{d}X_{f_i}$. The function $p(X_f; \alpha_f, \beta_f)$ denotes a beta density function, cf. Equation 2.

Although the complexity of calculating the circuit is always #P-hard, the complexity of answering the query given the circuit depends on the type of query. For standard ProbLog inference it suffices to propagate real numbers through the circuit and evaluation is linear in the circuit size. Under specific conditions, calculating higher order moments can be done in polynomial time as well, as shown by Khosravi et al. (2019). More general computations, referred to as weighted model integration (Belle, Passerini, and Van den Broeck 2015; Morettin et al. 2021), are known to be #P-hard even with the circuit given (Zuidberg Dos Martires, Dries, and De Raedt 2019).

4.2 Monte Carlo Evaluation

The integral in Equation 10 is hard to solve in general but can be approximated by Monte Carlo sampling. In order to obtain a sample of X_q, we can generate a sample for every random variable X_f and evaluate the circuit in these samples. Instead of sampling directly from $X_f \sim \text{Beta}(\alpha_f, \beta_f)$, however, we consider random variables $U_f \sim \text{Uniform}(0, 1)$ and apply a reparametrization function r to these variables such that $r(U_f, \alpha_f, \beta_f) \sim \text{Beta}(\alpha_f, \beta_f)$ (Kingma and Welling 2014). This allows us to calculate gradients with respect to the distribution parameters, which enables learning (cf. Section 4.3).

Let $\boldsymbol{u}$ be a set containing a sample for each variable U_f and let $\mathcal{U}$ denote a collection of distinct $\boldsymbol{u}$'s. The integral in Equation 10 can then be approximated as

$$A(Q) = \frac{1}{|\mathcal{U}|} \sum_{\boldsymbol{u} \in \mathcal{U}} g(\Gamma_q^r(\boldsymbol{u}, \boldsymbol{\theta})) \quad (11)$$

Γ_q^r denotes the reparametrized circuit and $\boldsymbol{\theta}$ the set of parameters of all beta distributions:

$$\boldsymbol{\theta} = \{\alpha_{f_1}, \beta_{f_1}, ..., \alpha_{f_{|\mathcal{B}|}}, \beta_{f_{|\mathcal{B}|}}\} \quad (12)$$

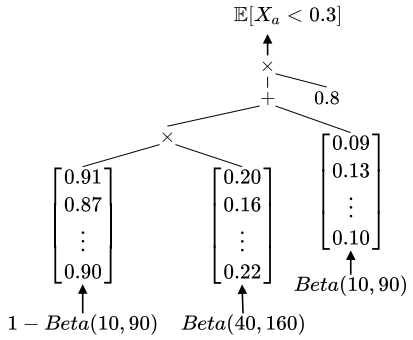

Figure 3: Example of an arithmetic circuit with samples

As the circuit structure does not change between circuit evaluations we can parallelize the evaluations for the different $\boldsymbol{u}$ by performing them on a GPU. We have implemented this idea, adapted from (Zuidberg Dos Martires, Dries, and De Raedt 2019), using PyTorch's tensor operations (Paszke et al. 2019).

Example 8 *The second-order query previously introduced in Example 4, $P(X_{alarm} < 0.3)$, can be answered by replacing the random variables in circuit Γ_a (Figure 2) with samples from the associated beta distributions, performing tensor operations throughout Γ_a, and using the resulting samples to compute $\mathbb{E}[X_{alarm} < 0.3]$. This process is depicted in Figure 3, only showing the relevant non-zero branch of Γ_a.*

4.3 A Learning Algorithm

A question that naturally arises is whether the beta parameters in the program can be learned from data. Work on learning these parameters in the domain of second-order networks has so far been limited to networks of only two nodes (Kaplan et al. 2020), while SLProbLog considered it as future work. To resolve this, we propose a parameter learning approach for BetaProbLog that can be applied to various learning settings involving model uncertainty. It can be used for probability estimation, discriminative classification, distribution matching and cost optimization.

BetaProbLog performs parameter learning by minimizing a loss function. The total loss $\mathcal{L}$ is a sum of loss functions L applied to a second-order query, which is compared to a target value. More concretely, given a BetaProbLog program with a set of probabilistic or distributional parameters $\boldsymbol{\theta}$, a dataset $\mathcal{D}$ of tuples (d, t) with d a partially observed model of the program and t a target value of $A(Q_d, \boldsymbol{\theta})$, the total loss is:

$$\mathcal{L}(\boldsymbol{\theta}) = \textstyle\sum_{(d,t)\in\mathcal{D}} L(A(Q_d, \boldsymbol{\theta}), t) \tag{13}$$

where we explicitly write the dependency of $A(\cdot)$ on $\boldsymbol{\theta}$, unlike in definition 3, as we would like to learn these parameters.

The loss L can take on any form. Depending on the learning setting, some useful choices are

- negative log likelihood loss $L = -\log(A(Q_d))$ with $Q_d = (d, I)$ and I the identity function, for learning the most probable underlying parameters,
- mean absolute error or squared error $L = (A(Q_d) - t)^2$, for training classifiers,
- cross entropy $L = t\log(A(Q_d)) + (1-t)\log(1-A(Q_d))$, for distribution matching

BetaProbLog optimizes the loss function in Equation 13 using stochastic gradient descent. When g is differentiable, the gradients can be obtained by taking the derivative of Equation 11 and using automatic differentiation methods available in PyTorch (Paszke et al. 2019). This approach is similar to the learning setting found in the work of Gutmann et al. (2008) and DeepProbLog (Manhaeve et al. 2018). The difference is that BetaProbLog goes beyond mere point probabilities and allows for expressing distributions over probabilities. This requires the reparametrization trick.

During learning, the beta parameters can become arbitrarily large. To avoid this issue, we impose the number of data points $|\mathcal{D}|$ as an upper bound on the beta parameters by applying a normalization after every iteration:

$$(\alpha_f, \beta_f) \rightarrow \left(\tfrac{\alpha_f|\mathcal{D}|}{\alpha_f+\beta_f}, \tfrac{\beta_f|\mathcal{D}|}{\alpha_f+\beta_f}\right) \tag{14}$$

Having the number of data points as an upper bound mirrors the same property in Baeysian learning, which arises naturally in the latter. Our learning method differs from Bayesian learning in that it can use any loss function and as such can be used for discriminative learning. For example, it can *minimize* the probability of some undesired data points, whereas Bayesian learning returns a distribution over the parameters that most likely generated that data. With missing data, BetaProbLog also has the benefit that during learning the beta facts $\mathcal{B}$ remain beta distributed, in contrast to Bayesian learning.

5 Experiments

Our experiments involve three Bayesian networks proposed in (Kaplan and Ivanovska 2018) and five larger networks from the BNLearn repository[1]. Each network is transformed into a ProbLog program using a script available from the ProbLog code repository. Our implementation can be found on GitHub[2].

5.1 Inference Experiments

Experimental Setup We construct two models for each ProbLog program, one with epistemic uncertainty and one without. The model M_T without epistemic uncertainty is a ProbLog program that represents the true model. In contrast, the model M_E with epistemic uncertainty is a BetaProbLog program that models the uncertainty of the probability parameters, and represents a user constructed or learned model. For our experiments, M_T is generated by replacing each probability in the program with a probability randomly sampled from the unit uniform distribution. The model M_E is obtained from M_T by replacing each probability parameter p with a beta distribution. The parameter α of the beta distribution is sampled from $\alpha \sim \text{Binomial}(N, p)$

[1] https://www.bnlearn.com/bnrepository

[2] https://github.com/ML-KULeuven/betaproblog

Net	$\#n$	$\#p$	Citation
A1	9	17	Kaplan and Ivanovska (2018)
A2	9	19	Kaplan and Ivanovska (2018)
A3	9	21	Kaplan and Ivanovska (2018)
Child	20	230	Spiegelhalter (1992)
Alarm	37	509	Beinlich et al. (1989)
Water	32	10083	Jensen et al. (1989)
Win95pts	76	574	BNLearn
Hepar2	70	1453	Onisko (2003)

Table 1: The number of nodes ($\#n$) and parameters ($\#p$) for each used Bayesian network.

and $\beta = N - \alpha$, with N an experimental parameter controlling the number of data points.

A description of the networks used in the experiments investigating the inference accuracy and the time scaling is given in Table 1. The time scaling experiments were run on Intel Xeon CPU with 2.20GHz. BetaProbLog additionally took advantage of a GeForce GTX 1080 Ti GPU.

Question 1: Can BetaProbLog capture the uncertainty with respect to the true model? We follow the experiment of Kaplan and Ivanovska (2018) and Cerutti et al. (2019) and analyse how well our expression of uncertainty for a specific query q captures the spread between the inferred probability and the actual probability. In this experiment, we infer the probability distribution of X_q from the uncertain model M_E, and the true probability $P(q)$ from the true model M_T. After the distribution of X_q is inferred, we calculate the root mean squared error of X_q and the underlying probability $P(q)$ (TE). Furthermore, we calculate the predicted error (PE) as the standard deviation $\sigma[X_q]$ and verify whether it is representative of the true error (TE). These results are averaged out over 100 iterations and are tabulated in Table 2 for various values of N. We compare against Cerutti et al. (2019) as they exhibited similar performance to state-of-the-art methods for reasoning under uncertainty: subjective Bayesian networks (Ivanovska et al. 2015; Kaplan and Ivanovska 2016, 2018), credal networks (Fagiuoli and Zaffalon 1998) and belief networks (Smets 1993).

The predicted error (PE) tends to underestimate the true error (TE) but it does decrease along with the true error when N is increased and therefore is indicative of the true error. We can see that increasing the sample count does not have a huge impact on the error, and that 10 inference samples often suffice. The true error for BetaProbLog is smaller than for SLProbLog in almost all cases, indicating it can more accurately predict the underlying probability.

Question 2: How well does the computation time scale? We investigate the compilation and evaluation time for inference in BetaProbLog and SLProbLog, and average out the results over 5 runs. We include larger Bayesian networks to illustrate that our approach is not only viable for small problems. The results shown in Table 3 illustrate that due to the parallelization, the evaluation time remains constant with an increasing number of samples. It also shows that our approach outperforms SLProbLog in terms of run time

Net	N		BP10	BP100	BP1000	SLP
A1	10	TE	0.156	0.154	**0.144**	0.152
		PE	0.099	0.106	0.103	0.186
	50	TE	0.065	**0.064**	**0.064**	0.076
		PE	0.046	0.048	0.048	0.107
	100	TE	0.051	**0.045**	0.048	0.057
		PE	0.034	0.035	0.035	0.078
A2	10	TE	**0.134**	0.142	0.139	0.138
		PE	0.096	0.100	0.099	0.178
	50	TE	**0.058**	**0.058**	0.059	0.070
		PE	0.042	0.044	0.045	0.098
	100	TE	0.043	**0.042**	0.043	0.055
		PE	0.029	0.033	0.032	0.073
A3	10	TE	0.134	0.127	**0.126**	0.150
		PE	0.092	0.095	0.093	0.172
	50	TE	0.056	0.053	**0.050**	0.073
		PE	0.039	0.039	0.039	0.090
	100	TE	0.039	**0.037**	0.038	0.055
		PE	0.029	0.027	0.028	0.066

Table 2: For networks A1, A2 and A3 the true error (TE) and predicted error (PE) between X_q and the underlying probability $P(q)$ are shown. The results for BetaProbLog (BP) with 10, 100 and 1000 inference samples are shown, as well as the results of SLProbLog (SLP). Different values for the number of data points N were used. The smallest values are indicated in bold.

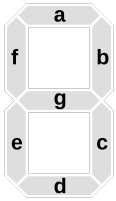

Figure 4: Seven-segment digit display.

– even with 10000 inference samples our method is an order of magnitude faster. The method can also scale up to larger networks with more than 1000 parameters.

Question 3: How does our method compare to credal inference? In (Mattei et al. 2020) an experiment is outlined where digits are displayed on an error-prone seven-segment display, visually represented in Figure 4. A digit is sampled uniformly at random to be displayed, but every required segment might fail to turn on independently with a probability following a beta distribution with sample size $N = \alpha + \beta$ and mean $\mu = \alpha/N$. The goal is to predict which segments should be on, given an observed configuration of segments. We have used BetaProbLog, SLProbLog and ProbLog to classify segments in order to compare to the credal approach from (Mattei et al. 2020). A segment is classified as 'on' if the mean probability of being 'on' given the observed segments is larger than $1/2$. Both the accuracy and

Net		BP				SLP
		10^1	10^2	10^3	10^4	
A1	C	0.045	0.031	0.027	0.029	0.053
	E	0.002	0.002	0.002	0.002	0.697
A2	C	0.033	0.027	0.030	0.030	0.068
	E	0.002	0.002	0.002	0.003	0.851
A3	C	0.031	0.030	0.032	0.030	0.063
	E	0.002	0.002	0.002	0.002	1.025
Child	C	0.201	0.210	0.210	0.202	—
	E	0.017	0.019	0.017	0.019	
Alarm	C	0.475	0.468	0.475	0.480	—
	E	0.040	0.046	0.040	0.054	
Water	C	9.869	9.817	10.202	9.945	—
	E	0.730	0.872	0.824	0.804	
Win95pts	C	0.796	0.618	0.638	0.614	—
	E	0.035	0.031	0.034	0.031	
Hepar2	C	1.422	1.455	1.456	1.438	—
	E	0.141	0.116	0.123	0.115	

Table 3: Time (in seconds) taken by BetaProbLog (BP) to compile (C) and evaluate (E) circuits for inference in various networks with 10^1, 10^2, 10^3 or 10^4 samples. Networks with multi-valued variables are not supported by SLProbLog (SLP) and are denoted with '—'.

the u_{80} score (Zaffalon, Corani, and Mauá 2012), were used as a metric and are plotted in Figure 5 for various values of μ and N.

The u_{80} score is a so-called utility-discounted accuracy metric (Zaffalon, Corani, and Mauá 2012), which are used in the credal set literature to compare the precision of imprecise probability models (to precise models) as an alternative to F-metrics (del Coz, Díez, and Bahamonde 2009). The u_{80} score is a specific instance of this utility-discounted metric. The score requires computing the so-called discounted accuracy, which is the expected value of the accuracy taken over the outputted labels. For example, a credal classifier that outputs a set of n labels has a discounted accuracy of $1/n$ if the correct label is in the set and 0 if it is not. Then a quadratic map Q is applied to the discounted accuracy such that $Q(0) = 0$, $Q(1) = 1$ and $Q(0.5) = 0.8$, with this last requirement giving the score its name. In symbols,

$$u_{80} = Q(a) = -0.6a^2 + 1.6a \tag{15}$$

with a the discounted accuracy. The u_{80} score is the result of this quadratic map applied to the discounted accuracy, averaged over all the data points. A more detailed exposition can be found in (Zaffalon, Corani, and Mauá 2012).

BetaProbLog can be seen to outperform both SLProbLog and credal inference for both metrics in Figure 5. Notice that the accuracy of BetaProbLog and standard ProbLog overlap. This experimentally confirms that when taking mean values BetaProbLog reduces to regular ProbLog. For SLProbLog the u_{80} score is lower than the accuracy, indicating a large part of the distribution lies on the wrong side of $1/2$ for a correct classification. This illustrates the drawbacks of the approximations SLProbLog makes during inference.

5.2 Learning From Target Probabilities

Experimental Setup We consider the three networks used by Cerutti et al. (2019), convert them to BetaProbLog programs, and use them to illustrate the parameter learning algorithm for BetaProbLog. The aim of the learning task in this experiment is to tune the beta parameters of the network in order to match the probability of a set of selected nodes to a target probability. This learning situation can occur when combining the results of multiple sources into one network and trying to find the parameters of the network that conform to the sources. For example, given the simple network $smoking \rightarrow cancer$, and the evidence $P(smoking) = 0.3$, $P(cancer) = 0.01$ and $P(smoking \mid cancer) = 0.4$, we can easily learn the other probabilities, such as $P(cancer \mid \neg smoking)$, by imposing the above three probabilities in the network. Note that the given probabilities are not the network parameters and that discriminative learning with these target probabilities does not fit the standard Bayesian learning framework.

To generate a dataset $\mathcal{D}$ for our problem, we first create a ProbLog program by uniformally sampling point probabilities for all facts in the program. After that, we randomly decide on an atom q, and determine a target probability t by inferring the probability of q from the program. We repeat this N_q times to form $\mathcal{D} = \{(q_i, t_i) \mid i = 1, ..., N_q\}$. After the dataset is generated, we randomly select N_p probabilistic facts and omit their probability for them to be learned.

Question 4: How well can BetaProbLog recover parameters based on the number of given target probabilities and the number of parameters to be learned? To learn the N_p missing parameters from N_q target probabilities, we utilise BetaProbLog's parameter learning approach to minimize the squared error between the inferred probability of q and the target probability t with the loss function

$$L(A(Q, \boldsymbol{\theta}), t) = (A(Q, \boldsymbol{\theta}) - t)^2 \tag{16}$$

where $Q = (q, I)$ and I the identity function. During learning we use the Adam optimizer, a sample count of 1000 and allow for 100 epochs. We evaluate the learning by repeating the same experiment 10 times for each N_q and N_p, averaging out the absolute error between the parameters omitted and the parameters learned (Table 4). Provided with enough data for the number of parameters to be learned, BetaProbLog can accurately recover the parameters. Even with less data, the method is quite robust.

6 Conclusion

Modeling epistemic uncertainty allows for reasoning even when the underlying probabilistic model is not exactly known. We extended the PLP language ProbLog with sound semantics to support epistemic uncertainty through beta distributions. These semantics are implemented in the language BetaProbLog for which we also provide an inference and learning algorithm. The inference method achieves state-of-the-art performance for second-order Bayesian networks

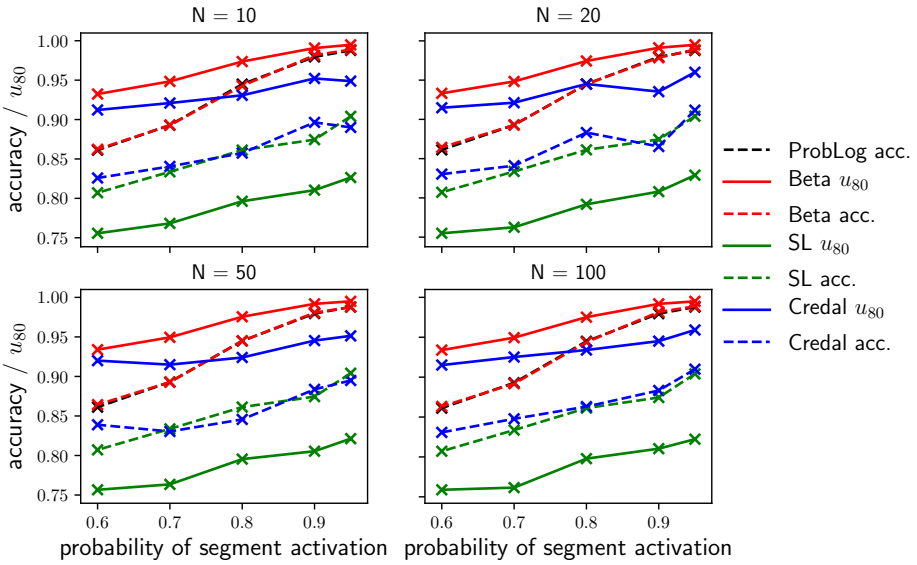

Figure 5: Accuracy and u_{80} for classifying segments with $N \in \{10, 20, 50, 100\}$.

Net			N_q		
			9	6	3
A1	N_p	4	0.030	0.008	0.110
		8	0.036	0.098	0.149
		12	0.102	0.123	0.173
A2	N_p	4	0.005	0.066	0.107
		8	0.056	0.097	0.127
		12	0.088	0.131	0.152
A3	N_p	4	0.004	0.022	0.077
		8	0.070	0.090	0.148
		12	0.085	0.124	0.160

Table 4: Mean absolute error for the target probability learning experiment with different values of N_p and N_q.

and the learning is more scalable than previous epistemic uncertainty learning methods. In future work, we would like to integrate BetaProbLog with DeepProbLog (Manhaeve et al. 2018), a ProbLog extension that supports neural networks, in order to reason over the uncertainty present in them. Furthermore, we also envisage to apply this work to a utility learning problem in the context of probabilistic routing networks (Derkinderen and De Raedt 2020).

Acknowledgements

VV received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme. VD is supported by an SB PhD fellowship at the Research Foundation – Flanders [1SA5520N]. PZD has received support from the KU Leuven Special Research Fund. LDR is also funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [694980] SYNTH: Synthesising Inductive Data Models) and the Wallenberg AI, Autonomous Systems and Software Program. The authors thank Angelika Kimmig for insightful discussions and feedback.

References

Beinlich, I. A.; Suermondt, H. J.; Chavez, R. M.; and Cooper, G. F. 1989. The ALARM Monitoring System: A Case Study with two Probabilistic Inference Techniques for Belief Networks. In *AIME 89*, 247–256. Berlin, Heidelberg: Springer Berlin Heidelberg.

Belle, V.; Passerini, A.; and Van den Broeck, G. 2015. Probabilistic Inference in Hybrid Domains by Weighted Model Integration. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8): 677–691.

Cerutti, F.; Kaplan, L.; Kimmig, A.; and Şensoy, M. 2019. Probabilistic logic programming with beta-distributed random variables. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7): 772–799.

Darwiche, A. 2000. A Differential Approach to Inference in Bayesian Networks. In *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence (UAI)*, 123–132. Morgan Kaufmann.

Darwiche, A. 2009. *Modeling and reasoning with Bayesian networks*. Cambridge university press.

Darwiche, A.; and Marquis, P. 2002. A Knowledge Compilation Map. *Journal Artificial Intellence Research*, 17: 229–264.

del Coz, J. J.; Díez, J.; and Bahamonde, A. 2009. Learning Nondeterministic Classifiers. *J. Mach. Learn. Res.*, 10: 2273–2293.

Derkinderen, V.; and De Raedt, L. 2020. Algebraic Circuits for Decision Theoretic Inference and Learning. In *Proceedings of the 24th European Conference on Artificial Intelligence*.

Fagiuoli, E.; and Zaffalon, M. 1998. 2U: An Exact Interval Propagation Algorithm for Polytrees with Binary Variables. *Artificial Intelligence*, 106(1): 77–107.

Fierens, D.; Van den Broeck, G.; Renkens, J.; Shterionov, D.; Gutmann, B.; Thon, I.; Janssens, G.; and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15: 358–401.

Gutmann, B.; Kimmig, A.; Kersting, K.; and De Raedt, L. 2008. Parameter learning in probabilistic databases: A least squares approach. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 473–488. Springer.

Hüllermeier, E.; and Waegeman, W. 2021. Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods. *Mach. Learn.*, 110(3): 457–506.

Ivanovska, M.; Jøsang, A.; Kaplan, L.; and Sambo, F. 2015. Subjective Networks: Perspectives and Challenges. In *Graph Structures for Knowledge Representation and Reasoning*, 107–124.

Jensen, F. V.; Kjærulff, U.; Olesen, K. G.; and Pedersen, J. 1989. An expert system for control of waste water treatment—a pilot project. Technical report, Judex Datasystemer A/S, Aalborg, 1989. In Danish.

Jøsang, A. 2016. *Subjective Logic: A Formalism for Reasoning Under Uncertainty*. Springer.

Kaplan, L.; Cerutti, F.; Şensoy, M.; and Mishra, K. V. 2020. Second-Order Learning and Inference using Incomplete Data for Uncertain Bayesian Networks: A Two Node Example. In *23rd International Conference on Information Fusion (FUSION)*, 1–8. IEEE.

Kaplan, L.; and Ivanovska, M. 2016. Efficient subjective Bayesian network belief propagation for trees. In *19th International Conference on Information Fusion (FUSION)*, 1300–1307. IEEE.

Kaplan, L.; and Ivanovska, M. 2018. Efficient belief propagation in second-order Bayesian networks for singly-connected graphs. *International Journal of Approximate Reasoning*, 93: 132–152.

Khosravi, P.; Choi, Y.; Liang, Y.; Vergari, A.; and den Broeck, G. V. 2019. On Tractable Computation of Expected Predictions. In *NeurIPS*, 11167–11178.

Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2011. An algebraic Prolog for reasoning about possible worlds. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2017. Algebraic model counting. *Journal of Applied Logic*, 22: 46–62.

Kingma, D. P.; and Welling, M. 2014. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations (ICLR)*.

Levi, I. 1983. *The enterprise of knowledge: An essay on knowledge, credal probability, and chance*. MIT press.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. DeepProbLog: Neural Probabilistic Logic Programming. In *NeurIPS*, 3753–3763.

Mantadelis, T.; and Janssens, G. 2011. Nesting probabilistic inference. In *Proceedings of the Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS)*, 1–16.

Mattei, L.; Antonucci, A.; Mauá, D. D.; Facchini, A.; and Villanueva Llerena, J. 2020. Tractable inference in credal sentential decision diagrams. *International Journal of Approximate Reasoning*, 125: 26–48.

Mauá, D. D.; De Campos, C. P.; Benavoli, A.; and Antonucci, A. 2014. Probabilistic Inference in Credal Networks: New Complexity Results. *J. Artif. Int. Res.*, 50(1): 603–637.

Morettin, P.; Zuidberg Dos Martires, P.; Kolb, S.; and Passerini, A. 2021. Hybrid probabilistic inference with logical and algebraic constraints: a survey. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*.

Onisko, A. 2003. Probabilistic causal models in medicine: Application to diagnosis of liver disorders. In *Ph. D. dissertation, Inst. Biocybern. Biomed. Eng., Polish Academy Sci., Warsaw, Poland*.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Köpf, A.; Yang, E. Z.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 8024–8035.

Rainforth, T. 2018. Nesting Probabilistic Programs. In *Proceedings of the 34th Conference on Uncertainty in Artificial Intelligence (UAI)*.

Sato, T. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *In Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*, 715–729. MIT Press.

Smets, P. 1993. Belief functions: the disjunctive rule of combination and the generalized Bayesian theorem. *International Journal of approximate reasoning*, 9(1): 1–35.

Spiegelhalter, D. J. 1992. Learning in probabilistic expert systems. *Bayesian statistics*, 4: 447–466.

Zaffalon, M.; Corani, G.; and Mauá, D. 2012. Evaluating credal classifiers by utility-discounted predictive accuracy. *International Journal of Approximate Reasoning*, 53(8): 1282–1301.

Zuidberg Dos Martires, P.; Dries, A.; and De Raedt, L. 2019. Exact and approximate weighted model integration with probability density functions using knowledge compilation. In *Proceedings of the AAAI Conference on Artificial Intelligence*.