Probabilistic Abductive Logic Programming using Dirichlet Priors

Calin Rares Turliuc[a,*], Luke Dickens[b], Alessandra Russo[a], Krysia Broda[a]

[a] *Department of Computing, Imperial College London, United Kingdom*
[b] *Department of Information Studies, University College London, United Kingdom*

Abstract

Probabilistic programming is an area of research that aims to develop general inference algorithms for probabilistic models expressed as probabilistic programs whose execution corresponds to inferring the parameters of those models. In this paper, we introduce a probabilistic programming language (PPL) based on abductive logic programming for performing inference in probabilistic models involving categorical distributions with Dirichlet priors. We encode these models as abductive logic programs enriched with probabilistic definitions and queries, and show how to execute and compile them to boolean formulas. Using the latter, we perform generalized inference using one of two proposed Markov Chain Monte Carlo (MCMC) sampling algorithms: an adaptation of uncollapsed Gibbs sampling from related work and a novel collapsed Gibbs sampling (CGS). We show that CGS converges faster than the uncollapsed version on a latent Dirichlet allocation (LDA) task using synthetic data. On similar data, we compare our PPL with LDA-specific algorithms and other PPLs. We find that all methods, except one, perform similarly and that the more expressive the PPL, the slower it is. We illustrate applications of our PPL on real data in two variants of LDA models (Seed and Cluster LDA), and in the repeated insertion model (RIM). In the latter, our PPL yields similar conclusions to inference with EM for Mallows models.

Keywords: probabilistic programming, abductive logic programming, Markov Chain Monte Carlo, latent Dirichlet allocation, repeated insertion model

*Corresponding author.
Email address: `ct1810@imperial.ac.uk` (Calin Rares Turliuc)

1. Introduction

Probabilistic programming is an area of research that aims to develop general inference algorithms for probabilistic models expressed as probabilistic programs whose execution corresponds to inferring the parameters of the probabilistic model. A wide range of probabilistic programming languages (PPLs) have been developed to express a variety of classes of probabilistic models. Examples of PPLs include Church [1], Anglican [2], BUGS [3], Stan [4] and Figaro [5].[1] Some PPLs, such as Church, enrich a functional programming language with exchangeable random primitives, and can typically express a wide range of probabilistic models. However, inference is not always tractable in these expressive languages. Other PPLs are logic-based. They typically 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 [7] or to logic programming, e.g. PRiSM [8], ProbLog [9]. Most logic-based PPLs focus only on discrete models, and consequently are equipped with more specialized inference algorithms, with the advantage of making the inference more tractable.

However, logic-based PPLs generally do not consider Bayesian inference with prior distributions. For instance, Alchemy implements Markov logic, encoding a first-order knowledge base into Markov random fields. Uncertainty is expressed by weights on the logical formulas, but it is not possible to specify prior distributions on these weights. ProbLog is a PPL that primarily targets the inference of conditional probabilities and the most probable explanation (maximum likelihood solution); it does not feature the specification of prior distributions on categorical distributions. PRiSM is a PPL which introduces Dirichlet priors over categorical distributions and is deigned for efficient inference in models with non-overlapping explanations.

This paper contributes to the field of logic-based PPL by proposing an alternative approach to probabilistic programming. Specifically, we introduce a PPL based on logic programming for performing inference in probabilistic models involving categorical distributions with Dirichlet priors. We encode these models as abductive logic programs [10] enriched with probabilistic definitions and inference queries, such that the result of abduction allows overlapping explanations. We propose two Markov Chain Monte Carlo (MCMC) sampling algorithms for the PPL: an adaptation of the uncollapsed

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

Gibbs sampling algorithm, described in [11], and a newly developed collapsed Gibbs sampler. Our PPL is similar to PRiSM and different from ProbLog in that it can specify Dirichlet priors. Unlike PRiSM, but similarly to ProbLog, we allow overlapping explanations. However, in this paper, all the models we study have non-overlapping explanations.

We show how our PPL can be used to perform inference in two classes of probabilistic models: Latent Dirichlet Allocation (LDA, [12]), a well studied approach for topic modelling, including two variations thereof (Seed LDA and Cluster LDA); and the repeated insertion model (RIM, [13]), a model used for preference modelling and whose generative story can be expressed using recursion. Our experiments demonstrate that our PPL can express a broad class of models in a compact way and scale up to medium size real-data sets, such as LDA with approximately 5000 documents and 100 words per document. On synthetic LDA data, we compare our PPL with two LDA-specific algorithms: collapsed Gibbs sampling (CGS) and variational expectation maximization (VEM), and two state-of-the-art PPLs: Stan and PRiSM. We find that all methods, with the exception of the chosen VEM implementation, perform similarly, and that the more expressive the method, the slower it is, in the following order, with the exception of VEM: CGS (fastest), PRiSM, VEM, our PPL, Stan (slowest).

The paper is organised as follows. Section 2 presents the class of probabilistic models supported by our PPL. In Section 3 we outline the syntax and the semantics of the PPL, whereas our two Gibbs sampling algorithms are discussed in Section 4. Section 5 shows our experimental results and Section 6 relates our PPL to other PPLs and methods. Finally, Section 7 concludes the paper.

2. The Probabilistic Model

This section begins with the description of a particular approach to probabilistic programming. Then we introduce `peircebayes`[2] (PB), our probabilistic abductive logic programming language designed to perform inference in discrete models with Dirichlet priors. Throughout the paper we will use normal font for scalars (α), arrow notation for vectors ($\vec{\alpha}$), and bold font for collections with multiple indexes ($\boldsymbol{\alpha}$), e.g. sets of vectors.

[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].

Probabilistic programs, as defined in [14], are " 'usual' programs with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observe statements". We can instantiate this general definition by considering the notions of hyperparameters $\boldsymbol{\alpha}$, parameters $\boldsymbol{\theta}$, and observed variables $\vec{f}$ and assuming the goal to be the characterisation of the conditional distribution $P(\boldsymbol{\theta}|\vec{f};\boldsymbol{\alpha})$.

Most PPLs assume such a conditional with a continuous sample space, i.e. they allow, in principle, probabilistic programs with an uncountably infinite number of unique outputs, should one not take into account issues of real number representations. In our approach, the conditional sample space is assumed to be finite, i.e. one can enumerate all possible executions. Specifically, in our PPL we restrict the class of distributions from which we can draw to categorical distributions with Dirichlet priors. The Dirichlet priors are chosen for their conjugacy, which supports efficient marginalisation.

The generality of our PPL is not given by the range of probability distributions that we can draw from, but rather by the way the draws from categorical distributions interact in the "generative story" of the model. We choose our "usual programs" to be abductive logic programs enriched with probabilistic primitives. Similarly to Church, Anglican and other PPLs this is declarative programming language, but one in which the generative story is expressed as an abductive reasoning task responsible for identifying relevant draws from the categorical distributions given the observations. Our choice is motivated by the significant amount of related work in probabilistic logic programming, although both functional and logic programming are Turing complete, so they are equally general. In what follows we present the class of probabilistic models that are supported by our PPL. We define them first as uncollapsed models, then show how they can be collapsed. As demonstrated in Section 4, this dual formalisation leads naturally to the possibility of using in our PPL uncollapsed as well as collapsed MCMC sampling methods.

2.1. The Uncollapsed PB model

The class of probabilistic models that can be expressed in PB, inspired by "propositional logic-based probabilistic (PLP) models"[11], is depicted in Figure 1, using the general plate notation [15]. In the figure, unbounded nodes are constants, circle nodes are latent variables, shaded nodes are observed variables, diamond nodes are nodes that can be (but are not always)

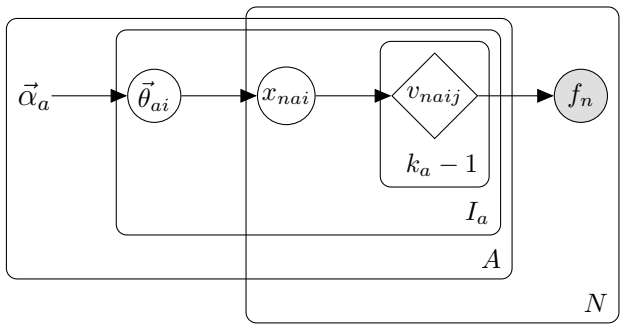


Figure 1: The PB plate model.

deterministic given their parent x (we discuss the details below), and A, I_a, N, and k_a are positive integers, with $k_a \geq 2$.
The above plate model encodes the following (joint) probability distribution:

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

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

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

$$\vec{\theta}_{ai}|\vec{\alpha}_a \sim \text{Dirichlet}(\vec{\alpha}_a) \qquad a = 1, \ldots, A \ , \ i = 1, \ldots, I_a$$

The samples $\vec{\theta}_{ai}$ are parameters of categorical distributions. For each i and a, there are N samples x_{nai} from the associated categorical distribution of the form:

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

Each $x_{nai} \in \{1, \ldots, k_a\}$ is encoded, similarly to [16], as a set of propositional variables $v_{naij} \in \{0, 1\}$, for $j = 1, \ldots, k_a - 1$, in the following manner:

$$P(\vec{v}_{nai*}|x_{nai}=l) = \begin{cases} 2^{l-(k_a-1)}\overline{v_{nai1}}\dots\overline{v_{nail-1}}v_{nail} & \text{, if } l<k_a \\ \overline{v_{nai1}}\dots\overline{v_{nail-1}} & \text{, if } l=k_a \end{cases} \tag{2}$$

where $\overline{v}$ denotes boolean negation, and $2^{l-(k_a-1)}$ is a normalization constant.

Note that not all variables in $\boldsymbol{v}$ are deterministic, more specifically for all j such that $l<j<k_a$, $\vec{v}_{naij}$ are not determined by the realization $x_{nai}=l$. Our presentation deviates from [11] in that we present both $\boldsymbol{x}$ and $\boldsymbol{v}$ in the same model, rather than considering two types of equivalent models ("base models" and "PLP models"). But the probabilistic semantics of $\boldsymbol{v}$ is the same as the one derived from the annotated disjunction (AD) compilation (cf. Section 3.3.1 of [17]). In our PB model:

$$\begin{aligned} &P(v_{nai1}=0,\dots,v_{nail-1}=0,v_{nail}=1|\vec{\theta}_{ai}) = \\ &\qquad = P(v_{nail}=1|\vec{\theta}_{ai})^{1-[l=k_a]}\prod_{j=1}^{l-1}\Big(1-P(v_{naij}=1|\vec{\theta}_{ai})\Big) = \theta_{ail} \end{aligned} \tag{3}$$

$$a=1,\dots,A \qquad i=1,\dots,I_a \qquad n=1,\dots,N \qquad l=1,\dots,k_a$$

where $[i=j]$ is the Kronecker delta function δ_{ij}. Therefore, it follows that:

$$P(v_{nail}=1|\vec{\theta}_{ai}) = \frac{\theta_{ail}}{\prod_{j=1}^{l-1}\Big(1-P(v_{naij}=1|\vec{\theta}_{ai})\Big)}$$

Example 2.1. For the reader unfamiliar with ADs, we offer a brief example. Let there be a four-sided die with probabilities 0.3, 0.2, 0.4, 0.1. The AD compilation involves three boolean variables, with probabilities computed as defined in Equation (3): 0.3, $\frac{0.2}{1-0.3}\approx 0.285$, $\frac{0.4}{(1-0.3)(1-0.285)}\approx 0.8$. To recover the original probabilities, we use Equation (2), e.g. $0.1 \approx (1-0.3)(1-0.285)(1-0.8)$. □

The observed variables of the model, $f_n \in \{0,1\}$, represent the output of boolean functions of v, such that:

$$P(f_n|\boldsymbol{v}_{n*}) = [f_n = \text{Bool}_n(\boldsymbol{v}_{n*})] \qquad n=1,\dots,N$$

where $\text{Bool}_n(\boldsymbol{v})$ denotes an arbitrary boolean function of variables $\boldsymbol{v}$. In our approach it is assumed the observed value for each f_n to be 1 (or true).

Inference in PB can be described in terms of a general schema of probabilistic inference presented at the beginning of this section, i.e. the characterization of $P(\boldsymbol{\theta}|\vec{f};\boldsymbol{\alpha})$. The parameters and the hyper-parameters correspond to $\boldsymbol{\theta}$ and $\boldsymbol{\alpha}$, respectively. The observed data $\vec{f}$ is a vector of N data points (observations) where, by convention, $f_n = 1$ ensures that the n-th observation is included in the model (assume this to be always the case). Furthermore, observation f_n is independent of any other observations $f_{n'}$, $n \neq n'$, if conditioned on $\boldsymbol{x}$ (since $\boldsymbol{x}$ determines $\boldsymbol{v}$); this is implied by the joint distribution given in Equation (1). The various ways in which an n-th data point can be generated, as well as the distributions involved in this process, are encoded through the boolean function $\text{Bool}_n(\boldsymbol{v}_{n*})$.

Example 2.2. LDA as a PB model

We illustrate the encoding of a popular probabilistic model for topic modelling, the latent Dirichlet allocation (LDA) [12], 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 there exist T topics. There are two sets of categorical distributions: D distributions over T categories (one topic distribution per document), each distribution parametrized by $\vec{\mu}_d$; and T distributions over V categories, (one token distribution per topic), each distribution parametrized by $\vec{\phi}_t$. The tokens of a document d are produced independently by sampling a topic t from $\vec{\mu}_d$, then sampling a token from $\vec{\phi}_t$. Furthermore, each distribution in $\boldsymbol{\mu}$ is sampled using the same Dirichlet prior with parameters $\vec{\gamma}$, and, similarly, each distribution in $\boldsymbol{\phi}$ is sampled using $\vec{\beta}$. Note that $\{\boldsymbol{\mu}, \boldsymbol{\phi}\}$ correspond to the parameters $\boldsymbol{\theta}$ in the general model, and $\{\vec{\gamma}, \vec{\beta}\}$ correspond to $\boldsymbol{\alpha}$. To instantiate a minimal LDA model, let us consider a corpus with 3 documents, 2 topics and a vocabulary of 4 tokens. The plate notation of the PB model for this minimal LDA, depicted in full, is given in Figure 2. Relating to the PB model, we have $A = 2$, i.e. one plate for each of $\vec{\gamma}$ and $\vec{\beta}$, $I_1 = 3$, one topic mixture for each of the 3 documents, $I_2 = 2$ and $k_1 = 2$ for the two topics, and $k_2 = 4$ for the 4 tokens in the vocabulary.

Let the first data point to be the observation of token 2 in document 3. Then the associated boolean function is:

$$\text{Bool}_1(\boldsymbol{v}_{1*}) = v_{13}\overline{v_{141}}v_{142} + \overline{v_{13}v_{151}}v_{152} \tag{4}$$

The literals v_{13} and $\overline{v_{13}}$ denote the choice, in document 3, of topic 1 and 2,

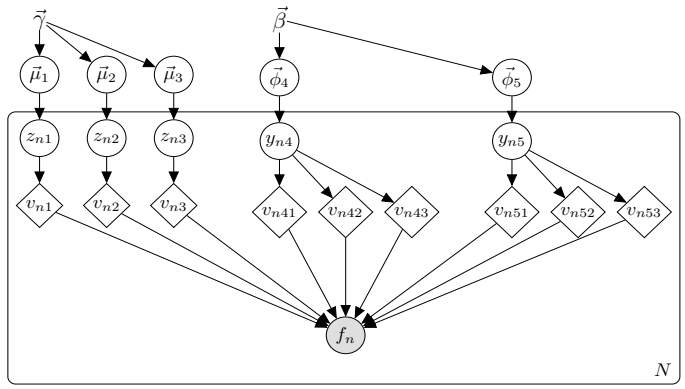


Figure 2: The PB model for an LDA example with 3 documents, 2 topics and 4 tokens.

respectively, and the conjunctions $\overline{v_{141}}v_{142}$ and $\overline{v_{151}}v_{152}$ denote the choice of the second token 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 will necessarily and/or simultaneously affect the probability of f_n. For instance, the value of $\text{Bool}_1(\boldsymbol{v}_{1*})$ doesn't depend on the value of v_{12}, which is related to document 2 and observations about document 3 and document 2 are independent. Also, $\text{Bool}_1(\boldsymbol{v}_{1*})$ cannot depend at the same time on both v_{141} and v_{151}, since the truth value of v_{13} effectively filters out the effect of one or the other. □

2.2. The Collapsed PB model

The above model, described as uncollapsed, can be inefficient to sample from, due to the large number of variables. The same PB model can be reformulated as collapsed model where the parameters of the categorical distributions are marginalised out. This is straightforward since we assume conjugate priors. We present the collapsed model here in order to introduce the distributions used in the derivation of the collapsed Gibbs sampler (CGS) given in Section 4.

Consider a and i fixed. Since we make multiple draws from a categorical distribution parametrized by $\vec{\theta}_{ai}$ it is convenient to work with count summaries. Therefore we overload the notation $\vec{x}_{*ai}$ to denote a k_a sized vector of counts, i.e. for all categories $l = 1, \ldots, k_a$, $x_{*ail} = \sum_{n=1}^{N}[x_{nai} = l]$. Similarly, $\boldsymbol{x}$ is overloaded to denote a set of such vectors for each $a = 1, \ldots, A$, $i = 1, \ldots, I_a$. Integrating out $\vec{\theta}$ for a single Dirichlet-categorical pair yields:

$$P(\vec{x}_{*ai};\vec{\alpha}_a) = \frac{\Gamma(\Sigma(\vec{\alpha}_a))}{\Gamma(\Sigma(\vec{\alpha}_a)+\Sigma(\vec{x}_{*ai}))}\prod_{l=1}^{k_a}\frac{\Gamma(x_{*ail}+\alpha_{al})}{\Gamma(\alpha_{al})}$$

where $\Sigma(\vec{v})$ denotes the sum of the elements of some vector $\vec{v}$ and Γ denotes the gamma function. Also, from the conditional independence of our x_{*ai}:

$$P(\boldsymbol{x};\boldsymbol{\alpha}) = \prod_{a=1}^{A}\prod_{i=1}^{I_a} P(\vec{x}_{*ai};\vec{\alpha})$$

The joint distribution of the collapsed PB model becomes:

$$P(\vec{f},\boldsymbol{v},\boldsymbol{x};\boldsymbol{\alpha}) = P(\boldsymbol{x};\boldsymbol{\alpha})P(\boldsymbol{v}|\boldsymbol{x})P(\vec{f}\,|\boldsymbol{v}) \tag{5}$$

where

$$P(\boldsymbol{v}|\boldsymbol{x}) = \prod_{a=1}^{A}\prod_{i=1}^{I_a}\prod_{n=1}^{N} P(\vec{v}_{nai*}|x_{nai})$$

$$P(\vec{f}\,|\boldsymbol{v}) = \prod_{n=1}^{N} P(f_n|\boldsymbol{v}_{n*})$$

The joint distribution in Equation (5) is simply Equation (1) with $\boldsymbol{\theta}$ integrated out. Note that $\boldsymbol{v}$ must take values from the models of the formulas Bool_n in order to have $P(\vec{f}|\boldsymbol{v}) = 1$ (recall that $\vec{f}$ is observed). Furthermore, given a realization of $\boldsymbol{v}$, it is uniquely decoded to a realization of $\boldsymbol{x}$. In practice, we always make sure that these conditions are met, cf. Section 3.

In summary, inference in PB models means to characterize $P(\boldsymbol{\theta}|\vec{f},\boldsymbol{\alpha})$. Since Dirichlet priors are conjugate to categorical distributions, the posterior distributions are also Dirichlet distributions with parameters $\boldsymbol{\alpha}'$:

$$\vec{\theta}_{ai}|\vec{f},\boldsymbol{\alpha} \sim \text{Dirichlet}(\vec{\alpha}'_{ai}) \qquad a = 1,\ldots,A\ ,\ i = 1,\ldots,I_a$$

Therefore, the inference task is to estimate $\boldsymbol{\alpha}'$.

Informally, the PB inference task is computed in two steps. The first step, described in Section 3, consists of representing a given probabilistic model as an abductive logic program enriched with probabilistic primitives and executing this program. The latter yields the formulas Bool_n, and thus the PB model is completely specified. The second step consists of sampling the PB model and is described in Section 4.

3. Syntax and Semantics

In this section we formally define the syntax and semantics (up to MCMC sampling) of probabilistic programs in PB. PB programs will be defined as abductive logic programs where the set of abducibles correspond to the sample space of $P(\boldsymbol{x}|\boldsymbol{\theta})$. Each observation in the model corresponds to an abductive query. The execution of the abductive query given the program means explaining that observation, and the result of this execution will be a formula in terms of abducibles. This formula is then translated into a boolean formula Bool expressed in terms of the boolean variables $\boldsymbol{v}$ from the PB model. We argue that all observations need not be explained individually, i.e. by executing the previous steps for each observation, and instead propose a more efficient approach. Finally, we describe a compact encoding of each boolean formula Bool into a (reduced ordered) binary decision diagram, which will be used for MCMC sampling.

3.1. Abductive Logic Programming and PB

A PB program is an abductive logic program [10] enhanced with probabilistic predicates. Adapting the definitions from [18], an *abductive logic program* is a tuple (Π, AB), where Π is a normal logic program, AB is a finite set of ground atoms called *abducibles*. In PB, an abductive logic program encodes the generative story of the model, as well as the observed data.

A query Q is a conjunction of existentially quantified literals. In PB, there exists one query for each observation, describing how that observation should be explained. An abductive solution for a query Q is a set of ground abducibles $\Delta \subseteq AB$:

- $comp_3(\Pi \cup \Delta) \models Q$
- $comp_3(\Pi \cup \Delta) \models CET$

where CET denotes the Clark Equality Theory axioms [19], and $comp_3(\Pi)$ the Fitting three-valued completion of a program Π [20]. CET is needed to define the semantics of universally quantified negation in programs containing variables. The result of a query is the disjunction of all the abductive solutions computed by the abductive logic program, where each abductive solution is considered a conjunction of its elements. This computation is described in Appendix A.

The syntax of the non-probabilistic predicates is similar to Prolog, and is documented in [21]. The domain of the abducibles is specified through a probabilistic predicate `pb_dirichlet`. This predicate defines a set of categorical distributions with the same Dirichlet prior, i.e. it allows draws from $P(x_{nai}|\vec{\theta}_{ai})$, for a fixed a and all n and i. A conjunction of such predicates defines the plate indexed by a. In PB, the syntax of the predicate is `pb_dirichlet(Alpha_a, Name, K_a, I_a)`. The first argument, `Alpha_a`, corresponds to $\vec{\alpha}_a$ 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.

Example 3.1. Consider the LDA example from Example 2.2. To specify the probability distributions, assuming flat symmetric priors over $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$, we need the following predicates:

```
pb_dirichlet(1.0, mu, 2, 3).
pb_dirichlet(1.0, phi, 4, 2).
```

□

Declaring `pb_dirichlet` predicates simply states that the distributions on the a-indexed plate exist. The draws from these distributions, i.e. samples from $P(\boldsymbol{x}|\boldsymbol{\theta})$, are realized using a predicate `Name(K_a, I_a)`. The first argument denotes a category from $1, \ldots, k_a$, and the second argument a distribution from $1, \ldots, I_a$. Informally, the meaning of the predicate is that it draws a value `K_a` from the `I_a`-th distribution with Dirichlet prior parametrized by $\vec{\alpha}_a$. All predicates `Name(K_a, I_a)` are assumed to be ground atoms when called. Therefore the set of abducibles can be defined as:

$$AB = \{\texttt{Name(K\_a, I\_a)} | \forall a, \texttt{K\_a}, \texttt{I\_a}\}$$

Example 3.2. Continuing Example 3.1, we show the rest of the abductive logic program. Note that this is not the complete PB program, since we have not defined all probabilistic predicates.

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

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

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

Each `observe` fact encodes a document, indexed using the first argument, and consisting of a bag-of-words in the second argument. The bag-of-words is a list of pairs: a token with an index and its (positive) count per document.

The `generate` rule describes how each observation, characterized by a particular `Token` in a `Document`, is generated (or explained). The variable `Topic` is grounded as either 1 or 2, in general 1 up to T. Note that `observe` and `generate` are not keywords, but descriptive conventional names. In this example, the abducibles are the predicates with functors `mu` and `phi`. □

In PB, each abducible corresponds to a draw from $P(x_{nai}|\vec{\theta}_{ai})$, represented as a tuple (a, i, l) via a bijection $\rho : AB \rightarrow \{(a, i, l)|a \in \{1, \ldots, A\}, i \in \{1, \ldots, I_a\}, l \in \{1, \ldots, k_a\}\}$. The tuple consists of an index of the distribution (a, i) and the drawn category l. For an abducible `Name(K_a, I_a)`, `K_a` corresponds to l, `I_a` corresponds to i, and a is determined by the order of the `pb_dirichlet` definition involving `Name`. For instance, in the context of Example 3.1, $\rho($`phi(3,2)`$) = (2, 2, 3)$.

Abusing notation, the mapping ρ is also used to map abductive solutions, i.e. $\rho(\Delta)$ is a list of tuples (a, i, l).

The semantics of the abductive logic program ensures that abductive solutions are minimal, which means no abducibles are included in an abductive solution unless they are necessary to satisfy the query. Probabilistically, this means that no draws are made except the ones needed to explain an observation. This implicit marginalisation and compactness (e.g. queries that produce v_1 and $v_1 v_2 + v_1 \overline{v_2}$ are equivalent) is effectively enforced by BDD compilation, described in Section 3.2.

The PB model constrains the draws from the categorical distributions such that, for each observation, we can draw once per distribution, and since each observation is explained by an abductive query, we must enforce this constraint on our abductive solutions, i.e. there can be no (a, i, l_1), $(a, i, l_2) \in \rho(\Delta)$ such that $l_1 \neq l_2$.

Example 3.3. Continuing Example 3.2, consider observing token 2 in document 3. The corresponding query is `generate(3,2)`. The execution of the abductive logic program with this query produces two solutions corresponding to the explanations of the token, i.e. it can be produced either from topic 1 or topic 2. The solutions are $\{$`mu(1,3)`, `phi(2,1)`$\}$ and $\{$`mu(2,3)`, `phi(2,2)`$\}$.

We obtain the result of the query by applying ρ to the abductive solutions and taking their disjunction:

$$((1,3,1) \wedge (2,1,2)) \vee ((1,3,2) \wedge (2,2,2))$$

□

3.2. Knowledge compilation and multiple observations

The purpose of abduction is to produce, for each observation $n = 1, \ldots, N$, a formula $\text{Bool}_n(\boldsymbol{v}_{n*})$. Having computed the result of a query, all that is left is to translate each draw (a, i, l) into boolean variables. We define this translation, denoted conditional AD compilation, as ordering all draws by their distribution index (a, i), i.e. first by a then i, and compiling each distribution as an AD with respect to the categories present in the result of the query.

Example 3.4. Continuing Example 3.3, in the result of the query, we have the sorted distributions (a, i):

1. $(1, 3)$, with two outcomes present in the solution, i.e. $(1, 3, 1)$ and $(1, 3, 2)$, and one variable v_1.
2. $(2, 1)$, with one outcome present in the solution, i.e. $(2, 1, 2)$, and one variable v_2.
3. $(2, 2)$, with one outcome present in the solution, i.e. $(2, 2, 2)$, and one variable v_3.

Therefore the result of the query is parsed using conditional AD compilation into the formula:

$$v_1 v_2 + \overline{v_1} v_3$$

Note that for a fixed number of topics, this representation is invariant to V, the number of tokens in the vocabulary, due to the fact, when explaining an observation, we draw a single token from some topic. □

The proposed representation of conditional AD compilation is different from conventional AD compilation (cf. Section 3.3.1 of [17]) in that instead of considering the sample space of the distributions, it considers the conditional sample space of the distributions given a particular observation. In models such as LDA this difference is crucial, since in typical inference tasks, the size of the vocabulary V is of the order of tens of thousands of tokens, and AD compilation would thus create conjunctions of up to $V - 1$ variables.

So far we have discussed the explanation of a single observation, i.e. the computation of a single abductive query and the translation of the result of the query into a boolean formula. It would be inefficient to treat multiple observations by computing a query for every single observation. For instance, in the LDA example, the same token `Token` can be produced multiple times from the same document `Document`. The queries corresponding to these observations are identical: `generate(Document, Token)`, therefore it is redundant to execute the query more than once for all such observations. We make a further remark, namely that all observations generated from the same set of topics in an LDA task produce the same formula after conditional AD compilation, although the particular distributions used are different for different document-token pairs.

We exploit these properties when expressing queries in a PB program. To do so, we introduce the probabilistic predicate `pb_plate(OuterQ, Count, InnerQ)`, where the queries `OuterQ` and `InnerQ` are conjunctions represented as lists, and `Count` is a positive integer. For each successful solution of `OuterQ`, the `Count` argument and the arguments of `InnerQ` are instantiated, then the predicate executes the query `InnerQ`. We assume that all observations defined by a `pb_plate` predicate yield the same formula from the abductive solutions of `InnerQ`. If one didn't know whether the formulas are identical, one could simply write a `pb_plate` definition for each observation, with an empty outer query, and a count argument of 1.

Example 3.5. Continuing the previous examples of this section, we specify the last part of the PB program for an LDA task. The `pb_plate` predicate iterates through the corpus and generates each token according to the model. In this model, iterating through observations means selecting different pairs of document and token indexes.

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

```
    Count,
    [generate(Doc, Token)]
).
```

An example of a PB program with multiple `pb_plate` definitions is given for the Seed LDA model, discussed in Section 5, in Table 1. □

The formula for each `pb_plate` definition is compiled into a reduced ordered binary decision diagram (ROBDD, in the rest of the paper the RO attributes are implicit) [22, 23], with the variables in ascending order according to their index. The BDD of a boolean formula Bool allows one to express in a compact manner all the models of Bool. This enables us to sample (a subset of) the variables $\boldsymbol{v}$ such that $P(\vec{f}|\boldsymbol{v})$ equals 1. Furthermore, the conditional AD compilation allows us to correctly decode $\boldsymbol{v}$ into $\boldsymbol{x}$, such that all the deterministic constraints of the model are satisfied.

Compiling the abductive solutions into a BDD allow PB to perform inference in models with overlapping explanations, similarly to ProbLog.

The final step of inference in PB is sampling via Markov Chain Monte Carlo (MCMC), which we discuss in Section 4.

4. MCMC sampling

Inference in high-dimensional latent variable models is typically done using approximate inference algorithms, e.g. variational inference or MCMC sampling. We use the latter, in particular two algorithms: an adaptation to PB models of Ishihata and Sato's uncollapsed Gibbs sampling for PLP models [11], and our novel collapsed Gibbs sampling (CGS) for PB models.

4.1. Uncollapsed Gibbs sampling

Following [11], we can perform uncollapsed Gibbs sampling along two dimensions ($\boldsymbol{\theta}$ and $\boldsymbol{x}$) by alternatively sampling from $P(\boldsymbol{x}|\hat{\boldsymbol{\theta}}, \vec{f})$ and $P(\boldsymbol{\theta}|\hat{\boldsymbol{x}}, \boldsymbol{\alpha})$. We use hat to denote the samples in some iteration, and if the variable is a vector, we omit vector notation. A sample from $P(\boldsymbol{\theta}|\hat{\boldsymbol{x}}, \boldsymbol{\alpha})$ yields an estimate $\hat{\boldsymbol{\theta}}$ that is averaged over the sampling iterations to produce our posterior belief of $\boldsymbol{\theta}$.

Sampling from $P(\boldsymbol{\theta}|\hat{\boldsymbol{x}}, \boldsymbol{\alpha})$ means sampling:

$$\hat{\theta}_{ai} \sim \text{Dirichlet}(\vec{\alpha}_a + \hat{x}_{*ai}) \qquad a = 1, \ldots, A \ , \ i = 1, \ldots, I_a$$

Assume this is implemented by a function:

$$\boldsymbol{\theta} \leftarrow \textsc{sample\_theta}(\boldsymbol{\alpha}, \boldsymbol{x})$$

Sampling from $P(\boldsymbol{x}|\hat{\boldsymbol{\theta}}, \vec{f})$ can be done by sampling a path from root to the "true" leaf[3] in each of the BDDs for Bool_n, $n = 1, \ldots, N$, which yield a sample from $\boldsymbol{v}$ that is then decoded to $\boldsymbol{x}$. To sample a BDD, we sample the truth value of each node, and therefore we need to use $P(\boldsymbol{v}; \boldsymbol{\theta})$, as defined in Section 2.1.

According to the conditional AD compilation defined in Section 3.2, each compilation is performed for each individual observation, and the observations are grouped if they correspond to the same query, and grouped again when they correspond to the same compiled boolean formula Bool. This means that for each formula Bool, the probabilities of the boolean variables $P(\boldsymbol{v}; \boldsymbol{\theta})$ must be computed from $\boldsymbol{\theta}$ for each distinct query. Let N_{bdd} be the number of boolean variables in BDD *bdd*, and $N_{Q,bdd}$ be the number of distinct queries whose result yields the same formula Bool encoded in *bdd*. Then $\boldsymbol{\theta}_{bdd}$ is a $N_{Q,bdd}$ by N_{bdd} matrix, computed from $\boldsymbol{\theta}$ using a function:

$$\boldsymbol{\theta}_{bdd} \leftarrow \textsc{reparametrize}(\boldsymbol{\theta}, bdd)$$

The sampling algorithm is given in Algorithm 1, and the algorithm for sampling a single BDD (SAMPLE_X() from Algorithm 1) is explained in Appendix B.

In Algorithm 1, the input to the sampler are the priors $\boldsymbol{\alpha}$, a set of BDDs *bdds*, and a number of iterations *maxit*. We define some additional notation: we use [ ] to denote an empty list, APPEND(*list*, *el*) to append element *el* to a list *list*, ZEROS() to create empty vectors or matrices of shapes specified by the arguments and AVG() to take the average of a vector or list. To update $\boldsymbol{x}$, we must sample all BDDs (one for each `pb_plate` definition), and all observations within each BDD. We assume that an observation *obs* takes values in $1, \ldots, N_{Q,bdd}$, and *count* represents the number of times the observation is repeated, as specified by the `Count` argument of a `pb_plate` predicate. For each observation *obs* we will use only the *obs*-th row from $\boldsymbol{\theta}_{bdd}$, denoted as $\boldsymbol{\theta}_{bdd}[obs, :]$.

[3]The "true" leaf of a BDD is a node such that the paths from the root to it encode all models of the formula represented by the BDD.

Algorithm 1 Uncollapsed Gibbs sampling for PB.

```
function UNCOLLAPSED_GIBBS(α, bdds, maxit)
    samples ← [ ]
    for it = 1, ..., maxit do
        θ ← SAMPLE_THETA(α, x)                          ▷ updates θ
        for bdd ∈ bdds do
            θ_bdd ← REPARAMETRIZE(θ)
        end for
        for a = 1, ..., A do                            ▷ resets x
            for i = 1, ..., I_a do
                x⃗_*ai ← ZEROS(k_a)
            end for
        end for
        for bdd ∈ bdds do                               ▷ updates x
            for (obs, count) ∈ bdd do
                θ_obs ← θ_bdd[obs, :]
                SAMPLE_X(bdd, count, θ_obs)
            end for
        end for
        samples ← APPEND(samples, θ)
    end for
    return AVG(samples)
end function
```

Algorithm 2 Collapsed Gibbs sampling for PB.

```
function COLLAPSED_GIBBS(α, O, maxit)
    samples ← [ ]
    x ← INITIALIZE(α)
    for it = 1, ..., maxit do
        for (obs, bdd) ∈ SHUFFLE(O) do
            for (a, i, l) ∈ DRAWS(obs) do
                x⃗_*ai[l] ← x⃗_*ai[l] − 1
                α⃗'_ai[l] ← α⃗'_ai[l] − 1
            end for
            θ ← AVG(α')
            θ_bdd ← REPARAMETRIZE(θ)
            θ_obs ← θ_bdd[obs, :]
            SAMPLE_X(bdd, 1, θ_obs)
            for a = 1, ..., A do
                for i = 1, ..., I_a do
                    α⃗'_ai ← α⃗_a + x⃗_*ai
                end for
            end for
        end for
        samples ← APPEND(samples, α')
    end for
    return AVG(samples)
end function
```

As explained in [11], it is not feasible to perform CGS in PLP models, as a generalization of CGS for LDA in [24]. It is, however, possible to define a general CGS procedure for PB. This uses the independence property of our observations $\vec{f}$ given $\boldsymbol{x}$. The advantage of collapsed versus uncollapsed Gibbs sampling is the faster convergence of the collapsed sampler. We show that this is the case for an experiment using synthetic data in an LDA task in Section 5.1.

CGS for PB models entails sampling $P(\boldsymbol{x}_{n*}|\boldsymbol{\alpha}, \hat{\boldsymbol{x}}_{-n*})$, for $n = 1, \ldots, N$, where the subscript $-n$ means the range of values $1, \ldots, n-1, n+1, \ldots, N$. Note that in constrast to uncollapsed Gibbs sampling, we sample one observation at a time, and $\boldsymbol{\theta}$ is integrated out. Since sampling a BDD requires

probabilities on boolean variables, $P(\boldsymbol{v}|\boldsymbol{\theta})$, we "uncollapse" $\boldsymbol{\theta}$ for this purpose as the average of its current Dirichlet posterior parametrized by $\boldsymbol{\alpha}'$:

$$\vec{\theta}_{ai} = \frac{\vec{\alpha}'_{ai}}{\Sigma(\vec{\alpha}'_{ai})} \qquad a = 1, \ldots, A \ , \ i = 1, \ldots, I_A$$

We assume this is implemented by $\textsc{avg}(\boldsymbol{\alpha}')$.

The CGS is shown in Algorithm 2. Before sampling, we initialize $\boldsymbol{x}$ by setting $\boldsymbol{\theta}$ to the average of the Dirichlet prior, then x is sampled like in an iteration of the uncollapsed Gibbs sampler. We assume this setup is implemented in a function $\textsc{initialize}(\boldsymbol{\alpha})$. We switch representation from a set of BDDs to a set of O of observations, each observation having its own BDD. Each iteration loops over the observations and removes the draws of the current observation from $\boldsymbol{x}$, then updates $\boldsymbol{\theta}$, then re-samples the current observation to update $\boldsymbol{x}$. To remove the draws of the current observation, we use a function $\textsc{draws}(obs)$, that, for some observation obs, returns the list of draws required to explain that observation, i.e. a list of (a, i, l) tuples. The draws represent the path sampled from the BDD in the previous iteration. Unlike the uncollapsed Gibbs sampling algorithm, where all observations were sampled given the same $\boldsymbol{\theta}$, in CGS, we update $\boldsymbol{\theta}$ after each observation. Furthermore, the samples we record are not $\boldsymbol{\theta}$, but the posterior Dirichlet parameters $\boldsymbol{\alpha}'$.

5. Evaluation

In this section we present experiments with PB[4]. The first two experiments are quantitative, while the rest are qualitative. In the latter, we show that inference in PB yields reasonable results given our intuition about the models, and in the case of the repeated insertion model, we reach similar conclusions compared to a different model (the Mallows model).

5.1. PB and collapsed Gibbs sampling (CGS) for LDA on synthetic data

We run a variation of the experiment performed in [24, 11]. A synthetic corpus is generated from an LDA model with parameters: 25 words in the

[4]See supplementary materials for details on implementation and software availability (Appendix Appendix C). In all but the first experiment we use only uncollapsed Gibbs sampling for PB, since, although it converges slower than CGS, it is better optimized in the current implementation.

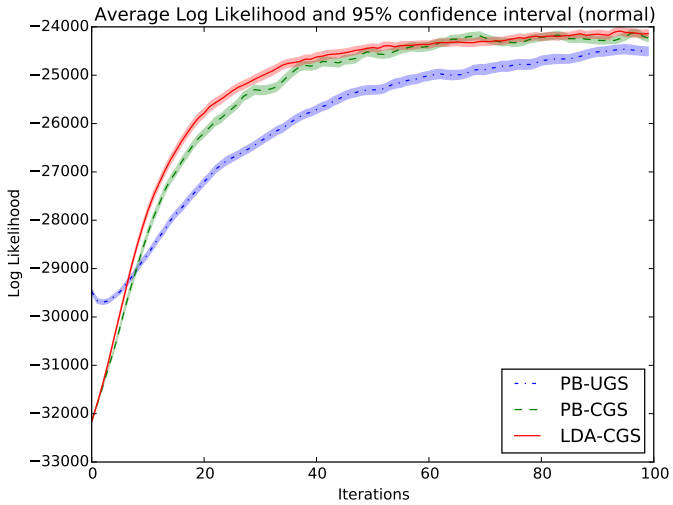


Figure 3: Comparison between PB samplers and LDA-specific CGS on 10 sampled synthetic corpora (10 runs per corpus).

vocabulary, 10 topics, 100 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. [24, 11]. We evaluate the convergence of the PB samplers and an LDA-specific CGS implementation in the `topicmodels` R package. The parameters are: $\beta = \gamma = 1$ as (symmetric) hyper-parameters, and we run 100 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 yields two conclusions: 1) similarly to [11], we find that uncollapsed Gibbs sampling converges slower than CGS and 2) the PB-CGS performs similarly to the LDA-specific CGS, which supports our claim that CGS in PB generalizes CGS in LDA.

5.2. PB, CGS-LDA, VEM-LDA, PRiSM and Stan for LDA on synthetic data

We compare PB with two LDA-specific methods from the `topicmodels` R package: collapsed Gibbs sampling (tm-gibbs) and variational expectation maximization (tm-vem), and two state-of-the-art PPLs: PRiSM [8] and Stan [4]. The metrics we are interested in are: 1) the intrinsic metric of each method, used to subjectively decide when a sampler has converged, 2) log

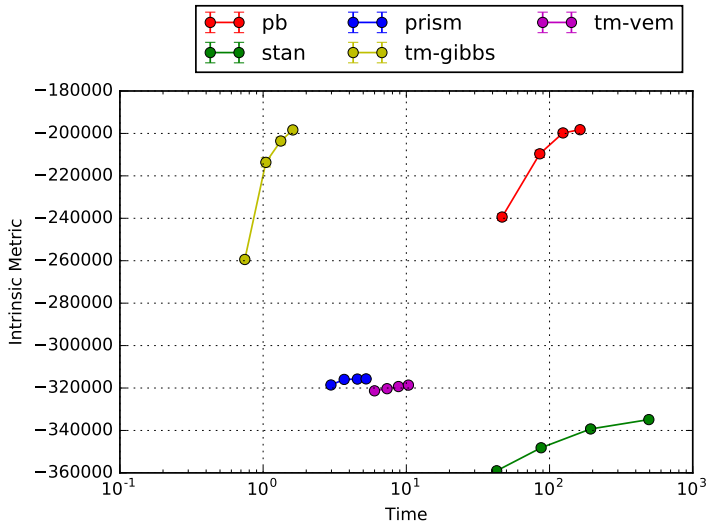


Figure 4: Intrinsic metrics by log average execution time. The methods should not be compared based on these values.

likelihood, used to asses the quality of fit to the observed data and 3) fold-average perplexity in 5-fold cross-validation, used to asses the generalization power of each method. In a previous paper, we showed that Church [1] doesn't seem to converge in a reasonable amount of time on simple LDA tasks, cf. Appendix C of [18].

We use a similar setup to the previous experiment ($T = 10, \gamma = 1$), except that we generate only one corpus of 1000 documents and average over 10 runs with different RNG seeds.

The first problem in evaluating the methods is deciding when has the model converged. We use the default settings of all implementations unless specified and their intrinsic measures: the likelihood attributes of LDA objects for tm-gibbs and tm-vem (`@logLiks`), the LDA-likelihood for PB, cf. Appendix B of [18], the `get_logposterior` function for Stan and the variational free energy (VFE) for PRiSM. Time is measured in seconds and averaged across runs, the metrics are averaged across runs and the error bars show one standard error, though very often the methods show little variation.

We plot the results in Figures 4 and 5. Note that the methods should not be compared with each other based on these figures. The last value of the number of iterations is the one where we deem there is convergence. For sampling methods, we ran up to 200 iterations for tm-gibbs, 400 iterations

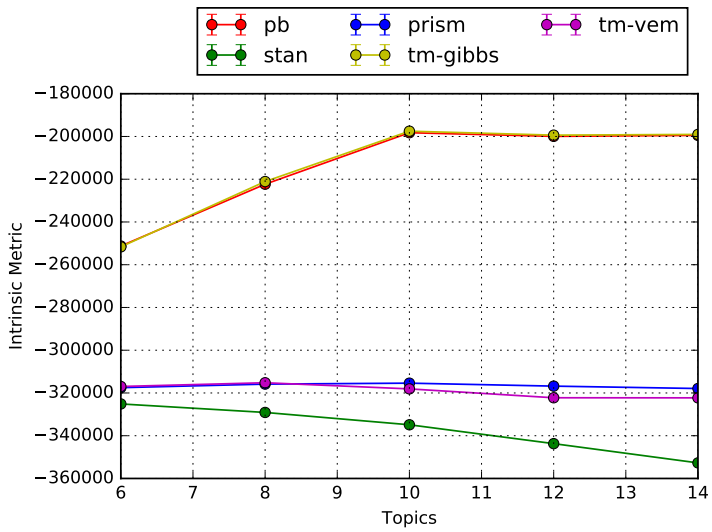


Figure 5: Intrinsic measures by number of topics in the evaluated model.

for PB and 25 iterations for Stan to make this decision. For PB we use 50, 100, 150 and 200 iterations, for Stan 5, 10, 15, 20 iterations, for Prism 50, 100, 150 and 200 iterations (based on the intrinsic convergence of VEM), for tm-gibbs 25, 50, 75, 100 iterations, for tm-vem 20, 30, 40, 50 iterations (based on the intrinsic convergence of VEM). When varying the number of topics, we used the maximum value of iterations for sampling methods, and the rest ran until convergence.

When we vary the number of topics in the probabilistic program, e.g. in Figure 5, we typically expect the metric to increase up to 10 topics, and then decrease or remain the same. This behaviour is illustrated by the intrinsic measures of PB and tm-gibbs, while that of Stan behaves oddly with respect to this expectation. In what follows we shall see that when we track likelihood and perplexity, these behaviours change.

To be able to compare the methods, we used the following definition of "likelihood":

$$\mathcal{L}(\mathcal{C}) = \prod_{(w,d)\in\mathcal{C}} \sum_{t=1}^{T} \mu_d(t)\phi_t(w)$$

, where $\mathcal{C}$ is a corpus of tokens (w, d), w is the token index, d is the document index, and T, μ and ϕ are the same as in Example 2.2.

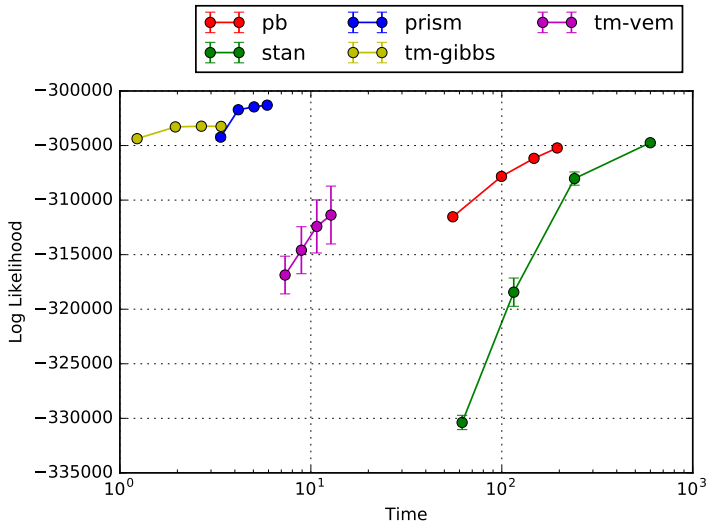


Figure 6: Likelihood against log average execution time. Higher is better. The results are consistent with Figure 4.

We plot the results in Figures 6 and 7. We use the same settings for iteration numbers. As in the intrinsic metric experiment, the more expressive the method, the slower it is, with the exception of PRiSM being faster than tm-vem. The difference of more than an order of magnitude between PRiSM and PB can be explained by the fact that the assumption of non-overlapping explanations allows for significant optimizations in PRiSM, as well as the fact that the implementation of the latter is much more mature. Concretely, PRiSM is written in C, Stan compiles its programs to C++ code, while our current PB prototype sampling implementation is written in Python and Cython.

With respect to the number of topics, all methods perform similarly

Finally, we show the average per fold perplexity in a 5-fold cross-validation split on every document of the corpus in Figures 8 and 9.

We define perplexity similarly to [25]:

$$\mathcal{P}(\mathcal{C}) = \exp\left(-\frac{\mathcal{L}(\mathcal{C})}{\sum_{(w,d)\in\mathcal{C}} 1}\right)$$

We run the methods only four times, and due to faster overall convergence, we tune the number of iterations as follows: 10, 20, 30, 40, 200 for

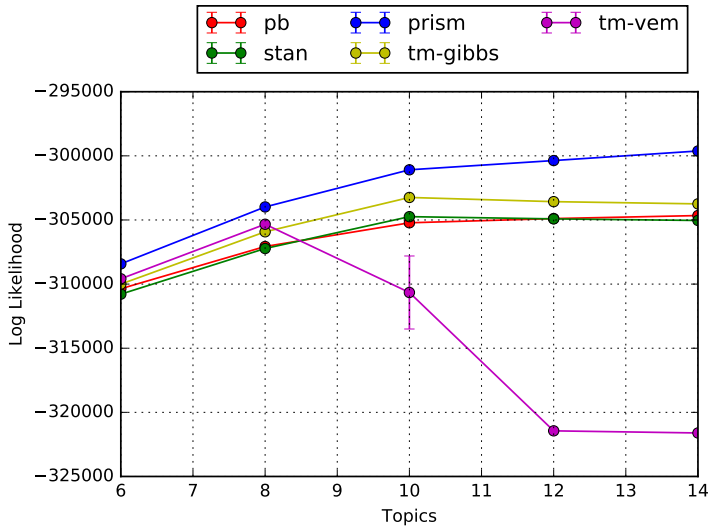


Figure 7: Likelihood against number of topics in the evaluated model. Notice the difference w.r.t. Figure 5.

PB, 2, 4, 6, 8, 20 for Stan, 10, 20, 30, 40, 200 for PRiSM, 10, 20, 30, 40, 200 for tm-gibbs, and 5, 10, 15, 20, 50 for tm-vem. All methods perform well, with the exception of tm-vem. Stan yields very similar values for perplexity for 2, 4, 6 and 8 iterations.

5.3. PB for seed LDA on 20 newsgroups dataset.

Inspired by an experiment from [26], we use the computing related (comp.*) newsgroups in the 20 newsgroups dataset [27]. We tokenize, lemmatize and remove stop words from all documents to obtain a corpus with $V = 27206$ unique tokens in $D = 4777$ documents with average length of approx. 72 tokens. We set $T = 20$ topics and priors $\gamma = 50/T$, $\beta = 0.01$. We seed two topics with hardware (hardware, machine, memory, cpu), and software (software, program, version, shareware) related terms. Seeding a token in a topic means that whenever we observe that token we will assign it only that topic.

We show the PB program, with the observe facts omitted, in Table 1. The omitted observe facts encode the corpus in the same manner as in Example 3.2. The seed predicate specifies a token as its first argument and a list of allowed topics as the second argument. The observations are split into seed tokens and non-seed tokens, and are placed on separate `pb_plate` pred-

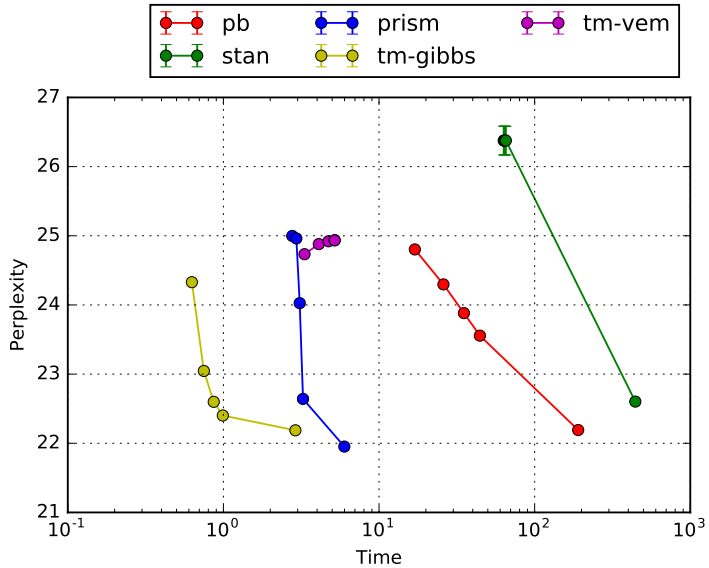


Figure 8: Average fold perplexity in 5CV against average execution time. Lower is better. The first four markers for Stan almost overlap.

icates. To distinguish between the two types of tokens we use the predicate `seed_naf` because negation is universally quantified.

We run PB for 400 iterations and summarize the seeded topics, as word clouds, in Figure 10. We observe that both topics give high weights to the seed words, and other hardware (mhz, rom, disk, bios, board) or software (source, library, utility, server, user) related terms.

5.4. PB for cluster LDA on arXiv abstracts

We consider a different variant of the LDA model, one in which we define a partition C over the topics. In the experiment, we consider 25 topics clustered into 5 clusters of 5 topics. Each token is then generated by choosing a topic cluster according to a document-specific mixture of clusters, then a topic from the cluster, again according to a document-specific mixture, and finally the token is chosen from the topic. Note that this model is different from the parametric version of hierarchical Dirichlet processes [28], equation 29, namely that model considers a global set of topics, and each document (or group) selects a subset from the global set. In cluster LDA, all documents can use any of the topics, however the topic clusters are disjoint, as opposed

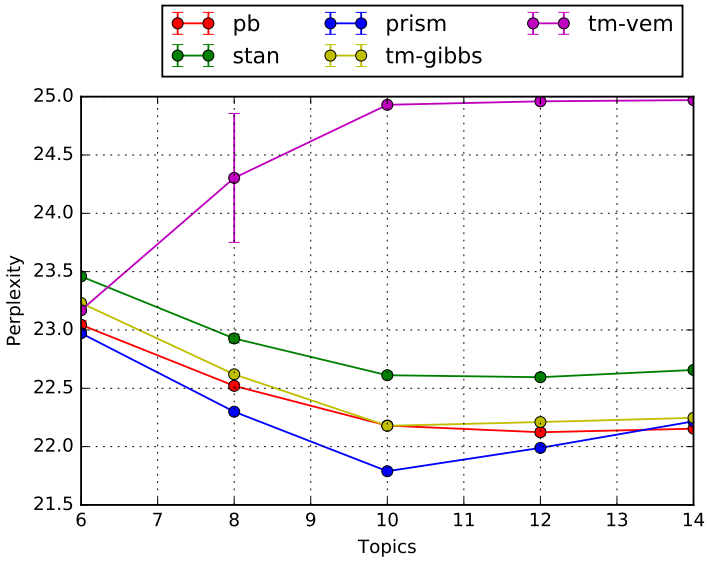
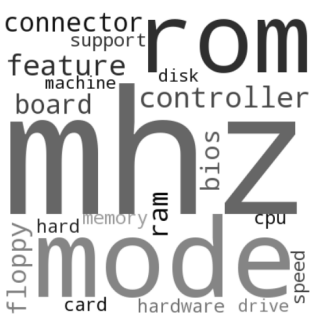


Figure 9: Average fold perplexity in 5CV against number of topics in the evaluated model. Note the consistency w.r.t. Figure 7.



(a) Hardware.

(b) Software.

Figure 10: Seeded topics in the seed LDA task.

```
% 'observe' facts are ommited
pb_dirichlet(2.5, theta, 20, 4777).
pb_dirichlet(0.01, phi, 27206, 20).
seed(9398, [1]).    seed(21247, [2]).
seed(13167, [1]).   seed(17982, [2]).
seed(13813, [1]).   seed(24490, [2]).
seed(4483, [1]).    seed(20682, [2]).
seed_naf(Token) :- seed(Token, _).
pb_plate(
    [observe(d(Doc), TokenList),
        member((w(Token), Count), TokenList),
        \+ seed_naf(Token)],
    Count,
    [Topic in 1..20, theta(Topic,Doc), phi(Token,Topic)]).
pb_plate(
    [observe(d(Doc), TokenList),
        member((w(Token), Count), TokenList),
        seed_naf(Token)],
    Count,
    [seed(Token, TopicList), member(Topic, TopicList),
        theta(Topic,Doc), phi(Token,Topic)]).
```

Table 1: PB program for seed LDA.

to the possibly overlapping subsets in the parametric hierarchical Dirichlet process.

We collect all abstracts on arXiv submitted in 2007, from five categories: quantitative finance (q-fin), statistics (stats), quantitative biology (q-bio), computer science (cs), and physics (physics). We tokenize and remove stop words to obtain a corpus with $V = 26834$ unique tokens in $D = 5769$ documents with average length of approx. 80 tokens. We use priors of $\gamma = 50/T = 10$ for each cluster mixture and topic mixture per cluster, and $\beta = 0.1$ for the topics.

The PB program for the cluster LDA task is shown in Table 2. We use `psi` to denote the draw of a topic cluster, and a predicate `create_term` to create terms that, when called with `pb_call`, choose between the topics within a cluster, as well as the tokens from the topics.

In Figure 11 we plot, as heat maps, the cluster mixtures for each document in each category. We observe that quantitative biology is well represented by cluster 2 and physics is well represented by cluster 3. Computer science is characterized by cluster 5, but also 1 and 4, while quantitative finance and

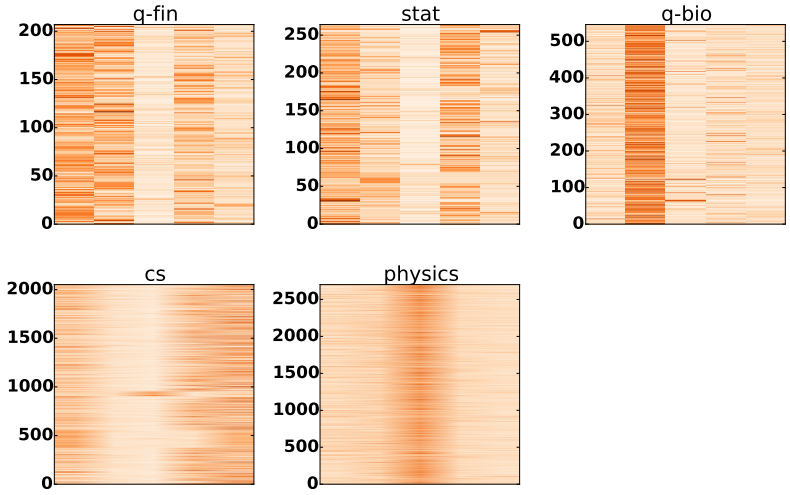


Figure 11: Cluster mixture for each category (x - topic clusters, y - documents, darker colour - higher probability).

statistics are very similar, consisting mainly of clusters 1,2 and 4. The latter effect may be due to the small number of documents in both quantitative finance and statistics, as well as the fact that most quantitative finance papers focus on statistical methods.

Furthermore, if we inspect, for example, cluster 2, corresponding to quantitative biology, shown in Table 3, we find that most topics give high probability to terms used in biology, e.g. "proteins" in topic 2, "genetic" and "brain" in topic 3, "response" and "diffusion" in topic 4. The results agree with our intuition about the model: topics within the same cluster are similar.

```
% 'observe' facts are ommited
pb_dirichlet(10.0, psi, 5, 5769).
pb_dirichlet(10.0, theta1, 5, 5769).
pb_dirichlet(10.0, theta2, 5, 5769).
pb_dirichlet(10.0, theta3, 5, 5769).
pb_dirichlet(10.0, theta4, 5, 5769).
pb_dirichlet(10.0, theta5, 5, 5769).
pb_dirichlet(0.1, phi1, 26834, 5).
pb_dirichlet(0.1, phi2, 26834, 5).
pb_dirichlet(0.1, phi3, 26834, 5).
pb_dirichlet(0.1, phi4, 26834, 5).
pb_dirichlet(0.1, phi5, 26834, 5).
pb_plate(
    [observe(d(Doc), TokenList),
        member((w(Token), Count), TokenList)],
  Count,
  [generate(Doc, Token)]).
create_term(Functor, Idx, Cat, Distrib, Term) :-
    number_chars(Idx, LIdx),
    atom_chars(Functor, LFunctor),
    append(LFunctor, LIdx, LF),
    atom_chars(F, LF),
    Term =.. [F, Cat, Distrib].
generate(Doc, Token) :-
    Cluster in 1..5,
    Topic in 1..5,
    psi(Cluster, Doc),
    create_term(theta, Cluster, Topic, Doc, Term1),
    pb_call(Term1),
    create_term(phi, Cluster, Token, Topic, Term2),
    pb_call(Term2).
```

Table 2: PB program for cluster LDA.

important	0.0092	physical	0.0099	scaling	0.0115
expression	0.0092	proteins	0.0092	free	0.0105
model	0.0092	interaction	0.0092	similar	0.0089
fluctuations	0.0085	individual	0.008	genetic	0.0079
large	0.0082	scales	0.0072	transfer	0.0067
mechanism	0.0077	transition	0.0072	agent	0.0062
specific	0.0077	mathematical	0.0069	brain	0.006
factors	0.0075	investigate	0.0066	chemical	0.0058
recent	0.0074	process	0.0066	exhibit	0.0056
highly	0.0073	dynamical	0.0062	normal	0.0056
simulations	0.0098	structural	0.0096		
response	0.0093	short	0.0095		
activity	0.0087	stability	0.0094		
mean	0.0084	global	0.009		
diffusion	0.0084	equilibrium	0.0087		
rate	0.0081	studies	0.0081		
present	0.008	role	0.008		
temporal	0.0075	experimental	0.0078		
mechanics	0.0075	statistics	0.0074		
correlations	0.0073	influence	0.0071		

Table 3: Topic Cluster 2 (top 10 tokens and probabilities).

5.5. PB for RIM on Sushi dataset

A repeated insertion model (RIM, [13]) 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 [29], on a dataset published in [30]. The data consists of 5000 permutations over $M = 10$ Sushi ingredients, each permutation expressing the preferences of a surveyed person. Following [29], 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 show the PB program used in Table 5, assuming that the `create_term` predicate is defined as in Table 2. 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

$\pi_1 = 0.155$	$\pi_2 = 0.194$	$\pi_3 = 0.134$	$\pi_4 = 0.194$	$\pi_5 = 0.197$	$\pi_6 = 0.126$
fatty tuna	fatty tuna	fatty tuna	fatty tuna	fatty tuna	fatty tuna
shrimp	tuna	sea eel	sea urchin	tuna	shrimp
salmon roe	shrimp	tuna	salmon roe	shrimp	tuna
sea eel	squid	shrimp	shrimp	squid	sea eel
squid	egg	squid	sea eel	sea eel	squid
tuna	tuna roll	tuna roll	tuna	tuna roll	salmon roe
tuna roll	sea eel	salmon roe	tuna roll	salmon roe	tuna roll
sea urchin	cucumb. roll	sea urchin	squid	sea urchin	sea urchin
egg	salmon roe	egg	egg	cucumb. roll	egg
cucumb. roll	sea urchin	cucumb. roll	cucumb. roll	egg	cucumb. roll

Table 4: Mixture parameters and modes of preference profiles on the Sushi dataset.

is produced by selecting a latent profile, then generating that permutation by consecutively inserting elements from a permutation called 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 naïvely generating all the possible permutations is intractable (and unnecessary).

We run PB 10 times for 100 iterations and average the parameters. For each preference profile, we show its mixture parameter and its mode in Table 4. The inference yields similar conclusions to [29]: 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.

```
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),
    % make probabilistic choice
    create_term(p, Idx, Pos, K, Pred),
    pb_call(Pred),
    % increment position and recurse
    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 5: PB program for a RIM with $K = 6$ preference profiles.

6. Related Work

In this section, we will briefly describe the relation between the approach proposed in this paper and other relevant methods. We begin with each of the two fundamental steps of PB: 1) enumeration of the conditional sample space and 2) sampling thereof, and conclude with a high-level comparison with other probabilistic programming languages.

The first step of PB, the enumeration of the conditional sample space through abductive logic programming, could be compared to "logical inference" in ProbLog [9]. While both languages aim to generate a propositional formula and compile it into a decision diagram, "logical inference" in PB is based on abductive logic programming, while ProbLog grounds the relevant parts of the probabilistic program. Moreover, in PB compilation of the boolean formulas is performed using (RO)BDDs, while ProbLog can use a wider range of decision diagrams, e.g. sentential decision diagrams (SDD), deterministic, decomposable negation normal form (d-DNNF). These differences reflect the different aims of the two PPLs: ProbLog focuses on models where "logical inference" needs to be efficient, and the resulting representation, the decision diagrams, need to be compact, while PB focuses on models where "logical inference" is typically easy, however it must be applied repeatedly, according to the nature and the number of the observations. However, in future work, PB could benefit from the use of more compact decision diagrams.

The second step of PB is inspired by the uncollapsed Gibbs sampling algorithm from [11], which we have adapted to PB. However, sampling in the PB model, unlike PLP models, can be performed using collapsed Gibbs sampling, an algorithm with faster convergence that the uncollapsed version, as shown in Section 5.1.

In relation to Church [1] 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. On the other hand, inference in discrete models such as LDA is difficult in highly expressive PPLs, whereas in PB inference is tractable on various discrete models.

Both PB and the logic-based PPL Alchemy [6] focus on discrete models, however they differ at a fundamental level due to the fact that the probabilistic model of PB is directed, whereas that of Alchemy is undirected (Markov networks). Consequently, the specification of probaiblistic programs is also

different: in Alchemy, programs are expressed as weighted formulas, whereas in PB specifies models using abductive logic programs where the abducibles represent draws from a categorical distribution with Dirichlet priors. Furthermore, by using abductive logic programming instead of a first-order knowledge base, PB can easily encode recursive generative models, such as RIM. It is much less obvious how to do so using Alchemy.

7. Conclusions and Future Work

In this paper, we introduced PB, a probabilistic logic programming language for dicrete models with Dirichlet priors. 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. The main contribution to representation is the conditional AD compilation, while the main contribution to inference is the collapsed Gibbs sampling algortihm that generalizes the one proposed for LDA in [24].

We have shown, through the experiments in Section 5, that PB yields reasonable inference results both on synthetic and real datasets.

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 in PB models.

On the other hand, we wish to relax the important restriction to discrete models using Dirichlet processes, that allow discretization of any continuous distribution specified as the base distribution of the process.

[1] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, J. B. Tenenbaum, Church: a language for generative models, Uncertainty in Artificial Intelligence 2008.
URL http://danroy.org/papers/church_GooManRoyBonTen-UAI-2008.pdf

[2] B. Paige, F. Wood, A compilation target for probabilistic programming languages, in: ICML, 2014.

[3] D. Lunn, D. Spiegelhalter, A. Thomas, N. Best, The bugs project: Evolution, critique and future directions, Statistics in Medicine 28 (25) (2009) 3049–3067. doi:10.1002/sim.3680.
URL http://dx.doi.org/10.1002/sim.3680

[4] Stan Development Team, Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0 (2014).
URL http://mc-stan.org/

[5] A. Pfeffer, Figaro: An object-oriented probabilistic programming language.

[6] P. Domingos, S. Kok, H. Poon, M. Richardson, P. Singla, Unifying logical and statistical ai, in: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1, AAAI'06, AAAI Press, 2006, pp. 2–7.
URL http://dl.acm.org/citation.cfm?id=1597538.1597540

[7] B. Milch, B. Marthi, S. Russell, Blog: Relational modeling with unknown objects, in: ICML 2004 Workshop on Statistical Relational Learning and Its Connections, 2004, pp. 67–73.

[8] T. Sato, Y. Kameya, New advances in logic-based probabilistic modeling by prism, in: L. De Raedt, P. Frasconi, K. Kersting, S. Muggleton (Eds.), Probabilistic Inductive Logic Programming, Vol. 4911 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 118–155. doi:10.1007/978-3-540-78652-8_5.
URL http://dx.doi.org/10.1007/978-3-540-78652-8_5

[9] D. Fierens, G. V. den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. D. Raedt, Inference and learning in probabilistic logic programs using weighted boolean formulas, CoRR abs/1304.6810.
URL http://arxiv.org/abs/1304.6810

[10] A. C. Kakas, R. A. Kowalski, F. Toni, Abductive logic programming (1993).

[11] M. Ishihata, T. Sato, 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, 2011, pp. 81–96.
URL http://www.jmlr.org/proceedings/papers/v20/ishihata11/ishihata11.pdf

[12] D. M. Blei, A. Y. Ng, M. I. Jordan, J. Lafferty, Latent dirichlet allocation, Journal of Machine Learning Research 3 (2003) 2003.

[13] J.-P. Doignon, A. Peke, M. Regenwetter, The repeated insertion model for rankings: Missing link between two subset choice models, Psychometrika 69 (1) (2004) 33–54. doi:10.1007/BF02295838.
URL http://dx.doi.org/10.1007/BF02295838

[14] A. D. Gordon, T. A. Henzinger, A. V. Nori, S. K. Rajamani, Probabilistic programming, in: International Conference on Software Engineering (ICSE Future of Software Engineering), IEEE, 2014.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=208585

[15] W. L. Buntine, Operations for learning with graphical models, J. Artif. Intell. Res. (JAIR) 2 (1994) 159–225. doi:10.1613/jair.62.
URL http://dx.doi.org/10.1613/jair.62

[16] L. D. Raedt, A. Kimmig, H. Toivonen, Problog: A probabilistic prolog and its application in link discovery., in: M. M. Veloso (Ed.), IJCAI, 2007, pp. 2462–2467.
URL http://dblp.uni-trier.de/db/conf/ijcai/ijcai2007.html#RaedtKT07

[17] A. Kimmig, A Probabilistic Prolog and its Applications (Een probabilistische prolog en zijn toepassingen), Ph.D. thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, de Raedt, Luc (supervisor) (Nov. 2010).
URL https://lirias.kuleuven.be/handle/123456789/280932

[18] C. Turliuc, N. Maimari, A. Russo, K. Broda, 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, 2013, pp. 759–775. doi:10.1007/978-3-642-45221-5_51.
URL http://dx.doi.org/10.1007/978-3-642-45221-5_51

[19] K. L. Clark, Negation as failure, in: Logic and Data Bases, 1977, pp. 293–322.

[20] M. Fitting, A kripke-kleene semantics for logic programs, J. Log. Program. 2 (4) (1985) 295–312.

[21] J. Ma, Abductive reasoning module for sicstus prolog (2012).
URL http://www-dse.doc.ic.ac.uk/cgi-bin/moin.cgi/abduction

[22] S. B. Akers, Binary decision diagrams, IEEE Trans. Comput. 27 (6) (1978) 509–516. doi:10.1109/TC.1978.1675141.
URL http://dx.doi.org/10.1109/TC.1978.1675141

[23] R. Bryant, Graph-based algorithms for boolean function manipulation, Computers, IEEE Transactions on C-35 (8) (1986) 677–691. doi:10.1109/TC.1986.1676819.

[24] T. L. Griffiths, M. Steyvers, Finding scientific topics, Proceedings of the National Academy of Sciences 101 (suppl 1) (2004) 5228–5235. arXiv:http://www.pnas.org/content/101/suppl_1/5228.full.pdf, doi:10.1073/pnas.0307752101.
URL http://www.pnas.org/content/101/suppl_1/5228.abstract

[25] E. Hörster, R. Lienhart, M. Slaney, Image retrieval on large-scale image databases, in: Proceedings of the 6th ACM International Conference on Image and Video Retrieval, CIVR '07, ACM, New York, NY, USA, 2007, pp. 17–24. doi:10.1145/1282280.1282283.
URL http://doi.acm.org/10.1145/1282280.1282283

[26] D. Andrzejewski, X. Zhu, M. Craven, B. Recht, A framework for incorporating general domain knowledge into latent dirichlet allocation using first-order logic, in: IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011, 2011, pp. 1171–1177.
URL http://ijcai.org/papers11/Papers/IJCAI11-200.pdf

[27] K. Lang, Newsweeder: Learning to filter netnews, in: in Proceedings of the 12th International Machine Learning Conference (ML95), 1995.

[28] Y. W. Teh, M. I. Jordan, M. J. Beal, D. M. Blei, Hierarchical dirichlet processes, Journal of the American Statistical Association 101.

[29] T. Lu, C. Boutilier, Effective sampling and learning for mallows models with pairwise-preference data, Journal of Machine Learning Research 15 (2014) 3783–3829.
URL http://jmlr.org/papers/v15/lu14a.html

[30] T. Kamishima, H. Kazawa, S. Akaho, Supervised ordering - an empirical survey, in: Data Mining, Fifth IEEE International Conference on, 2005, pp. 4 pp.–. doi:10.1109/ICDM.2005.138.

Appendix A. Execution of an abductive logic program

In this section we explain how to compute all abudctive solutions for an abductive program and a query. We use a proof procedure similar to ASystem [18, 21], however our treatment of abducibles is different. The proof procedure is top-down and can be viewed as a tree where every node represents a state.

A *denial* is a formula $\forall Y \bot \leftarrow \Gamma$, where Γ is a set of literals and Y is a set of logical variables. In the rest of the paper $\bot$ is considered implicit.

An ASystem state S is a tuple $(\mathcal{G}, \Delta, \mathcal{N})$, where:

- $\mathcal{G}$ is a set of goals where each goal can be a literal or a denial. All the variables except the ones universally quantified in the denials are existentially quantified.
- Δ is the abducible store, a set of abducibles.
- $\mathcal{N}$ is the denial store, a set of denials.

A *selection strategy* Ξ has a two-fold role: it selects a goal G_i from the set $\mathcal{G}$, and if the goal is a denial $\forall Y \leftarrow \Gamma$ it further selects a literal from Γ. A selection strategy is called *safe* if, in a denial goal, it never selects a negative literal if the arguments of the predicate include a universally quantified variable.

Given an abductive framework (Π, AB), a query Q and a selection strategy Ξ, an *ASystem derivation tree* is a tree such that:

- every node of the tree is an ASystem state.
- children nodes are generated by selecting a goal (and if the goal is a denial, further selecting a literal in the denial) according to Ξ, and then applying the proof procedure rules (defined later in the section) on the selected goal.
- the initial state is $S_0 = (Q, \emptyset, \emptyset)$.
- a success state is one in which $\mathcal{G} = \emptyset$. If the derivation flounders, i.e. we cannot safely apply Ξ, then that state is a failure state. A state is a leaf of the tree iff it is either a success or failure state.

We restrict our programs such that in a success state, the denial store is empty. Let SS_Q denote the set of success states for query Q. Then, the result of an abductive query is the formula:

$$\mathcal{M}(Q) = \bigvee_{\Delta \in SS_Q} \left(\bigwedge_{ab \in \Delta} ab \right)$$

It is not difficult to extend the result to non-empty denial stores, a case in which $\mathcal{M}(Q)$ will not be in a disjunctive normal form (DNF), but rather an arbitrary formula, as we do not take advantage of the particular DNF encoding in the rest of the paper.

In defining an ASystem derivation tree, we mentioned the application of proof procedure rules, which we proceed to define in the rest of the section. Given an abductive framework $\langle \Pi, AB \rangle$, let $S_k = (\mathcal{G}_k, \Delta_k, \mathcal{N}_k)$ denote an ASystem state, and let F be a goal selected by Ξ from $\mathcal{G}_k$, such that $\mathcal{G}_k^- = \mathcal{G}_k \setminus \{F\}$. By applying a suitable proof procedure rule, we obtain a child state $S_{k+1} = (\mathcal{G}_{k+1}, \Delta_{k+1}, \mathcal{N}_{k+1})$. Some inference rules create more than one child, and the children states will be separated by **OR**. In the description of the rules, we mention only the updated goal and elements in the store, the rest remain the same as in the parent state.

We use the following notation: Y denotes a set of variables, u denotes a term, U, and Z denote vectors of terms. Equality between variables and terms denotes unification, and is extended to vectors component-wise. Φ is used to denote rule bodies, i.e. conjunctions of literals. The symbols a, i, l and k_a have the same meaning as in Section 2, p denotes a functor, L denotes a literal. $Var(U)$ denotes the set of variables in U, where U can be either a predicate, a set of terms or a literal.

Based on the type of the selected goal (i.e. literal or denial) we distinguish three inference rules for each type.

(1) If the selected goal is a literal F, then:

(D1) if $F = p(u)$ and p is `pb_call`, then:

$\mathcal{G}_{k+1} = \{u\} \cup \mathcal{G}_k^-$

else if $F = p(U)$ is a non-abducible, let $p(Z_r) \leftarrow \Phi_r$, $r = 1, \ldots, R$, be R rules in Π, then:

$\mathcal{G}_{k+1} = \{U = Z_1\} \cup \Phi_1 \cup \mathcal{G}_k^-$

OR

$\mathcal{G}_{k+1} = \{U = Z_2\} \cup \Phi_2 \cup \mathcal{G}_k^-$

OR

...
$\mathcal{G}_{k+1} = \{U = Z_R\} \cup \Phi_R \cup \mathcal{G}_k^-$.

(A1) if $F = p(l, i)$ is an abducible,

if $F \in \Delta_k$, then:

$\mathcal{G}_{k+1} = \mathcal{G}_i^-$

else if $\nexists p(l_2, i) \in \Delta_i$, $l_2 \neq l$:

$\mathcal{G}_{k+1} = \{F\} \cup \mathcal{G}_k^-$

else **FAIL**

(N1) if $F = \neg p(U)$ then:

$\mathcal{G}_{k+1} = \{\leftarrow p(U)\} \cup \mathcal{G}_k^-$.

(2) In the case that the selected goal is a denial F, let $F = \forall Y \leftarrow \Gamma$, with $\Gamma \neq \emptyset$, and let L be a literal selected by Ξ from Γ, such that $\Gamma^- = \Gamma \setminus \{L\}$. The rules for L are:

(D2) if $L = p(u)$ and p is `pb_call`, then:

$\mathcal{G}_{k+1} = \{\forall Y \leftarrow \{u\} \cup \Gamma^-\} \cup \mathcal{G}_k^-$.

else if $L = p(U)$ is a non-abducible, then:

$\mathcal{G}_{k+1} = \{\forall Z^+ \leftarrow \Gamma^+ \mid p(Z) \leftarrow \Phi \in \Pi$ and $Z^+ = Y \cup Var(p(Z)) \cup Var(\Phi)$ and $\Gamma^+ = \{U = Z\} \cup \Phi \cup \Gamma^-\} \cup \mathcal{G}_k^-$.

(A2) $F = p(l, i)$ is an abducible, then:

if $F \notin \Delta_k$, let k_a be the number of categories of the distribution inexed by (a, i):

$\mathcal{G}_{i+1} = \{p(1, i)\} \cup \mathcal{G}_i^-$

OR

...

OR

$\mathcal{G}_{i+1} = \{p(l-1, i)\} \cup \mathcal{G}_i^-$

OR

$\mathcal{G}_{i+1} = \{p(l+1, i)\} \cup \mathcal{G}_i^-$

OR

...

OR

$\mathcal{G}_{i+1} = \{p(k_a, i)\} \cup \mathcal{G}_i^-$

else **FAIL**

(N2) if $L = \neg p(U)$ such that $Var(L) \cap Y = \emptyset$, then:

$\mathcal{G}_{k+1} = \{p(U)\} \cup \mathcal{G}_k^-$

OR

$\mathcal{G}_{k+1} = \{\leftarrow p(U), Y \leftarrow \Gamma^-\} \cup \mathcal{G}_k^-$

Note that the differences with respect to the original ASystem rules concern abducibles, i.e. **(A1)** and **(A2)**, and additionally we allow calling terms as abducibles by modifying **(D1)** and **(D2)**. Furthermore, since we are working with ground abducibles, we don't need rules for (in)equalities or constraints. Similarly to ProbLog, the abducibles are always memoized, i.e. if we call the same abducible twice in a query, the second call will not modify the state.

Appendix B. BDD sampling

To sample a BDD, as described in Algorithm 3, we first compute the backward probability of the BDD, then we sample a path from root to the "true" leaf. We use *bdd.NNodes* to denote the number of nodes in the BDD. Furthermore, we assume the nodes of the BDD are sorted breadth-first from the "true" leaf (node 1) to the root (node *bdd.NNodes*), and that the "false" leaf is not represented in the BDD. Additionally, we use the following functions for a BDD *bdd*:

- IS_LEAF($bdd, node$) returns "true" for node 1 and "false" for all other nodes.
- LABEL($bdd, node$) returns the label of a non-leaf node, i.e. the level of the node in the BDD (from 1 for the root node to N_{bdd} for the last variable).
- CHILDREN($bdd, node$) returns the high ("true") and the low ("false") children of a non-leaf node *node*.
- IS_LEAF($bdd, node$) returns "true" for node *bdd.NNodes* and "false" for all other nodes.
- BINOMIAL($p, counts$) returns the counts (success, failure) that result from *counts* independent Bernoulli trials with probability of success p.
- LABEL_DISTRIB($bdd, node, obs$) returns a tuple (a, i, l) corresponding to the draw represented by making node *node* "true" for observation *obs* (decoding $\boldsymbol{v}$ to $\boldsymbol{x}$).
- PENULTIMATE(bdd, obs, a, i) returns the penultimate category in the conditional AD compilation of distribution indexed by a and i in observation *obs*.

- LAST(bdd, obs, a, i) returns the last category in the conditional AD compilation of distribution indexed by a and i in observation obs. Note that in general this is not an increment of the penultimate category, since the PB program determines which categories are chosen in the explanations of a particular observation.

Appendix C. 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 see:
`http://www.github.com/raresct/peircebayes_experiments`

Algorithm 3 BDD sampling (based on [11]).

```
function SAMPLE_X(bdd, count, θ_obs)
    beta ← BACKWARD(bdd, θ_obs)
    SAMPLE_BDD(beta, bdd, count, θ_obs)
end function
function BACKWARD(bdd, θ_obs)
    beta ← ZEROS(bdd.NNodes)
    for node = 1, ..., bdd.NNodes do
        if IS_LEAF(bdd, node) then
            beta[node] ← 1
        else
            p ← θ_obs[LABEL(bdd, node)]
            (high, low) ← CHILDREN(bdd, node)
            beta[node] ← p × beta[high] + (1 − p) × beta[low]
        end if
    end for
    return beta
end function
function SAMPLE_BDD(beta, bdd, count, θ_obs)
    counts ← ZEROS(bdd.NNodes)
    for node = bdd.NNodes, ..., 2 do
        if IS_ROOT(bdd, node) then
            counts[node] ← count
        end if
        (high, low) ← CHILDREN(bdd, node)
        p ← (θ_obs[LABEL(bdd, node)] × beta[high]) / beta[node]
        (high_counts, low_counts) ← BINOMIAL(p, counts[node])
        (a, i, l) ← LABEL_DISTRIB(bdd, node, obs)
        x⃗_*ail ← x⃗_*ail + high_counts
        if l = PENULTIMATE(bdd, obs, a, i) then
            l_2 ← LAST(bdd, obs, a, i)
            x⃗_*ail_2 ← x⃗_*ail_2 + low_counts
        end if
        counts[high] ← counts[high] + high_counts
        counts[low] ← counts[low] + low_counts
    end for
end function
```