probKanren: A Simple Probabilistic extension for microKanren

Robert Zinkov[1], William E. Byrd[2]

[1]Dept. of Engineering Science, University of Oxford, 25 Banbury Rd, Oxford, UK

[2]Hugh Kaul Precision Medicine Institute, University of Alabama at Birmingham, 705 20th Street S., Birmingham, AL 35233, United States of America

Abstract

Probabilistic programming can be conceptually seen as generalisation of logic programming where instead of just returning a set of answers to a given query, we also return a probability distribution over those answers. But many contemporary probabilistic logic programming languages implementations are not simple extensions of existing logic programming languages but instead involve their own unique implementations. Here we introduce probKanren, a simple extension to microKanren that transforms it into a probabilistic programming language without needing to make any modifications to the underlying logic language's search. We use several illustrative examples from the probabilistic programming and program synthesis literature to demonstrate the practicality of the approach.

Keywords

Probabilistic Logic Programming, miniKanren, Probabilistic Programming, Sequential Monte Carlo

1. Introduction

Conceptually, logic programming provides a way to model non-determinism. This is accomplished by maintaining a set of answers that satisfy a set of logical constraints. A natural generalisation to this domain is adding a notion of uncertainty to this set of answers by associating with them a probability distribution.

But the conceptual simplicity of this sort of generalisation is not reflected in the complexity of many existing probabilistic logic programming systems. They often involve making implementing sophisticated algorithms and the underlying systems are not just implemented on top of existing logic programming systems.

We believe a conceptually simple extension deserves a conceptually simple implementation to go along with it. We thus contribute a simple way to extend microkanren a small logic programming DSL such that it becomes a probabilistic programming language.

1.1. Illustrated Example

To help explain how to use probKanren we introduce the following example:

PLP '21: The 8th Workshop on Probabilistic Logic Programming, September 20–27, 2021, Porto, Portugal

zinkov@robots.ox.ac.uk (R. Zinkov); webyrd@uab.edu (W. E. Byrd)

https://zinkov.com/ (R. Zinkov); http://webyrd.net/ (W. E. Byrd)

```
(run 1000 (q)
 (conj
  (normal 0 3 q)
  (normal q 2 4))]
```

This probKanren program draws 1000 samples from a normal distribution truncated below 0.

2. Related Work

There is a rich history of extending logic programming formalisms to support probabilistic inference. Early systems like PRISM[1] and ProbLog[2] allowed associating discrete distributions with facts. Early versions of ProbLog were also built on top of Prolog matching one of the goals of our work. Later work[3] introduces distribution clauses so that some continuous distributions. Other work extended these methods further while focusing on efficient exact inference algorithms like model weight integration[4, 5]. Our work is most similar to [6] except that while they combine their forward reasoning with an importance sampler we use a particle cascade instead which can be more sample efficient.

3. Background

3.1. microKanren

microKanren[7, 8] is a pure logic programming language embedded in Scheme. The language consists of a set of terms, a set of goal primitives, and two run functions to answer queries in the language. The goal primitives consist of a `fresh` form for introducing logic variables, a unification primitive `==`, a conjunction combinator `conj`, a disjunction combinator `disj`, and ways to define and apply relations.

3.2. Grammar and Definitions

To create probKanren we extend this grammar with distribution clauses such as `normal` and `bern`. These are just another type of goal expression. We do not need to add a notion of probabilistic variables as they can just be seen as logic variables constrained in a particular way.

3.3. Probabilistic Programming

Probabilistic Programming Languages[9, 10] are a family of domain specific languages for posing and efficiently solving probabilistic modelling problems. At their core, all have a way to `sample` from a probability distribution and `observe` data generated from a probability distribution.

There are many ways to implement inference algorithms for probabilistic programming languages but methods based on likelihood-weighting and sequential monte carlo algorithms are the easiest.

⟨*prog*⟩ ::= (run ⟨*number*⟩ (⟨*id*⟩) ⟨*goal-expr*⟩)

⟨*goal-expr*⟩ ::= (disj ⟨*goal-expr*⟩ ⟨*goal-expr*⟩)
| (conj ⟨*goal-expr*⟩ ⟨*goal-expr*⟩)
| (fresh (⟨*id*⟩) ⟨*goal-expr*⟩)
| (== ⟨*term-expr*⟩ ⟨*term-expr*⟩)
| (letrec-rel ((⟨*id*⟩ (⟨*id*⟩ ...) ⟨*goal-expr*⟩) ...)
⟨*goal-expr*⟩)
| (call-rel ⟨*lexical-var-ref*⟩ ⟨*term-expr*⟩ ...)
| (prim-rel-call ⟨*lexical-var-ref*⟩ ⟨*term-expr*⟩ ...)
| (delay ⟨*goal-expr*⟩)

⟨*term-expr*⟩ ::= (quote ⟨*datum*⟩)
| ⟨*lexical-var-ref*⟩
| (cons ⟨*term-expr*⟩ ⟨*term-expr*⟩)
| ⟨*term*⟩

⟨*term*⟩ ::= ⟨*number*⟩
| #f | #t
| ⟨*symbol*⟩
| (⟨*term*⟩ . ⟨*term*⟩)
| ⟨*logic-var*⟩

3.4. Sequential Monte Carlo

Sequential Monte Carlo[11] are an efficient online way to sample from probabilistic models especially suited for state-space domains. If we imagine our probabilistic programs as straight-line programs with no control-flow we can imagine numbering every sample function $f_1, f_2, \dots, f_n$ and every observe function $g_1, g_2, \dots, g_n$ then our probability density over our random variables x and observed data y can be defined as:

$$p(x, y) = \prod_i f(x_i \mid x_{0:n-1}) g(y_i \mid x_{0:n}) \tag{1}$$

3.5. Sequential Importance Sampling

If we imagine running this program each execution trace can be seen as a sample from the distribution. If we then weight by their likelihood as we run them that collection execution traces will be an empirical distribution of the given program. We call each of these execution traces particles and the below method is how we obtain them along with their weights w.

$$\begin{align} x_n &\sim f(x_n \mid x_{0:n-1}) \tag{2} \\ w_n^{(k)} &= \frac{g(y_n \mid x_{0:n}) f(x_n \mid x_{0:n-1})}{q(x_n \mid x_{0:n-1})} \tag{3} \\ W_n^{(k)} &= W_{n-1}^{(k)} w_n^{(k)} \tag{4} \end{align}$$

The weights are then normalised at the end to make a proper probability distribution.

3.6. Sequential Importance Resampling

The problem with the above algorithm is over time for many particles W_k is going to become low and that particle will stop being very informative of the underlying distribution. To mitigate this issue, each time we encounter an observation we resample our particles. Resampling effectively removes particles with low weight and duplicates particles with higher weight by sampling with replacement our existing particles.

The above is called multinomial resampling but there are other methods as well. A survey[12] of resampling methods suggests all of them are helpful to reduce particle degeneracy.

3.7. Particle Cascade

As the SMC resampling step was defined in the previous section Particle Cascades [13] remove this barrier allowing every particle to be resampled asynchronously with the associated weights being relative to a global running average.

4. Proposed Method

We propose to extend microKanren by augmenting each of the search streams with a set of particles. These particles represent the empirical distribution of that stream. Each particle has associated with it a substitution of all the logic and random variables as well as a weight that is proportional to the likelihood of the substitution.

An initial set of particles is created from the probabilistic program when it is first run. As disjunctions (`disj`) are encountered, we split evenly the number of particles allocated to each stream. Whenever we encounter a unification primitive, we run a resampling step. This helps to prune low-weight particles and replicate high weight ones.

As an optimisation we may create more particles during resampling based on a globally stored a counter of the effective sample size of all particles across all streams.

We follow [3] and place the following restrictions on our distribution clauses and the random variables they specify.

Firstly, the arguments of distribution clauses must be ground. Secondly, a random variable cannot unify with any arithmetic expression.

This extension does not modify the search and streams are managed exactly as in microkanren. An additional advantage of this is thanks to the microkanren search being complete, if we generate enough particles we are guaranteed to recover the true posterior as all paths of the search space will eventually be explored.

5. Semantics of probKanren

6. Experiments

We validate that probKanren is at least as expressive as other probabilistic logic programming languages by implementing the Friends who Smoke model.

Friends who Smoke is a probabilistic logic program which models the social nature of who smokes cigarettes. The model predicts that people who are friends with people who smoke are more likely to smoke. We replicate the example on https://dtai.cs.kuleuven.be/problog/tutorial/basic/05_smokers.html using 2000 particles and get an empirical distribution that seems to match up with the discrete distribution returned from ProbLog.

7. Conclusions

We made a simple to implement extension to microKanren that let's us support probabilistic inference on both discrete and continuous distributions. The approach does not require modifying the underlying search algorithm or touch any of the backtracking code and comes with a theoretical guarantee that if the underlying search is complete then the probabilistic extension will require the true posterior given enough particles.

References

[1] T. Sato, Y. Kameya, Prism: a language for symbolic-statistical modeling, in: IJCAI, volume 97, Citeseer, 1997, pp. 1330–1339.

[2] L. De Raedt, A. Kimmig, H. Toivonen, Problog: A probabilistic prolog and its application in link discovery., in: IJCAI, volume 7, Hyderabad, 2007, pp. 2462–2467.

[3] B. Gutmann, M. Jaeger, L. De Raedt, Extending problog with continuous distributions, in: International Conference on Inductive Logic Programming, Springer, 2010, pp. 76–91.

[4] M. A. Islam, C. Ramakrishnan, I. Ramakrishnan, Inference in probabilistic logic programs with continuous random variables, Theory and Practice of Logic Programming 12 (2012) 505–523.

[5] V. Belle, A. Passerini, G. Van den Broeck, Probabilistic inference in hybrid domains by weighted model integration, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.

[6] B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, L. De Raedt, The magic of logical inference in probabilistic programming, Theory and Practice of Logic Programming 11 (2011) 663–680.

[7] J. Hemann, D. P. Friedman, W. E. Byrd, M. Might, A small embedding of logic programming with a simple complete search, in: Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Association for Computing Machinery, 2016, p. 96–107. URL: https://doi.org/10.1145/2989225.2989230. doi:10.1145/2989225.2989230.

[8] D. P. Friedman, W. E. Byrd, O. Kiselyov, J. Hemann, The Reasoned Schemer, 2nd ed., MIT Press, 2018.

[9] F. Wood, J. W. Meent, V. Mansinghka, A new approach to probabilistic programming inference, in: Artificial Intelligence and Statistics, PMLR, 2014, pp. 1024–1032.

[10] J. van de Meent, B. Paige, H. Yang, F. Wood, An introduction to probabilistic programming, CoRR abs/1809.10756 (2018). URL: http://arxiv.org/abs/1809.10756.

[11] N. Chopin, O. Papaspiliopoulos, et al., An Introduction to Sequential Monte Carlo, volume 4, Springer, 2020.

[12] R. Douc, O. Cappé, Comparison of resampling schemes for particle filtering, in: ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005., IEEE, 2005, pp. 64–69.
[13] B. Paige, F. D. Wood, A. Doucet, Y. W. Teh, Asynchronous anytime sequential monte carlo, in: Advances in Neural Information Processing Systems, 2014, pp. 3410–3418. URL: https://proceedings.neurips.cc/paper/2014/hash/7eb7eabbe9bd03c2fc99881d04da9cbd-Abstract.html.