Embedded probabilistic programming in Clojure

Nils Bertschinger
Max-Planck Institute for Mathematics in the Sciences
Leipzig, Germany
bertsch@mis.mpg.de

ABSTRACT
Probabilistic programming is a powerful tool to specify probabilistic models directly in terms of a computer program. Here, I present a library that allows to embed probabilistic computations into Clojure. Automatic tracking of dependencies between probabilistic choice points enables an efficient way to sample from the distribution corresponding to a probabilistic program.

1. INTRODUCTION
Nowadays, many problems of artificial intelligence are formulated as probabilistic inference. Also the machine learning community makes heavy use of probabilistic models [2]. Thanks to the steady increase of computing power and algorithmic advances it is now feasible to apply such models to real-world data from various domains. Many different types of inference algorithms are available, but most of them fall into just a few well-studied and understood classes, such as Gibbs sampling, Variational Bayes etc. Even if such standard algorithms are used, still a lot of hand-crafting is required to turn a probabilistic model into code.

Probabilistic programming tries to remedy this problem, by providing specialized programming languages for expressing probabilistic models. The inference engine is hidden from the user and can handle a large class of models in a generic way. WinBUGS [6] is a popular example of a language for specifying Bayesian networks which are then solved via Gibbs sampling. Instead of designing a custom domain-specific language (DSL) for probabilistic computations, it is also possible to extend existing languages with probabilistic semantics. This has been done mainly for logic and functional programming languages [3, 8]. Especially, in the case of functional programming languages, rather lightweight embeddings have been developed [5, 13]. Furthermore, the structure of the embedding is well understood in terms of the probability monad [11].

Here, I present a shallow embedding of probabilistic programming into Clojure, a modern Lisp dialect running on the JVM. The implementation is based on the bher compiler [13], but does not require a transformational compilation step. Instead, it utilizes the dynamic features of the host language for a seamless and efficient embedding. As the bher compiler, my library uses the general Metropolis-Hastings algorithm (Sec. 2.1) to draw samples from a probabilistic program. In contrast to previous implementations, dependencies between the probabilistic choice points encountered during a run of the program are tracked (Sec. 3). This allows to speed up sampling by exploiting conditional independence relations. Sec. 4 illustrates the resulting efficiency on an example application. Finally, Sec. 5 concludes with some general comments on the development of this library.

2. PROBABILISTIC PROGRAMMING
A functional program consists of a sequence of functions which are applied to some input values in order to compute a desired output. Each function in the program can be understood as a mathematical function, i.e., it produces the same output when given the same input values. A function with this property, that the output can only depend on its inputs, is called “pure”. The semantics of a functional program is thus given by a mapping from inputs to outputs. Probabilistic programs extend the notion of a function to probabilistic functions which map inputs to a distribution over their outputs. A probabilistic program thus specifies a (conditional) probability distribution instead of a deterministic function.

Probabilistic functions are not pure, in the sense that they could return different values – drawn from a fixed distribution – for the same input. Thus, a run of a probabilistic program gives rise to a computation tree where each node represents a basic random choice, such as flipping a coin. The children correspond to the possible outcomes of the choice and the edges of the tree are weighted with the probability of the corresponding choice (see Fig. 1 for a simple example). Each path through the computation tree corresponds to a possible realization of the probabilistic program and will be called a trace in the following. The probability of such a trace is simply the product of all edge weights along the path.

Similar trees also arise from non-deterministic operations which have often been embedded into functional languages [1]. The main difference is that probabilistic choices are weighted and the probabilities of different choices add up in the final result. Therefore, the common practice to implement non-determinism, i.e., following just one possible
trace and back-tracking if no solution could be found, is not suitable to evaluate a probabilistic program. Instead, either all possible traces have to be considered or an approximate scheme to sample from the probability distribution corresponding to the program is necessary.

2.1 Metropolis-Hastings sampling

Metropolis-Hastings is a general method to draw samples from a probability distribution. Since it only needs to evaluate ratios of probabilities, it is particularly useful if the probabilities can only be calculated up to normalization. Metropolis-Hastings is a Markov-Chain Monte-Carlo (MCMC) method which, instead of directly sampling from the desired distribution \( p(x) \), constructs a Markov chain \( p(x' | x) \) with a unique invariant distribution \( p(x) \) (see [2] for details). Assume that the current sample is \( x \) and \( p(x) \) can be evaluated up to normalization. Then, a proposal distribution \( q(x' | x) \) is used to generate a new candidate sample. \( x' \) is then accepted as a new sample with probability

\[
\min \left\{ \frac{p(x') q(x | x')}{p(x) q(x' | x)} \right\}.
\]

Otherwise the old sample \( x \) is retained. It is well known that this defines a Markov chain with invariant distribution \( p(x) \). The efficiency of this algorithm depends crucially on the proposal distribution. Ideally the proposed values should be from a distribution close to the desired one, but at the same time it should be easy to sample from.

In the case of a probabilistic program, each sample corresponds to a trace through the computation tree. A global store of all probabilistic choices is used to keep track of the current trace. Thus, the probabilistic program is purified and deterministically evaluates to the particular trace through the computation tree that is recorded in the global store. From this, a new proposed trace is constructed using the steps illustrated in Fig. 2:

- Select a random choice \( c \) (with value \( v \)) from the old trace.
- Propose a new value \( v' \) for this random choice using a local proposal distribution \( q(c, v') \).

Using the global store of all random choices, a new trace is obtained by re-running the probabilistic program with a new store reflecting the proposed value. Pseudo-code for this algorithm can be found in [13]. Unfortunately, they do not fully specify how the so-called forward and backward probabilities \( q(x' | x) \) and \( q(x | x') \) are obtained. Here, care needs to be taken that choice points, which only occur in the new (old) trace but not in the other trace, are accounted for in the forward (backward) probability.

An important point is how choice points are identified between the different traces. If they would be considered as unrelated, each trace would draw completely new choice points and the method boils down to rejection sampling. The change becomes more local the more choice points can be identified between the two traces. In [13] it is argued that choice points should be identified according to their structural position in the program. Currently, this is not yet implemented and instead the user has to explicitly tag each choice point. With dynamically bound variables it should be possible to pass along information about the structural position of each choice point. Implementing a naming scheme based on this idea is left for future work.

2.2 Memoization and conditioning

Memoization of choice points, as introduced in [3], can be implemented by simply changing the tag of a memoized choice point to reflect its type and arguments which should be memoized. This way, the same random choice is fetched from the global store if the memoized choice point is called again with the same arguments. The form \(<\text{basic choice point}>\langle\text{parameters}\rangle<\text{optional arguments}\rangle\) is provided to memoize a basic choice point on its parameters and possibly further identifying arguments.

Conditioning can be implemented in different ways. The most general form allows to condition on any predicate and can be achieved by invalidating the trace, i.e. set its probability to zero, if the predicate evaluates to false. Unfortunately, this is rather inefficient way to implement conditioning since an invalid trace is always rejected and thus a form of rejection sampling is obtained. In the common case of conditioning a basic random choice on a single, specified value, a better implementation is possible. In this case, the choice point is fixed at the conditioning value and deterministically reweights the trace by the probability of this value.

3. NETWORKS OF CHOICE POINTS

Whenever a new proposal is evaluated, the whole program is run again in order to compute the updated trace. In contrast, a hand-crafted sampling algorithm, e.g. for a graphical model, only recomputes those choice points which are neighbors of the node where a change is proposed. Especially, in the common case of sparsely connected models with many conditional independence relations, this leads to huge performance gains. Quite often, each sampling step can be computed in constant time, independent of the total number of choice points.

Similar observations have lead some researchers away from probabilistic functional programming. Instead, they have developed object-oriented frameworks [10, 7] which represent each choice point as an object. The program is then used to connect such objects and imperatively defines a model consisting of interconnected choice points. This model can then be solved either by exact methods based on belief propagation or approximate methods such as Markov-Chain Monte Carlo. This approach has the further advantage that subclassing allows the user to provide custom choice points and control the proposal distributions that are to be used. Unfortunately, the direct relation between the program and the model is lost and it usually not possible to specify models with a changing topology (e.g. Fig. 1). Thus, the object-oriented implementation is more of a library for Bayesian networks (or Markov random fields) than a language extension for embedded probabilistic programming\(^2\).

Here, I propose an implementation which keeps track of the dependencies between probabilistic choices. Then only the actual dependents of a choice points need to be recomputed if a new value is proposed. This approach is inspired by reactive programming which allows to set variables via spreadsheet-like formulas. If a variable changes, all dependencies

\(^2\)A better embedding can be achieved if language constructs are overloaded for choice point objects. This works rather nicely for standard operators on numbers, list, etc. but looks somewhat clumsy for special forms, e.g. if. Furthermore, a small additional overhead is introduced for every operation whereas, ideally, the deterministic parts of the program should be able to run at full speed.
A:  
```clojure
(let [c1 (flip 0.6)
    c2 (flip (if c1 0.8 0.4))
    c3 (if (= c1 c2) (flip 0.3) true)]
  (and c1 c3))
```

B:  
```clojure
(let [c1 (flip-cp :c1 0.5)
    c2 (flip-cp :c2 (if (gv c1) 0.8 0.4))
    c3 (flip-cp :c3 (if (= (gv c2) (gv c1)) (gv (flip-cp :c3-a 0.3)) true))]
  (det-cp : result
           (and (gv c1) (gv c3))))
```

Figure 1: A simple probabilistic program and its computation tree. A: The probabilistic program in Clojure pseudo-code: flip is a basic probabilistic function which returns true with the specified probability and false otherwise. B: The computation tree corresponding to the program on the left. Note that the number of random choices depends on the outcome of previous choices.

![Computation Tree](image)

Figure 2: Example of proposing a new trace, by changing the value of the first random choice c1. The value of choice c2 can be reused, but has to be reweighted with the new probability. The random choice at c3 is created anew when moving from the old to the new trace.

Overall, the resulting programming style is somewhat intermediate between a functional probabilistic programming language and the object-oriented style mentioned above. In the latter approach, objects are used to represent choice points and dependencies are explicitly constructed between them via methods that take parent choice points as arguments. Thus, the program is not a specification of the probabilistic model itself, but is used to construct it as a static network of choice points. Here, choice points are also represented as a special data structure, but as in a functional probabilistic programming the program itself is the probabilistic model. The explicit choice points are merely introduced in order to speed up recomputations necessary during Metropolis-Hastings sampling. In addition, having an explicit representation of choice points enables extensibility, as in the object oriented approach. The macro `def-prob-cp` allows to add custom choice points which support additional probability distributions or use specialized proposal distributions.

### 4. Example
To illustrate the power of probabilistic programming Fig. 3 shows how a Gaussian mixture model with a Dirichlet prior for the mixture components can be implemented. The program closely follows the structure of the statistical model:

1. Draw the component weights from a Dirichlet prior.
2. Assign each data point to one of the components.
3. Draw a mean for the corresponding component model from a Gaussian prior.
4. Condition the Gaussian component model on the observed data point.

Note, how memoization is used to share parameters of the component models between different data points assigned.
A:

```clojure
(defn mixture-memo [comp-labels data]
  (let [comp-weights (dirichlet-cp :weights (for [_ comp-labels] 10.0))]
    (doseq [[idx point] (indexed data)]
      (let [comp-weights (dirichlet-cp :weights (for [_ comp-labels] 10.0))]
        (det-cp :mixture (zipmap comp-labels (gv comp-weights))
                (gv (memo [:mu comp] (gaussian-cp :mu 0.0 10.0) comp)))))))
```

and returns a sequence of samples from the posterior distribution of the model with three mixture components, labeled \(a, b\) and \(c\), when conditioned on a vector of observed data points.

Panel B of Fig. 3 shows a single sample from the posterior overlaid on a histogram of the data points. Here, a data set consisting of 750 points drawn from a mixture of three Gaussians has been used. Since each Metropolis-Hastings step only needs to recompute the choice points which are affected by the proposal, the program scales to considerably larger data sets. Preliminary tests show that it is slightly faster than the Python library PyMC [9] which also implements MCMC sampling.

5. CONCLUSIONS

Adjusting the language towards different semantics, such as non-determinism or logic programming, is a common way to design Lisp programs (see for example [4]). The presented Clojure library demonstrates that probabilistic programming can as easily be integrated into Lisp. Indeed, the code required for running the example from Sec. 4 is surprisingly compact with just below 1000 LoC. The implementation utilizes dynamically bound variables and syntactic extensions via macros to achieve a seamless embedding of the probabilistic programming model.

In addition, Clojure’s immutable data structures allow for an easy and efficient implementation of Metropolis-Hastings sampling: Reverting to the old trace is always possible without creating a copy of the global store first. Thus, they serve as efficient difference data structures which had to be hand-crafted in [8] for this purpose.

During development the dynamic nature of Clojure was of great help. I implemented different prototypes, which helped to clarify unforeseen corner cases making the tracking of dependencies somewhat tricky. Also in this case, the immutable data structures prevented bugs due to state updates and thus simplified the task of keeping the dependencies consistent.

Overall, probabilistic programming is a powerful tool to specify probabilistic models. Many different types of models, e.g., mixture models, regression models etc., can be implemented with ease and conciseness. Thus, even though the performance falls short of hand-crafted algorithms, it is rather useful for rapid prototyping of probabilistic models and the presented library now allows to play with different types of probabilistic models directly in Lisp.

6. REFERENCES