Learning the Parameters of Probabilistic Answer Set Programs

Damiano Azzolini[1], Elena Bellodi[2], and Fabrizio Riguzzi[1]

[1] Dipartimento di Scienze dell'Ambiente e della Prevenzione – Università di Ferrara
damiano.azzolini@unife.it
[2] Dipartimento di Ingegneria – Università di Ferrara
elena.bellodi@unfe.it
[3] Dipartimento di Matematica e Informatica – Università di Ferrara
fabrizio.riguzzi@unife.it

Abstract. Probabilistic Answer Set Programming (PASP) is a powerful formalism that allows to model uncertain scenarios with answer set programs. One of the possible semantics for PASP is the credal semantics, where a query is associated with a probability interval rather than a sharp probability value. In this paper, we extend the learning from interpretations task, usually considered for Probabilistic Logic Programming, to PASP: the goal is, given a set of (partial) interpretations, to learn the parameters of a PASP program such that the product of the lower bounds of the probability intervals of the interpretations is maximized. Experimental results show that the execution time of the algorithm is heavily dependent on the number of parameters rather than on the number of interpretations.

Keywords: Probabilistic Answer Set Programming · Parameter Learning · Statistical Relational Artificial Intelligence.

1 Introduction

Probabilistic Answer Set Programming (PASP) [8,20] extends the capabilities of Answer Set Programming (ASP) [5] by introducing uncertain data representable with probabilistic facts. In traditional semantics for Probabilistic Logic Programming (PLP) [23], such as the Distribution Semantics [25], usually each world has a total well-founded model and thus a single answer set [29]. In the case of PASP, it is not guaranteed that every world has a unique answer set. Thus, there are two layers that must be considered: the worlds, identified by the choices made for the probabilistic facts, and the answer sets for every world. To handle this, we adopt here the credal semantics [6], that associates every query with a lower and upper probability bound, according to the number of models where the query is satisfied for a given world.

Parameter learning is a central topic in the field of Statistical Relational Artificial Intelligence (StarAI) [22]. Given a set of examples, the goal is to find a probability assignment to probabilistic facts such that the likelihood of the

examples is maximized. There are two main types of parameter learning tasks: learning from interpretations and learning from entailment. Both have been extensively studied in the context of PLP [4,16,26] but none of these algorithms works for PASP under the credal semantics.

In this paper, we adapt the learning from interpretations setting described in [16] and propose the first algorithm to perform parameter learning (learning the probabilities of probabilistic facts) in PASP: given a set of interpretations and a PASP program with unknown parameters, the goal is to set the values of the parameters such that the products of the lower probabilities of the interpretations is maximized.

The paper is structured as follows: in Section 2 we review some background concepts, in Section 3 we introduce our algorithm for parameter learning, that is tested in Section 4. Section 5 discusses some related work and Section 6 concludes the paper.

2 Background

We assume the reader is familiar with the basic concepts of Logic Programming, such as atoms, literal, clauses, etc. For a book on the topic see [19].

2.1 Answer Set Programming

In addition to the basic elements of Logic Programming, ASP also allows the definition of aggregate atoms [1] with the syntax $g_0 \circ_0 \#f\{e_1; \ldots; e_n\} \circ_1 g_1$ where g_0 and g_1 can be either constants or variables, f is an aggregate function symbol, and $\circ_0$ and $\circ_1$ are arithmetic comparison operators (that may be omitted). Every e_i has the structure $t_1, \ldots, t_l : E$ where E is a conjunction of literals and every t_i is a term with variables appearing in E. There are several aggregate function symbols, such as *count* or *sum*.

A *disjunctive rule*, or simply *rule* for short, is a rule with multiple heads, i.e.,

$$\underbrace{h_1; \ldots; h_m}_{\text{head}} \leftarrow \underbrace{b_1, \ldots, b_n}_{\text{body}}.$$

where each h_i is an atom and each b_i is a literal. While describing actual code, $\leftarrow$ is usually replaced with `:-`. The meaning is: "if all the literals in the body are true, then one of the heads h_i is true". We consider here only rules where variables in the head also appear in a positive literal in the body: these are called *safe* rules. An *integrity constraint* is a rule with no atoms in the head, a *fact* is a rule with only one atom in the head and no literals in the body. An answer set program is a set of rules. A rule is called *ground* when it does not contain variables. If a rule is not ground, we can ground it through a process called *grounding* that consists in replacing all the variables with constants in the program in all possible ways.

To illustrate the semantics of answer set programs, we need to introduce some additional concepts. The Herbrand base (B_P) of an answer set program

P is the set of all ground atoms that can be constructed using the symbols in the program. An *interpretation* I for P is such that $I \subseteq B_P$ and it satisfies a ground rule if at least one head atom is true in it when every literal in the body is true in it. An interpretation that satisfies all the groundings of all the rules of P is named *model*. Given a ground program P_g and an interpretation I, if we remove from P_g the rules in which a literal in the body is false in I we obtain the *reduct* [12] of P_g with respect to I. Finally, an *answer set* I for a program P is a minimal model (in terms of set inclusion) of the reduct of P_g with respect to I. With $AS(P)$ we denote the set of all the answer sets of P.

Consider the following example:

```
0{noise}1.
0{tired}1.
angry ; relaxed:- tired.
angry:- noise.
```

This program has 5 answer sets: `{}`, `{relaxed,tired}`, `{angry,tired}`, `{angry, noise}`, `{angry,noise,tired}`. The cautious consequences, i.e., the set of atoms that are present in every answer set, form an empty set, and the solutions projected [14] on the `noise/0` and `tired/0` atoms are `{}`, `{noise}`, `{tired}` and `{noise,tired}`.

2.2 Probabilistic Logic Programming

Following the ProbLog [11] syntax, Probabilistic Logic Programming [10,23] allows the definition of *probabilistic facts* of the form

$$\Pi :: f$$

where f is an atom and $\Pi \in]0, 1]$ is its probability. Intuitively, f is true with probability Π and it is false with probability $1 - \Pi$. If we consider the Distribution Semantics [25], an *atomic choice* indicates whether a grounding for a probabilistic fact f, denoted with $f\theta$ is selected or not. A set of atomic choices is called *composite choice* and if it contains an atomic choice for every grounding of every probabilistic fact is called *total composite choice* or *selection*. The probability of a composite choice κ is computed as:

$$P(\kappa) = \underbrace{\prod_{(f_i,\theta,1)\in\kappa} \Pi_i}_{\text{selected}} \cdot \underbrace{\prod_{(f_i,\theta,0)\in\kappa} (1 - \Pi_i)}_{\text{not selected}} \tag{1}$$

where with $(f_i, \theta, 1)$ we indicate that the grounding θ of f_i is selected and with $(f_i, \theta, 0)$ that it is not. We consider here only consistent sets of atomic choices, i.e., if $(f_i, \theta, 0) \in \kappa$ then $(f_i, \theta, 1) \notin \kappa$ and vice versa. A selection identifies a logic program called *world* composed by the rules of the program and the selected probabilistic facts. The probability of a world, $P(w)$, is given by the probability of the correspondent selection. The probabilities of all the worlds sum up to 1.

A query is a conjunction of ground literals, and its probability can be computed with the formula:

$$P(q) = \sum_{w \models q} P(w) \tag{2}$$

For example, if we consider the program

```
0.2::noise.
0.6::tired.
angry:- noise.
angry:- tired.
```

there are 2 probabilistic facts, `noise` and `tired` that identify $2^2 = 4$ worlds: w_1 where both probabilistic facts are false, with $P(w) = (1-0.2) \cdot (1-0.6) = 0.32$, w_2 where `noise` is true and `tired` is false, with $P(w_2) = 0.2 \cdot (1-0.6) = 0.08$, w_3 where `noise` is false and `tired` is true, with $P(w_3) = (1-0.2) \cdot 0.6 = 0.48$, and w_4 where both `noise` and `tired` are true, with $P(w_4) = 0.2 \cdot 0.6 = 0.12$. Note that $P(w_1)+P(w_2)+P(w_3)+P(w_4) = 0.32+0.08+0.48+0.12 = 1$. If we are interested in the probability of the query $q =$ `angry`, $P(q) = P(w_2) + P(w_3) + P(w_4) = 0.08 + 0.48 + 0.12 = 0.68$. For this example, every world has a unique answer set. However, if we consider an ASP program extended with probabilistic facts, this usually does not hold, and other semantics must be adopted.

2.3 Credal Semantics

As previously discussed, with the Distribution Semantics we can manage only programs with a unique answer set for every world. If this does not hold, we need to consider an alternative semantics, such as the credal semantics [6]. Under this semantics, a query q has a lower probability $\underline{\mathrm{P}}(q)$ and an upper probability $\overline{\mathrm{P}}(q)$, and thus is associated with a probability interval. A world contributes to the lower probability if the query is true in every answer set and contributes to the upper probability if the query is true in at least one answer set. Thus $\overline{\mathrm{P}}(q) \geq \underline{\mathrm{P}}(q)$. In formulas:

$$\overline{\mathrm{P}}(q) = \underbrace{\sum_{w_i | \exists m \in AS(w_i),\ m \models q}}_{\text{query true in at least one answer set}} P(w_i) \tag{3}$$

$$\underline{\mathrm{P}}(q) = \underbrace{\sum_{w_i | |AS(w_i)|>0 \wedge \forall m \in AS(w_i),\ m \models q}}_{\text{query true in every answer set}} P(w_i) \tag{4}$$

Note that the credal semantics [6] is defined only for programs where all the worlds have at least one answer set. In fact, if a world w had no answer sets, $P(w)$ would neither contribute to the probability of the query nor to the probability of the negation of the query. If this happened, $\underline{\mathrm{P}}(q) + \overline{\mathrm{P}}(\neg q) < 1$, and there would be a loss of probability mass (with $\neg q$ we indicate the query *not* q, adopting negation as failure).

Example 1 (Smokers). Consider the following example, adapted from [16].

```
smokes(Y) ; not_smokes(Y):- smokes(X), friend(X,Y).

:- #count{Y,X:smokes(X),friend(X,Y)} = F,
   #count{Y,X:smokes(X),friend(X,Y),smokes(Y)} = SF,
   10*SF < 4*F.

smokes(a).
smokes(c).
smokes(e).
```

The first disjunctive rule states that a person can either smoke or not smoke if he/she is a friend with a person that smokes. The constraint imposes that at least in the 40% of the pairs person-friend the friend smokes. Finally, we know for certain that `a`, `c`, and `e` smoke. Suppose that there are the following 5 probabilistic facts

```
0.5::friend(a,b).
0.5::friend(b,c).
0.5::friend(c,e).
0.5::friend(b,d).
0.5::friend(d,e).
```

With these values, the probability of the query `smokes(b)` lies in the range $[0.25, 0.5]$.

Also conditional queries are described by a lower and an upper probability bound: the upper conditional probability for a query q given evidence e is given by [7]

$$\overline{\mathrm{P}}(q \mid e) = \frac{\overline{\mathrm{P}}(q,e)}{\overline{\mathrm{P}}(q,e) + \underline{\mathrm{P}}(\neg q,e)} \tag{5}$$

If $\overline{\mathrm{P}}(q,e) + \underline{\mathrm{P}}(\neg q,e) = 0$ and $\overline{\mathrm{P}}(\neg q,e) > 0$, $\overline{\mathrm{P}}(q \mid e) = 0$. This value is undefined if both $\overline{\mathrm{P}}(q,e)$ and $\overline{\mathrm{P}}(\neg q,e)$ are 0.

Similarly, the formula for the lower conditional probability is

$$\underline{\mathrm{P}}(q \mid e) = \frac{\underline{\mathrm{P}}(q,e)}{\underline{\mathrm{P}}(q,e) + \overline{\mathrm{P}}(\neg q,e)} \tag{6}$$

If $\underline{\mathrm{P}}(q,e) + \overline{\mathrm{P}}(\neg q,e) = 0$ and $\overline{\mathrm{P}}(q,e) > 0$, $\underline{\mathrm{P}}(q \mid e) = 1$. As before, this value is undefined if both $\overline{\mathrm{P}}(q,e)$ and $\overline{\mathrm{P}}(\neg q,e)$ are 0.

3 Parameter Learning in PASP

To learn the parameters of PASP programs, we adapt the learning from interpretations settings described in [16].

A partial interpretation $I = \langle I^+, I^- \rangle$ is such that I^+ is the set of atoms that should be considered true and I^- is the set of atoms that should be considered false. A partial interpretation specifies the truth value of only some atoms. In [16], the authors define the probability of an interpretation I, $P(I)$, as the probability of the query $q_I = \bigwedge_{i^+ \in I^+} i^+ \bigwedge_{i^- \in I^-} \textit{not } i^-$. We call q_I *interpretation query*.

Here we consider probabilistic answer set programs under the credal semantics so the probability of an interpretation is associated with a probability interval rather than a sharp probability value, since a world may have more than one answer set. Given a PASP program $\mathcal{P}(\Pi)$, where Π is the set of parameters that can be tuned, the lower probability of an interpretation I, $\underline{\mathrm{P}}(I \mid \mathcal{P}(\Pi))$ is the sum of the probabilities of the worlds $w \in \mathcal{P}(\Pi)$ where all the literals of the interpretation query q_I are true (i.e., the interpretation query is true) in *all* the answer sets of w. Similarly, the upper probability of an interpretation I, $\overline{\mathrm{P}}(I \mid \mathcal{P}(\Pi))$, is the sum of the probabilities of the worlds $w \in \mathcal{P}(\Pi)$ where the interpretation query q_I is true in *at least one* answer set of w. In formulas:

$$\overline{\mathrm{P}}(I \mid \mathcal{P}(\Pi)) = \sum_{w \in \mathcal{P}(\Pi) \ \mid \ \exists m \in AS(w),\ m \models I} P(w) \tag{7}$$

$$\underline{\mathrm{P}}(I \mid \mathcal{P}(\Pi)) = \sum_{w \in \mathcal{P}(\Pi) \ \mid \ \forall m \in AS(w),\ m \models I} P(w) \tag{8}$$

For example, if we consider the program shown in Example 1 with the probabilities of all the probabilistic facts set to 0.5, the partial interpretation $I = \langle$`{smokes(b),smokes(c)},{smokes(d)}`$\rangle$ has lower probability 0.125 and upper probability 0.5. This is computed by asking the query `smokes(b),smokes(c), not smokes(d).`

To compute the parameters of a PASP program with partial interpretations, we adapt the algorithm presented in [16]. Here we consider a PASP program and a set of *ground* probabilistic facts of the form $\Pi :: f$ whose probabilities can be learned.

Definition 1 (Parameter Learning in probabilistic answer set programs). *Given a PASP $\mathcal{P}(\Pi)$ with a set of ground probabilistic facts f_i whose probabilities Π_i can be learned, and a set of (partial) interpretations $\mathcal{I}$, the goal is to find a probability assignment to the probabilistic facts such that the product of the lower probabilities of the partial interpretations is maximized, i.e., solve:*

$$\Pi^* = \arg\max_\Pi \underline{\mathrm{P}}(\mathcal{I} \mid \mathcal{P}(\Pi)) = \arg\max_\Pi \prod_{I \in \mathcal{I}} \underline{\mathrm{P}}(I \mid \mathcal{P}(\Pi)) \tag{9}$$

This is equivalent to maximize the sum of the log likelihood of the interpretations:

$$\Pi^* = \arg\max_\Pi \ \log(\underline{\mathrm{P}}(\mathcal{I} \mid \mathcal{P}(\Pi))) = \arg\max_\Pi \sum_{I \in \mathcal{I}} \log(\underline{\mathrm{P}}(I \mid \mathcal{P}(\Pi))) \tag{10}$$

To solve the parameter learning problem, we consider a scenario with partial observability, i.e., some atoms may be not observed, and adopt the Expectation

Maximization (EM) algorithm. The EM process consists in finding the expected value of the probabilistic facts given the interpretations and then updating the probability values accordingly. Since we consider ground probabilistic facts, in the expectation step we need to compute, for each probabilistic fact f_j,

$$\underline{\mathrm{E}}[f_{jk}] = \sum_{i=1}^{|\mathcal{I}|} \underline{\mathrm{P}}(f_{jk} \mid I_i) \tag{11}$$

where in f_{ik} $k \in \{0, 1\}$ indicates whether it is true or false. In the maximization step each parameter Π_i is updated as

$$\Pi_j = \frac{\underline{\mathrm{E}}[f_{j1}]}{\underline{\mathrm{E}}[f_{j0}] + \underline{\mathrm{E}}[f_{j1}]} = \frac{\sum_{i=1}^{|\mathcal{I}|} \underline{\mathrm{P}}(f_j = \top \mid I_i)}{\sum_{i=1}^{|\mathcal{I}|} \underline{\mathrm{P}}(f_j = \bot \mid I_i) + \underline{\mathrm{P}}(f_j = \top \mid I_i)} \tag{12}$$

The whole pipeline is described in Algorithm 1. Function LearnParametersPASP works as follows: first, we create two data structures (*LLComputations* and *CondComputations*) to store the computations made during the main loop of line 6. Both data structures, initially empty, will contain the worlds and the probabilities for the ones involved in the computation of the lower probability of the interpretations and of the probabilistic facts. The stored values depend on the probabilities of the parameters: in this way, we can call only once the ASP solver to compute the worlds, and simply update their probabilities when the parameters of the program change. Before entering the EM cycle, we first compute the log likelihood of all the examples using function ComputeLogLikelihood. Then, iteratively, as long as the log likelihood does not converge, in the expectation step (line 8) we generate the query for the current interpretation with the function GenerateInterpretationQuery and then compute the conditional probability for every probabilistic fact f_i, both negated ($\bot$, with the query *not* f_i) and not negated ($\top$, with the query f_i), given the current interpretation. The computation of the conditional probability is either performed by calling the solver (function ComputeConditionalProbability, needed only for the first iteration) or by retrieving the values from the *CondComputations* data structure. In the maximization step (line 23) we update the parameters of the probabilistic facts according to Equation 12 and update the values for the stored computations (line 26). The algorithm can be straightforwardly adapted to find the parameters that maximize the upper probability (that may not coincide).

To compute the lower (and upper) probability of a query (function ComputeLowerProbability [2]), we first translate every ground probabilistic fact $\Pi :: f$ into an ASP rule `0{f}1`. The probability is stored in a data structure. First, we check whether there are some probabilistic facts that must always be true by computing the cautious consequences of the ASP program plus the probabilistic facts converted as previously described. These facts constitute the *minimal set of probabilistic facts* (that may be empty). This is possible because every world has at least one answer set. Then, we add every element of this set to the initial program as a constraint, i.e., if an atom `a` is in this set, we add

the constraint `:- not a`. Finally, to obtain the probability interval of a query, we consider the program composed of the ASP program plus the converted probabilistic facts, the inserted constraints, and two additional rules of the form `query:- q` and `not_query:- not q`, and project the solutions on the probabilistic facts and the `query` and `not_query` atoms. We then analyse all the solutions and identify every world, evaluate the probability, and update the lower (and upper) probability bound according to the number of associated answer sets. In case of conditional queries with evidence `e` (function ComputeConditionalProbability), the process remains almost the same, but we add two additional rules, `evidence:- e` and `not_evidence:- not e` and project the solutions also on the `evidence` and `not_evidence` atoms. This process avoids the generations of all the answer sets for every world, as happens in [28], but is still exponential in the number of worlds.

4 Experiments

We ran several experiments on a computer with Intel® Xeon® E5-2630v3 running at 2.40 GHz. For all the experiments, 60% of the dataset is used as a training set and the remaining 40% is used as a test set[4]. The initial value for all the facts whose probabilities can be learned is set to 0.5 and ϵ of Algorithm 1 is set to 10^{-5}. We used clingo as ASP solver [13] and Python as programming language.

The first dataset (*smoke*) contains programs with the structure shown in Example 1. We ran the experiments by increasing both the number of smokers and the number of interpretations. The interpretations are generated by running the ASP version of the program (with the probabilistic facts converted as previously described) and extracting the `smoke/1` and `friend/2` atoms (or their negation if they are not present) from a random answer set. The number of atoms of every interpretation randomly ranges between 1 and the total number of atoms of the selected answer set. In a first experiment, we fix the number of parameters to 5 (*smoke5*) and increase the number of interpretations. Results are shown in Figure 1a, where the execution time seems to scale almost linearly by increasing the number of interpretations. We then analyse how the number of parameters influences the execution time. Results are shown in Figure 2b, where the execution time increases exponentially by increasing the number of parameters.

The second dataset (*shop*, adapted from [24]) encodes a scenario where some person can buy some products. There are 4 instances of this dataset, with 4 (*shop4*), 8 (*shop8*), 10 (*shop10*), and 12 (*shop12*) people each. There is a disjunctive rule for each person. The dataset *shop4* has the following structure:

```
shops(a). shops(b).
shops(c). shops(d).
bought(spaghetti,a) ; bought(steak,a) :- shops(a).
bought(spaghetti,b) ; bought(beans,b) :- shops(b).
bought(tomato,c) ; bought(onions,c) :- shops(c).
```

[4] Source code and datasets available at https://github.com/damianoazzolini/pasta

Algorithm 1 Function LearnParametersPASP: learning the parameters of a PASP program $\mathcal{P}(\Pi)$ from interpretations $\mathcal{I}$.

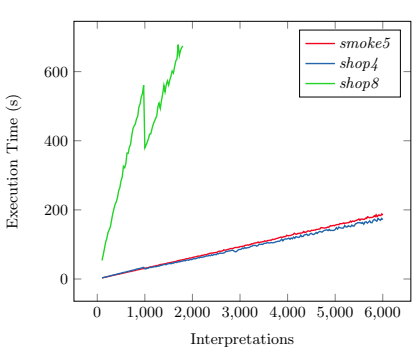
function LearnParametersPASP($\mathcal{P}(\Pi), \mathcal{I}$)
  $LL_0 \leftarrow -\infty$
  $LLComputations \leftarrow \emptyset$
  $CondComputations \leftarrow \emptyset$
  $LL_1 \leftarrow$ ComputeLogLikelihood($\mathcal{P}(\Pi), \mathcal{I}$, $LLComputations$)
  while $LL_1 - LL_0 > \epsilon$ **do**
    $LL_0 \leftarrow LL_1$
    for all $f_i \in \mathcal{P}(\Pi)$ **do** ▷ Expectation.
      $lp^{\top}_{f_i} \leftarrow 0$, $lp^{\bot}_{f_i} \leftarrow 0$
      for all $I \in \mathcal{I}$ **do**
        if $(I, f_i) \in CondComputations$ **then** ▷ Computation steps already stored.
          $lp^{\bot}_{f_i} \leftarrow lp^{\bot}_{f_i} + CondComputations[I, f_i].f$
          $lp^{\top}_{f_i} \leftarrow lp^{\top}_{f_i} + CondComputations[I, f_i].t$
        else
          $evidence \leftarrow$ GenerateInterpretationQuery(I)
          $lp^{\bot\prime}_{f_i} \leftarrow$ ComputeConditionalProbability($not\ f_i$,$evidence$)
          $lp^{\bot}_{f_i} \leftarrow lp^{\bot}_{f_i} + lp^{\bot\prime}_{f_i}$
          $lp^{\top\prime}_{f_i} \leftarrow$ ComputeConditionalProbability(f_i,$evidence$)
          $lp^{\top}_{f_i} \leftarrow lp^{\top}_{f_i} + lp^{\top\prime}_{f_i}$
        end if
      end for
    end for
    for all $f_i \in \mathcal{P}(\Pi)$ **do** ▷ Maximization.
      $\Pi_i \leftarrow \frac{lp^{\top}_{f_i}}{lp^{\top}_{f_i} + lp^{\bot}_{f_i}}$
    end for
    UpdateCondComputations($\Pi, \mathcal{I}$, $CondComputations$)
    $LL_1 \leftarrow$ ComputeLogLikelihood($\mathcal{P}(\Pi), \mathcal{I}$)
  end while
end function
function ComputeLogLikelihood($\mathcal{P}(\Pi), \mathcal{I}$, $LLComputations$)
  $LogP_{int} \leftarrow 0$
  for all $I \in \mathcal{I}$ **do**
    if $I \in LLComputations$ **then** ▷ Computation steps already stored.
      $LogP_{int} \leftarrow LogP_{int} + LLComputations[I]$
    else
      $LP \leftarrow$ ComputeLowerProbability($I \mid \mathcal{P}(\Pi)$)
      $LogP_{int} \leftarrow LogP_{int} + log(LP)$
      $LLComputations[I] \leftarrow log(LP)$
    end if
  end for
  return $LogP_{int}$
end function

```
bought(steak,d) ; bought(onions,d) :- shops(d).

cs(C):-
    #count{X : bought(spaghetti,X)} = C0,
    #count{X : bought(onions,X)} = C1,
    C = C0 + C1.
ce(C):- #count{X,Y : bought(Y,X)} = C.

:- cs(S), ce(C), 10*S < 4*C.
```

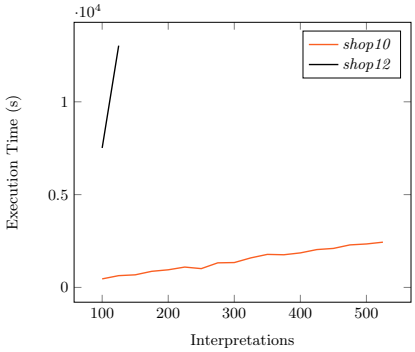
The `shops/1` facts are probabilistic, and their probabilities must be learned. The constraint states that at least 40% of the buying actions involve spaghetti or onions. The dataset *shop8* has four additional disjunctive rules with 3 possible heads, the dataset *shop10* has 2 additional rules with respect to *shop8* with 4 possible heads, the dataset *shop12* has 2 additional rules with respect to *shop10* with 5 possible heads. The generation of the interpretations follows the same pattern used for the *smoke* dataset. However, we consider only the first $n/2$ disjunctive rules during this process, where n is the size of the instance. Figure 1a and 1b show that, by increasing the number of involved people and the number of interpretations, the overhead of execution time becomes substantial. It is interesting to note that, even if *shop4* has one less parameter than *smoke5*, the computations time are comparable: this may be due to an increasing number of answer sets that must be computed for every interpretation for the *shop4* instance. For this dataset, we also tested how the log likelihood varies during the learning process by fixing the interpretations to 125. Figure 2a shows that after a few iterations the log likelihood has almost reached its maximum value. *shop8* and *shop10* seem to be faster than *shop4* and *shop12* (whose plots are almost identical, even thus *shop4* requires some more iterations to fully converge with the given ϵ). Finally, we plotted the number of iterations needed to converge for the *shop4* and *shop8* instances up to 800 interpretations. Figure 3 shows that *shop4* seems to require more iterations to converge. This is may be due to the structure of the program, since it has less parameters to tweak. Overall, the main limitation of the algorithm is that, by increasing the number of parameters and probabilistic facts, the execution time increase exponentially, since the inference task is exponential in the number of parameters. This may seem unavoidable given the expressiveness of the ASP syntax. By limiting the possible types of rules, it will be maybe possible to leverage knowledge compilation [9] to represent the program and the answer sets in more compact forms where inference can be performed faster. This is a possible direction for a future work.

5 Related Work

Most of the related work involves parameter learning in PLP. One of the first approaches to learn the parameters of a probabilistic logic program can be found in [25], where the authors propose a learning algorithm based on Expectation Maximization that starts from a set of positive and negative examples (ground

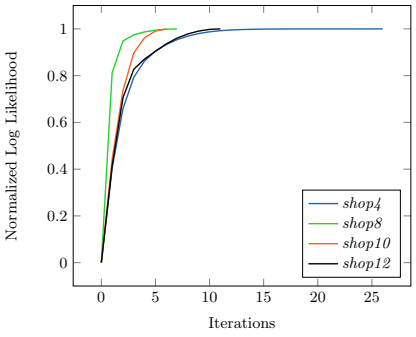


(a) Results for *smoke5*, *shop4*, and *shop8*.

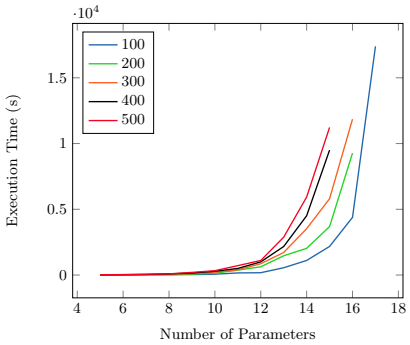


(b) Results for *shop10* and *shop12*.

Fig. 1: Variation of the execution time by increasing the number of interpretations.



(a) Variation of the (normalized) log likelihood during learning for the *shop* experiment.



(b) Variation of the execution time by increasing the number of parameters for a different number of interpretations for the *smoke* experiment.

Fig. 2: Results for the *shop* and *smoke* experiments.

atoms). This has been later extended in [26]. LeProbLog [15] is another approach to learn the parameters of a probabilistic logic program that starts from a series of examples annotated with a probability and uses gradient descent. In this setting, the goal is to learn the parameters such that the probabilities of the examples are as close as possible to the ones provided. EMBLEM [4] is another algorithm that adopts EM to learn the parameters of logic programs with annotated disjunction [30]. LFI-ProbLog [16] is one of the first algorithms to perform parameter learning given a set of interpretations, and it has been already discussed in this paper. The authors of [17] propose an algorithm to learn the parameters of an extended PRISM [25] program, i.e., a probabilistic logic program that also allows continuous random variables, and the authors of [3]

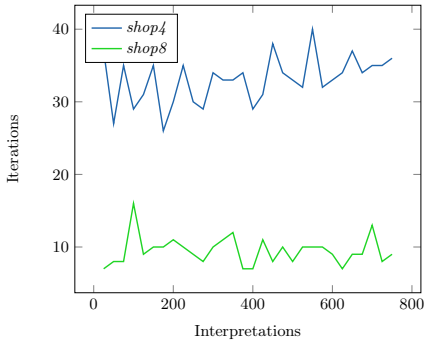


Fig. 3: Number of iterations until convergence for the *shop4* and *shop8* instances.

introduce an algorithm to learn the parameters of some facts in the programs given some constraints involving their probabilities. In [27] the authors propose SMProbLog, a framework that extends the system (and language) ProbLog [11] and the Distribution Semantics to manage worlds with multiple models. With their system, it is possible to compute the probability of a query (that is a sharp value) by dividing the probability of every world by the number of models. Differently from them, we adopt the credal semantics and associate a probability range rather than a probability value to a query. Moreover, they use the ProbLog syntax while we support the ASP syntax, so all the possible types of rules and constraints described in the previous sections. Few systems allow parameter learning in ASP: LP^{MLN} [18] is an extension to ASP that allows the representation of weighted rules that also include an algorithm for parameter learning, but it does not adopt the credal semantics. PrASP [21] is similar and also does not use the credal semantics. Some recent work also combine probabilistic ASP with neural networks, such as NeurASP [31], but with a different semantics.

6 Conclusions

In this paper, we proposed the first algorithm to perform learning from interpretations in Probabilistic Answer Set Programming under the credal semantics. The goal is, given a set of interpretations, to maximize the product of the lower probabilities of the interpretations by tweaking the parameters of the program. We tested the algorithm on two different datasets and showed how the algorithm scales by increasing the number of interpretations and parameters. As future work we plan to develop approximate algorithms for the inference part to scale this approach on real world datasets.

Acknowledgements Damiano Azzolini was supported by IndAM - GNCS Project with code CUP_E55F22000270001.

References

1. Alviano, M., Faber, W.: Aggregates in answer set programming. KI-Künstliche Intelligenz **32**(2), 119–124 (2018). https://doi.org/10.1007/s13218-018-0545-9
2. Azzolini, D., Bellodi, E., Riguzzi, F.: Statistical statements in probabilistic logic programming. In: Gottlob, G., Inclezan, D., Maratea, M. (eds.) Logic Programming and Nonmonotonic Reasoning. pp. 43–55. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-15707-3_4
3. Azzolini, D., Riguzzi, F.: Optimizing probabilities in probabilistic logic programs. Theor. Pract. Log. Prog. **21**(5), 543–556 (2021). https://doi.org/10.1017/S1471068421000260
4. Bellodi, E., Riguzzi, F.: Expectation maximization over binary decision diagrams for probabilistic logic programs. Intell. Data Anal. **17**(2), 343–363 (2013). https://doi.org/10.3233/IDA-130582
5. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (december 2011). https://doi.org/10.1145/2043174.2043195
6. Cozman, F.G., Mauá, D.D.: The structure and complexity of credal semantics. In: Hommersom, A., Abdallah, S.A. (eds.) PLP 2016. CEUR Workshop Proceedings, vol. 1661, pp. 3–14. CEUR-WS.org (2016)
7. Cozman, F.G., Mauá, D.D.: On the semantics and complexity of probabilistic logic programs. J. Artif. Intell. Res. **60**, 221–262 (2017). https://doi.org/10.1613/jair.5482
8. Cozman, F.G., Mauá, D.D.: The joy of probabilistic answer set programming: Semantics, complexity, expressivity, inference. Int. J. Approx. Reason. **125**, 218–239 (2020). https://doi.org/10.1016/j.ijar.2020.07.004
9. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002). https://doi.org/10.1613/jair.989
10. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming, LNCS, vol. 4911. Springer (2008)
11. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI 2007. vol. 7, pp. 2462–2467. AAAI Press (2007)
12. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: European Workshop on Logics in Artificial Intelligence. pp. 200–212. Springer (2004). https://doi.org/10.1007/978-3-540-30227-8_19
13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theor. Pract. Log. Prog. **19**(1), 27–82 (2019). https://doi.org/10.1017/S1471068418000054
14. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.J., Hooker, J.N. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 71–86. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01929-6_7
15. Gutmann, B., Kimmig, A., Kersting, K., Raedt, L.D.: Parameter learning in probabilistic databases: A least squares approach. In: European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2008). LNCS, vol. 5211, pp. 473–488. Springer (2008). https://doi.org/10.1007/978-3-540-87479-9_49

16. Gutmann, B., Thon, I., De Raedt, L.: Learning the parameters of probabilistic logic programs from interpretations. In: Gunopulos, D., Hofmann, T., Malerba, D., Vazirgiannis, M. (eds.) ECML PKDD 2011. LNCS, vol. 6911, pp. 581–596. Springer (2011). https://doi.org/10.1007/978-3-642-23780-5_47
17. Islam, M.A., Ramakrishnan, C., Ramakrishnan, I.: Parameter learning in PRISM programs with continuous random variables. arXiv **abs/1203.4287** (2012). https://doi.org/10.48550/ARXIV.1203.4287
18. Lee, J., Wang, Y.: Weight learning in a probabilistic extension of answer set programs. In: Thielscher, M., Toni, F., Wolter, F. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018. pp. 22–31. AAAI Press (2018)
19. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987)
20. Mauá, D.D., Cozman, F.G.: Complexity results for probabilistic answer set programming. Int. J. Approx. Reason. **118**, 133–154 (2020). https://doi.org/10.1016/j.ijar.2019.12.003
21. Nickles, M.: A tool for probabilistic reasoning based on logic programming and first-order theories under stable model semantics. In: Michael, L., Kakas, A. (eds.) Logics in Artificial Intelligence. pp. 369–384. Springer International Publishing, Cham (2016). https://doi.org/doi.org/10.1007/978-3-319-48758-8_24
22. Raedt, L.D., Kersting, K., Natarajan, S., Poole, D.: Statistical relational artificial intelligence: Logic, probability, and computation. Synthesis Lectures on Artificial Intelligence and Machine Learning **10**(2), 1–189 (2016)
23. Riguzzi, F.: Foundations of Probabilistic Logic Programming: Languages, semantics, inference and learning. River Publishers, Gistrup, Denmark (2018)
24. Riguzzi, F., Di Mauro, N.: Applying the information bottleneck to statistical relational learning. Mach. Learn. **86**(1), 89–114 (2012). https://doi.org/10.1007/s10994-011-5247-6
25. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press (1995). https://doi.org/10.7551/mitpress/4298.003.0069
26. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. J. Artif. Intell. Res. **15**, 391–454 (2001)
27. Totis, P., Kimmig, A., Raedt, L.D.: Smproblog: Stable model semantics in problog and its applications in argumentation. arXiv **abs/2110.01990** (2021). https://doi.org/10.48550/ARXIV.2110.01990
28. Tuckey, D., Russo, A., Broda, K.: PASOCS: A parallel approximate solver for probabilistic logic programs under the credal semantics. arXiv **abs/2105.10908** (2021). https://doi.org/10.48550/ARXIV.2105.10908
29. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38**(3), 620–650 (1991)
30. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3131, pp. 431–445. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27775-0_30
31. Yang, Z., Ishay, A., Lee, J.: NeurASP: Embracing neural networks into answer set programming. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020. pp. 1755–1762. ijcai.org (2020). https://doi.org/10.24963/ijcai.2020/243