International Journal of Approximate Reasoning 80 (2017) 52–66

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

www.elsevier.com/locate/ijar

Distributional logic programming for Bayesian knowledge representation

Nicos Angelopoulos [a,b,*], James Cussens [c]

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

ARTICLE INFO

Article history:
Received 23 January 2016
Received in revised form 28 July 2016
Accepted 8 August 2016
Available online 17 August 2016

Keywords:
Probabilistic logic programming
Bayesian inference
Bayesian networks
Classification and regression trees
Knowledge representation
Logic programming

ABSTRACT

We present a formalism for combining logic programming and its flavour of nondeterminism 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 Bayesian networks having N nodes while at the same time it defines a prior that penalises networks 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.

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:

$$\textit{posterior} \propto \textit{prior} \times \textit{evidence}$$

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.

* Corresponding author.
E-mail address: nicos.angelopoulos@sanger.ac.uk (N. Angelopoulos).

http://dx.doi.org/10.1016/j.ijar.2016.08.004
0888-613X/

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 [4–7]. 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.

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 $A_1, \ldots, A_n$. 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 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:

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

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

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