Distributional Logic Programming for Bayesian Knowledge Representation

Nicos Angelopoulos[c], James Cussens[d]

[a]Welcome Trust Sanger Institute, Hinxton, CB10 1SA, UK
[b]Department of Computer Science,University of York, York UK

We present a formalism for combining logic programming and its flavour of non-determinism with probabilistic reasoning. In particular, we focus on representing prior knowledge for Bayesian inference. Distributional logic programming (Dlp), is considered in the context of a class of generative probabilistic languages. A characterisation based on probabilistic paths which can play a central role in clausal probabilistic reasoning is presented. We illustrate how the characterisation can be utilised to clarify derived distributions with regards to mixing the logical and probabilistic constituents of generative languages. We use this operational characterisation to define a class of programs that exhibit *probabilistic determinism*. We show how Dlp can be used to define generative priors over statistical model spaces. For example, a single program can generate all possible BNs having N nodes while at the same time it defines a prior that penalises BNs with large families. Two classes of statistical models are considered: Bayesian networks and classification and regression trees. Finally we discuss: (1) a Metropolis-Hastings algorithm that can take advantage of the defined priors and the probabilistic choice points in the prior programs and (2) its application to real-world machine learning tasks.

Distributional Logic Programming for Bayesian Knowledge Representation

Nicos Angelopoulos[c], James Cussens[d]

[c]Welcome Trust Sanger Institute, Hinxton, CB10 1SA, UK
[d]Department of Computer Science,University of York, York UK

1. Introduction

Bayesianism provides a powerful framework for reasoning with statistical knowledge. The result of reasoning is captured by the posterior distribution. Knowledge is captured by the prior and the evidence. The former can represent expert knowledge or belief in a domain, while the latter can take the form of data to be analysed. The basic Bayesian premise can be summarised as:

$$posterior \propto prior \times evidence$$

A plethora of algorithms operate on the above principle to either locate important members of the posterior, such as the maximum a posteriori mode (MAP), or to characterise the whole distribution. Computation in both cases is often prohibitively lengthy to allow exact algorithms, so approximations are routinely used. These include variational methods [26] which approximate the inference on the evidence by considering a simpler inference task while Markov chain Monte Carlo (MCMC) simulations approximate the whole posterior by means of a stochastic search.

Bayesian algorithms that take into account the prior part of the above premise often do so in a restricted form. For instance, [10] uses a conjugate prior over classification trees and in [21] the authors use an uninformative prior over BNs. Reasons for such restrictions include both the lack of relevant knowledge and the limited availability of formalisms that can express the known biases and for which effective inference procedures exist. However, in application areas such as computational biology and bioinformatics a growing amount of formalised knowledge is becoming available. The ability to represent complex biological knowledge would greatly benefit the application of Bayesian methods in these areas as it can focus computational resources in parts of the solution space that are most likely to hold the answer or of particular interest to the biologists. On the other hand, Bayesian methods provide a convenient, clean framework in which such knowledge can be incorporated.

The incorporation of prior knowledge is playing an increasingly important role in bioinformatics and computational biology. A vast array of experimental data is becoming publicly available in unprecedented volumes. Summarising and incorporating extracted knowledge in the analyses of new data is a route already taken by many labs. In addition formal frameworks for representing knowledge such as Gene Ontology [37] and the Kyoto Encyclopedia of Genes and Genomes (KEGG, [27]) are gaining ground in both depth and breadth of the knowledge they store.

Logic programming (LP) is an attractive formalism for representing crisp knowledge. Probabilistic extensions to logic programming have been previously proposed for the purpose of representing Bayesian priors ([13, 3, 6]). Here, we present an expressive language that extends logic clauses with probabilities which are calculated by guards encoding arbitrary relations. We also provide a characterisation that elucidates the interplay of nondeterminism in LP and probabilistic reasoning for a number of generative languages. Additionally, we put emphasis on representing knowledge for effective probabilistic problem solving via a number of examples that represent knowledge over model structures and which are drawn from the literature. We detail how the probabilistic aspects of our formalism enable Bayesian learning that can exploit both the prior information and the internal probabilistic choice points to create a search space constrained Metropolis-Hasting algorithm. This paper provides the knowledge representation machinery for the conceptual framework of [3] and the machine learning results of [4, 5, 6, 7]. The full syntax is presented for the first time, along with semantic considerations and a thorough discussion and mathematical framework for probabilistic paths (Section 4). Furthermore, details on constructing effective priors that model priors from the literature are discussed. Finally, we illustrate how the knowledge representation discussed in this paper connects to already published research that concentrated on machine learning results [4, 5, 6, 7].

2. Preliminaries

In this section we review the necessary terminology from logic programming. A logic program L is a set of clauses of the form $Head$:- $Body$ defining a number of predicates. $Head$ is an atom, a single positive literal constructed from a predicate symbol and a number of term arguments. $Body$ is a conjunction of zero or more atoms $A_1, \dots A_n$. Each term is a recursively defined structure that might be an atomic value, a variable or a function constructed by an atomic function symbol and n term arguments. The form $Head$:- $Body$ is syntactic sugar for the disjunction $Head \vee \neg A_1 \vee \cdots \vee \neg A_n$ with all variables implicitly universally quantified. We follow LP conventions and have variables starting by a capital letter ($List$) and atoms by a lower case letter ($constant$). An example term of 4 arguments is: $cart(f1, v1, L, R)$. It represents a classification tree which splits some data at the top level on feature $f1$ and value $v1$, while the left (L) and right (R) branches are as yet to be constructed and are shown here as free variables.

A query or goal G_1 is a disjunction of negative literals $(\neg A_{(1,1)} \vee \ldots \vee \neg A_{(1,n)})$ which the logic engine attempts to refute using the clauses in L. This is done by employing SLD (linear resolution of definite clauses with a selection rule). Linear resolution at step i will resolve $\neg A_{(i,1)}$ with the head (H_i) of the first matching clause found (M_i) and replace it with the body of the clause thus generating a new goal. Matching is via the unification algorithm, which when successful, provides a substitution θ_i such that $A_{(i,1)}/\theta_i = H_i/\theta_i$. Intuitively, successful unification is a method for selecting which of the clauses in the program are applicable in answering the query while θ_i possibly makes the free variables in G_i more concrete thus helping to build an answer to the query. A computation terminates when the current goal is the empty one or no matching clause is found. In the former case an overall substitution is constructed $\theta = (\theta_N, \ldots, \theta_1)$ where θ is the composition of the substitutions $\theta_N, \ldots, \theta_1$ with G/θ being the computed answer.

The logic engine explores yet unexplored parts of the space, by returning to the latest matching step and attempting to find alternative resolution clauses. In logic programming parlance, this is a backtracking step. In the case where no matching clauses are found, the engine will backtrack to the the second latest matching step, and thus recursively search until an alternative can be found. Computed answers in the form of θ_x substitutions done after the backtracking point are undone. The complete search ends when all alternatives have been exhausted. In what follows we will use A_i to refer to $A_{(i,1)}$, that is, the literal used for the ith resolution step.

As an illustrating example program, consider the following two clauses defining the $member/2$ predicate:

$$member(H, [H|T]). \tag{C_1}$$

$$member(El, [H|T]) \ :- \tag{C_2}$$

$$member(El, T).$$

The first clause, (C_1), states that the head of a list is its first element, while the second clause states that element El is a member of the list, if it is a member of the tail (T) of the list. Lists are convenient recursive term structures commonly used in logic programming to hold a collection of data objects. Posed with a query of the form $?{-}member(X, [a, b, c])$ (which is syntactic sugar for $\neg member(X, [a, b, c])$) the logic programming engine uses SLD resolution which scans the query left to right and the program top to bottom as to provide all possible answers in the form of values for X. In our example these are the alternative values a, b, and c which are formally written as $\theta = \{X = a\}$, $\theta = \{X = b\}$ and $\theta = \{X = c\}$.

2.1. Probability Theory and Logic Programming

Logic programming implements a complete search of a nondeterministic space. We review the difficulties of mixing such spaces with probabilistic ones and present one way to achieve this. The formalism we propose here assigns probabilities to clauses defining a single probabilistic space based on a clear distribution over computed answers G/θ. Furthermore, the thesis we propose is that for a class of generative logic programming formalisms that treat probabilities as top-level constructs the definition of a single probabilistic space with a clear distribution over computed answers G/θ is a key concept.

Current probabilistic formalisms include those that completely replace nondeterminism with a probabilistic operator. Such approaches limit nondeterminism. In this category, SLPs [29] under the semantics presented in [13] replace the SLD with sampling over pure programs, which only contain stochastic clauses. An example of a formalism that subjugates probabilities within logical inference is that of PRISM [35]. It was introduced for parameter learning in the context of Probabilistic Context Free Grammars (PCFGs) and hidden Markov models (HMMs). PRISM provides a single probabilistic construct that instantiates an unbound variable from the elements of a list according to the probability values attached to each element. Dlp is closely related to PRISM and to SLPs; we will explain the similarities and differences in Section 4.1. Another language is ProbLog [28]. This generalises the concept of assigning probabilities to a list of values by removing the requirement for independence between random variables.

PCLP [33] and clp(pfd(Y)) [1] employ constraint programming to create, in distinct ways, two separate spaces. Nondeterminism is expressed via clausal syntax while the probabilistic is constructed in the constraint store with the constraint solver used to reason/infer from this information. A crucial point in understanding the differences of nondeterminism to probabilistic reasoning as discussed here, is that of a single substitution (θ) appearing twice. In terms of standard logic programming, the repetition of an already seen solution is immaterial, as the second appearance adds nothing new in terms of logical consequence with the possible exception of explanation-based interpretations. On the other hand, multiple probabilistic derivations of the same consequences are intrinsically important as they alter the probability attached to each θ.

2.2. Statistical relational learning

Related research areas include that of statistical relational learning with important recent contributions such as: probabilistic relational models (PRMs, [20]), Markov logic networks (MLNs, [19]) and ProPPR [39]. These approaches use relations and logic for representing knowledge. Inference is layered on top by the statistical machinery. ProPPR has been presented as an extension to SLPs which locally grounded and was introduced with the specific aim of a efficient alternative for implementing the personalised PageRank algorithm [11]. In comparison, distributional logic programming (Dlp) focuses on logic programming as a general purpose AI language by extending logical reasoning with probabilistic knowledge. The statistical machinery is built in confluence with the underlying logic inference. Figaro [30] is a powerful object oriented probabilistic programming language also coupled to a Metropolis-Hastings (MH). It is more general in that it allows objects to have constraints and relationships to other objects. In contrast, Dlp is based on logic programming which makes programs readily amenable to mathematical analysis and program transformations.

It is worth noting that probabilistic programming has seen an increase in research interest, with a number of formalisms from other paradigms. Two such systems are Church [23] and Anglican [41] that extend functional programming with probabilistic constructs.

3. Syntax

We extend LP's clausal syntax with probabilistic guards that associate a resolution step using a particular clause with a probability whose value is computed on-the-fly. The computed value can then be used as the probability with which the clause is selected for resolution. The main intuition is that in addition to the logical relation a clause defines over the objects that appear as arguments in its head, it also defines a probability distribution over aspects of this relation.

Definition 1. *Let* $H :- B$ *be a syntactically correct Prolog clause and* $\phi(H) = functor(H)/arity(H)$ *be the structure denoting the functor and arity of* H. *Let* $V_{G_{\phi(H)}}$ *be a list of variables and* $E_{G_{\phi(H)}}$ *an arithmetic expression involving the variables in* $V_{G_{\phi(H)}}$ *and no other variables. A probabilistic predicate* $P_{\phi(H)}$ *is associated with a single guard* $G_{\phi(H)}$ *and consists of clauses of the form:*

$$E_{G_{\phi(H)}} :: V_{G_{\phi(H)}} :: H \text{ :- } B \qquad (C_3)$$

We will refer to $E_{G_{\phi(H)}}$ as the probabilistic expression of the clause and to $V_{G_{\phi(H)}}$ as the probability measure variables or simply measure variables of the clause. The probabilistic expression of a specific clause will be evaluated at resolution time to a number: the probability label of the resolved clause. The evaluation is preceded by the matching of $V_{G_{\phi(H)}}$ to a list of ground arithmetic values generated by the guard $G_{\phi(H)}$ associated with $P_{\phi(H)}$. Note that all clauses belonging to the same predicate should share equal length probability measure variable lists ($V_{G_{\phi(H)}}$), as they will be matched to a list of numbers generated by a single guard. The computation of the associated guard $G_{\phi(H)}$ occurs deterministically and if it fails or if it does not instantiate $V_{G_{\phi(H)}}$ to a list of numbers, the computation halts.

Definition 2. *Let* $A_{\phi(H)}$ *be a goal,* $V_{A_{\phi(H)}}$ *a list of variables in* $A_{\phi(H)}$ *and* $P_{\phi(H)}$ *a probabilistic predicate as defined in Definition 1. Let* H^ϵ *be a form of* $P_{\phi(H)}$*'s head* (H) *with all its arguments replaced by variables and* Λ_{H^ϵ} *be a list of some of these variables. The single guard* $G_{\phi(H)}$ *associated with* $P_{\phi(H)}$ *is defined by:*

$$V_{A_{\phi(H)}} :: A_{\phi(H)} \sim \Lambda_{H^\epsilon} :: H^\epsilon \qquad (G_1)$$

The list of variables $V_{A_{\phi(H)}}$ constitutes the measure variables of the guard, and its length should match the length of the measure variables list in the clauses of $P_{\phi(H)}$, $V_{G_{\phi(H)}}$ in (C_3), as they will be matched to them at run-time. H^ϵ will typically share variables with $A_{\phi(H)}$. These shared variables are the measure defining variables of the guard, and are distinct to those in Λ_{H^ϵ}, which are the distributional or probabilistic variables of the guard and associated predicate. Having a single guard means that the probabilistic dependencies of the data objects present as arguments to a predicate definition, can be defined in one place, capturing the probabilistic relation of the data and allowing the distillation onto the measure variables which in turn are used in the evaluation of the run-time probability expressions.

The intuitive reading of the measure variables in $V_{A_{\phi(H)}}$ of (G_1) is that the probability by which data corresponding to the Λ_{H^ϵ} variables (by extension $P_{\phi(H)}$ too) are generated, depends on the data in the shared variables of H^ϵ and $A_{\phi(H)}$. The magnitude of the dependency is defined by $A_{\phi(H)}$- an arbitrary logic goal, called at run-time. $A_{\phi(H)}$ should be a pure logic goal, with no internal calls to probabilistic predicates. Communicating the results from the guard to the clauses is done via matching the positioning of the variables in $V_{G_{\phi(H)}}$ to those of $V_{A_{\phi(H)}}$. Concrete probabilistic labels for each clause can then be calculated by clausal, probabilistic expressions ($E_{G_{\phi(H)}}$).

For example the following guard:

$$[L] :: length(List, L) \sim [El] :: umember(El, List) \qquad (G_2)$$

declares that the length L of the predicate's $umember/2$ input list $List$ is the numerical information needed to define a distribution over selecting an element from the input. Furthermore, in the case where $V_{A_{\phi(H)}}$ and $\Lambda_{H^{\epsilon}}$ are lists of one element we simplify the notation by dropping the surrounding square brackets and writing (G_2) as

$$L :: length(List, L) \sim El :: umember(El, List) \qquad (G_3)$$

The complete program corresponding to the $umember/2$ example guard shown in (G_2) is:

$$L :: length(List, L) \sim El :: umember(El, List)$$

$$\frac{1}{L} :: L :: umember(El, [El|Tail]). \qquad (C_4)$$

$$1 - \frac{1}{L} :: L :: umember(El, [H|Tail]) \ :- \qquad (C_5)$$

$$umember(El, Tail).$$

The above program defines a uniform distribution over element selection for an input list irrespective of the length of this list. However, its execution according to what described so far, is computationally wasteful when consecutively calling the guard for $umember/2$ ($G_{umember/2}$) via the recursive call of (C_5). For each application of the recursive clause (C_5) the length of the remaining list needs to be recalculated as to provide the measure variables of $umember/2$. In order to address this inefficiency we introduce new syntax in the form of probabilistic goals.

Definition 3. *Let F be a standard logic goal, $H :- B$ a logic clause C or the logic part of a probabilistic clause, with B a conjunction of atoms $A_1, \ldots, A_n$ (Section 2). Let V_C be a list combining variables in C and numeric values, with its length being equal to $V_{A_{\phi(F)}}$ (Defn. 2). For A_i calling the probabilistic predicate $P_{\phi(F)}$, we refer to A_i as a probabilistic goal if and only if it is of the form*

$$V_C :: F \qquad (G_4)$$

V_C are the measure variables of the goal. The intuition behind probabilistic goals is that when calling F in the context of C, we might already have the necessary information regarding the relevant guard's measure variables. So there will be no need to re-evaluate $G_{\phi_{(F)}}$. A typical category of predicates for which this is the case, is that of recursive predicates. When a probabilistic predicate recurses, it might be possible to generate the guard measure values incrementally. This is an efficiency consideration, and as before, if V_C is not instantiated to a list of arithmetic values the computation stops. In terms of correctness, one could have a special execution mode which ensures that for a given query the values in V_C of a probabilistic goal as shown in (G_4) are equal to those calculated by the guard in $V_{G_{\phi(A)}}$ of Definition 2.

The example program can now be written as:

$$L :: length(List, L) \sim El :: umember(El, List) \qquad (G_5)$$

$$\frac{1}{L} :: L :: umember(El, [El|Tail]). \qquad (C_6)$$

$$\begin{aligned} 1 - \frac{1}{L} :: L :: umember(El, [H|Tail]) \ :- & \qquad (C_7) \\ K \text{ is } L - 1, & \\ K :: umember(El, Tail). & \end{aligned}$$

The probabilistic expressions of clauses (C_6, C_7) will be computed at resolution time. Clause (C_6) is labelled by $\frac{1}{L}$ where L is the length of the input list (as defined by standard predicate $length/2$). L is computed at run-time by calling $?-length(List, L)$ which when called with a free variable L and a list of values $List$ instantiates L to the length of the list. Clause (C_7) claims the residual probability $(1 - \frac{1}{L})$. The recursive call has a label which carries forward the length of $Tail$, the tail of the input list. This is one less than that of list $[H|Tail]$. By adding K as the label to the goal $K :: umember(El, Tail)$, we avoid recomputing the guard. For the query $?-umember(X, List)$ where $List$ is a known list, the above program defines a uniform distribution over all the possible element selections from this list. The probabilities for the three possible selections (substitutions) for query

$$?-umember(X, [a, b, c]). \qquad (Q_1)$$

are all equal to $\frac{1}{3}$ and are computed by $\frac{1}{3}$, $\frac{2}{3} \times \frac{1}{2}$, and $\frac{2}{3} \times \frac{1}{2} \times 1$ for $X = a$, $X = b$ and $X = c$ respectively.

A distributional logic program R is the union of a set of definite clauses L and a set of probabilistic clauses D. Both sets define the logical semantics of R in unison while D also defines a distribution over the substitution of the logical queries posed against R.

4. Probabilistic paths

Recall the description of SLD-resolution given in Section 2. For query goal A let A_i be the selected goal for resolution at step i and M_i a matching probabilistic clause. We define I the index of M_i in R, such that $M_i = R_I$ (where R_I is the Ith clause in R). E_i the probabilistic expression of M_i, G_i the guard of M_i (G_i is short for $G_{\phi(M_i)}$ as defined in Defn. 1) and θ'_{G_i} to be a complete substitution that grounds all the input variables in G_i ($V_{A_{\phi(M_i)}}$ in Defn. 1). Let $\vdash$ be the normal derives operator defined on the logical parts of R and eval(E) a function that evaluates arithmetic expression E. The refutation choice π_i, that is the selection of which clause was used for the ith step can be captured by

$$\pi_i = \begin{cases} (I, eval(E_i/\theta'_{U_i})) & \text{if } M_i \in D \wedge R \vdash U_i/\theta'_{U_i} \\ I & \text{if } M_i \in \mathrm{L} \end{cases}$$

The intuition is that a choice point can be abstracted to the identity of the clause used and the value of the label (probability) at run-time. Each refutation that involves a probabilistic clause produces answers of the form $A/\theta\pi$ where $\pi = (\pi_1, \ldots, \pi_N)$ is the path of the refutation along with the probabilities attached to each choice. Similarly for iteration i we define $\pi(A_i) = (\pi_1, \ldots, \pi_{i-1})$, the partial path to that point. The procedure described above does not specify which literal $\neg A$ in the current goal is chosen for the next resolution step as our analysis is independent of this choice.

For query A we define Λ_A to be the set of probabilistic variables in A. $V \in \Lambda_A$ if and only if it is a variable in A and it will be matched during refutation to the output variables of a guard ($V \in \Lambda_{H^\epsilon}$ as per Defn. 1). We restrict the class of allowable combinations of programs and queries (A) so that each V can only be further instantiated by unifications to the same predicate. In the $umember/2$ example (G_5,C_6,C_7) and for query $?-umember(X, [a, b, c])$, X is a probabilistic variable and it only gets instantiated by the first argument of $umember/2$ which belongs to $\Lambda_{umember/2}$ (see, (G_5) and Defn. 2). Similarly, the probabilistic variables of a clause are those identified by the predicate guard, thus, $\Lambda_{\phi(M_i)}$ are the probabilistic variables of M_i.

We define $\pi^{\sim}(A_i)$ as the list of choice points (clause indices) in the path that have so far involved unification of the probabilistic variables of M_i. For example at the end of deriving $?-umember(X, [a, b, c])$. for $X = c$ against the program in clauses (G_5,C_6,C_7), we have $\pi^{\sim}(member(X, [a, b, c])_{X=c}) = \{C_7, C_7, C_6\}$, that is the probabilistic variable was passed through twice the recursive clause (C_7) and once through the base, terminating, clause (C_6). In this example, $\pi(member(X, [a, b, c])_{X=c}) = \{(C_7,2/3),(C_7,1/2),(C_6,1)\}$. In general, $\pi^{\sim}(A_i)$ is a subset of the choice points that appear in $\pi(A_i)$. Choice points that are in $\pi(\cdot)$ but not $\pi^{\sim}(\cdot)$ are either missing due to probabilistic goals that do not involve the probabilistic variables in the query, or due to refutations of non-probabilistic goals.

More formally, $\pi^{\sim}(A_i) = (J \cdot (J, X_j) \in \pi(A_i) \wedge T \in \Lambda_{\phi(M_j)} \wedge V \in \Lambda_{A_j} \wedge V/\theta_j \prec T)$, where $V/\theta_j \prec T$ is true if and only if θ_j is a non empty substitution and $V/\theta_j = T$. That is, V is not a variable renaming equivalent term to T. Recall that θ_j is the j step unification substitution within $\pi(A_i)$. Function $uniq(\theta, \pi, i)$ is defined to be true if and only if $\pi^{\sim}(A_i)$ is unique for each distinct $A/\{\theta_1, \theta_2, \ldots, \theta_{i-1}\}$. Let $\pi^{\triangleleft}(A_i)$ be the sum of probabilities for clauses that lead to at least one refutation when they are resolved against A_i given that the choices leading to A_i are $(\pi_1, \ldots, \pi_{i-1})$.

Definition 4. *For program R, goal A and substitution θ we define the following probability distribution:*

$$P_R(A/\theta) = \sum_{\pi \cdot R \vdash A/\theta\pi} \prod_{i \cdot uniq(\theta,\pi,i)} \frac{eval(E_i)}{\pi^{\triangleleft}(A_i)} \tag{1}$$

The formulation of $P_R(A/\theta)$ allows R to contain both probabilistic and logical predicates. However, only programs that can guarantee the uniqueness constraint,

$uniq(\cdot)$ have well-defined distributions. The constraint states that each $A/\{\theta_1, \theta_2, \ldots, \theta_{n-1}\}$ should correspond to a unique probabilistic path $\pi^{\sim}(A_i)$ thus ensuring that a well-defined probabilistic space is considered.

The denominator in (1) normalises each choice ensuring that $\sum_\theta P_R(G/\theta) = 1$. In the naive case calculating $\pi^{\triangleleft}(A_i)$ is a prohibitively expensive operation, as at least one derivation for each possible continuation needs to be found. However, there are two interesting special cases. The first is when R can be shown to have no probability mass loss in which case $\pi^{\triangleleft}(A_i) = 1 \Rightarrow \sum_{\theta \in \Theta} P(G/\theta) = 1$ and the second when all unifiable clauses for A_i $(u(A_i))$ lead to at least one θ, then $\pi^{\triangleleft}(A_i)$ is replaced by $\sum_{a \in u(A_i)} eval(E_a)$. Both properties may hold even in the case of infinite i and are static properties of a program with regard to a class of queries. The main intuition behind Definition 4 is that the narrowing or instantiation of probabilistic variables in G can be traced via the probabilistic variables in clauses used in resolution. Unification operations involving these variables are of particular interest as they define a single probabilistic space. Programs and associated queries for which nondeterministic and probabilistic predicates that incur no direct manipulation to the query probabilistic variables produce a single $\pi^{\sim}(A_i)$ for each probabilistic A_i define a unique probabilistic space that provides a sound platform for doing inference. This is certainly the case in the context of the inference we describe later in this paper, but likely to also hold for other types of probabilistic inference. In terms of the relation between $\pi(\cdot)$ and $\pi^{\sim}(\cdot)$ these well behaved programs are characterised by the fact that there is an one-to-one mapping between $\pi(\cdot)$ and $\pi^{\sim}(\dot{})$. This is a property of the program, and not a test of admissibility for the inference described herein. We will term such programs and associated queries as exhibiting *"probabilistic determinism"*. In general, $\pi^{\sim}(\cdot)$ provides an abstract operational means for describing probabilistic aspects of probabilistic logic programs.

4.1. Paths in generative languages

The distribution we defined over substitutions has similarities with both the distributions over observables by [18] in PCCP and yields by [15] in pure normalised SLPs. However, both these formalisms steer clear of mixing probabilistic reasoning and nondeterminism. PCCP replaces the nondeterministic operator of CCP by a probabilistic one and thus it does not allow any nondeterminism. This corresponds to programs where non probabilistic predicates are defined by single clauses. Clearly the uniqueness constraint is satisfied in this case. Furthermore, as PCCP is not concerned with failures, the sum of matching $eval(E_i)$ can be used instead of $\pi^{\triangleleft}(A_i)$.

A stochastic clause in SLPs is a clause labelled by a number. [13] presented log-linear semantics for pure normalised SLPs. This is a program that only contains labelled clauses. Following standard statistical practice, their semantics normalise over the partition function (Z), which for an SLP is equal to the sum of probabilities over $\cup\theta$. A pure normalised SLP corresponds to a Dlp program that has just numbers as expressions, guards are the true goal having no guard variables and all head variables in a clause are considered probabilistic. When there is no probability mass loss in the C_i values then the SLP semantics and those presented here are equivalent since both normalising factors (Z and $\pi^{\triangleleft}(A_i)$) are equal to 1. It is straightforward to see that any pure normalised SLP can be mapped to a valid Dlp program. However, the opposite is not true. For instance, there is no intuitive way to code the $umember/2$ program defined by clauses (C_6-C_7). The semantic arguments we have put forward in this paper can be viewed as an operational extension to the semantics given by Cussens [15]. Probabilistic paths provide a constructive history of the proof that can be used to pinpoint classes of programs and queries with desirable properties. In particular, we have argued that the desirable behaviour of pure, normalised SLPs can be seen as one manifestation of the general property of *probabilistic determinism*.

The logical-statistical programming language PRISM [35] provides a single primitive, $msw/3$, which implements a *switch*: a selection of an element from a list according to an explicit fixed distribution over the list elements. This primitive is a fact, that is a clause with an empty body. In order for a query to have correct distributional semantics, each θ should have a one-to-one correspondence with possible switch outcomes. PRISM presents an appealing primitive abstract machine for mixing logic programming and probabilistic reasoning. It has been used in efficient parameter estimation [34].

SLPs, PRISM and the much earlier formalism the *Independent Choice Logic (ICL)* [31, 32] have been shown to be very closely related. In each case a simple 'base' joint probability distribution is defined as a product of independent random variables. This is then extended to a more complex probability distribution using a logic program. In SLPs, the base distribution is over the choice of clause. In PRISM the base distribution is defined by the *switches*. In ICL the base distribution is defined using *alternatives*: sets of logical atoms exactly one of which can be true. Details can be found in [16].

Dlp extends this approach in one crucial way: the probabilities defining the 'base' distribution are not *fixed*. In contrast to SLPs, PRISM and ICL, Dlp probabilities can be computed 'on the fly'. They can be functions of the current goal. As explained above Dlp is most easily understood as an extension of SLPs that allow for dynamic computation of clause selection probabilities. Since Dlp allows both probabilistic and non-probabilistic clauses it is, more precisely, an extension of *impure SLPs* [14]. Impure SLPs have both fixed-probability-labelled and unlabelled clauses, a combination which is necessary for an SLP to represent a PRISM program (which can have clauses with or without switches).

5. Priors over statistical models

We have shown previously [13] how graphical models can be represented as logic programming terms. For instance, the structure of a BN with nodes $1, 2$ and 3, and two parent edges: $1 \rightarrow 3$ and $2 \rightarrow 3$ is mapped to term $[1 - [3], 2 - [3], 3 - []]$. Logic programs can be written to define the space of all models (e.g. all BNs with N nodes). The theoretically sound properties of logic programs can provide a suitable platform for representing domain knowledge. Further, Dlp ascribes probabilities to each possible statistical model. We show how Dlp can define effective priors over two model spaces: classification trees and BNs.

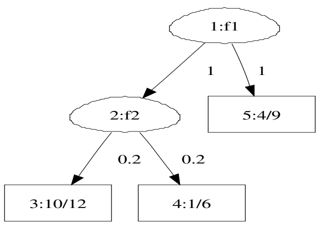

Figure 1: Classification tree: ovals are decisions on a single feature with an associated splitting value. Leaf nodes present a distribution over classes.

5.1. Classification and regression trees

Classification and regression trees [17] use a number of decisions on features to classify each element to one of a number of possible classes (classification) or fit a distribution over a range (regression). An example is shown in Fig. 1. A classification tree (similarly read for regression tree), splits a dataset on a number of features as to make decisions on which class does specific data-points belong to. Internal nodes are decisions, where those data with a value above a threshold on a specific feature go to the right branch and those with values below are placed to the left branch of the decision. Leaf nodes present a distribution over classes that is proportional to the ratio of class data-points at the leaf node. The feature of the root node of the tree in Fig. 1 is *f1* and the threshold value is 1. In the rightmost leaf node, 4 examples belong to class 0 and 9 examples belong to class 1. Having learned such a tree from training data, it can be used to provide predictions of class for data in which the class is unknown. [12] uses a prior over the set of trees T that depends on the probability of splitting individual nodes:

$$\psi_\eta = \alpha(1 + e_\eta)^{-\beta} \tag{2}$$

$$p(T) = \prod_{\eta \in H^I} \psi_\eta \prod_{\eta \in H^L} 1 - \psi_\eta \tag{3}$$

where e_η is the depth of node η, α and β are user defined parameters controlling the size of the trees, H^I is the set of internal nodes for T and H^L is its set of leaf nodes. $p(T)$ is the prior probability of tree T and ψ_η is the probability of splitting node η. In what follows we present a Dlp program for this prior. The operator 'is' assigns an arithmetic value to a free variable. Note that logic programming does not support destructive assignment. The corresponding Dlp program, from which we first shown a fragment in [4, Fig. 4], is:

$$\begin{array}{lrl}
(A_0) & cart(D, Cart) :- & \\
 & & parameters(\psi_0, \beta), \\
 & \psi_0:: & split(0, D, Cart).
\end{array}$$

$$\begin{array}{lrl}
(A_1) & \psi_H :: \psi_H:: & split(E_H, D_H, c(F, Val, L, R)) :- \\
 & & parameters(\alpha, \beta), \\
 & & E_{H_1} \text{ is } E_H + 1, \\
 & & \psi_{H_1} \text{ is } \alpha * E_{H_1}^{-\beta}, \\
 & & r\_select(F, Val, D_H, D_L, D_R), \\
 & & \psi_{H_1}:: split(E_{H_1}, D_L, L), \\
 & & \psi_{H_1}:: split(E_{H_1}, D_R, R).
\end{array}$$

$$(A_2)\ 1 - \psi_H :: \psi_H:: \quad split(E_H, D_H, l(D_H)).$$

$$(A_3) \quad parameters(\alpha, \beta).$$

Note that the above program does not contain any guards as the probability labels are calculated within the clauses and are passed via the recursive call via probabilistic clauses as defined in Definition 3. Clause (A_0) is non-stochastic and serves as a convenient entry point. It is called once at the very start of each invocation and it declares that $Cart$ is a valid representation of a tree with prior probability as defined in (3). For each split at depth E_H the leaf nodes are considered in turn with a decision made for each of them as to whether a node is to be split or not. Clauses (A_1) and (A_2) correspond to the two possibilities. The node will either become an internal one (A_1) or a leaf (A_2). Note that although the data D is part of the input to the program, this is purely for populating the tree and has no effect on the probability for each split decision which defines the shape of the tree and the associated prior value.

Clause (A_1) states that a sub-tree is initiated at node H by randomly selecting $(r\_select/5)$ a feature F and an associated splitting value Val and using those to partition the data D_H into two distinct parts D_L and D_R. It then increases the depth by one, computes ψ_{H_1} and recursively calls itself on D_L and D_R producing the left (L) and right (R) sub-trees. Clause (A_2) constructs a leaf node at level E_H which it populates with all data partitioned to this branch. Each time a split is considered (A_1) is selected with probability ψ_H and (A_2) with the complementary probability $1 - \psi_H$. For instance, at $H = 0$ it is the case that $\psi_H = \alpha$. The parameters are recorded in (A_3). The Dlp program captures the essence of the prior in an elegant and abstract way. SLPs cannot model such a generic prior as they only allow fixed values as labels.

The recursive way in which nodes are split has the attractive property that as far as it can go on splitting for ever, the sum of the, infinite, prior is equal to 1. In terms of the paths introduced previously this is a case where $\pi^{\triangleleft}(A_i) = 1$. When applying the prior, however, it is often the case that a lower limit is imposed to the number of data in each leaf. Too few data points lead to over-fitting. From the prior's perspective, trees that do not meet the condition have zero probability leading to a probability mass loss $(Z < 1)$. Note that the MH algorithm presented in this paper only requires the ratio of priors, so the value of Z is immaterial. In cases where we need to redress the probability loss the Dlp program can be changed to perform a one step look-ahead and redistribute the lost mass.

5.2. Bayesian Networks

One approach to constructing the acyclic directed graph for a BN is by recursively choosing parents for each of its nodes as we have shown in [6, Fig. 2]. Care must be taken however as to avoid introducing cycles in the graph. This method is well suited to situations where prior information regarding edges in the graph is available. The top level, non-stochastic part of the selection expressed in logic programming is:

$$
\begin{array}{ll}
(B_1) & bn(Nds, BN) :- \\
& \quad bn(Nds, Nds, BN), \\
& \quad no\_cycles(BN). \\
\\
(B_2) & bn([\,], \_Nds, [\,]). \\
(B_3) & bn([Nd|Nds], AllNds, BN) :- \\
& \quad parents\_of(Nd, AllNds, Pa), \\
& \quad BN = [Nd - Pa|TBN], \\
& \quad bn(Nds, AllNds, TBN).
\end{array}
$$

Given a list of nodes, Nds, for which we wish to construct a Bayesian network, BN, predicate $bn/2$ constructs a candidate graph and then checks that the graph is acyclic. Predicate $bn/3$ traverses the nodes selecting parents for each one of them. When an ordering is known for the variables in the BN, its construction can proceed without checking for cycles. The ordering constraint [21] specifies that the order of nodes (Nds) is significant and that each node can only have parents from the section of the ordering that follows it.

$$
\begin{array}{ll}
(B_4) & bn(Nds, BN) :- \\
& \quad bn(Nds, [\,], BN). \\
\\
(B_5) & bn([\,], \_AllNds, [\,]). \\
(B_6) & bn([Nd|Nds], PossPa, BN) :- \\
& \quad parents\_of(Nd, PossPa, Pa), \\
& \quad BN = [Nd - Pa|TBN], \\
& \quad bn(Nds, [Nd|PossPa], TBN).
\end{array}
$$

Clauses ($B_4 - B_6$) provide a compact implementation of the ordering constraint. The program is also robust in relation to the probabilistic paths associated with the model instances they generate. Each model has a unique non-probabilistic part with regard to this program segment and it never leads to a failure. On the contrary clauses ($B_1 - B_3$) lead to failure and loss of probability mass when a cycle is introduced. This can only be detected after some probability is assigned to the failed path. Clause (B_6) selects parents for a node from the set of possible parents rather than the set of all nodes. Also, when the ordering is not known (program $B_1 - B_3$) there is no good reason why child variables should be selected in order. The following program merges these two ideas:

$$
\begin{array}{ll}
(B_7) & bn(Nds, BN) :- \\
& \quad bn(Nds, Nds, [\,], BN).
\end{array}
$$

$$
\begin{array}{ll}
(B_8) & bn([\,], \_All, BN, BN). \\
(B_9) & bn(Nds, All, BnSoFar, BN) :- \\
& \quad p\_member(Nds, Nd, RemNds), \\
& \quad poss\_pa(Nd, BnSoFar, All, PossPa), \\
& \quad parents\_of(Nd, PossPa, Pa), \\
& \quad add(Nd - Pa, BnSoFar, NextBnSF), \\
& \quad bn(RemNds, All, NextBnSF, BN).
\end{array}
$$

The parent node is selected with relative probability ($p\_member/3$) from the nodes available. Clause (B_9) utilises an auxiliary structure $BnSoFar$ which accumulates the graph of the BN at the current level. This is used by $poss\_pa/4$ to eliminate cycle-introducing parents which is taken care by $add/3$. Clause (B_8) terminates the recursion and unifies the auxiliary structure to the BN model. A number of distributions from the literature can be fitted over the edge selection that connects children in the BN to their parents. [24] introduced $p(BN) \propto \kappa^{\delta}$ where κ is a user defined parameter and δ is the number of differing edges/arcs between BN and a 'prior network' which encapsulates the user's prior belief about the network structure. [9] suggested a generalisation of the above that allows for arbitrary weights for each missing edge: $p(BN) \propto \sum_{ij} \kappa_{ij}$ where i and j refer to the end nodes of an edge. Assuming Ws is a list of pairs matching nodes to weights for parents for this node ($Ws = [Nd_1 - W_1, \ldots Nd_n - W_n]$ with $W_i = [W_{i,1}, \ldots, W_{i,n}]$) and $KnownPa_i$ is the list of parents of the ith child in the prior network, then we have:

$$
\begin{array}{ll}
(B_10) & bn([\,], \_All, \_Ws, \_KnownBn, BN, BN). \\
(B_11) & bn(Nds, All, Ws, KnownBn, BnSoFar, BN) :- \\
& p\_member(Nds, Nd, RemNds), \\
& poss\_pa(Nd, BnSoFar, All, PossPa), \\
& member(Nd - NWs, Ws), \\
& member(Nd - KnownPa, KnownBn), \\
& parents\_of(PossPa, KnownPa, NWs, Pa), \\
& add(Nd - Pa, BnSoFar, NextBnSF), \\
& bn(RemNds, All, Ws, KnownBn, NextBnSF, BN).
\end{array}
$$

$$
\begin{array}{ll}
(B_{12}) & parents\_of([\,], \_KnownPa, \_NWs, [\,]) \qquad . \\
(B_{13}) & parents\_of([PP|PPs], KnownPa, NWs, Pa) :- \\
& member(PP, KnownPa), \\
& parents\_of(PPs, KnownPa, NWs, Pa).
\end{array}
$$

$$
\begin{aligned}
(B_{14})\quad & parents\_of([PP|PPs], KnownPa, [W_{ij}|NWs], Pa) :- \\
& \qquad not(member(PP, KnownPa)), \\
& \quad W_{ij} :: parent\_edge(PP, Pa, TPa), \\
& \qquad parents\_of(PPs, KnownPa, NWs, TPa).
\end{aligned}
$$

$$
\begin{aligned}
(B_{15})\quad & W :: W :: parent\_edge(PP, [PP|TPa], TPa). \\
(B_{16})\; 1-W & :: W :: parent\_edge(PP, TPa, TPa).
\end{aligned}
$$

Clause (B_{14}) utilises probabilistic goal calling by attaching W (W_{ij}) to the goal $parent\_edge/2$. As far as the W_{ij}s sum up to 1 for a single value of i the program leads to no probability loss. This can be enforced by the use of a simple guard that first sums the weights, making sure the add up to 1, and then simply selects the j item from i's list. In the context of learning from expression array data [40] constructed tabular priors over the existence of some edges. This is complementary to penalising missing edges. A very similar program to that presented above can capture such knowledge. Other constraints such as the one proposed in [21] where the number of parents is limited can be naturally encoded in our language.

5.3. Likelihood based learning

Bayesian learning methods either look in the posterior distribution for single models that maximise some measure or seek to approximate the whole posterior. The posterior over models given some data $P(M|D)$ is proportional to the prior and a likelihood function, $P(M|D) \propto p(M)P(D|M)$. Since the space of all possible models is, in all but trivial examples, too large to enumerate, various approximate methods have been introduced. Variational methods [26] approximate the inference on the evidence by considering a simpler inference task while Markov chain Monte Carlo algorithms sample from the posterior indirectly. In this section we discuss the use of the described priors in a Bayesian model averaging scenario and how this general framework can be used in machine learning tasks on biological datasets.

5.3.1. Metropolis-Hastings

Metropolis-Hastings (MH) algorithms approximate the posterior distribution by making stochastic moves through the model-space. A chain of visited models is constructed. At each iteration the last model added to the chain, the current model M, is used as a base from where a new model M' is proposed. M' is stochastically accepted or rejected. The distribution with which M' is reached from M is the proposal $q(M, M')$ and the acceptance probability is given by

$$\propto \frac{p(M')P(M'|D)q(M, M')}{p(M)P(M|D)q(M', M)} \tag{4}$$

To our knowledge all MH algorithms in the literature have distinct functions for computing the prior and the proposal. Standard MH requires two separate computations. The first is the prior over models: $p(M)$, and the second is a distribution for proposing a new model M' from current model M. The proposed model is accepted with probability that is propositional to the ratio given above which also includes the marginal likelihood of the model (measure of goodness of fit to the data $P(M|D)$). This often leads to restricting the choices of either the prior [21] or the proposal [12]. Furthermore writing two programs that manipulate the same model-space means that the algorithms are hard to extend to other spaces. The MH algorithm over Dlp requires the construction of a single program, that of the prior.

Our MH scheme follows that of SLPs, which was briefly sketched in [13] and which we then formally introduced in [3]. The main idea is to use the choices in the probabilistic path as points from which alternative models can be sampled. In effect, the resulting MH is a search space constrained algorithm with the proposal reduced to choosing a backtracking strategy rather than defining operations on growing and reducing model structures. Proposals are thus tightly coupled to the prior and take the form of a function f such that $\pi_j^M = f(\pi^M)$ where π^M is the path produced for deriving model M. π_j is the point from which M' will be sampled. A consequence of defining the proposal q in terms of the prior p is that, due to 'cancelling out', the quantity $\frac{p(M')q(M,M')}{p(M)q(M',M)}$ is easy to compute. The acceptance probability (4) is just $\frac{p(M')q(M,M')}{p(M)q(M',M)}$ multiplied by the likelihood ratio $\frac{P(M'|D)}{P(M|D)}$, so as long as we can compute the likelihood ratio we have an effective MH method which can use any prior defined using Dlp. We will see how likelihoods (and thus likelihood ratios) can be computed for C&RT s and BNs in Sections 5.4 and 5.5 respectively.

Dlp provides a clear connection between computed instantiations and probabilistic choices. The Metropolis-Hasting algorithm can then operate over the priors defined by these programs. Dlp is a powerful probabilistic representation that marries work from the AI community with Bayesian statistics.

5.4. Marginal likelihood for C&RT

Given a particular C&RT classification tree T, let $\Theta = \{\theta_i\}_i$ be the parameters of the tree, where θ_i is the distribution over classes in leaf i of the tree. Let (X, Y) be the data, where Y is to be classified into one of K possible classes based on predictor variables X. The likelihood we need for our MH algorithm is the marginal likelihood $P(Y|X, T)$, which requires Θ to be integrated out of the full likelihood $P(Y|X, \Theta, T)$. This integration is with respect to the structure-conditional parameter prior $P(\Theta|T, X)$, which we define in the standard way. Firstly, we choose to define the prior to be independent of X so that $P(\Theta|T, X) = P(\Theta|T)$. Secondly, we assume independence between the θ_i in Θ, so that $P(\Theta|T) = \prod_{i=1}^{b} P(\theta_i|T)$ where I is the number of leaves in T. θ_i is $(p_{i1}, \dots p_{iK})$: a class probability distribution in T, with p_{ik} being the probability of a sample being of the kth class given that it is placed in the ith leaf of tree T. For all i, we set $P(\theta_i|T)$ to the same distribution: a Dirichlet distribution with parameters $(\alpha_1, \dots, \alpha_K)$, for some user-defined choice of α_k $(1 \leq k \leq K)$.

With such a parameter prior there is a closed form for $P(Y|X, T)$ as noted by [12]. Let n_i denote the number of examples at leaf i in tree T. Let n_{ik} be the number of examples of class k reaching leaf i, (so that $n_i = \sum_k n_{ik}$). Then the marginal likelihood is:

$$p(Y|X, T) = \left(\frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \right) \prod_{i=1}^{b} \frac{\prod_k \Gamma(n_{ik} + \alpha_k)}{\Gamma(n_i + \sum_k \alpha_k)} \qquad (5)$$

5.5. Marginal likelihood for BNs

As for C&RT models, we require the marginal likelihood for Bayesian networks (BNs), and so again have to integrate away the parameters. In the case of BNs these parameters are conditional probabilities and we have to define a prior distribution for each of them. We adopt the entirely standard approach of requiring these to be Dirichlet distributions. For the details of this approach see [25]. If various independence assumptions are made about the joint distribution over parameters we have previously shown in the context of Dlp [6, Eq. 5] that for any BN G and observed data X:

$$p(X|G) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(n_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(n_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})} \tag{6}$$

where i ranges over the n nodes in the BN, j is a joint instantiation of the parents of such a node in G and k is a value of the node. n_{ijk} is the data count of node i having value k when its parents have configuration j. α_{ijk} is the corresponding Dirichlet parameter. Finally, $n_{ij} = \sum_{k=1}^{r_i} n_{ijk}$ and $\alpha_{ij} = \sum_{k=1}^{r_i} \alpha_{ijk}$ where r_i is the number of values for the ith BN variable. In our past experiments we have set $\alpha_{ijk} = N/(r_i q_i), \forall i, j, k$ where N is the *prior precision*) parameter. In the experiments we reported in [6, Section 4.2] we used $N = 1$ and $N = 10$.

5.6. Priors and MH

Using priors for learning statistical models effectively, depends on the structural complexity of the models and the amount of probabilistic bias we wish to impart via the prior. Increased model complexity leads to nondeterministic programs having non-unique probabilistic paths. For instance, the classification trees shown previously are simple recursive structures, the generation of which can be done in a straightforward manner as there are no structural dependencies between the left and right branches of a split node. Thus, the prior program provided was very succinct and it was easy to calculate the probabilities of nodes and leaves. On the other hand, Bayesian networks are more complex as simple programs encoding priors might introduce cycles. As we have shown, guarding against such contextual dependencies leads to more complex programs which are harder to compute with.

Techniques such as using auxiliary arguments can help with unravelling the MH space. Here, we elucidate the effect of priors on MH and explore the difficulties arising in more complex settings.

5.6.1. Classification and regression trees

As detailed above, our approach comprises: a prior, in the form of a Dlp program, an integrated backtracking strategy that proposes new models from old ones and a likelihood which guides the choice between the current and the proposed models. By using a fake likelihood that always returns 1 as the ratio of the two models, we can investigate the effect of priors on the proposed models.

Classification trees have a recursive structure where a node is either a leaf or an internal branching point holding two branches each of which is a tree. The prior detailed above, (Section 5.1), focuses on trees of certain depth and number of leaves. The depth

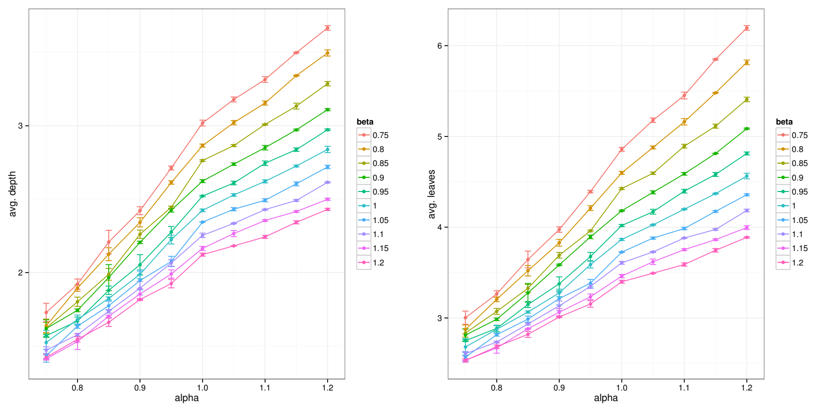

Figure 2: Classification trees prior parameters' effect on models. Left panel: mean depth. Right panel: mean number of leaves.

of a tree is defined as the length of the longest path from a leaf to the root. The higher the two parameters, α and β the deeper the trees and the larger the number of their leaves.

We ran 3 chains of length $10,000$ while varying α and β. The range for the two parameters is $0.75 - 1.2$ with a step increment of 0.05. We then count average depth for the trees visited and number of leaves. Having isolated the effect of likelihood, the shape metrics for the trees solely reflect the effect of prior on the chain construction. Figure 2 shows that increasing values for the two parameters increase the depth and number of leaves for the visited trees.

The Dlp described in clauses ($A0 - A3$) succinctly encodes the desired effect from the mathematical description (2 and 3). Furthermore, the probabilistic and logical parts work recursively in tandem. The probabilistic path that pertains to the decisions of constructing the right branch of an internal node is unperturbed by any decision in the left branch. In the terms of (1), if G_P, G_L and G_R are the goals for the parent, left and right tree respectively,

$$P_R(G_P/\theta_P) = P_R(G_L/\theta_L) * P_R(G_R/\theta_R)$$

Importantly, $\theta_L \perp\!\!\!\perp \theta_R$ and $\pi^{\triangleleft}(A_i) = 1$. The latter states that within each branch the sum of probabilities for all possible choices is equal to 1 thus making the probability of a derivation the exact prior probability of the derived model.

5.6.2. Bayesian networks

Bayesian networks comprise an acyclic directed graph and accompanying conditional tables. Here we focus on learning the structure of BNs. Structurally BNs are more complex than classification trees, as the nodes can be connected to each other arbitrarily. Furthermore, the constraint for acyclicity introduces a contextual interpretation, in which certain edges lead to invalid models.

As shown above, a strategy for programmatically constructing BNs is by working through the list of possible nodes and selecting parents for each node. This recursive

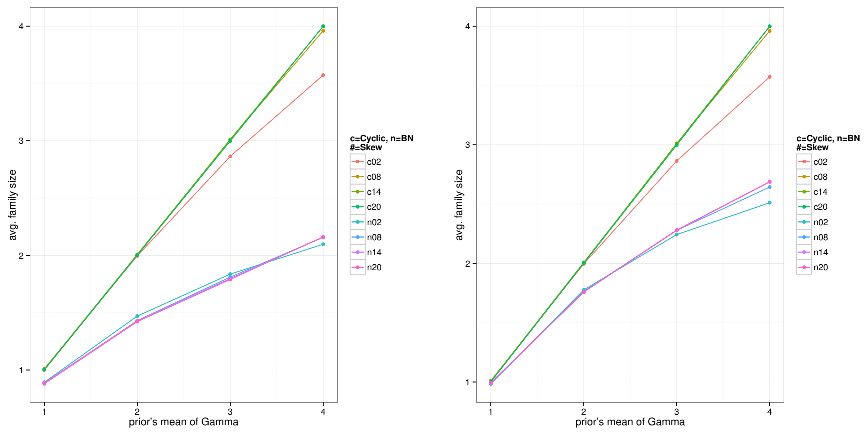

Figure 3: Bayesian networks prior parameters' effect on models. Left panel: cyclic versus deterministic removal of edges. Right panel: cyclic versus prior using current partial network.

logic fits logic programming well and local probabilistic preferences such as controlling the average size of (BN) families. Problems arise from the fact that the choice of parents for one node cannot be done without reference to the complete BN structure. Logic for excluding invalid edges can be added either progressively via auxiliary arguments that hold the part of the network which has been already constructed, or by removing enough conflicting edges at the end, as to produce a legal BN structure. It both cases though, it becomes difficult to keep the prior probabilities to simple, well understood constructs that map intuitively to the logical components.

To explore the discrepancy between intended prior distribution and actual model metrics we use a naive prior that uses the Γ distribution described by a mean, μ, and skewness parameter κ. The construction proceeds unconcerned by any cycles introduced. Those are removed in a deterministic manner by removing edges until an acyclic structure is derived (not shown).

$$(G_2) \quad P :: \Gamma(\mu, \kappa, X), length(Nodes, L), P\ is\ (L-1)/X \\ \sim Pa :: family(Nodes, \mu, \kappa, Pa).$$

$$(D_1) \quad cyclic\_bn([], \_, \_, \_, []).$$

$$(D_2) \quad cyclic\_bn([H|T], \mu, \kappa, Nodes, [H-Pa|TCy]) :- \\ select(H, Nodes, PotPa), \\ family(PotPa, \mu, \kappa, Pa), \\ cyclic\_bn(T, \mu, \kappa, Nodes, TCy).$$

$$(D_3) \quad P :: P :: family([H|T], \mu, \kappa, [H|TPa]) :- \\ P :: family(T, \mu, \kappa, TPa).$$

$$(D_4)\ 1-P :: P :: family([\_H|T], \mu, \kappa, TPa) :- \\ P :: family(T, \mu, \kappa, TPa).$$

We ran 3 chains of length $10,000$ while varying μ and κ. The ranges for the μ and κ parameters are $1-4$ and $2-20$ with step increments of 1 and 6 respectively. The average family size for the structures visited is then calculated. We first ran the program as shown in clauses ($D_1 - D_4$), experiments $c02 - c20$ in Figure 3 which produce cyclic structures. The prior was used to construct 8-node BNs. We then ran experiments where edges were removed deterministically until a valid BN is reached. The left panel of Figure 3 shows the differential in average family size between the cyclic structures, which faithfully produce networks having average family size equal to the mean of the prior. In contrast, the removal step in the acyclic experiments reduce the average family size. The higher the prior mean the higher the proportion of edges removed as the networks become denser.

We repeated the experiments with an alternative prior in which an auxiliary variable held the partial network already constructed. Only candidate edges that do not introduce cycles are considered. For small values of the mean (μ) this prior follows the cyclic distribution closely (Figure 3, right panel). However, as μ comes closer to the number of nodes in the net and the networks become denser, the prior encounters smaller candidate sets from which to chose parental edges that would not introduce a cycle. Thus, the overall objective of a family size of μ cannot be fulfilled.

Writing prior programs for Bayesian network structure is a much harder task than that of composing priors for classification trees. In the cases where cycles are removed from full cyclic networks, the precise probability of a single model is hard to determine as there are multiple paths leading to the same mode. As has been shown experimentally the precision of even simple priors on the graphs suffer. When auxiliary arguments are used, the results can be better particularly for reasonably sparse networks. In addition, each structure has a single derivation path.

Although BN priors are complex to characterise mathematically, in practice experimentation with unit likelihoods which set the likelihood of every mode to 1 can provide close enough approximations that can be effectively used for experimenting with constructing BNs in a specific domain and evaluate the effect of the prior by factoring out the effect of the likelihood.

6. Practical MH

Although Bayesian reasoning is a conceptually simple framework, in practice a number of hurdles have to be cleared before one can use this framework for effective inference. These include expedience of calculations and visiting representative sections of the posterior within a reasonable number of iterations. Improvements in the literature include parallel tempering [22] and fast likelihood calculations. Tempering is a mixing promoting technique, where a number of chains are run in parallel with swapping moves between chains occurring stochastically. Each parallel chain runs at a different likelihood temperature, a parameter that controls the smoothness of the likelihood space as to, in turn, control fluidity of chain movement.

Bayesian reasoning over model structures defined by Dlp priors has to address much the same issues as other approaches of MH inference. Dlp benefits from the fact that it needs no proposal and in particular it does not need to calculate the proposal probabilities for proposing M' from M and vice versa 4. Instead, the probabilistic choices in the derivation of M are used as possible backtrack points from which the computation for M' can be embarked on as an alternative probabilistic path. Selecting these points uniformly can lead to unbiased exploration of the space. As all instantiations done for M below the chosen point are undone these type of backtracking can lead to inefficient sampling. Interestingly, the independence of branches in classification trees with regard to θ can also be exploited to provide more efficient backtracking. Returning to consider the construction of a branch in a tree should leave choices for the other branch unaffected. Executing in a naive fashion would mean that backtracking to a choice in the left branch would remove all choices for the right branch.

A system implementing the main MH algorithm over Dlp priors of model structures as well as advanced features such as declaration of independent paths have been implemented in the Bims system [2]. The system supports expression of Dlp priors, implements a number of in-built likelihoods as well as providing a simple interface for expressing additional likelihoods. The system produces chains of visited models and provides a number of functions for extracting information from such chains. Here we present summaries from two publications that have used priors similar to those presented here.

6.1. Ligand discovery with classification trees

[7] has shown that MH over Dlp scales well and does at least as well as other machine learning algorithms in a cross validation experiment over a large feature space. [7] employed the language and algorithm presented here, to learn the binding affinity of molecules to the pyruvate protein (PYK). Their data were constructed from molecules with known binding affinities from a large NIH funded screening programme. The datasets used contained 582 molecules from each of the two classes: binders and non-binders to PYK. A high dimensional learning space was constructed by associating each molecule with a large number of features (1572). The study established, by means of a ten-fold cross-validation, that the MH algorithm performed at least as well if not better, in terms of overall accurate prediction of positives, than Support Vector Machines and Feed-Forward Neural Networks. [7] also describes advanced tempering techniques in the context of their experiments.

6.2. Bayesian networks for binding assays

MH over Dlp-defined BN priors was used in [8] to build Bayesian networks (BNs) from binding affinity data on 43 chromatin proteins. A gamma distribution over parents was used with a mean value of 1.2 for the average family. The dataset comprised 4380 measurements and was discretised by setting the strongest 5% of protein measurements to 1 and setting the rest to 0. The dataset was originally presented in [38] and was analysed using bootstrapping of simulated annealing searches as implemented in the Banjo software for learning BNs [36]. [38] built a consensus network at the 80% of edge presence. The networks built using Dlp were also summarised with the same criterion on the posterior and the resulting network was compared to the original. The two networks agreed to a large extent, with 40 common edges and 9 edges appearing in only the first network and 6 only in the second network. These experiments demonstrated that Dlp can digest substantial datasets and learn core interactions that are in agreement with other algorithms.

7. Conclusions

This paper introduced a general programming language for combining nondeterminism and probabilistic reasoning in logic programming specially for the purpose of defining prior Bayesian knowledge. The language syntax is presented along with the mathematical concept of probabilistic path that can be used to give semantics in such languages. Furthermore, we focused discussion on how to represent knowledge from the literature.

The paper also presented a characterisation that can be useful in the context of deriving probabilistic information in the form of distributions over variables. This characterisation is of relevance not only to Dlp but also to other generative formalisms that combine logic and probability. We have argued that for certain classes of programs the kind of knowledge that can be represented is substantially expanded. Furthermore, we illustrated via examples on how to write correct and efficient programs that capture knowledge from the Bayesian learning literature.

The main challenges ahead lie in the use of static analysis and program transformations to create programs that exhibit the desirable properties that have been described here, from more arbitrary ones.

We have highlighted that the theoretical framework can compete in producing practical results against standard, well established machine learning approaches. The ability to express prior knowledge can give this approach extra leverage in domains where such knowledge is available.

Availability

The programming language and reasoning techniques described in this paper are implemented in the Bims system which is written in Prolog and freely available from `http://stoics.org.uk/~nicos/sware/bims`. It can be easily installed and used in two Prolog system: *SWI-Prolog* and YAP.

References

[1] Nicos Angelopoulos. *Probabilistic Finite Domains.* PhD thesis, City University, London, UK, 2001.

[2] Nicos Angelopoulos. Bims: Bayesian Inference of Model Structure, 2015. `http://stoics.org.uk/~nicos/sware/bims`.

[3] Nicos Angelopoulos and James Cussens. MCMC using tree-based priors on model structure. In *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 16–23, University of Washington, Seattle, Washington, USA, August 2-5 2001.

[4] Nicos Angelopoulos and James Cussens. Exploiting informative priors for Bayesian classification and regression trees. In *19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 641–646, Edinburgh, UK, August 2005.

[5] Nicos Angelopoulos and James Cussens. Tempering for Bayesian C&RT. In *22nd International Conference on Machine Learning (ICML)*, Bonn, Germany, August 2005.

[6] Nicos Angelopoulos and James Cussens. Bayesian learning of Bayesian networks with informative priors. *Journal of Annals of Mathematics and Artificial Intelligence*, 54(1-3):53–98, 2008.

[7] Nicos Angelopoulos, Andreas Hadjiprocopis, and Malcolm D. Walkinshaw. Bayesian ligand discovery from high dimensional descriptor data. *ACS Journal of Chemical Information and Modeling*, 49(6):1547–1557, 2009.

[8] Nicos Angelopoulos and Lodewyk Wessels. Effective priors over model structures applied to DNA binding assay data. In *Probabilistic Problem Solving in BioMedicine, (ProBioMed'11)*, 2011.

[9] Wray Buntine. Theory refinement on Bayesian networks. In *Proceedings of the 7th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 52–60, Los Angeles, California, USA, 1991.

[10] Wray Buntine. Learning classification trees. *Statistics and Computing*, 2(2):63–73, 1992.

[11] Soumen Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *Proceedings of the 16th international conference on World Wide Web*, pages 571–580, 2007.

[12] Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. Bayesian CART model search (with discussion). *Journal of the American Statistical Association*, 93:935–960, 1998.

[13] James Cussens. Stochastic logic programs: Sampling, inference and applications. In *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 115–122, Stanford University, Stanford, CA, USA, 2000.

[14] James Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.

[15] James Cussens. Statistical aspects of stochastic logic programs. In *Artificial Intelligence and Statistics*, pages 181–186, 2001.

[16] James Cussens. Integrating by separating: Combining probability and logic with ICL, PRISM and SLPs. APRIL project report, January 2005.

[17] David G. T. Denison, Christopher C. Holmes, Bani K. Mallick, and Adrian F. M. Smith. *Bayesian Methods for Nonlinear Classification and Regression*. Chichester: Wiley, March 2002.

[18] Alessandra Di Pierro and Herbert Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *Proceedings of IEEE Computer Society Conference on Computer Languages*, pages 174–183, 1998.

[19] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for AI*. Morgan & Claypool, 2009.

[20] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 99, pages 1300–1309, 1999.

[21] Nir Friedman and Daphne Koller. Being Bayesian about network structure. In *16th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 201–210, 2000.

[22] Walter R. Gilks, Sylvia Richardson, and David Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC, 1996.

[23] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[24] David Heckerman, Dan Geiger, and David M. Chickering. Learning Bayesian Networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[25] David Heckerman, Dan Geiger, and David M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[26] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.

[27] Minoru Kanehisa and Susumu Goto. KEGG: Kyoto encyclopedia of genes and genomes. *Nucleic Acids Research*, 28(1):27–30, 2000.

[28] Angelika Kimmig, Bart Demoen, Luc de Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic LP language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.

[29] Stephen H. Muggleton. Stochastic logic programs. In *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.

[30] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

[31] David Poole. Probabilistic horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

[32] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):5–56, 1997.

[33] Stefan Riezler. *Probabilistic Constraint Logic Programming*. PhD thesis, Tübingen Univ., Germany, 1998.

[34] Taisuke Sato and Yoshitaka Kameya. Prism: A symbolic-statistical modeling language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1330–1335, 1997.

[35] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[36] V. Anne Smith, Erich D. Jarvis, and J. Alexander Hartemink. Evaluating functional network inference using simulations of complex biological systems. *Bioinformatics*, 18:s216 – s224, 2002.

[37] The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25(1):25–9, May 2000.

[38] Bas van Steensel, Ulrich Braunschweig, Guillaume J. Filion, Menzies Chen, Joke G. van Bemmel, and Trey Ideker. Bayesian network analysis of targeting interactions in chromatin. *Genome Research*, 20(2):190–200, 2010.

[39] William Yang Wang, Kathryn Mazaitis, Ni Lao, and William W. Cohen. Efficient inference and learning in a large knowledge base. *Machine Learning*, 100(1):101–126, 2015.

[40] Andriano V. Werhli and Dirk Husmeier. Reconstructing gene regulatory networks with Bayesian networks by combining expression data with multiple sources of prior knowledge. *Statistical Applications in Genetics and Molecular Biology*, 6(1), 2007.

[41] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.