Loopy Logic

Article · January 2002

CITATION
1

READS
7

2 authors:

Daniel Pless
Sandia National Laboratories
8 PUBLICATIONS **26** CITATIONS
SEE PROFILE

George F. Luger
University of New Mexico
110 PUBLICATIONS **1,776** CITATIONS
SEE PROFILE

Loopy Logic

Daniel J. Pless and **George F. Luger**
Department of Computer Science,
University of New Mexico
Albuquerque, NM 87131
{dpless, luger}@cs.unm.edu

Abstract

In this paper we describe a new logic-based stochastic modeling language. It is an extension of the Bayesian logic programming approach of Kersting and De Raedt [1]. We specialize the Kirsting and De Raedt formalism by suggesting that product distributions are an effective combining rule for Horn clause heads. We use a refinement of Pearl's *loopy belief propagation* [2] for the inference algorithm. We also extend the Kirsting and De Raedt language by adding learnable distributions. We propose a message passing algorithm based on *Expectation Maximization* for estimating the learned parameters in the general case of models built in our system. We have also added some additional utilities to our logic language including second order unification and equality predicates.

1 Introduction

Several researchers [1], [3], [4] have proposed forms of first-order logic for the representation of probabilistic systems. In their paper "Bayesian Logic Programs", Kersting and De Raedt [1] extract a particularly elegant kernel for developing probabilistic logic programs. They replace Horn clauses with conditional probability formulas. For example, instead of saying that `x` is implied by `y` and `z` (`x <- y, z`) they write that `x` is conditioned on `y` and `z` (`x | y, z`). They then annotate these conditional expressions with the appropriate probability distributions. In two valued logic, every symbol is `true` or `false`. To support variables that can range over more than two values, they allow the domain of the logic to vary by predicate symbol. Kirsting and De Raedt allow some predicates to range over other sets such as {`red`, `green`, `blue`}.

Ngo and Haddawy [3] construct a logic-based language for describing probabilistic knowledge bases. Their knowledge database consists of a set of sentences giving a conditional probability distribution and a context under which this distribution holds. Such context rules do not appear in the language developed by Kersting and De Raedt [1]. Both of these papers propose using Bayesian networks for inference. In our approach we construct Markov random fields for doing inference (see Section 3).

Ng and Subramanian [4] have a well developed formalism for probabilistic logic. Their system represents ranges of probabilities and provides rules for propagating such ranges through a probabilistic logic program. A simple range of possible probability values is inherently non-Bayesian in nature. In a Bayesian framework, uncertainty in the value of a

probability is handled through higher order probabilities. Ng and Subramanian's declarative language consists of sentences that contain horn clauses with terms that are annotated with probability ranges. The terms in the clauses are two valued. If the terms in the body are provably true, then the head is true with a probability bounded by the given range. Ng and Subramanian also show how to prove queries through PROLOG style SLD tree construction.

In a related approach, Friedman et al. [5] develop a formalism based on the entity-relationship model that underlies most databases. This results in a logic that in some ways is more restrictive than that of Kersting and De Raedt [1], but which allows second order aggregation functions.

Friedman et al. [5] and Ngo and Haddawy [3] can be viewed as extensions to the kernel extracted by Kersting and De Raedt [1]. Our approach to probabilistic logic and inference further extends the language of Kersting and De Raedt by supporting product distributions and learning. Product distributions have been found to be an effective way of representing stochastic models for domains such as handwriting recognition [6].

We have fully implemented our Loopy Logic probabilistic inference system. We have tested it in some standard domains such as Bayesian networks and Hidden Markov Models. Although the trials so far are on simple cases, we have evaluated the full functionality of the language including its ability to do parameter estimation or learning.

In the next section of this paper we describe our new language. In Section 3 we present inference through the construction of Markov fields and the use of loopy belief propagation. In Section 4 we show how the same structure can be used for Expectation Maximization (EM) style parameter updates. In Section 5 we demonstrate our language through an example of a Hidden Markov Model. Finally we present conclusions and future work in Section 6.

2 Language description

We follow Kersting and De Raedt [1] in the basic structure of our language. A sentence in the language is of the form `head | body`$_1$`, body`$_2$`, …, body`$_n$ `= [`p_1`, `p_2`, …, `p_m`]`. The size of the conditional probability table (m) at the end of the sentence has a size equal to the arity (number of states) of the head times the product of the arities of the body. The probabilites are naturally indexed over the states of the head and the clauses in the body, but is shown with a single index for simplicity. For example, suppose `x` is a predicate that is valued over {`red`, `green`, `blue`} and `y` is boolean. P(x|y) is defined by the sentence `x | y = [[0.1, 0.2, 0.7], [0.3, 0.3, 0.4]]`, here shown with the structure over the states of `x` and `y`. Terms (such as `x` and `y`) can be full predicates with structure and contain PROLOG style variables. For example, the sentence `a(X) = [0.5,0.5]` indicates that `a` is universally equally probable to take on either of two values.

If we want a query to be able to unify with more than one rule head, some form of combining function is needed. Kersting and De Raedt [1] allow for general combining functions. In our language, we restrict this combining function to one that is simple, useful, and works well with our inference algorithm. Our choice for combining sentences is a product distribution. For example, suppose we have two simple rules (facts) about some Boolean predicate `a` and one says that `a` is `true` with probability `0.4`, the other says it is `true` with probability `0.7`. The resulting probability for `a` is proportional to the product of the two. Thus `a` is `true` proportional to 0.4 * 0.7 and `a` is `false` proportional to 0.6 * 0.3. Normalizing, `a` is `true` with probability of about 0.61. Thus the overall distribution defined by a database in our language is the normalized product of the distributions defined for all the sentences.

One advantage of using this product rule for defining the resulting distribution is that observations and probabilistic rules are now handled uniformly. An observation is represented by a simple fact with a probability of 1.0 for the variable to take on the observed value. Thus a fact is simply a Horn clause with no body and a singular probability distribution, i.e., all the state probabilities are zero except for a single state.

We extend the basic structure of our probabilistic logic language in a number of ways. First, we allow second order terms, i.e., we can use variables for the function symbol in predicates. A useful example of using this occurs with Boolean functions. If we have a group of predicates whose domain is `{true, false}` we can create a general `or` predicate:

```
or(X,Y) | X, Y = [1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0]
```

Here `X` and `Y` in the body of the clause are higher order predicates. Now if we have two arbitrary predicates representing Boolean random variables, say `a(n)` and `b(m,q)`, then we can form the predicate `or(a(n),b(m,q))` to get a random variable that is distributed according to the logical "or" of the two previous variables.

Our probabilistic logic language also supports simple Boolean equality predicates. These are denoted by angle brackets `<>`. For example, if the predicate `a(n)` is defined over the domain `{red, green, blue}` then `<a(n) = green>` is a variable over `{true, false}` with the obvious distribution. That is, the predicate is `true` with the same probability that `a(n)` is `green` and is `false` otherwise.

The final addition to our logic language is parameter fitting, i.e., learning. An example of a statement that indicates a learnable distribution is `a(X) = A` The capital "`A`" indicates that the distribution for `a(X)` is to be fitted. The data for this is obtained from the facts and rules in the database itself. To specify an observation, one adds a fact to the database in which the variable `X` is bound. For example, suppose that we have the rule above and we add a set of five observations (the `di`) to give the following database:

```
a(X)  = A
a(d1) = true
a(d2) = false
a(d3) = false
a(d4) = true
a(d5) = true
```

In this case we have a single learnable distribution and five completely observed data points. The resulting distribution for `a` will be `true` 60% of the time and `false` 40% of the time. In this case the variables at each data point are completely determined. In general, this is not necessarily so since there may be learnable distributions for which there are no direct observations. But a distribution can be inferred in the other cases and used to estimate the value of the adjustable parameter. In essence, this provides the basis for an Expectation Maximization (EM) [6] style algorithm for simultaneously inferring distributions and estimating the learnable parameters (see Section 4).

Learning can also be applied to conditional probability tables, not just to variables with simple prior distributions. Also learnable distributions can be parameterized with variables just as any other logic term. For example, one might have a rule `rain(X,City) | season(X,City) = R(City)` indicating that the probability distribution for `rain` depends on the `season` and varies by `city`.

All the elements described above have been implemented and tested. We are in the process of developing other predicate types, including pure logic and probabilistic logic

predicates. These are described in a preliminary form in the concluding section. We next describe our inference mechanism for the probabilistic language.

3 Inference

One of the simplest possible inference algorithms for Bayesian networks is the message passing algorithm known as *loopy belief propagation* first proposed by Pearl [2]. In presenting our inference algorithm, we take an approach similar to Murphy et al. [7] who represent stochastic models as Markov fields rather than Bayesian networks.

In Kersting and De Raedt's work, inference proceeds by constructing an SLD tree (a selective literal resolution system for definite clauses) and then converting it into a Bayesian network. We follow a similar path, but we convert the SLD tree to a Markov field instead. The advantage of our approach is that the product distributions that arise from goals that unify with multiple heads can be handled in a completely natural way. The basic idea is that random variable nodes are generated as goals are found. Cluster nodes are created as goals are unified with rules. If one were constructing a Bayesian network, then the node created corresponding to the clause in the head would be the child of the nodes corresponding to the clauses in the body. To construct a Markov field, we add a cluster node between the child and parents. This is illustrated in Figure 1. If more than one rule unifies with the rule head, then the variable node is connected to more than one cluster node. This idea for creating a product distribution is shown in Figure 2.

As a result of the addition of the cluster nodes, the graphs that are generated for inference are bipartite as shown in Figure 1. There are two kinds of nodes in these graphs, the variable and the cluster nodes. The variable nodes hold distributions for the random variables they define. The cluster nodes contain joint distributions over the variables to which they are linked. Messages between nodes are initially set randomly. On update,

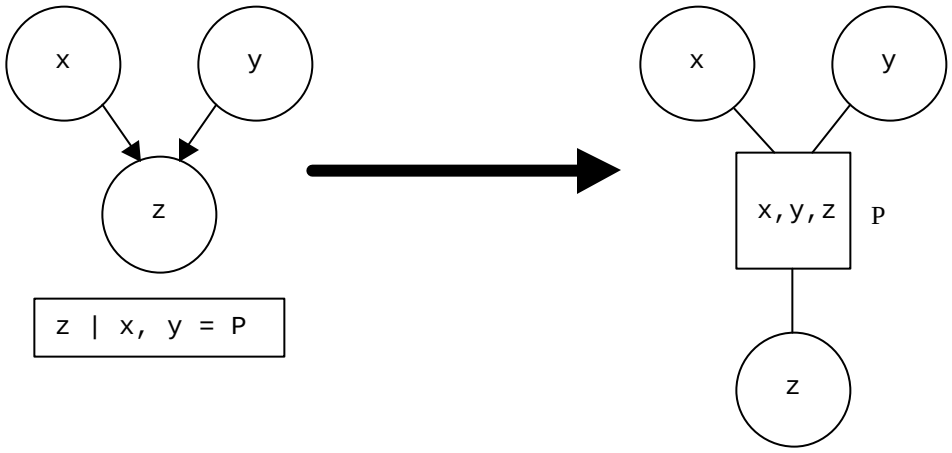

Figure 1: The transition of a piece of a Bayesian network into an equivalent piece of a Markov random field. Note that this generates a bipartite graph due to the addition of the cluster node, the square node which is annotated with the conditional probability distribution P.

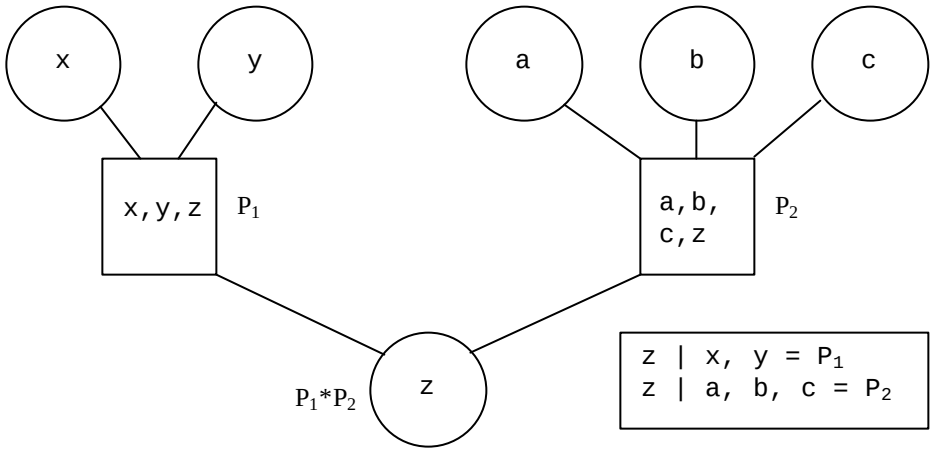

Figure 2: A product distribution is formed from two rules. This is represented in the Markov network as two cluster nodes attached to a single variable node.

the message from variable node X to cluster node Y is the normalized product of all the messages incoming to X other than the message from Y. In the other direction, the message from a cluster node Y to a variable node X is the product of all the messages to Y except the message from X and the conditional probability table (local potential) at Y. This product is marginalized over the variable in X before being sent to X. This process, starting from random messages, and iterating until convergence, has been found to be effective for stochastic inference [7].

The algorithm works by starting from a query (or possibly a set of queries) and generating the variable nodes that are needed. Each query is matched against all unifying heads in the database. The resulting bodies are then converted to new goals in the search. Our current system is limited in that goals produced by this search must be ground terms. Kersting and De Raedt [1] place a range restriction on variables in terms: a variable may appear in the head of a rule only if it also appears in the body. As a result of this requirement, all facts entailed from the database are ground. We require that all entailed goals be ground. We find that this requirement makes for better construction of useful models.

4 Learning

To support learning, we expand the process of building the Markov fields. When a cluster node is created that has a learnable distribution, a new learnable node is created (unless the appropriate node already exists). The parameter estimation example of Section 2, a small database based on the rule `a(X) = A`, is illustrated in Figure 3.

We do parameter estimation with a message passing algorithm. Each learnable node is initially assigned a random normalized distribution. The conditional probability table is the learnable node's message to each of its linked cluster nodes. When the node is updated, each cluster node sends a message which is the product of all messages coming into that cluster. These (unnormalized) tables are an estimate of the joint probability at each cluster node. This is a distribution over all states of the conditioned and conditioning variables. The learnable node takes the sum of all these cluster messages. The result is then converted to a normalized conditional probability table.

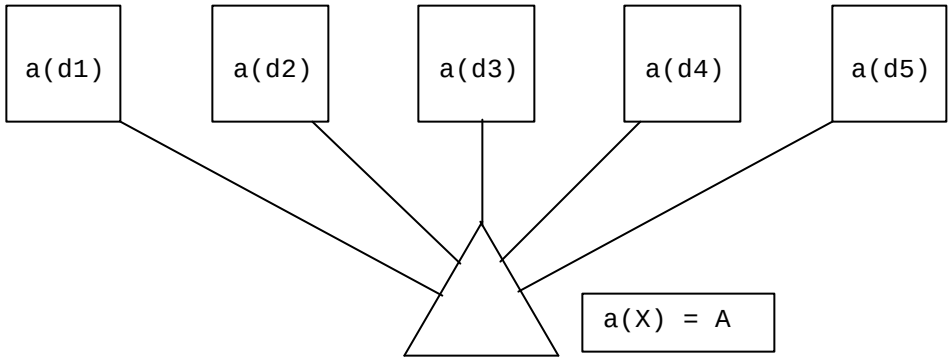

Figure 3: The learnable node and the associated cluster nodes that result from a learnable distribution with five datapoints.

By doing inference (*loopy belief propagation*) on the cluster and variable nodes, we compute the message for the learnable nodes. Applying the propagation algorithm until convergence yields an approximation of the expected values. This is equivalent to the Expectation step in the EM algorithm. The averaging that takes place over all the clusters gives a maximum likelihood estimate of the parameters in a learnable node. Thus, allowing convergence in the variable and cluster nodes followed by updating the learnable nodes and iterating this process is equivalent to the full EM algorithm.

In the algorithm just described, we update all variables synchronously. This is not necessary and may not even be optimal. The nodes can be changed in any order, and updates of cluster and variable nodes may be overlapped with the updates of learning nodes. This iterative update process gives a family of EM style algorithms, some of which may be more efficient than standard EM for certain domains. An algorithmic extension that this framework easily supports is the *generalized belief propagation* of Yedidia et al. [8].

5 Hidden Markov Model example

We present here an example showing how to construct a Hidden Markov Model (HMM) in our Bayesian logic. In our example, we have two states (x, y). The system can start in either one, and at each time step, cycle to itself or transition to the other state. The probability of these events is a learnable distribution. In both states, the system can output one of two symbols (a, b). The conditional distribution for these emissions is also represented in our model by an adjustable distribution.

```
state <- {x,y}
emit <- {a,b}

state(s(N)) | state(N) = State
emit(N) | state(N) = Emit
```

The Hidden Markov Model works as follows. We represent each state with an integer, that is zero or the successor of another integer. We have implemented integer shorthand in our system, i.e., `2` is shorthand for `s(s(0))`. In the model, each state is conditioned on the previous state with the learnable distribution `State`. Each state emits its output with the learnable distribution `Emit.`

Strictly speaking, these four lines of code are sufficient to specify an HMM. We include the next five lines to demonstrate the utility of several of our other extensions. Note, for example, the definition of the `and` predicate:

```
observed,o,and <- {true,false}

and(X,Y) | X,Y = [true,false,false,false]
o([],N) = true
o([H|T],N) = and(<emit(N)=H>,o(T,s(N)))
observed(L) = o(L,0)
```

Without these last five lines, one must specify an observed sequence by incluing in the database a separate fact for each emision that is seen. That is, one must state `emit(0) = a`, `emit(1) = b`, `emit(2) = b` and so on. With the additional five lines, three observations can be included with the predicate `observed([a,b,b])`.

We can easily express a product of HMMs by adding a new predicate to indicate the states of a second HMM. This new HMM can be coupled to the existing one through a product distribution by using the same `emit` predicate. Here is an example of a second HMM with three states:

```
state2 <- {z,q,w}
state2(s(N)) | state2(N) = State2
emit(N) | state2(N) = Emit2
```

Note that the final line uses the previous `emit` predicate which creates the product distribution. As a final comment, our language is far more general than is required for simple HMMs.

6 Conclusion

We have presented a new logic-based stochastic modeling language. We have also presented a translation of this language to a well-known effective inference algorithm, *loopy belief propagation*. This combination produces a first-order probabilistic language with the ability to represent product distributions effectively. We have also shown that learning is supported naturally within this framework.

In our view, each type of logic (deductive, abductive, and inductive) can be mapped to elements of our loopy logic language. The ability to represent rules and chains of rules is equivalent to deductive reasoning. Probabilistic inference, particularly from symptoms to causes, represents abductive reasoning. Finally, learning through the fitting of parameters to known datasets is a form of induction.

Some interesting extensions to our language are possible. For example, we plan to add continuous variables to our language. Using continuous variables it may be possible to support decision theory in the same framework. We are also developing meta predicates for the construction of rules, similar in style to those of Ngo and Haddawy [3]. Furthermore, we would like to relax the constraint on rules that requires all goals be ground, thus producing a more expressive language. Finally, it may be interesting to allow the construction of the Markov field to be interleaved with the inference iterations, so that goals with an infinite SLD tree can be approximated.

Acknowledgements

This work was supported in part by NSF Research Grant IIS-9800929.

References

[1] K. Kirsting and L. De Raedt, (2000) Bayesian Logic Programs. *AAAI-2000 Workshop on Learning Statistical Models from Relational Data*, Menlo Park CA: AAAI Press.

[2] J. Pearl (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, San Francisco CA: Morgan Kaufmann.

[3] L. Ngo and P. Haddawy (1997) Answering Queries from Context-Sensitive Knowledge Bases. *Theoretical Computer Science*, 171:147-177.

[4] R. Ng and V. Subrahmanian (1992) Probabilistic Logic Programming. *Information and Computation*, 101(2):150-201.

[5] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer (1999) Learning Probabilistic Relational Models, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, Stockholm, Sweden, 1300—1307, Menlo Park CA: AAAI Press.

[6] G. Mayraz and G. Hinton (2000) Recognizing Hand-written Digits Using Hierarchical Products of Experts. *Advances in Neural Information Processing Systems 13*, 953-959.

[7] K. Murphy, Y. Weiss, and M. Jordan (1999) Loopy Belief Propagation for Approximate Inference: An Empirical Study. *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 467-475, San Francisco CA: Morgan Kaufmann.

[8] J. Yedidia, W. Freeman, and Y. Weiss (2000) Generalized Belief Propagation. *Advances in Neural Information Processing Systems 13*, 689-695, Cambridge MA: MIT Press.