dPASP: A Probabilistic Logic Programming Environment For Neurosymbolic Learning and Reasoning

Renato Lui Geh[*1] , **Jonas Gonçalves**[2] , **Igor C. Silveira**[2] , **Denis D. Mauá**[2] , **Fabio G. Cozman**[3]
[1]University of California, Los Angeles
[2]Instituto de Matemática e Estatística, Universidade de São Paulo
[3]Escola Politécnica, Universidade de São Paulo
renatolg@cs.ucla.edu, {jonasrlg,ddm,igorcs}@ime.usp.br, fgcozman@usp.br

Abstract

We present dPASP, a novel declarative probabilistic logic programming framework that allows for the specification of discrete probabilistic models by neural predicates, relational logic constraints, and interval-valued probabilistic choices. This expressive combination facilitates the construction of models that combine low-level perception (images, texts, etc) and common-sense reasoning, thus providing an excellent tool for neurosymbolic reasoning. To support all such features, we discuss several semantics for probabilistic logic programs that allow one to express nondeterminism, nonmonotonic reasoning, contradiction, and (vague) probabilistic knowledge. We also discuss how gradient-based learning can be performed with neural predicates and probabilistic choices under selected semantics. To showcase the possibilities offered by the framework, we present case studies that exploit different semantics and constructs.

github.com/kamel-usp/dpasp

1 Introduction

One can combine logic programming and probabilistic modeling in a myriad of ways: by adopting different syntax and semantics for the logic language, for the probabilistic part, or yet by considering different ways of marrying the two (Sato and Kameya 1997; Muggleton 2003; Poole 2008; Fuhr 2000; Costa et al. 2002; Vennekens, Denecker, and Bruynooghe 2009; Nickles and Mileo 2014; Fierens et al. 2015; Hadjichristodoulou and Warren 2012; Baral, Gelfond, and Rushton 2009; BáRány et al. 2017; Alviano et al. 2023; Lee and Wang 2016).

Probabilistic logic programming offers an expressive and intuitive language for the specification of probabilistic models with context-specific independence, determinism and parameter sharing (Riguzzi 2018; De Raedt et al. 2008). While some of these languages are radically different, many differ only by the choice of the logic part, with a common probabilistic syntax and semantics, or by how probability mass is distributed among the models of underlying logic programs.

In this paper we describe dPASP (differentiable Probabilistic Answer Set Programming), a highly expressive language and accompanying computational framework that encompasses many of the existing formalisms for probabilistic logic programming that fit into Sato's distribution semantics (Sato 1995). dPASP handles logic programs with disjunctive heads, aggregation, integrity constraints, and recursion through negation (under the stable model semantics); it can cope with inconsistencies (e.g., contradictions) by means of least undefined partial stable model semantics (Przymusinski 1991) or smProbLog semantics (Totis, De Raedt, and Kimmig 2023); it allows for interval-valued probabilistic facts (hence decision-theoretic programs) and neural annotated disjunctions (i.e., facts annotated with probabilities specified by neural probabilistic classifiers). Finally, it performs EM-based parameter learning of probabilistic facts and end-to-end gradient-based learning of neural annotated disjunctions, thus enabling neurosymbolic reasoning.

We start by presenting the syntax and semantics of dPASP's language (Sections 2–6), then discuss the implementation details of inference and parameter learning of the system (Section 7), and conclude with some experiments that showcase the framework's features (Section 8).

2 Stratified Probabilistic Programs

We start with the simplest syntax and semantics of locally stratified probabilistic logic programs. To make this description succinct yet accessible, we follow a more intuitive exposition and defer to (Gebser et al. 2012; Lifschitz 2019) for a full presentation of logic programming concepts.

A *probabilistic logic program* is a finite set of *Annotated Disjunctive Rules* (ADRs). dPASP adopts a syntax similar to ProbLog (Fierens et al. 2015). ADRs, or simply rules, are written as

$$p_1::a_1;\ldots;p_k::a_k \texttt{ :- } b_1,\ldots,b_n,\texttt{not } b_{n+1},\ldots,\texttt{not } b_{n+m}\,.$$

Each p_i is a probability value (a number in $(0, 1]$) such that $\sum_{i=1}^{k} p_i \leq 1$. We omit the corresponding value if $p_i = 1$. Each a_i or b_i is an *atom* represented by an expression $p(t_1, \ldots, t_o)$, where p is a string starting with a lower case letter representing a predicate name, and each t_j, $j = 1, \ldots, o$, is either a string starting with lower case, denoting *constants*, or a string starting with an upper case letter representing (logical) *variables*. The atoms $a_1, \ldots, a_k$ are called the *head* of the rule, and the atoms $b_1, \ldots, b_n$ (resp.,

[*] This work was partially done while RLG was at the Instituto de Matemática e Estatística, Universidade de São Paulo.

$b_{n+1}, \ldots, b_{n+m}$) are called the *positive body* (resp., negative body). A rule is called a(n integrity) *constraint* if its head is empty, and it is called a *(probabilistic) fact* if the head is a singleton and the body is empty.

Example 1. *Here is an example of a simple (constraint-free) program stating the relationship between stress, smoking and peer influence:*

```
person(anna). person(bob). smokes(anna).
0.4::stressed(X) :- person(X).
0.3::influences(anna,bob).
smokes(X) :- stress(X).
smokes(X) :- influences(Y,X), smokes(Y).
```

The body of rules with non-probabilistic heads can also contain arithmetic operations and comparisons over variables with integer domains, e.g.

```
sum(A,B,Z) :- digit(A,X), digit(B,Y), Z=X+Y, X<Y.
```

The grounding of the program produces a variable-free program by replacing every rule with every possible variable-free form of the rule in a consistent way (i.e., each occurrence of a variable symbol in a rule is mapped into the same constant).

Example 2. *The grounding of the program in Example 1 is:*

```
person(anna). person(bob). smokes(anna).
smokes(anna).
0.4::stressed(anna) :- person(anna).
0.4::stressed(bob) :- person(bob).
0.3::influences(anna,bob).
smokes(anna) :- stress(anna).
smokes(anna) :- influences(anna,anna), smokes(anna).
smokes(anna) :- influences(bob,anna),  smokes(bob).
smokes(bob) :- stress(bob).
smokes(bob)  :- influences(bob,bob),   smokes(bob).
smokes(bob)  :- influences(anna,bob),  smokes(anna).
```

The semantics of a program is given by the semantics of its grounding, thus we now consider a variable-free program P. A *total choice* is a function σ that maps each rule r to a head atom a_i or to the symbol $\bot_r$, which does not appear in the program. We define the probability of a selection $a_i = \sigma(r)$ for a given rule r as the corresponding probability value $\mathbb{P}(\sigma(r)) = p_r$, or as $1 - \sum_{i=1}^{k} p_i$ if $\sigma(r) = \bot_r$. A total choice denotes a collection of fully independent categorical random variables

$$\mathbb{P}(\sigma) = \prod_{r \in P} \mathbb{P}(\sigma(r)). \tag{1}$$

Intuitively, a total choice produces a pure logic program by replacing each ADR r by the rule

```
$\sigma(r)$ :- $b_1$,...,$b_n$,not $b_{n+1}$,...,not $b_{n+m}$.
```

A σ-*interpretation* is a function I_σ mapping each ground atom to either 0 or 1. The interpretation *satisfies* a set of atoms $\mathbf{a} = (a_1, \ldots, a_m)$, written $I_\sigma \models \mathbf{a}$, if $I_\sigma(a_i) = 1$ for all $a_i \in \mathbf{a}$. The interpretation I_σ is a *model* if it satisfies each rule that is not mapped into $\bot_r$, that is, if for each $r \in P$ we have that either $\sigma(r) = \bot_r$ or $I(\sigma(r)) \geq \min(\{I(b_i) : i = 1, \ldots, n\}, \{1 - I(b_i) : i = n+1, \ldots, n+m\})$. Intuitively, a rule is satisfied if the atom of the head is selected by σ whenever all atoms of the body are satisfied. A model is *minimal* if $I_\sigma \leq I_{\sigma'}$ implies $\sigma = \sigma'$, where $\leq$ is taken coordinate-wise. We denote by $\Gamma(\sigma)$ the set of minimal σ-models of P.

The *dependency graph* of a ground probabilistic program P contains a node for every ground atom. There is an edge $X \to Y$ if there is a rule in P such that X is some atom b_i in its body and Y is an atom a_j in its head. If b_i is positive, the arrow is labeled as positive, otherwise we label it as negative (note that we admit more than one arrow between a pair of nodes, i.e., it is a multigraph). If the graph does not contain a cycle involving a negative arrow, then we say that the program is *stratified*. The ground program in Example 2 is stratified (as it contains no negation). A well-known result is that constraint-free stratified logic programs have exactly one minimal model (Dantsin 1992). This implies that for any total choice σ, we have that $|\Gamma(\sigma)| = 1$.

Given a stratified probabilistic program, for each atom a, we associate an indicator random variable χ_a such that $\chi_a(\sigma) = 1$ if the respective single minimal model satisfies a (i.e., $I_\sigma(a) = 1$), else $\chi_a(\sigma) = 0$. This is Sato's *distribution semantics* for probabilistic logic programs (Sato 1995).

Example 3. *The stratified ground program in Example 2 has 8 total choices with non-zero probability, listed below along with some events (we abbreviate predicate names and constants):*

$\chi_{\texttt{str(a)}}$	$\chi_{\texttt{str(b)}}$	$\chi_{\texttt{inf(a,b)}}$	$\chi_{\texttt{smo(b)}}$	$\mathbb{P}(\sigma)$
0	0	0	0	0.252
1	0	0	0	0.168
0	1	0	1	0.168
1	1	0	1	0.112
0	0	1	1	0.108
1	0	1	1	0.072
0	1	1	1	0.072
1	1	1	1	0.048

2.1 Inference

The most typical inference one draws with probabilistic models is to compute the marginal probability of some set of query atoms, say $\mathbf{q} = \{q_1, \ldots, q_n\}$, possibly conditional on some evidence, say $\mathbf{e} = \{e_1, \ldots, e_n\}$. First, note that the query and evidence can be associated to an indicator random variable $\chi_{\mathbf{a}}$, that returns 1 iff each atom a_i in $\mathbf{a}$ is satisfied by some minimal model I_σ. We define $\mathbb{P}(\mathbf{a}) := \mathbb{P}(\chi_{\mathbf{a}} = 1) = \mathbb{P}(\{\sigma : I_\sigma(\mathbf{a}) = 1\})$. Thus, we are generally interested in computing:

$$\mathbb{P}(\mathbf{q}|\mathbf{e}) = \frac{\sum_{\sigma : I_\sigma \models \mathbf{q}, \mathbf{e}} \mathbb{P}(\sigma)}{\sum_{\sigma : I_\sigma \models \mathbf{e}} \mathbb{P}(\sigma)} . \tag{2}$$

Such a query is encoded in dPASP by the special directive `#query`.

Example 4. *Consider the (stratified ground) program in Example 2 and extend it with the following queries:*

```
#query smokes(bob).
#query smokes(bob), stressed(anna).
#query stressed(anna) | smokes(bob).
```

By collecting the probabilities in Example 3, one can verify that $\mathbb{P}(\texttt{smo(b)}) = 0.58$, $\mathbb{P}(\texttt{smo(b)}, \texttt{str(a)}) = 0.232$ *and* $\mathbb{P}(\texttt{str(a)}|\texttt{smo(b)}) = 0.4$.

2.2 Neural Predicates

The probabilities associated with ADRs often result from statistical models such as neural networks. In such cases, it is convenient to allow a tight integration of statistical modeling tools and logic programming, so that one can build solutions that combine machine learning and probabilistic and logical reasoning.

One of the main advantages of dPASP over similar systems – e.g. DEEPPROBLOG (Manhaeve et al. 2021), NEURASP (Yang, Ishay, and Lee 2020) and SCALLOP (Li, Huang, and Naik 2023) – is the ability to easily integrate Python code that specifies a statistical model, such as a probabilistic classifier, with a probabilistic program. This is performed via the `#python` and `#end.` directives, whose purpose is to encapsulate arbitrary Python code. Code within this block is executed and the functions can be used to assign a probabilistic semantics to predicates of the logic program, using expressions of the form:

```
?::atom(X) as @PythonClass :- input(X).
```

The predicate names `atom` and `input` are user-defined; the first represents the output of the network, the second represents its input. `PythonClass` is the Python object (e.g., a PyTorch module specialization). Variable `X` is used to associate possibly multiple groundings of the predicate `atom` or `input` to different inputs of the model. The symbol `?` indicates the neural predicate's parameters are learnable from data. The user can instead use a `!` symbol to denote them as fixed.

Example 5. *Consider an event whose occurrence is governed by a Poisson distribution with expected rate equals one. Say that there is a 20% chance that preventive measures will be taken to counter the effect of the event; if such measures are not taken, then a disaster follows.*

```
#python
import torch as t
class Poisson(t.nn.Module):
 def __init__(self):
   super().__init__()
   self.l = t.nn.Parameter(t.tensor([1.0]))
   self.p = t.distributions.poisson.Poisson(self.l)
 def forward(self,x):
   return t.exp(self.p.log_prob(x))
#end.
!::event(X) as @Poisson :- input(X).
0.2::counter_measures.
disaster :- event(X), not counter_measures.
#query disaster. % P(disaster)
```

The predicate `event` *is true (selected) with probability given by the output of the statistical model represented by the PyTorch class* `Poisson`*, which in turn depends on the input of the object, represented by the (user-defined) predicate* `input`*. For example, if* `input(2021)` *represents an input of* 2.0 *to the model, then* `event(2021)` *is true with probability* ≈ 0.184*, and thus* $\mathbb{P}(\texttt{disaster}) \approx 0.8 * 0.184 = 0.147$.

The interface between raw data (fed to the neural network in the python code part) and program constants is managed by a special rule of the form

```
atom(x) ∼ test(@func1).
```

In this rule, `atom` is a user-defined one-place predicate name used to represent the input of the statistical model in the program, `x` is an arbitrary constant identifying a particular object (since the same network can be used several times in the same program) and `test` is a reserved predicate, whose argument is a Python function name defined in `#python` that returns data in the appropriate format. Arguments can be optionally passed as input to `@func1`, which are then passed down to the Python function that define the data. Data can be shared among neural networks in the program by associating the neural rules with the same data predicate.

Example 6. *Suppose we have a Python function that takes a year and returns the number of occurrences of an event in that year (e.g., extracted from some web service or data base). We are interested in the* `joint` *probability of the observed frequency of events for three consecutive years.*

```
#python
class Poisson(t.nn.Module): # ...
def get_data(year):
   data = {2020: 0., 2021: 2., 2022: 4.}
   return [[data[year]]]
#end.
input(2020) ∼ test(@get_data(2020)).
input(2021) ∼ test(@get_data(2021)).
input(2022) ∼ test(@get_data(2022)).
!::evt(X) as @Poisson :- input(X).
joint :- evt(Y1),evt(Y2),evt(Y3),Y2=Y1+1,Y3=Y2+1.
#query joint.
```

The program outputs $\mathbb{P}(\texttt{joint}) \approx 0.001$.

A neural predicate can be associated with a multiclass classifier by specifying the domain of the output of a statistical model. This is performed by Neural Annotated Disjunctive Rules (NADRs):

```
?::atom(X, {v_1,...,v_k}) as @model :- input(X).
```

where $v_1, \ldots, v_k$ are constants relating to the class labels of the statistical model defined by the Python class `model`.

Example 7. *Suppose we want to reason over the values of images containing handwritten digits, whose values are predicted by some neural probabilistic classifier implemented by some class* `DigitNet`*. We can represent this by the following program.*

```
input(d0) ∼ test(@get_img(0)).
input(d1) ∼ test(@get_img(1)).
?::digit(X, {0..9}) as @DigitNet :- input(X).
sum(Z)  :- digit(d0,X), digit(d1,Y), Z=X+Y.
odd_sum :- digit(d0,X), digit(d1,Y), X\2=0, Y\2=1.
odd_sum :- digit(d0,X), digit(d1,Y), X\2=1, Y\2=0.
```

The two images are represented by constants `d0` *and* `d1` *in the program, and are manipulated by the Python function* `get_img`*. The neural predicate* `digit` *is equivalent to:*

```
p_1::digit(X,0); ...; p_10::digit(X,9) :- input(X).
```

with the probabilities p_i *as given by the 10-dimensional output of the network for the input associated to some grounding of* X *(either* `d0` *or* `d1`*).*

Neural predicates can also be extended to handle networks whose output contains multiple independent predictors. Syntactically, this is done through a semicolon (e.g., `p(X; {a, b, c})` instead of a colon (e.g., `p(X, {a, b, c})`) in the neural predicate declaration.

Example 8. *Suppose we are interested in knowing the probability that a Poisson distributed event will occur with some frequency between 3 and 7 times.*

```
#python
class Poisson(t.nn.Module): # ...
def get_data(): return [list(range(10))]
#end.
input(data) ~ test(@get_data).
?::event(X; {0..9}) as @Poisson :- input(X).
target :- event(X,K), K > 2, K < 8.
#query target.
```

The syntax `0..9` *is a shorthand for the list* $(0, \dots, 9)$*. Note that the predicate* `event` *takes two arguments: the first one is the constant that represents the input to the Python class (a 10-dimension tensor, in this case); the second one is the constant representing the dimension of the output. This program thus outputs* $\mathbb{P}(\texttt{target}) \approx 0.079$*.*

One can also combine the two extensions, and produce NADRs that are linked to models that output several independent multiclass classifiers, e.g. `p(X, {c1, c2, c3}; {o1, o2})` defines an NADR with two independent classifiers `o1` and `o2`, both with three classes `c1`, `c2` and `c3` each.

3 Normal Probabilistic Programs

We now examine non-stratified programs, i.e. probabilistic programs that induce logic programs with more than one intended model. We focus our attention to *normal* programs of this kind: ones restricted to only a single atom as head.

The semantics of programs with cycles that go through negations relies on the notion of a program's reduct. Given a σ-interpretation I_σ, the reduct of program P w.r.t. I_σ is the program where each rule r is substituted by

```
σ(r) :- b_1, ..., b_n.
```

if $\max\{I(b_{n+i}) : i = 1, \dots, m\} = 0$, else the rule is discarded. Note that the transformation produces a `not`-free program; hence the program reduct is stratified and thus admits a single minimal model. We say that an interpretation I_σ is a *stable model* if it is the minimal model of the program reduct w.r.t. it. This is the *stable model semantics* for normal programs (Gebser et al. 2012; Lifschitz 2019). A program might have 0, 1 or multiple σ-stable models for a fixed σ.

Example 9. *Consider the non-stratified program:*

```
0.4::stressed(anna).
work(anna) :- not nap(anna).
nap(anna) :- not work(anna), not stressed(anna).
```

For $\sigma \mapsto$ `stressed(anna)`*, we have a single* σ*-stable model that assigns* $I_\sigma(\texttt{work(anna)}) = 1$ *and* $I_\sigma(\texttt{nap(anna)}) = 0$*. For* $\sigma \mapsto \bot$*, we have two* σ*-stable models: one that assigns* $I_\sigma(\texttt{work(anna)}) = 1$ *and* $I_\sigma(\texttt{nap(anna)}) = 0$*, and another that assigns* $I_\sigma(\texttt{work(anna)}) = 0$ *and* $I_\sigma(\texttt{nap(anna)}) = 1$*.*

Note that under the stable model semantics, the indicator random variable χ_{a} related to an atom a is not well-defined since there might be σ-stable models assigning different values $I_\sigma(\mathsf{a})$ to a. To make it a proper random variable, we need to assign a probability distribution over such σ-stable models.

Let $\Gamma_{\text{stable}}(\sigma)$ denote the σ-stable models of a program, and $\Gamma^{\mathsf{a}}_{\text{stable}}(\sigma)$ be the subset of σ-stable models that satisfy an atom a. We denote by ω a selection of one particular σ-stable model in $\Gamma_{\text{stable}}(\sigma)$; that is, χ_{a} is a function of both σ and ω. The *maximum entropy* (maxent) probabilistic semantics assumes that ω is uniformly distributed:

$$\mathbb{P}_{\text{maxent}}(\mathsf{a}) = \mathbb{P}(\chi_{\mathsf{a}} = 1) = \sum_\sigma \mathbb{P}(\sigma)\frac{|\Gamma^{\mathsf{a}}_{\text{stable}}(\sigma)|}{|\Gamma_{\text{stable}}(\sigma)|}. \tag{3}$$

Maxent is named after the fact that the above defined probability maximizes entropy over all probabilities consistent with $\mathbb{P}(\sigma)$. That remains true even if we allow for arbitrary $\mathbb{P}(\sigma)$ whose marginals agree with the probabilities of body free rules (Dantsin 1992). While maxent shows interesting properties, it makes some unjustified assumptions regarding the selection of stable models; for one thing, it dilutes probability mass among the many stable models, and it does not allow one to differentiate between uncertainty that arises from severe uncertainty (e.g., logical non-determinism) and aleatory uncertainty (statistical estimates).

Another common choice of probabilistic semantics is to act conservatively, and consider the bounds obtained by assuming every possible extension of $\mathbb{P}(\sigma)$ to $\mathbb{P}(\sigma, \omega)$. The *credal semantics* computes tight lower and upper bounds on such probabilities:

$$\underline{\mathbb{P}}(a) = \mathbb{P}(\{\sigma : \Gamma^{\mathsf{a}}_{\text{stable}}(\sigma) = \Gamma_{\text{stable}}(\sigma)\}), \tag{4}$$
$$\overline{\mathbb{P}}(a) = \mathbb{P}(\{\sigma : \Gamma^{\mathsf{a}}_{\text{stable}}(\sigma) \neq \emptyset\}). \tag{5}$$

The lower and upper probabilities define sub and super-additive functions, respectively $\underline{\mathbb{P}}(a) + \underline{\mathbb{P}}(b) \leq 1$ and $\overline{\mathbb{P}}(a) + \overline{\mathbb{P}}(b) \geq 1$ for disjoint a and b. Those bounds are akin to cautious and brave semantics used in Answer Set Programming to draw logical inferences under multiple intended model programs (Gebser et al. 2012). Note that, by construction, $\mathbb{P}_{\text{maxent}}(a) \in [\underline{\mathbb{P}}(a), \overline{\mathbb{P}}(a)]$. Also, for stratified programs all three values coincide.

Example 10. *Consider again the program in Example 9. To select the semantics in* dPASP*, we use:*

```
#semantics maxent. % maxent semantics.
#semantics credal. % credal semantics.
```

The program's output (predicates and constants abbreviated) is

$$\underline{\mathbb{P}}(\texttt{n(a)}) = 0.0, \quad \mathbb{P}_{\text{maxent}}(\texttt{n(a)}) = 0.3, \quad \overline{\mathbb{P}}(\texttt{n(a)}) = 0.6,$$
$$\underline{\mathbb{P}}(\texttt{w(a)}) = 0.4, \quad \mathbb{P}_{\text{maxent}}(\texttt{w(a)}) = 0.7, \quad \overline{\mathbb{P}}(\texttt{w(a)}) = 1.0.$$

Example 11. *Here is a more intricate non-stratified program, taken from (Totis, De Raedt, and Kimmig 2023, Example 6), that represents a probabilistic argumentation scenario involving six abstract arguments* $(\texttt{a1}, \dots, \texttt{a6})$*, probabilistic support and attack relations, and prior argument probabilities.*

```
0.4::base(a1). 0.8::base(a2). 0.3::base(a3).
0.7::base(a4). 0.6::base(a5). 0.7::base(a6).
pos(A) :- arg(A).
```

```
0.6::neg(a6) :- arg(a1). 0.3::neg(a1) :- arg(a4).
0.8::neg(a1) :- arg(a2). 0.7::neg(a2) :- arg(a1).
0.6::pos(a4) :- arg(a5). 0.5::pos(a1) :- arg(a3).
arg(A) :- pos(A), not neg(A).
```

*The following table shows the marginal probabilities for each argument under the maxent and credal semantics. Arguments that are not attacked by others (*a3*,* a4 *and* a5*) have narrower intervals for the credal semantics.*

	a1	a2	a3	a4	a5	a6
$\underline{\mathbb{P}}$	0.13	0.59	0.30	0.81	0.60	0.57
$\mathbb{P}_{\text{maxent}}$	0.22	0.68	0.30	0.81	0.60	0.61
$\overline{\mathbb{P}}$	0.30	0.76	0.30	0.81	0.60	0.64

We conclude this section with two important observations. First, both the maxent and credal semantics require that $\Gamma_{\text{stable}}(\theta)$ is non-empty for each σ; this condition is called *consistency* (Cozman and Mauá 2020), and is required by most probabilistic logic frameworks. The second observation is that both probabilistic semantics are agnostic to the choice of logic semantics, that is, similar and alternative semantics can be created simply by adopting different sets of intended models $\Gamma(\theta)$, as long as such sets are non-empty. For example, an alternative semantics would be to adopt the well-founded models semantics (Fierens et al. 2015), which is currently not implemented in dPASP. We will discuss in Section 5 how to extend the maxent and credal semantics to three-valued semantics that cope with inconsistencies.

4 Disjunctive Programs

Another interesting feature that dPASP borrows from Answer Set Programming is to allow non-probabilistic disjunction in the head. A disjunctive rule is an expression:

```
a_1;...;a_k :- b_1,...,b_n,not b_{n+1},...,not b_{n+m}.
```

The rule encodes the knowledge that *some* atom of the head must be satisfied whenever the body is satisfied.

The (extended) *stable semantics* of probabilistic programs with disjunctive heads is given by the minimal σ-models of program reduct, with disjunctive rules r being discarded if there is an atom b_{n+i} in the negative body being satisfied (hence its negation is not), else they are transformed into `not`-free rules

```
a_1;...;a_k :- b_1,...,b_n.
```

Note that unlike in the transformation for ADRs, the heads of disjunctive rules are not affected by other rules that are not discarded in the transformation. Thus, disjunctive rules can encode non-determinism even in the program reduct. An interpretation is σ-stable if it is a minimal model (not necessarily unique) of its program reduct.

Example 12. *The following program computes the probability that a 3-node random graph is 2-colorable.*

```
node(1). node(2). node(3).
0.5::edge(X,Y) :- node(X), node(Y), X < Y.
edge(X,Y) :- edge(Y,X), X > Y.
fail :- edge(X,Y), color(X,C), color(Y,C).
color(X,red)  :- fail, node(X).
color(X,blue) :- fail, node(X).
color(X,red); color(X,blue) :- node(X).
colorable :- not fail.
#query colorable.
```

The program outputs $\underline{\mathbb{P}}(\texttt{colorable}) = \mathbb{P}_{maxent}(\texttt{colorable}) = \overline{\mathbb{P}}(\texttt{colorable}) = 7/8$.

The previous program is an example of an encoding technique called *saturation*, where non-solutions of a combinatorial problem are mapped into a single maximal model (one that satisfies the maximum number of atom), which is thus minimal iff no other (stable) model exists. In the example, the non-colorable graphs are represented by models that satisfy both `color(x,red)` and `color(x,blue)` for each node `x`. This technique requires the use of disjunction in heads.

5 Handling Inconsistencies

As noted previously, the probabilistic semantics are only defined for consistent programs. Yet, there are situations where one wants to admit σ-induced contradictory logic programs that allow no stable model (Lee and Wang 2016; Totis, De Raedt, and Kimmig 2023).

One approach to cope with such inconsistencies is to extend the logic semantics to admit an undefined state for atoms, denoting that they are involved in a contradiction. dPASP allows for two such semantics: one introduced in Totis, De Raedt, and Kimmig 2023 and one that extends the least undefined stable semantics of logic programs (Przymusinski 1991).

To enable the modeling of rules with negated atoms for representing an inhibition effect (thus differing from the usual classic negation interpretation of Answer Set Programming), smProbLog (Totis, De Raedt, and Kimmig 2023) proposes a three-valued semantics that coincides with the maxent stable model semantics whenever $\Gamma_{\text{stable}}(\sigma)$ is non-empty, otherwise assigning a single model $I_\sigma(a) = 0.5$ for every atom a. The value 0.5 denotes an undefined state (between false, represented by 0, and true, represented by 1). This type of modeling thus allows one to renormalize the probability function over the total choices with consistent programs by conditioning on any atom being not undefined. For instance, we can query for

```
#query smokes(anna) | not undef smokes(anna).
#query undef smokes(anna).
#semantics smproblog.
```

under the smProbLog semantics to obtain, respectively, the probability that `anna` smokes over the σ-induced programs where such atom (and thus all others) are defined, as well as the probability of the complement of the conditioning event.

The *least undefined stable model semantics* (L-stable, for short) acts strictly, and leaves only atoms that are involved in contradictions undefined. It is based on $\{0, 0.5, 1\}$-valued σ-interpretations, where $I_\sigma(a) = 0.5$ denotes that a is deemed undefined (or undecided) by the interpretation. The program reduct w.r.t. such I_σ changes only in that undefined atoms in the body are substituted with a reserved atom `undef`, which is not present in the program and such that $I_\sigma(\texttt{undef}) = 0.5$. Then we say that a three-valued σ-interpretation I_σ is a partial stable model in the same way as $\{0, 1\}$-valued σ-interpretations; that is, if it is a minimal model of its program reduct. Note that for three-valued interpretations, $I_\sigma(a) = 1 - I_\sigma(a)$ iff $I_\sigma(a) = 0.5$; thus, interpretations that leave all atoms undefined satisfy every rule and are hence models (but not necessarily minimal models).

The L-stable model semantics selects only stable models that are least undefined: $I_\sigma \in \Gamma_{\text{ls}}(\sigma)$ iff there is no different I'_σ such that $I_\sigma(a) \neq 0.5$ implies $I'_\sigma(a) \neq 0.5$. The *maxent L-stable* semantics replaces $\Gamma_{\text{stable}}(\sigma)$ with $\Gamma_{\text{ls}}(\sigma)$ in Eq. (3). Similarly, the *credal L-stable* semantics (Rocha and Cozman 2022) replaces the stable models with L-stable models in Eqs. (4) and (5).

Example 13. *We compare two semantics that allow for inconsistent probabilistic programs (i.e., programs with contradictions): L-stable and smProbLog. We use a reduced variant of the asthma example provided in (Totis, De Raedt, and Kimmig 2023), which extends Example 2 with knowledge about the development of asthma. To allow for rules that imply the negation of an atom* a *(when the body is satisfied), we create copies* a_pos *and* a_neg *that represent positive and negative variants, and a constraint that enforces consistency among the three atoms.*

```
person(1). person(2). person(3). person(4).
0.1::asthma(X) :- person(X).
0.3::stress(X) :- person(X).
0.3::influences(1,2). 0.6::influences(2,1).
0.2::influences(2,3). 0.7::influences(3,4).
0.9::influences(4,1).
0.4::smokes_pos(X) :- stress(X).
smokes_pos(X) :- influences(Y,X), smokes(Y).
smokes_neg(X) :- asthma(X).
smokes(X) :- smokes_pos(X), not smokes_neg(X).
0.4:: asthma(X) :- smokes(X).
```

We present the probability of the atoms smokes(X) *being undefined in the following table.*

undef smokes(i)	i			
	1	2	3	4
smProbLog	0.2223	0.2223	0.2223	0.2223
L-stable	0.1548	0.0828	0.0599	0.0909

This program has only one σ-model for each σ, so the credal and maxent semantics coincide.

6 Imprecisely Specified Probabilities

The credal semantics produces probability intervals related to the non-determinism created by multiple stable models for a fixed probabilistic choice. That is, the uncertainty over probability values is the result of the logical semantics of the program.

Some situations however are better captured by uncertainty about the values of probabilities given as input, that is, the probabilities associated with ADRs (Mauá and Cozman 2023). To address such situations, dPASP currently accepts *interval-valued probabilistic facts* of the form

```
[p, q]::a.
```

where $0 \leq p \leq q \leq 1$ and a is an arbitrary ground atom. The expression indicates that a is true with some probability $p \leq \mathbb{P}(a) \leq q$. Any credal network (i.e., a Bayesian network whose conditional probabilities are specified as intervals) can be represented as a probabilistic logic program with interval-valued probabilistic facts, with some local complexity overhead on the in-degree of nodes (Mauá and Cozman 2023).

The semantics of programs with interval-valued probabilistic facts extends the credal semantics as follows: let γ be a function that selects an end-point for each probability interval (i.e., it selects either p or q for $[p, q]$). Obtain a probabilistic program P_γ by replacing intervals by the point probability value selected by γ, for each interval-valued rule. Then the lower (resp., upper) probability $\underline{\mathbb{P}}(a)$ (resp., $\overline{\mathbb{P}}(a)$) of an atom a is the minimum (resp., maximum) probability of a of all such programs P_γ. We call such semantics also *credal semantics*, as it produces intervals and reduces to the previous credal semantics when intervals are singletons.

Example 14. *An interesting use of interval-valued probabilistic facts is decision making. Suppose we want to decide whether to take an umbrella when leaving home. According to the forecast, there is a 40% chance of rain. Say that we assign a utility of 0 if it rains and we did not take the umbrella, 0.1 if it does not rain and we took the umbrella, 0.5 if it rains and we took the umbrella, and 1.0 if it does not rain and we did not take the umbrella. The following program encodes such a situation:*

```
[0,1]::umbrella.   0.4::rain.
0.1::sad :- not rain, umbrella.
0.5::happy :- rain, umbrella.
very_happy :- not rain, not umbrella.
util :- sad.   util :- happy.   util :- very_happy.
#query util.
```

Then the credal semantics is $\underline{\mathbb{P}}(\texttt{util}) = 0.26$ and $\overline{\mathbb{P}}(\texttt{util}) = 0.6$, and reflects the expected utility of the worst and best actions, respectively.

7 The dPASP System

In this section, we describe the implementation details of the dPASP computational system, an open-source software available at https://github.com/kamel-usp/dpasp.[1] The system can be used as a stand-alone command that takes a program in dPASP domain specific language and prints out the results of queries, or as a Python library for integration into a larger system. Most of the system was designed with flexibility in mind, so that different combinations of semantics can be tested and implemented easily. The downside is that the system currently lacks semantics-specific optimizations (e.g. knowledge compilation and relevance pruning).

7.1 Inference

Inference in dPASP currently comes in two flavors: exact inference by exhaustive enumeration, and approximate inference through answer set enumeration by optimality (Pajunen and Janhunen 2021). Selecting which algorithm is used is done through the `#inference` directive. If no such directive is present in the program, exact inference is used by default.

[1]A quick-guide and setup tutorial on dPASP can be found at https://kamel.ime.usp.br/pages/learn_dpasp.

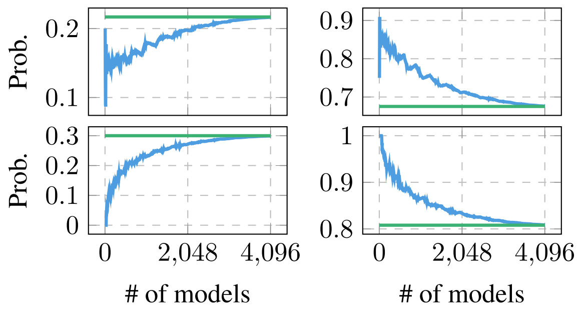

Figure 1: Approximate values (in blue) and ground truth (in green) versus number of models for the inference in Example 11.

Exact inference Exact inference in dPASP is done by enumerating all total choices and using clingo's solver (Gebser et al. 2017) to list all models for each induced logic program. Under the maxent semantics, this amounts to brute-force computing Equation (2) for each model I. For the credal semantics, we compute lower and upper bounds through the exact algorithm described in (Cozman and Mauá 2020). Performing exact inference by this exhaustive approach, under either (probabilistic) semantics, limits the scalability of inference to programs with few NADRs and ADRs.

Approximate inference An approximate inference algorithm based on answer set enumeration by optimality (Pajunen and Janhunen 2021) is available in a limited capacity through the directive `#inference aseo`.

Example 15. *Example 11 contains 12 probabilistic rules and facts, amounting to a total of 4096 total choices. To compute the query* $\mathbb{P}(\texttt{arg(a1)})$ *by answer set enumeration by optimality, we extend the program as follows.*

```
#inference aseo, nmodels=2048.
#query arg(a1).
```

Parameter `nmodels` *specifies the number of optimality models to be enumerated.*

Figure 1 shows the approximated values of $\mathbb{P}(\texttt{arg(a1)}), \dots, \mathbb{P}(\texttt{arg(a4)})$ (under the maxent and stable model semantics) as the number of models increases.

More scalable approximate inference based on knowledge compilation (Totis, De Raedt, and Kimmig 2023), sampling (Tuckey, Russo, and Broda 2021; Azzolini, Bellodi, and Riguzzi 2023) and variational methods are planned features for future versions of dPASP, which is currently in an early stage of development.

Partial, L-stable and smProbLog semantics Internally, dPASP only accepts the stable model semantics when performing inference or learning. To enable support to different semantics, we implement translation procedures to the stable model semantics.

The partial stable model semantics in dPASP is implemented via the translation described in (Janhunen et al. 2006). In a nutshell, dPASP creates an auxiliary atom and rule for each non-probabilistic atom in the program and duplicates logic rules in order to allow undefined values for non-probabilistic atoms. The L-stable semantics is implemented by checking, for each total choice, whether there exists a stable model for the program: in the positive case, dPASP performs inference over the stable models of such a program, otherwise it queries from the translated program's partial stable models. Inference under the smProbLog semantics follows a similar approach: when enumerating total choices, if it so happens that the induced program has no stable model, we add a model where all atoms are undefined.

Maximum entropy semantics If the maxent semantics is selected, it is sufficient to simply add up the (uniform) probabilities of each model that is consistent with the query; this is the same procedure done in Yang, Ishay, and Lee 2020. dPASP's exact inference computes the counts $|\Gamma^{\text{a}}_{\text{stable}}(\sigma)|$ and $|\Gamma_{\text{stable}}(\sigma)|$ by calling clingo's routine for enumerating stable models.

Credal semantics For the credal semantics, one is interested in the interval of all probabilities $\mathbb{P}(\mathbf{q}|\mathbf{e})$ obtained by some probability model. This interval can be described by its lower and upper values, which in dPASP are obtained by the exact algorithm described in (Cozman and Mauá 2020). In short, we compute the lower $\underline{\mathbb{P}}(\mathbf{q}|\mathbf{e})$ and upper $\overline{\mathbb{P}}(\mathbf{q}|\mathbf{e})$ probabilities by iterating over each total choice σ and counting the models where (a) every model satisfies both $\mathbf{q}$ and $\mathbf{e}$, (b) some model satisfies both $\mathbf{q}$ and $\mathbf{e}$, (c) every model satisfies $\mathbf{e}$ but does not satisfy some value in $\mathbf{q}$, and (d) some model satisfies $\mathbf{e}$ but does not satisfy some value in $\mathbf{q}$. The credal interval is then $[0, 0]$ if $b + c = 0$ and $d > 0$, $[1, 1]$ if $a + d = 0$ and $b > 0$, and $[a/(a + d), b/(b + c)]$ otherwise.

Credal facts in dPASP are only available when the credal semantics is selected. To perform inference with credal facts, dPASP constructs four multilinear polynomials corresponding to a, b, c and d; each term is a total choice θ, each coefficient is the probability of θ, and indeterminates in the polynomial are x if $\chi_x = 1$ in σ or $1 - x$ otherwise. The domain of the polynomial is the Cartesian product of all pairs of lower and upper probabilities in credal facts. The functions $a(\mathbf{x})/(a(\mathbf{x}) + d(\mathbf{x}))$ and $b(\mathbf{x})/(b(\mathbf{x}) + c(\mathbf{x}))$ are then optimized in order to find the two global minima and maxima respectively, with the first amounting to the lower and the second the upper probabilities of the queries.

7.2 Parameter Learning

dPASP currently implements three maximum likelihood parameter learning rules for the maxent stable model semantics: (i) a fixed-point learning procedure for non-neural programs that matches EM, (ii) a Lagrange multiplier derivation for gradient ascent, and (iii) an implementation of NeurASP's learning procedure (Yang, Ishay, and Lee 2020). All three procedures optimize the parameters (probabilities of ADRs and neural network parameters of NDRs) w.r.t. to the loglikelihood $\mathcal{L}(O)$ of a set of observations O (sets of atoms considered true), assumed i.i.d.

To understand the need for our alternative parameter learning method, we first discuss the shortcomings of the NeurASP learning rule. The procedure updates parameters $\mathbb{P}(\chi_x = 1) = p_x$ by the standard gradient rule $\mathbf{p} \leftarrow$

SYSTEM	STRUCTURE			PROBABILISTIC SEMANTICS			INFERENCE		LEARNING	
	Non-strat?	Incons?	CA?	PR?	Imp Prob?	Semantics	Exact	Approximate	PR	NR
DEEPPROBLOG	✗	✗	✓	✓	✗	Distribution	SDD	DPLA*	EM	SGD
NEURASP	✓	✗	✓	✗	✗	Maxent	Enumerative	✗	✗	SGD
SCALLOP	✗	✗	✗	✓	✗	Provenance	SDD	Top-k	SGD	SGD
dPASP	✓	✓	✓	✓	✓	Maxent, credal	Enumerative	ASEO	EM	SGD

Table 1: Overview of some of the features in DEEPPROBLOG, NEURASP, SCALLOP and dPASP and how they compare to each other.

$\mathbf{p} - \eta\nabla_\rho\mathcal{L}(O)$. When parameters are outputs of neural networks, the update rule can produce infeasible values (i.e., they are not normalized probability distributions). This issue can be mitigated by either projecting the parameters back to the feasible set or ensuring that parameter updates lie within the feasible set, for instance by using a softmax layer. To avoid this issue and allow for arbitrary output layers, we instead constrain parameters to remain within the feasible set by employing Lagrange multipliers.

For simplicity, we assume w.l.o.g. that the atoms in heads of ADRs and NADRS do not unify with heads of any other rule. Hence, such atoms are true in some stable model only if they are selected by the corresponding total choice. Let σ_x denote a total choice that selects atom x from its (N)ADR. The constrained derivative $\frac{\partial}{\partial p_x}\mathcal{L}(O)$ of the log-likelihood function with respect to the probability $\mathbb{P}(\chi_x = 1) = p_x$ subject to $\sum_{x'} p_{x'} = 1, p_x \geq 0$ is:

$$\left(1 - \frac{1}{m}\right)\frac{1}{\mathbb{P}(O)}\sum_{\sigma_x}\frac{\mathbb{P}(\sigma_x)}{\mathbb{P}(\chi_x = 1)}\cdot\frac{|\Gamma^O_{\text{stable}}(\sigma_x)|}{|\Gamma_{\text{stable}}(\sigma_x)|}$$
$$-\frac{1}{m}\sum_{\overline{x},\,\overline{x}\neq x}\frac{1}{\mathbb{P}(O)}\sum_{\sigma_{\overline{x}}}\frac{\mathbb{P}(\sigma_{\overline{x}})}{\mathbb{P}(\chi_{\overline{x}} = 1)}\cdot\frac{|\Gamma^O_{\text{stable}}(\sigma_{\overline{x}})|}{|\Gamma_{\text{stable}}(\sigma_{\overline{x}})|}. \quad (6)$$

where m is the number of atoms in the head.

Interestingly, Eq. (6) yields a similar expression to NEURASP's learning rule, with the only distinction being the factors $1-\frac{1}{m}$ and $\frac{1}{m}$. Thus, when $m = 2$, the Lagrangian rule is equivalent to halving the learning rate of NEURASP's rule. For $m > 2$, rule (6) assigns more weight to the probability of interpretations consistent with the observation, and less weight to its complement. This is more sensible, since the latter sums over more terms than the former.

The extension of (6) to the neural case is trivial: by applying the chain rule on the derivative of the log-likelihood with respect to the output p_x of the corresponding output of the neural network, we easily find that the resulting gradient is (6) multiplied by the derivative of the neural network with respect to network weights $\mathbf{w}$:

$$\frac{\partial}{\partial \mathbf{w}}\mathcal{L}(O) = \frac{\partial}{\partial p_x}\mathcal{L}(O)\frac{\partial}{\partial \mathbf{w}}p_x(\mathbf{x}), \quad (7)$$

where $\frac{\partial}{\partial \mathbf{w}}p_x(\mathbf{x})$ is the standard backward pass in a neural network computed using PyTorch's features.

7.3 Related Systems

Neurosymbolic programming languages have become increasingly popular in the last few years, with each implementation boasting a host of different features and semantics. Given the many (often subtle) differences between them, it is useful to compare some of these available systems and contrast them with dPASP. More specifically, we provide a short comparison of dPASP against DEEPPROBLOG, NEURASP and SCALLOP (Table 1).

In columns 2–4 of Table 1, systems are classified w.r.t. to the expressiveness of their language in three different dimensions: whether the language (i) accepts non-stratified programs (hence models non-determinism), (ii) handles inconsistencies (e.g. three-valued logic) or (iii) allows for complex arguments (i.e. complex terms as arguments for predicates, e.g. `f(g(x), y))`. Note that although ProbLog accepts non-stratified programs and handles inconsistencies with the smProbLog semantics, this extension is not yet implemented in DEEPPROBLOG. Both dPASP and NEURASP allow for non-stratified programs through their stable model semantics, but only dPASP handles inconsistencies in the program through the L-stable semantics. SCALLOP is based on Datalog (Abiteboul, Hull, and Vianu 1995), a syntactic subset of Prolog (Colmerauer 1990) – and thus stratified – with no support for complex arguments or inconsistencies.

Columns 5–6 classify the systems in terms of expressiveness for uncertainty specification. We distinguish between support for (i) probabilistic rules, and (ii) imprecise specification of probabilities (e.g., interval-valued probabilistic facts). Even though NEURASP does not natively support non-neural probabilistic rules, these can be expressed as neural predicates associated with a simple probabilistic classifier. dPASP is the only to allow for imprecise probabilities. In terms of semantics, DEEPPROBLOG follows Sato's distribution semantics (Sato 1995), SCALLOP defines its semantics by means of provenance semirings (Green, Karvounarakis, and Tannen 2007), NEURASP adopts the maxent semantics, and dPASP admits both maxent and credal. All these semantics coincide for stratified programs.

The last four columns summarize inference and learning routines for each system. Both DEEPPROBLOG and SCALLOP compile programs to sentential decision diagrams (SDDs, Darwiche 2011), a special form of logic circuit, on which they perform weighted model counting (WMC) for exact inference (Manhaeve et al. 2021). NEURASP and dPASP exhaustively enumerate all models. For approximate inference, DEEPPROBLOG uses an A*-like search over the SLD resolution tree to approximate probabilistic queries (Manhaeve, Marra, and De Raedt 2021), SCALLOP computes the top-k proofs and then performs WMC on the resulting compiled SDD (the exact version is the particular instance when k is the number of total choices) and dPASP performs answer set enumeration by optimality (ASEO, see Section 7.1). Parameter learning is done through expectation-maximization (EM) for non-neural probabilis-

	concept learners		ensemble
	recall	acc.	recall
Digit 0	0.93	0.97	0.99
Digit 1	0.86	0.94	0.96
Digit 2	0.81	0.93	0.97

Table 2: Performance of individual concept learners (on their task) and recall of the ensemble classifier.

tic programs in DEEPPROBLOG and dPASP, and stochastic gradient descent (SGD) for neural programs in all systems.

8 Case Studies

We now present two case studies showcasing dPASP. We first show how the system can be used to combine neural concept learners into an ensemble, increasing the overall accuracy of the system. In a following case study, we compare the performance of our system against two competitors on the task of parameter learning in image classification.

8.1 Data Fusion and Uncertainty Quantification

A concept learner is a binary classifier trained to recognize only objects of a certain class. Classifiers made of ensemble of concept learners are flexible in that classes can be added or removed without need for a full re-training, and that concept learners can be deployed and trained in a decentralized fashion (Verbaeten and Van Assche 2003; Cao, Brbić, and Leskovec 2021; Núñez, Fidalgo, and Morales 2007).

We perform an experiment on MNIST to showcase the ability of dPASP to combine predictions from a set of concept learners, comparing the maxent and credal probabilistic semantics. In this experiment, the digits 0, 1 and 2 of the dataset act as different concepts. For each digit, we train a convolutional neural network (CNN) to act as a specialized concept learner in a one-versus-all approach with negative sampling to ensure balancedness of positive/negative labels.

The first two columns of Table 2 show the recall (w.r.t. the concept learned) and accuracy of each model on MNIST test data (restricted to the digits labeled 0, 1 or 2). The relatively smaller recall shows a tendency of binary classifiers to classify objects as not being their learned concepts. Furthermore, the relatively higher accuracy is caused by the dataset being unbalanced for each concept.

```
data(x) ~ test(@test), train(@train).
?::concept(1,X) as @CLearner1 :- data(X,1).
?::concept(2,X) as @CLearner2 :- data(X,2).
?::concept(3,X) as @CLearner3 :- data(X,3).
class(1) :- not class(2),not class(3),not novel.
class(2) :- not class(1),not class(3),not novel.
class(3) :- not class(1),not class(2),not novel.
novel    :- not class(1),not class(2),not class(3).
:- class(C),not concept(C,X).
#query class(1). #query class(2). #query class(3).
```

The above program defines a multiclass classifier (with a novel class label) which can either use the maxent or credal semantics. The performance of the classifier based on the maxent semantics is presented in the right column of Table 2. The classifier accuracy (considering all classes/concepts) is 0.98, thus surpassing that of its components.

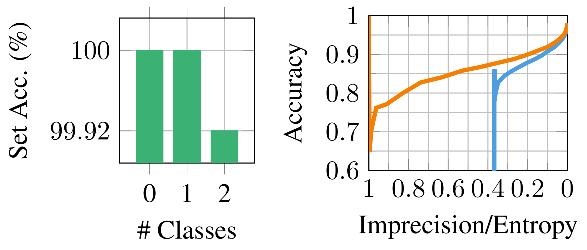

Figure 2: Left: set accuracy by size of credal classification. Right: accuracy versus uncertainty (**imprecision** or **entropy**).

We can use the credal semantics to either produce more conservative inferences or to assess the predictive uncertainty. We say that a concept learner *dominates* another one if the lower probability of the former concept is higher than the upper probability of the latter. A credal classification outputs only non-dominated class labels.

Figure 2 (left) contains the set accuracy (i.e. the percentage of credal classifications containing the correct label) as a function of the number of non-dominated classes. 2727 classifications were vacuous (when no class is dominated), and 410 were precise (a single class dominated others). Figure 2 (right) plots classification accuracy of the maxent classifier against the uncertainty as measured by either the entropy of the maxent classifier (blue) or by the length of the probability interval computed by the credal semantics (orange). One sees that both entropy and imprecision correlate well with accuracy, but imprecision is easier to interpret.

8.2 Distant Learning: MNIST Addition

We compare the performance of dPASP to NEURASP, DEEPPROBLOG, SCALLOP and a purely data-driven CNN on the task of learning addition of MNIST image digits, a common distant supervision benchmark for neural probabilistic logic programs (Manhaeve et al. 2021). Given two unlabelled images (e.g. and) of digits, and the corresponding atom (e.g. `sum(9)`) as a distant label, the program must learn to identify the sum of digits.

The code for this task is shown below, consisting of boilerplate data pre-processing and classifier definition functions (omitted for brevity), and a simple and short probabilistic logic section. The full program can be found in the project's repository.

```
#python
def DigitNet(): ...  # neural network classifier
def mnist_tr(i): ... # train images for i-th digit
def mnist_te(i): ... # test images for i-th digit
def labels(): ...    # sum(z) labels
#end.
input(0) ~ test(@mnist_te(0)), train(@mnist_tr(0)).
input(1) ~ test(@mnist_te(1)), train(@mnist_tr(1)).
?::digit(X, {0..9}) as @DigitNet :- input(X).
sum(Z) :- digit(0, X), digit(1, Y), Z = X+Y.
#semantics maxent.
#learn @labels, lr = 0.001, niters = 5, batch = 1000.
```

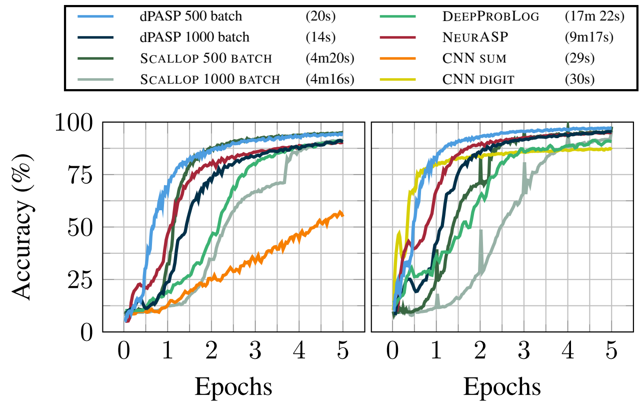

Figure 3: Sum and digit classification accuracy and training time (in parentheses) for dPASP, NEURASP, DEEPPROBLOG, SCALLOP and CNN. Left: accuracy per iteration of classifying sums. Right: accuracy of learned networks on classifying digits.

We follow the same neural network architecture and parameters used in Manhaeve et al. 2021 for the MNIST digit addition task. Figure 3 shows a comparison of the performance both in terms of classification accuracy and training time. The plot on the left compares the accuracy of classifying the correct sum of digits, while the plot on the right shows the digit classification accuracy of the program-embedded neural network during learning. CNN SUM corresponds to the performance of evaluating a CNN whose input is a single image consisting of concatenating the two digits and whose output are the probabilities of the 19 possible two-digit sum values; CNN DIGIT is the accuracy of a single digit classification network under the same parameter conditions as dPASP, SCALLOP, NEURASP and DEEPPROBLOG. We report the performance of dPASP and SCALLOP with training batch sizes of 500 and 1000.

Unsurprisingly, both purely data-driven CNN approaches performed poorly compared to neurosymbolic approaches. In particular, CNN SUM struggled to even break 50% accuracy, while CNN DIGIT quickly converged to the 80% mark, below that of neural probabilistic logic frameworks. We again stress the fact that these results were obtained by subjecting all systems to the same learning parameters. Comparing dPASP against SCALLOP, DEEPPROBLOG and NEURASP, we find that, under the same parameters, NEURASP converges faster, although the difference is small. This difference might be explained by the correction factor discussed in Section 7.2, which might require a different learning rate to equalize convergence.

Lastly, we note the surprisingly small difference gap in training times (in parentheses) for dPASP and CNNs. We conjecture that the main factor is implementation overhead: dPASP is mostly written in C, with a simple Python wrapper; in contrast, the pure PyTorch implementation still runs significant portions in Python.

9 Conclusion

We presented dPASP, a framework for neurosymbolic reasoning and learning based on probabilistic logic programming. We discussed syntax and semantics of the language, and commented on inference and learning implementations in the system. We showed use cases that illustrate the features and potential applications. There is still much to achieve to make the system more broadly applicable and effective. In particular, more efficient learning and inference routines need to be devised to scale to larger domains.

Acknowledgements

The authors of this work would like to thank the Center for Artificial Intelligence (C4AI-USP), the São Paulo Research Foundation (FAPESP grant #2019/07665-4), and IBM Corporation. This work was partially supported by FAPESP grants #2022/02937-9, CNPq grant no. 305136/2022-4 and CAPES Finance Code 001. RLG received partial support from FAPESP grant #2022/06373-2, DARPA PTG Program under award HR00112220005, the DARPA ANSR program under award FA8750-23-2-0004, and NSF grants #IIS-1943641.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases: The Logical Level*. USA: Addison-Wesley Longman Publishing Co., Inc., 1st edition.

Alviano, M.; Lanzinger, M.; Morak, M.; and Pieris, A. 2023. Generative Datalog with stable negation. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, 21–32. New York, NY, USA: Association for Computing Machinery.

Azzolini, D.; Bellodi, E.; and Riguzzi, F. 2023. Approximate inference in probabilistic answer set programming for statistical probabilities. In Dovier, A.; Montanari, A.; and Orlandini, A., eds., *AIxIA 2022 – Advances in Artificial Intelligence*, 33–46.

Baral, C.; Gelfond, M.; and Rushton, N. 2009. Probabilistic reasoning with answer sets. *Theory Pract. Log. Program.* 9(1):57–144.

BáRány, V.; Cate, B. T.; Kimelfeld, B.; Olteanu, D.; and Vagena, Z. 2017. Declarative probabilistic programming with Datalog. *ACM Transactions on Database Systems* 42(4).

Cao, K.; Brbić, M.; and Leskovec, J. 2021. Concept learners for few-shot learning. In *International Conference on Learning Representations*.

Colmerauer, A. 1990. An introduction to Prolog III. *Commun. ACM* 33(7):69–90.

Costa, V. S.; Page, D.; Qazi, M.; and Cussens, J. 2002. CLP(BN): constraint logic programming for probabilistic knowledge. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*, UAI'03, 517–524. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Cozman, F. G., and Mauá, D. D. 2020. The joy of probabilistic answer set programming: Semantics, complexity, expressivity, inference. *International Journal of Approximate Reasoning* 125:218–239.

Dantsin, E. 1992. Probabilistic logic programs and their semantics. In Voronkov, A., ed., *Logic Programming*, 152–164. Berlin, Heidelberg: Springer Berlin Heidelberg.

Darwiche, A. 2011. SDD: a new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, 819–826. AAAI Press.

De Raedt, L.; Frasconi, P.; Kersting, K.; and Muggleton, S., eds. 2008. *Probabilistic Inductive Logic Programming*. Springer Berlin Heidelberg.

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.

Fuhr, N. 2000. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science* 51(2):95–110.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. Answer set solving in practice. In *Answer Set Solving in Practice*.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2017. Multi-shot ASP solving with clingo. *CoRR* abs/1705.09811.

Green, T. J.; Karvounarakis, G.; and Tannen, V. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, 31–40. New York, NY, USA: Association for Computing Machinery.

Hadjichristodoulou, S., and Warren, D. S. 2012. Probabilistic logic programming with well-founded negation. In *IEEE 42nd International Symposium on Multiple-Valued Logic*, 232–237.

Janhunen, T.; Niemelä, I.; Seipel, D.; Simons, P.; and You, J.-H. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Logic* 7.

Lee, J., and Wang, Y. 2016. Weighted rules under the stable model semantics. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning*, KR'16, 145–154. AAAI Press.

Li, Z.; Huang, J.; and Naik, M. 2023. Scallop: A language for neurosymbolic programming. *Proc. ACM Program. Lang.* 7(PLDI).

Lifschitz, V. 2019. *Answer Set Programming*. Springer Cham.

Manhaeve, R.; Dumančić, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2021. Neural probabilistic logic programming in DeepProbLog. *Artificial Intelligence* 298.

Manhaeve, R.; Marra, G.; and De Raedt, L. 2021. Approximate Inference for Neural Probabilistic Logic Programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, 475–486.

Mauá, D. D., and Cozman, F. G. 2023. Specifying credal sets with probabilistic answer set programming. In Miranda, E.; Montes, I.; Quaeghebeur, E.; and Vantaggi, B., eds., *Proceedings of the Thirteenth International Symposium on Imprecise Probability: Theories and Applications*, volume 215 of *Proceedings of Machine Learning Research*, 321–332. PMLR.

Muggleton, S. 2003. Learning structure and parameters of stochastic logic programs. In Matwin, S., and Sammut, C., eds., *Inductive Logic Programming*, 198–206.

Nickles, M., and Mileo, A. 2014. Probabilistic inductive logic programming based on answer set programming. In *Proceedings of the 15th International Workshop on Non-Monotonic Reasoning*.

Núñez, M.; Fidalgo, R.; and Morales, R. 2007. Learning in environments with unknown dynamics: Towards more robust concept learners. *Journal of Machine Learning Research* 8(11).

Pajunen, J., and Janhunen, T. 2021. Solution enumeration by optimality in answer set programming. *Theory and Practice of Logic Programming* 21(6):750–767.

Poole, D. 2008. The independent choice logic and beyond. *Probabilistic inductive logic programming* 222–243.

Przymusinski, T. 1991. Stable semantics for disjunctive programs. *New Generation Computing* 9:401–424.

Riguzzi, F. 2018. *Foundations of Probabilistic Logic Programming*. River Publishers Series in Software Engineering. River Publishers.

Rocha, V. H. N., and Cozman, F. G. 2022. A Credal Least Undefined Stable Semantics for Probabilistic Logic Programs and Probabilistic Argumentation. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*.

Sato, T., and Kameya, Y. 1997. PRISM: a language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2*, IJCAI'97, 1330–1335. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*.

Totis, P.; De Raedt, L.; and Kimmig, A. 2023. smProbLog: Stable model semantics in problog for probabilistic argumentation. *Theory and Practice of Logic Programming* 23(6):1198–1247.

Tuckey, D.; Russo, A.; and Broda, K. 2021. PASOCS: A parallel approximate solver for probabilistic logic programs under the credal semantics. *CoRR* abs/2105.10908.

Vennekens, J.; Denecker, M.; and Bruynooghe, M. 2009. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and practice of logic programming* 9(03):245–308.

Verbaeten, S., and Van Assche, A. 2003. Ensemble methods for noise elimination in classification problems. In *Multiple Classifier Systems: 4th International Workshop*, 317–325. Springer.

Yang, Z.; Ishay, A.; and Lee, J. 2020. NeurASP: embracing neural networks into answer set programming. In *Proceedings of the 39th International Joint Conference on Artificial Intelligence*.