Probabilistic Abductive Logic Programming using Dirichlet Priors

Calin Rares Turliuc[1], Luke Dickens[2], Alessandra Russo[1], and Krysia Broda[1]

[1] Department of Computing, Imperial College London, United Kingdom
[2] Department of Information Studies, University College London
{ct1810, a.russo, k.broda}@imperial.ac.uk, l.dickens@ucl.ac.uk

Abstract. Probabilistic logic programming has traditionally focused on languages where probabilities or weights are specified or inferred directly, rather than through Bayesian priors. To address this limitation, we propose a probabilistic logic programming language that bridges the gap between logical and probabilistic inference in categorical models with Dirichlet priors. The language is described in terms of its general plate model, syntax, semantics and the relation between the three. A prototype implementation is evaluated on two case studies: latent Dirichlet allocation (LDA) on synthetic data, where we compare it with collapsed Gibbs sampling, and repeated insertion model (RIM) on real data. Universal probabilistic programming is not always scalable beyond toy examples on some models. However, our promising results show that the inference yields similar results to state-of-the-art solutions reported in the literature, produced with model-specific implementations.

Keywords: probabilistic programming, Bayesian inference, abductive logic programming, latent Dirichlet allocation, repeated insertion model

1 Introduction

Probabilistic programming is an area of research that aims to generalize inference in probabilistic models specified as inputs to a programming language. In this context, evaluating the program corresponds to prediction with or inference on the described model. A wide range of probabilistic programming languages (PPLs) have been developed, based on different programming languages and expressing a variety of classes of probabilistic models. Examples of PPLs include Church [8], Anglican [18], BUGS [15], Stan [22] and Figaro [19].[1] While some PPLs, such as Church, typically enrich a functional programming language with exchangeable random primitives, there also exist logic based PPLs that add probabilistic annotations or primitives to a logical encoding of the model. This encoding usually relates either to first-order logic, e.g. Alchemy[6], BLOG [17] or to logic programming PPLs, e.g. PRiSM [21], ProbLog [7].

[1] For a more comprehensive list cf. http://probabilistic-programming.org/.

Typical PPLs based on functional programming can express a wide range of probabilistic models, and inference is based on general sampling algorithms. Existing logic based PPLs mostly focus on discrete probabilistic models, and, generally, they do not consider Bayesian inference with prior distributions. For instance, Alchemy is a PPL which implements Markov logic, encoding a first order knowledge base into a Markov random field. Here, uncertainty is expressed by weights on the logical formulae and one cannot specify prior distributions on the weights. ProbLog is a PPL that primarily targets the inference of conditional probabilities and the most probable explanation (maximum likelihood solution) and it does not feature the specification of prior distributions. PRiSM is a PPL which introduces conjugate Dirichlet priors over categorical distributions; however, it is limited to probabilistic models described at the abductive level by non-overlapping explanations, such as hidden Markov models and probabilistic context-free grammars.

These observations motivate our present paper: we develop a logic programming based PPL specialized on probabilistic models involving categorical variables with conjugate Dirichlet priors that can be encoded as abductive logic programs with overlapping explanations. The programs evaluated by our PPL are abductive logic programs [11] enriched with probabilistic definitions and inference queries. We consider as case studies the latent Dirichlet allocation (LDA, [2]) and the repeated insertion model (RIM, [5]).

The contributions of this paper are:

- the design of `peircebayes`, a logic programming based PPL for inference in discrete models with categorical variables and Dirichlet priors.
- the description of the class of probabilistic models that can be expressed in the PPL, and their relation to the language.
- a prototype implementation of the language. For probabilistic inference, we adapt the Gibbs sampling algorithm described in [10].
- the formulation of RIM [5] as a probabilistic program.
- the evaluation of the PPL on an LDA task with synthetic data and on a RIM task with real data.

The rest of the paper is organized as follows. In Section 2 we describe the class of probabilistic models supported by our PPL. Section 3 explains the key features of the syntax and semantics of the PPL. We present the results of two experiments with our PPL in Section 4. Finally, in Section 5 we relate our PPL to other PPLs and methods, and we conclude.

2 The Probabilistic Model

This section introduces `peircebayes`[2], referred to in the rest of the paper as PB, a probabilistic logic programming language designed for inference in a subclass of the class of models called "propositional logic-based probabilistic models", described in [10].

[2] Named, in Church style, after Charles Sanders Peirce, the father of logical abduction, and Thomas Bayes, the father of Bayesian reasoning. Pronounced ['pɜrs'beɪz].

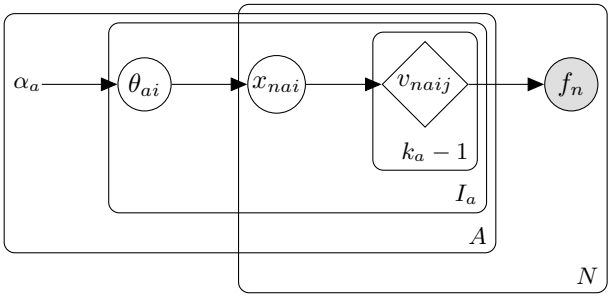

Fig. 1. The PB plate model. Unbounded nodes are constants, circle nodes are latent variables, shaded nodes are observed variables, diamond nodes are deterministic variables. A, I_a, N, and k_a are positive integers, $k_a \geq 2$.

The general plate notation [4] of the models expressible in PB is given in Figure 1. The plate model encodes the following (joint) probability distribution:

$$P(f, v, x, \theta; \alpha) = \left(\prod_{a=1}^{A} \prod_{i=1}^{I_a} P(\theta_{ai}; \alpha_a) \left(\prod_{n=1}^{N} P(x_{nai}|\theta_{ai}) P(v_{nai*}|x_{nai}) \right) \right) \prod_{n=1}^{N} P(f_n|v_{n*}) \quad (1)$$

We use $*$ to denote the set of variables obtained by iterating over the missing indexes, e.g. v_{nai*} is the set of all the variables v_{naij}, for $j = 1, \ldots, k_a - 1$, and v_{n*} is the set of all the variables v_{naij}, for $a = 1, \ldots, A$, $i = 1, \ldots, I_a$, $j = 1, \ldots, k_a - 1$. Unindexed variables are implicitly such sets, e.g. $x = x_*$.

In the model, each α_a, for $a = 1, \ldots, A$, is a vector of finite length $k_a \geq 2$ of positive real numbers. Each α_a may have a different length, and it represents the parameters of a Dirichlet distribution. From each such distribution I_a samples are drawn, i.e.:

$$\theta_{ai} \sim \text{Dirichlet}(\alpha_a) \qquad a = 1, \ldots, A \ , \ i = 1, \ldots, I_a$$

The samples θ are parameters of categorical distributions. Sampling N times from the latter yields:

$$x_{nai} \sim \text{Categorical}(\theta_{ai}) \qquad a = 1, \ldots, A \ , \ i = 1, \ldots, I_a \ , \ n = 1, \ldots, N$$

Each $x_{nai} \in \{1, \ldots, k_a\}$ is encoded, similarly to [20], as a set of propositional variables $v_{naij} \in \{0, 1\}$, for $j = 1, \ldots, k_a - 1$, in the following manner:

$$P(v_{nai*}|x_{nai} = l) = \begin{cases} \overline{v_{nai1}} \ldots \overline{v_{nail-1}} v_{nail} & , \text{ if } l < k_a \\ \overline{v_{nai1}} \ldots \overline{v_{nail-1}} & , \text{ if } l = k_a \end{cases}$$

where $\overline{v}$ denote boolean negation. Finally, the observed variables of the model, $f_n \in \{0, 1\}$, represent the output of boolean functions of v, such that:

$$P(f_n|v_{n*}) = [f_n = \text{Bool}_n(v_{n*})] \qquad n = 1, \ldots, N$$

$\text{Bool}_n(v)$ denotes an arbitrary boolean function of variables v, and $[i = j]$ is the Kronecker delta function δ_{ij}. The observed value for each f_n is 1 (or true) as we will explain in the following paragraph.

Inference in PB can be described in direct relation to a general schema of probabilistic inference, i.e. the characterization of $P(\theta|\Delta;\alpha)$, where θ are parameters of interest, α are constants (hyper-parameters) and Δ is some observed data. In PB, the parameters and the hyper-parameters correspond to θ and α, respectively. The observed data is captured by f and is assumed to be a set of N data points or observations. By convention, the realization $f_n = 1$ ensures that the n-th observation is included in the model, and, as such, we assume this is always the case. Furthermore, f_n is independent of the other observations given x (since x determines v), as implied by the joint distribution in Equation 1.

The various ways in which a data point can be generated, as well as the distributions involved in this process, are encoded through the boolean function $\text{Bool}_n(v_{n*})$ corresponding to the n-th data point. It is important to note that the data Δ can take any finite number of values, and $\text{Bool}_n(v_{n*})$ encodes the process of generating a single realization thereof.

Example. We illustrate the encoding of a popular probabilistic model for topic modelling, the latent Dirichlet allocation (LDA) [2] as a PB model. This will also serve as a running example throughout Section 3. LDA can be summarized as follows: given a corpus of D documents, each document is a list of tokens, the set of all tokens in the corpus is the vocabulary, with size V and assume there exist T topics. There are two sets of categorical distributions: D distributions over T categories, each distribution indexed μ_d, and T distributions over V categories, each distribution indexed ϕ_t. The words of a document d are produced independently by sampling a topic t from μ_d, then sampling a word from ϕ_t. Furthermore, each distribution in μ is sampled using the same Dirichlet prior with parameters γ, and, similarly, each distribution in ϕ is sampled using β. Note that μ and ϕ correspond to the parameters θ in the general model, and γ and β correspond to α. Assume that there is a corpus with 3 documents, 2 topics and a vocabulary of 4 words. The plate notation of the PB model of LDA is given in Figure 2.

Let the first data point be the observation of the second word of the vocabulary in document 3. Then the associated boolean function is:

$$\text{Bool}_1(v_{1*}) = v_{15}\overline{v_{111}}v_{112} + \overline{v_{15}}v_{121}v_{122}$$

The literals v_{15} and $\overline{v_{15}}$ denote the choice, in document 3, of topic 1 and 2, respectively, and the conjunctions $\overline{v_{111}}v_{112}$ and $\overline{v_{121}}v_{122}$ denote the choice of the second word from topic 1 and 2, respectively. Note that, in Figure 2, even though all possible edges between deterministic nodes and f_n are drawn, not all the variables must affect the probability of f_n, for instance the value of $\text{Bool}_1(v_{1*})$ doesn't depend on the value of v_{13}.

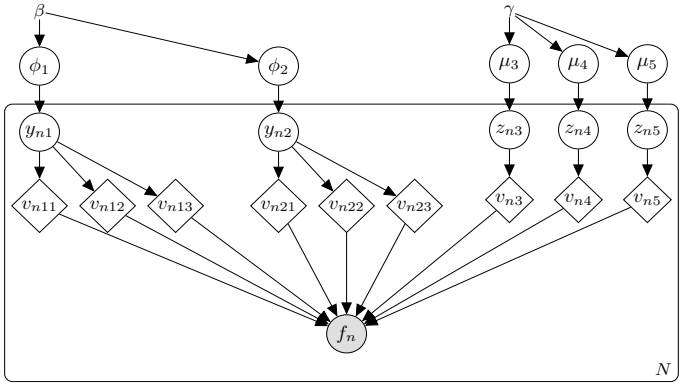

Fig. 2. The PB model for the LDA example.

3 Syntax and Semantics

Having established the semantics of the PB model, we proceed to describe the syntax and semantics of PB programs, and show how they relate to the probabilistic model.

A PB program is an abductive logic program [11] enhanced with probabilistic predicates. The abductive logic program encodes the generative story of the model, as well as the observed data. The most important probabilistic predicates are `pb_dirichlet` and `pb_plate`. The former provides a way to declare the probability distributions of the model, the latter is a query mechanism: it iterates through the data and computes all the possible ways it could have been generated according to the model, enabling the application of probabilistic inference algorithms.

The probabilistic predicate `pb_dirichlet` specifies a set of categorical distributions with the same Dirichlet prior, i.e. the elements on the outer plate indexed by a in Figure 1. Therefore, a set of such predicates express the whole outer plate. The syntax of the predicate is `pb_dirichlet(Alpha_a, Name, K_a, I_a)`. The first argument, `Alpha_a`, corresponds to α_a in the model, and can be either a list of k_a positive scalars specifying the parameters of the Dirichlet, or a positive scalar that specifies a symmetric prior. The second argument, `Name` is an atom that will be used as a functor when calling a predicate that represents a realization of a categorical random variable on the a-th plate. The third argument `K_a` corresponds to k_a, and `I_a` represents I_a, i.e. the number of categorical distributions having the same prior. The semantics of the predicate is that `Name` can be called in the program as a predicate, with the first argument denoting a category from $1, \ldots, k_a$, and the second argument a distribution from $1, \ldots, I_a$. In this paper, `Name(K_a, I_a)` is assumed to be a ground atom when called.

The probabilistic predicate `pb_plate(OuterQuery, Count, InnerQuery)` is the querying mechanism of PB. Informally, the first argument, `OuterQuery` is a usual Prolog query that iterates through the data. It must not call any predicates defined by `pb_dirichlet`. The argument `Count` is a positive integer that indicates that a particular observation is observed `Count` times. The final argu-

ment, `InnerQuery` is an abductive query that computes the explanation of the observations iterated in the `OuterQuery`. The formal semantics of `pb_plate` will be discussed after we introduce additional notation.

Example. Consider the LDA example from the previous section. Suppose we observe more data and we encode it in a PB program as:

```
observe(d(1),[(w(1),4),(w(4),2)]).
observe(d(2),[(w(3),1),(w(4),5)]).
observe(d(3),[(w(2),2)]).
```

Each `observe` fact encodes a document, indexed by an id using the first argument, and consisting of a bag-of-words in the second argument. The bag-of-words is a list of pairs: word index and its (positive) count per document. The next part of the PB program specifies the probability distributions as the following facts: `pb_dirichlet(1.0, mu, 2, 3)` and `pb_dirichlet(1.0, phi, 4, 2)`.

The `pb_plate` query iterates through document and word indexes and "explains" each such pair using the `generate` predicate. Note that `observe` and `generate` are not keywords, but descriptive conventional names.

```
pb_plate(
    [observe(d(Doc), TokenList), member((w(Token), Count), TokenList)],
    Count, [generate(Doc, Token)] ).

generate(Doc, Token) :- Topic in 1..2, mu(Topic, Doc), phi(Token, Topic).
```

Having described the core syntax of PB programs and its relation to the probabilistic model, we explain what is the result of executing a PB program and how probabilistic inference is performed to estimate $P(\theta|f, x, v; \alpha)$, or, in more typical applications, to produce an estimate $\hat{\theta}$.

In a traditional abductive logic programming setting [11,12,16], the result of evaluating a query is a list of abductive solutions, and an abductive solution is a list of abducibles. An abducible is a predicate that has no definition in the program. Some systems [20,23] represent an abductive solution as a pair of lists: a list of positive abducibles, i.e. abducibles that must be true, and a list of negative abducibles, i.e. abducibles that must be false. Since PB queries are abductive queries, an identical representation is obtained if the predicates defined by `pb_dirichlet` are parsed as annotated disjunctions [20].

Example. Consider the LDA example, more specifically the probabilistic predicate defining ϕ: `pb_dirichlet(1.0, phi, 4, 2)`. Let idx denote a positive integer such that `pa(Idx)` is a new abducible w.r.t. previously parsed `pb_dirichlet` predicates, i.e. the one defining μ. Abusing notation, $idx + incr$ is denoted by `Idx+incr`. The corresponding annotated disjunction is shown below:

```
phi(1,1) :- pa(Idx).
phi(2,1) :- \+pa(Idx), pa(Idx+1).
phi(3,1) :- \+pa(Idx), \+pa(Idx+1), pa(Idx+2).
phi(4,1) :- \+pa(Idx), \+pa(Idx+1), \+pa(Idx+2).
```

A similar annotated disjunction is produced for `phi(Token,2)`. Note that `pa(Idx)`, `pa(Idx+1)`, `pa(Idx+2)` represent v_{n11},v_{n12} and v_{n13} in Figure 2.

This encoding generates abductive representations of linear size in the number of categories of the distribution. In common LDA tasks, the size of the vocabulary of a corpus is frequently (much) more than 10.000 words, making it difficult to represent abductive solutions efficiently in a traditional way.

For the above reasons, PB uses a different representation of an abductive solution: a list of tuples (a, i, l), where a and i index the distribution as in the previous section, and l is a category $1 \leq l \leq k_a$. In the rest of the paper, the term "abductive solution" denotes this representation. The result of a PB query is a list of abductive solutions, and the result of calling `pb_plate` is a list of PB query results, one for each solution to `OuterQuery`. For the moment, consider programs with exactly one `pb_plate` definition. A generalization is presented once the necessary notation has been introduced.

Example. In the LDA program, the `OuterQuery` simply grounds `Doc` and `Token` in the order they are specified, e.g. the first grounding is $(1, 1)$, the second is $(1, 4)$, the third $(2, 3)$ etc. Assuming that square brackets represent lists, the result of `pb_plate` is shown below:

```
[   [ [(1,1,1),(2,1,1)], [(1,1,2),(2,2,1)] ],
    [ [(1,1,1),(2,1,4)], [(1,1,2),(2,2,4)] ],
    [ [(1,2,1),(2,1,3)], [(1,2,2),(2,2,3)] ],
    [ [(1,2,1),(2,1,4)], [(1,2,2),(2,2,4)] ],
    [ [(1,3,1),(2,1,2)], [(1,3,2),(2,2,2)] ] ]
```

The observation of word 2 in document 3 is the last element of the big list, and it can be produced using either topic 1 or 2, hence the two lists representing abductive solutions. The tuple $(1, 3, 1)$ means we choose topic 1 (last element) in document 3 (first two elements), and $(2, 1, 2)$ means we choose word 2 in topic 1 (with the same remarks).

Each result of a PB query is then parsed into a boolean formula which corresponds to $Bool_n(v_{n*})$ from the previous section. The key feature of PB is that it implicitly assumes that every result of a PB query on a `pb_plate` produces the same formula. This allows more concise query definitions, as well as improved time and memory performance, as a trade off with the user's expertise in PB. If the user were agnostic, she would write a `pb_plate` predicate for each data point (thus making the `OuterQuery` trivial). In future work we plan on investigating the automatic partition of the dataset in `pb_plate` definitions in an efficient way.

Example. The formula for the LDA example is $v_0v_1 + \overline{v_0}v_2$. Notice it is different from the one in Section 2, because the indexes have no semantic meaning w.r.t. the plate model, and the annotated disjunction is compiled w.r.t. the choices present in the same PB query, rather than the whole sample space.

The formula is compiled into a reduced ordered binary decision diagram (ROBDD, in the rest of the paper the RO attributes are implicit) [1,3], with the variables in ascending order according to their index. This means that the order of `pb_dirichlet` predicates matters, and that it should correspond to the sampling order in the generative story of the model. The BDD is the key data

structure that is used for probabilistic inference. The inference algorithm we use is an adaptation of Ishihata and Sato's Gibbs sampling for PLP models [10]. The algorithm is uncollapsed Gibbs sampling along two dimensions (θ and x).[3]

The difference between the original inference algorithm and the PB one is that instead of sampling from a Bernoulli, we sample from a multinomial with `Count` trials. We also sample all the (identical) BDDs for a `pb_plate` at once, using a single BDD, making sure to stop sampling a node when it isn't sampled in any of the implicit BDDs. Furthermore, sampling θ is followed by a re-parametrization such that the probabilities of the boolean variables in the BDD correspond to the new θ.

The generalization to multiple `pb_plate` predicates is straightforward: we sample each BDD, corresponding to one `pb_plate`, in turn, and all the samples update a common data structure representing x, and the probabilities of the boolean variables in each BDD are re-parametrized to adjust to the sampled θ.

In principle, it is possible to use the learned θ, or, to be more Bayesian, the posterior parameters of the Dirichlet α', to perform "forward" inference in a PB program: if we freeze θ, then the backward probability of the BDD yields the estimated parameter of a new observation. Otherwise, we sample x and θ using as priors α', record the backward probabilities of the BDD, then output the average thereof.

4 Evaluation

In this section we present experiments with PB[4]. No burn-in or lag was used in the experiments.

PB and collapsed Gibbs sampling (CGS) for LDA on synthetic data[5]. We run a variation of the experiment performed in [9,10]. A synthetic corpus is generated from an LDA model with parameters: 25 words in the vocabulary, 10 topics, 1000 documents, 100 words per document, and a symmetric prior on the mixture of topics μ, $\gamma = 1$. The topics used as ground truth specify uniform probabilities over 5 words, cf. [9,10]. We evaluate the convergence of PB and a traditional collapsed Gibbs sampling implementation[6]. The parameters are: $\beta = \gamma = 1$ as hyper-parameters, and we run 200 iterations of the samplers. The experiments are run 10 times over each corpus from a set of 10 identically sampled corpora, yielding 100 values of the log likelihoods per iteration. The average and 95% confidence interval (under a normal distribution) per iteration are shown in Figure 3. The experiment confirms the conclusions of the LDA experiment in [10]: both sampling algorithms converge, albeit PB converges slower

[3] The original Gibbs sampling for LDA [9] is collapsed Gibbs sampling along a number of dimensions equal to the number of words in the corpus.

[4] See supplementary materials for details on implementation and software availability (Appendix A).

[5] For details on likelihood formulation and comparison with the Church PPL, see supplementary materials (Appendices B and C).

[6] We use the `topicmodels` R package.

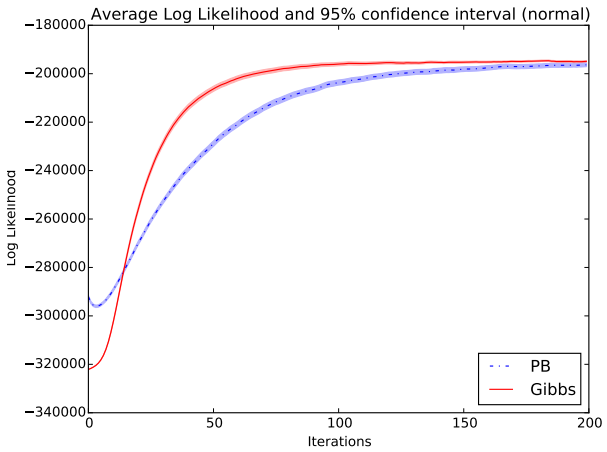

Fig. 3. Comparison between PB and collapsed Gibbs sampling on 10 sampled synthetic corpora (10 runs per corpus).

$\pi_1 = 0.144$	$\pi_2 = 0.191$	$\pi_3 = 0.153$	$\pi_4 = 0.185$	$\pi_5 = 0.187$	$\pi_6 = 0.138$
fatty tuna	fatty tuna	fatty tuna	fatty tuna	fatty tuna	fatty tuna
shrimp	tuna	sea urchin	sea urchin	tuna	tuna
sea eel	shrimp	salmon roe	salmon roe	shrimp	salmon roe
squid	squid	sea eel	shrimp	sea eel	shrimp
tuna	sea eel	tuna	sea eel	squid	squid
tuna roll	tuna roll	shrimp	squid	tuna roll	sea eel
salmon roe	egg	tuna roll	tuna roll	egg	tuna roll
sea urchin	cucumber roll	squid	tuna	salmon roe	sea urchin
egg	salmon roe	egg	egg	cucumber roll	egg
cucumber roll	sea urchin	cucumber roll	cucumber roll	sea urchin	cucumber roll

Table 1. Maximum likelihood preference profiles and mixture parameters on the Sushi dataset.

than CGS. However, we report different values for the log likelihood and note that PB takes 200 iterations rather than 100 to converge to a value that is close, under usual statistical assumptions, to the one produced by CGS.

PB for RIM on Sushi dataset. A repeated insertion model (RIM, [5]) provides a recursive and compact representation of K probability distributions, called preference profiles, over the set of all permutations of M items. This intuitively captures K different types of people with similar preferences. We evaluate a variant of the repeated insertion model in an experiment inspired by [14], on a dataset published in [13]. The data consists of 5000 permutations over $M = 10$ Sushi ingredients, each permutation expressing the preferences of a surveyed person. Following [14], we use $K = 6$ preference profiles, however we use the RIM rather than a Mallows model, and we train on the whole dataset. The parameters of the model are $50/K$ symmetric prior for the mixture of profiles, and 0.1 symmetric prior for all categorical distributions in all profiles. We run PB 10 times with 100 iterations and average the parameters. For each categorical

```
observe([5,0,3,4,6,9,8,1,7,2]).    observe([0,9,6,3,7,2,8,1,5,4]).
% ... 4998 'observe' facts ommited
pb_dirichlet(8.33333333333, pi, 6, 1).
pb_dirichlet(0.1, p2, 2, 6).       pb_dirichlet(0.1, p7, 7, 6).
pb_dirichlet(0.1, p3, 3, 6).       pb_dirichlet(0.1, p8, 8, 6).
pb_dirichlet(0.1, p4, 3, 6).       pb_dirichlet(0.1, p9, 9, 6).
pb_dirichlet(0.1, p5, 5, 6).       pb_dirichlet(0.1, p10, 10, 6).
pb_dirichlet(0.1, p6, 6, 6).
pb_plate( [observe(Sample)], 1,
    [generate([0,1,2,3,4,5,6,7,8,9], Sample)] ).
generate([H|T], Sample):-
    K in 1..6, pi(K, 1), generate(T, Sample, [H], 2, K).
generate([], Sample, Sample, _Idx, _K).
generate([ToIns|T], Sample, Ins, Idx, K) :-
    % insert next element at Pos yielding a new list Ins1
    append(_, [ToIns|Rest], Sample),
    insert_rim(Rest, ToIns, Ins, Pos, Ins1),
    % build prob predicate in Pred
    number_chars(Idx, LIdx), append(['p'], LIdx, LF),
    atom_chars(F, LF), Pred =.. [F, Pos, K],
    % call prob predicate and recurse
    pb_call(Pred), Idx1 is Idx+1,
    generate(T, Sample, Ins1, Idx1, K).
insert_rim([], ToIns, Ins, Pos, Ins1) :-
    append(Ins, [ToIns], Ins1), length(Ins1, Pos).
insert_rim([H|_T], ToIns, Ins, Pos, Ins1) :-
    nth1(Pos, Ins, H), nth1(Pos, Ins1, ToIns, Ins).
insert_rim([H|T] , ToIns, Ins, Pos, Ins1) :-
    \+member(H, Ins), insert_rim(T, ToIns, Ins, Pos, Ins1).
```

Table 2. PB program for a RIM with $K = 6$ preference profiles.

distribution in a profile, we select its maximum likelihood realization to build the corresponding maximum likelihood preference profile, shown in Table 1. The inference yields similar conclusions to [14]: there is a strong preference for fatty tuna, a strong dislike of cucumber roll and a strong positive correlation between salmon roe and sea urchin. We show the PB program used in Table 2, noting that `pb_call/1` is a special PB predicate that allows the evaluation of its argument as a predicate defined by `pb_dirichlet`. We are not aware of any other implementation of RIM in a PPL, therefore we briefly describe the program. The mixture of profiles is characterized by π, a set of K distributions, and for each profile there are $M - 1$ categorical distributions that specify the probabilities over the set of permutations of M elements. An observed permutation is produced by selecting a latent profile, then generating that permutation by consecutively inserting elements from an insertion order, e.g. $[0, 1, \ldots, 9]$, at the right position, according to the distributions in that profile. The right position is chosen using the `insert_rim` predicate, as naively generating all the possible permutations is intractable.

5 Related Work and Conclusions

In this paper, we introduced PB, a probabilistic logic programming language for categorical models with Dirichlet priors. The idea of PB was sparked by [10], which defines a similar class of probabilistic models and provides a Gibbs sampling algorithm for BDDs. However, BDDs are not a programming language, nor are they an intuitive representation for a non-expert. This paper bridges the gap between logical and probabilistic inference in the considered class of models, and addresses issues on representation of abductive solutions and inference on "syntactically" identical BDDs. Similarly to ProbLog [7], the pipeline of PB can be described as logical inference, followed by knowledge compilation, followed by probabilistic evaluation. Unlike ProbLog, the most difficult task in PB is probabilistic evaluation, rather than knowledge compilation, though for complex programs, PB could benefit from using more compact decision diagrams. In relation to Church [8] and many other related PPLs, PB is similar in that it uses a Turing-complete declarative language, but the set of probabilistic primitives available in PB is very restricted compared to Church. PB uses a different probabilistic model than Alchemy [6], and by using abductive logic programming instead of a first-order knowledge base, it can easily encode recursive generative models, such as RIM. Although in this paper we present well studied models, they can be easily adapted to include various constraints, e.g. seed words in LDA. In future work, we hope to explore more probabilistic models that fit the PB paradigm, and to design, implement, and compare efficient algorithms for generalized probabilistic inference.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. 27(6), 509–516 (Jun 1978), http://dx.doi.org/10.1109/TC.1978.1675141
2. Blei, D.M., Ng, A.Y., Jordan, M.I., Lafferty, J.: Latent dirichlet allocation. Journal of Machine Learning Research 3, 2003 (2003)
3. Bryant, R.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on C-35(8), 677–691 (Aug 1986)
4. Buntine, W.L.: Operations for learning with graphical models. J. Artif. Intell. Res. (JAIR) 2, 159–225 (1994), http://dx.doi.org/10.1613/jair.62
5. Doignon, J.P., Peke, A., Regenwetter, M.: The repeated insertion model for rankings: Missing link between two subset choice models. Psychometrika 69(1), 33–54 (2004), http://dx.doi.org/10.1007/BF02295838
6. Domingos, P., Kok, S., Poon, H., Richardson, M., Singla, P.: Unifying logical and statistical AI. In: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1. pp. 2–7. AAAI'06, AAAI Press (2006), http://dl.acm.org/citation.cfm?id=1597538.1597540
7. Fierens, D., den Broeck, G.V., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., Raedt, L.D.: Inference and learning in probabilistic logic programs using weighted boolean formulas. CoRR abs/1304.6810 (2013), http://arxiv.org/abs/1304.6810

8. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. Uncertainty in Artificial Intelligence 2008 (2008), http://danroy.org/papers/church_GooManRoyBonTen-UAI-2008.pdf
9. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National Academy of Sciences 101(suppl 1), 5228–5235 (2004), http://www.pnas.org/content/101/suppl_1/5228.abstract
10. Ishihata, M., Sato, T.: Bayesian inference for statistical abduction using Markov chain Monte Carlo. In: Proceedings of the 3rd Asian Conference on Machine Learning, ACML 2011, Taoyuan, Taiwan, November 13-15, 2011. pp. 81–96 (2011), http://www.jmlr.org/proceedings/papers/v20/ishihata11/ishihata11.pdf
11. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming (1993)
12. Kakas, Antonis, C., Van Nuffelen, B., Denecker, M.: A-system : Problem solving through abduction. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence. vol. 1, pp. 591–596. Morgan Kaufmann Publishers, Inc (2001), http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=34862
13. Kamishima, T., Kazawa, H., Akaho, S.: Supervised ordering - an empirical survey. In: Data Mining, Fifth IEEE International Conference on. pp. 4 pp.– (Nov 2005)
14. Lu, T., Boutilier, C.: Effective sampling and learning for Mallows models with pairwise-preference data. Journal of Machine Learning Research 15, 3783–3829 (2014), http://jmlr.org/papers/v15/lu14a.html
15. Lunn, D., Spiegelhalter, D., Thomas, A., Best, N.: The BUGS project: Evolution, critique and future directions. Statistics in Medicine 28(25), 3049–3067 (2009), http://dx.doi.org/10.1002/sim.3680
16. Ma, J.: Abductive reasoning module for SICStus prolog (2012), http://www-dse.doc.ic.ac.uk/cgi-bin/moin.cgi/abduction
17. Milch, B., Marthi, B., Russell, S.: Blog: Relational modeling with unknown objects. In: ICML 2004 Workshop on Statistical Relational Learning and Its Connections. pp. 67–73 (2004)
18. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. CoRR abs/1403.0504 (2014), http://arxiv.org/abs/1403.0504
19. Pfeffer, A.: Figaro: An object-oriented probabilistic programming language (2009)
20. Raedt, L.D., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI. pp. 2462–2467 (2007), http://dblp.uni-trier.de/db/conf/ijcai/ijcai2007.html#RaedtKT07
21. Sato, T., Kameya, Y.: New advances in logic-based probabilistic modeling by prism. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.) Probabilistic Inductive Logic Programming, Lecture Notes in Computer Science, vol. 4911, pp. 118–155. Springer Berlin Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78652-8_5
22. Stan Development Team: Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0 (2014), http://mc-stan.org/
23. Turliuc, C., Maimari, N., Russo, A., Broda, K.: On minimality and integrity constraints in probabilistic abduction. In: Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings. pp. 759–775 (2013), http://dx.doi.org/10.1007/978-3-642-45221-5_51

A Implementation

PB is implemented in YAP and Python (2.7), and is currently available as a command-line script. YAP is used to parse input files and produce files for probabilistic inference (e.g. solutions to each `pb_plate` query, information on the probability distributions). PyCUDD is used to compile ROBDDs and computationally intensive parts of the sampling algorithm are implemented in Cython. This prototype implementation and any additional files are released under a GNU General Public License (GPL3).

For more information and documentation see:
`http://raresct.github.io/peircebayes`
To access the source code see:
`http://www.github.com/raresct/peircebayes`
To reproduce the experiments, more concretely Figure 3 and Table 1, see:
`http://www.github.com/raresct/peircebayes_experiments`
On an Intel® Core™ i7-4710HQ CPU @ 2.50GHz ×8, the LDA experiment took: $\approx$ 265 minutes for PB, $\approx$ 4 minutes for CGS, and the RIM experiment took $\approx$ 10 minutes. Note that there is significant overhead for PB because we don't measure only sampling time, but also logical inference and knowledge compilation.

B A Note on the Joint Distribution of the PB model and Likelihood for LDA

The joint distribution of collapsed PB models is:

$$P(f, v, x; \alpha) = P(f|v)P(v|x)P(x; \alpha)$$

If x is the result of sampling the appropriate BDDs, then $P(f|v)P(v|x) = 1$, and the joint distribution reduces to $P(x; \alpha)$. This type of distribution has been well studied, cf. equations 2 and 3 in [9] for LDA, and in the case of PB models, it is:

$$P(x; \alpha) = \prod_{a=1}^{A} \left(\left(\frac{\Gamma(\sum_{l=1}^{k_a} \alpha_{al})}{\prod_{l=1}^{k_a} \Gamma(\alpha_{al})} \right)^{I_a} \prod_{i=1}^{I_a} \frac{\prod_{l=1}^{k_a} \Gamma(\sum_{n=1}^{N} [x_{nai} = l] + \alpha_{al})}{\Gamma(\sum_{n=1}^{N} x_{nai} + \sum_{l=1}^{k_a} \alpha_{al})} \right)$$

The likelihood for LDA is recovered by using the factors of the joint distribution involving only β, ϕ, y.

C PB and Church on LDA

In Section 4 we compare PB with CGS on a synthetic LDA task. We add to the comparison a much more expressive, universal PPL called Church [8]. The experimental setting differs from the LDA experiment in Section 4 in that we

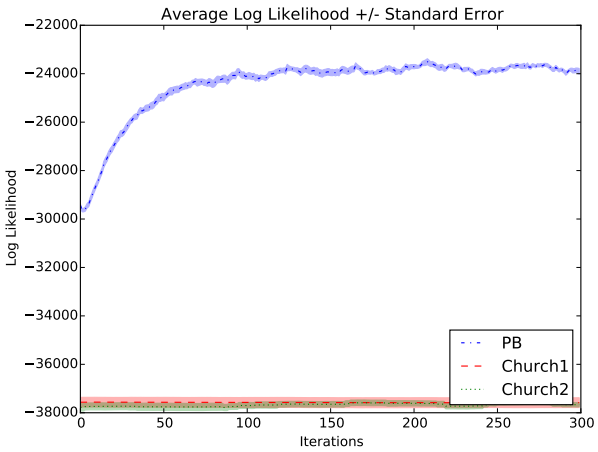

Fig. 4. Comparison between PB and Church on one sampled synthetic corpus (10 runs per corpus).

sample only one synthetic corpus of 100 documents. This is due to the fact that the Church implementation of LDA is slow. Furthermore, we take 300 samples (no lag, no burn-in for PB, 10 lag, no burn-in for Church). We use two implementations of LDA in Church[7], and report the results in Figure 4. Note that we use the uncollapsed likelihood for the Church models (which is more "optimistic" than the collapsed one), mainly due to the fact that we were unable to find an implementation of the $\log \Gamma$ function in Church.

Neither Church LDA programs seems to converge, while PB behaves consistently with the previous experiment. The average time per run is: ≈ 0.36 minutes for PB, ≈ 13.5 minutes for Church1 and ≈ 16.7 minutes for Church2.

[7] We use adaptations of the two programs shown here: `http://forestdb.org/models/lda.html` and run them with `webchurch` (`https://github.com/probmods/webchurch`) as command-line scripts.