A First-Order Stochastic Modeling Language for Diagnosis

Chayan Chakrabarti, Roshan Rammohan and George F. Luger

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
{cc, roshan, luger}@cs.unm.edu

Abstract

We have created a logic-based, first-order, and Turing-complete set of software tools for stochastic modeling. Because the inference scheme for this language is based on a variant of Pearl's loopy belief propagation algorithm, we call it Loopy Logic. Traditional Bayesian belief networks have limited expressive power, basically constrained to that of atomic elements as in the propositional calculus. Our language contains variables that can capture general classes of situations, events, and relationships. A Turing-complete language is able to reason about potentially infinite classes and situations, with a Dynamic Bayesian Network. Since the inference algorithm for Loopy Logic is based on a variant of loopy belief propagation, the language includes an Expectation Maximization-type learning of parameters in the modeling domain. In this paper we briefly present the theoretical foundations for our loopy-logic language and then demonstrate several examples of stochastic modeling and diagnosis.

1. Introduction[C]

We first describe our logic-based stochastic modeling language, Loopy Logic. We have extended the Bayesian logic programming approach of Kersting and De Raedt (2000). We have specialized the Kersting and De Raedt formalism by suggesting that product distributions are an effective combining rule for Horn clause heads. We also extend the Kersting and De Raedt language by adding learnable distributions. To implement learning, we use a refinement of Pearl's (1998) loopy belief propagation algorithm for inference. We have built a message passing and cycling - thus the term loopy - algorithm based on expectation maximization or EM (Dempster et al., 1977) for estimating the values of parameters of models built in our system. We have also added additional utilities to our logic language including second order unification and equality predicates.

A number of researchers have proposed logic-based representations for stochastic modeling. These first-order extensions to Bayesian Networks include probabilistic logic programs (Ngo and Haddawy, 1997) and relational probabilistic models (Koller and Pfeffer, 1998; Getoor et al., 1999). The paper by Kersting and De Raedt (2000) contains a survey of these logic-based approaches. Another approach to the representation problem for stochastic inference is the extension of the usual propositional nodes for Bayesian inference to the more general language of first-order logic. Several researchers (Kersting and De Raedt, 2000; Ngo and Haddawy, 1997; Ng and Subrahmanian, 1992) have proposed forms of first-order logic for the representation of probabilistic systems.

Kersting and De Raedt (2000) associate first-order rules with uncertainty parameters as the basis for creating Bayesian networks as well as more complex models. In their paper "Bayesian Logic Programs", Kersting and De Raedt extract a 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`, that is, `x <- y,z` they write that `x` is conditioned on `y` and `z`, or, `x|y,z`. They then annotate these conditional expressions with the appropriate probability distributions.

Section 2 describes our logic-based stochastic modeling language. In Section 3 we present several applications of Loopy Logic to diagnostic reasoning. It has been tested in some standard domains, including traditional as well as dynamic Bayesian networks and hidden Markov models. It has also been tested on failure data from aircraft engines provided by the US Navy (Chakrabarti 2005). The Java version of the tool called DBAYES is available for generic modeling applications from the authors.

2. The Loopy Logic Language

Our research approach follows Kersting and De Raedt (2000) in the basic structure of the language. A sentence in the language is of the form

$$\mathtt{head|body}_1\mathtt{,body}_2\mathtt{,...,body}_n\mathtt{=[p}_1\mathtt{,p}_2\mathtt{,...p}_m\mathtt{]}$$

The size of the conditional probability table (m) at the end of the sentence is equal to the arity (number of states) of the head times the product of the arities of the terms in the body. The probabilities are naturally indexed over the

states of the head and the clauses in the body, but are 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 likely to have either one 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 required. Kersting and De Raedt (2000) allow for general combining functions, while the Loopy Logic language restricts this combining function to one that is simple, useful, and works well with the selected inference algorithm. Our choice for combining sentences is the product distribution. For example, suppose there are 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 the language is the normalized product of the distributions defined for all of its 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 the observed value. Thus a fact is simply a Horn clause with no body and a singular probability distribution, that is, all the state probabilities are zero except for a single state.

Loopy Logic also supports 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 Loopy Logic is parameter fitting or learning. The representational form for a statement indicating a learnable distribution is a(X)=A. The "A" indicates that the distribution for a(X) is to be fitted. The data over which the learning takes place is obtained from the facts and rules presented in the database itself. To specify an observation, the user adds a fact (or rule relation) to the database in which the variable X is bound. For example, suppose, for the rule defined above, the set of five observations (the bindings for X) are added to produce the following database:

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

In this case there is 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) (Mayraz and Hinton 2000) style algorithm for simultaneously inferring distributions and estimating their learnable parameters. 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. A more complete specification of the Loopy Logic representation and inference system may be found in Pless and Luger (2001, 2003).

3. Diagnostic Reasoning with Loopy Logic

We now present two examples of fault diagnosis using Loopy Logic. These problems are intended to demonstrate the first-order representation and the expressive power of the language. We are currently addressing two complex problems, the propulsion system of a Navy aircraft, sponsored by Office of Naval Research, and a more complex and cyclic sequence of digital circuits, pieces of which have been presented here (Chakrabarti 2005).

3.1 Example: Diagnosing Digital Circuits

We next demonstrate how a first-order probabilistic language like Loopy Logic can be used for diagnosis of faults in a combinatorial (acyclic) digital circuit. We assume there is a database of circuits that are constructed from and, or and not gates and that we wish to model failures within such circuits. We assume that each component has a mode that describes whether or not it is working. The mode can have one of three values, it is good or has one of two failures, stuck_at_1 or stuck_at_0. We assume that the probability of the

various failure modes is the same for components of the same type, although this probability may vary across types of components.

There are two questions that a probabilistic model can answer. First, assume the probabilities of failure are known. Given a circuit that is not working properly, and one or more test cases (values for inputs and outputs of the circuit), it would be useful to know the probability for each component mode in order to diagnose where the problem might be. The second question comes from relaxing the assumption that the failure probabilities are known. If there is a database of circuits and tests performed on those circuits, we may wish to derive from these tests what the failure probabilities might be.

We next provide code for this model. We use some conventions for naming variables. We let `Cid` be a unique ID for each circuit, `Tid` be an ID for each different test, `N` be an ID for a component of the circuit, `Type` be the component type `(and,or,not)`, and `I` be inputs (a list of `N`s) for the component. The first two lines of the code are declarations to define which modes a component can be in as well as indicating that everything else is boolean:

```
mode <- {good,stuck_at_0,stuck_at_1}.
val,and,or,not <- {v0,v1}.
```

The `mode` and `val` statements provide the basic model for circuit diagnosis. The first indicates that the probability distribution for the mode of any component is a learnable distribution. One could enter a fixed distribution if the failure probabilities were known. Using the term `Mode(Type)` specifies that the probabilities may be different for different component types, but will be the same across different circuits. One could indicate that the distributions were the same for all components by using just `Mode` or that they differed across type and circuit by using `Mode(Type,Cid)`. The second statement of the two specifies how the possibility of failure interacts with normal operation of a component. The `val` predicate gives the output of component `N` in circuit `Cid` for test `Tid`. See Chakrabarti (2005) for details.

```
mode(Cid,N):-
comp(Cid,N,Type,_)= Mode(Type).

val(Cid,Tid,N):- comp(Cid,N,Type,I)|
mode(Cid,N),Type(Cid,Tid,I)=
[[v0,v1],[v0,v0],[v1,v1],
[[0.5,0.5],[0.5,0.5]]].
```

`<-` and `:-` are part of the Loopy Logic syntax. The `and`, `or` and `not` predicates model the random variables for what the output of a component would be if it is working correctly. The `and` and `or` are specified recursively. This allows arbitrary fan-in for both types of gates. The base case is handled by assigning a deterministic value for the empty list (`1` for `and`, `0` for `or`). The recursive case computes the appropriate function for the value of the head of the list of inputs and then recurs. The `not` acts on a single value, inverting the value of the input.

```
and(_,_,[])=v1.
and(Cid,Tid,[H|T])|val(Cid,Tid,H),
      and(Cid,Tid,T)=[[v0,v0],[v0,v1]].

or(_,_,[])=v0.
or(Cid,Tid,[H|T])=val(Cid,Tid,H),
      or(Cid,Tid,T)=[[v0,v1],[v1,v1]].

not(Cid,Tid,N)|val(Cid,Tid,N)=[v1,v0].
```

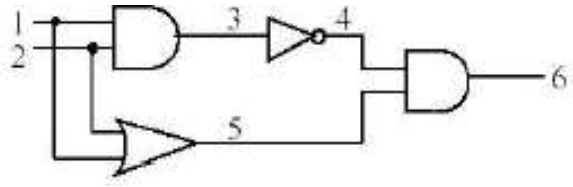

Figure 1. A sample circuit that implements XOR

The circuit of Figure 1 is described by the following four lines of code.

```
comp(1,3,and,[1,2]).
comp(1,4,not,3).
comp(1,5,or,[1,2]).
comp(1,6,and,[4,5]).
```

We now introduce failure probabilities into the different components. For the sake of simplicity we shall assume that the failure probabilities are fixed and are the same for all types of components.

```
mode=[0.989,0.01,0.001]
```

This indicates that the component is `good`, `stuck_at_0`, or `stuck_at_1` with a fixed probability distribution of 98.9%, 1% and 0.1%.

Next, we give the system a set of input values using the following statements.

```
val(1,1,1)=v0.
val(1,1,2)=v1.
```

Now, we query loopy logic about the output at gate 6 using the following statement from the loopy prompt.

```
val(1,1,6)?
```

We get the following response from the Loopy Logic interpreter.

```
val(6)
v0:  0.03064943979
v1:  0.96935056021
```

This output indicates that the output at component 6 is high with a 97% probability and low with 3% probability. This is consistent with our expectation.

Next we repeat the same test by introducing a very high failure rate in our model. We state that the component has only 50% probability of being good.

```
mode=[0.5,0.3,0.2].
```

We now query the Loopy Logic interpreter for the output at component 6 as follows.

```
val(1,1,1)  = v0.
val(1,1,2)  = v1.
val(1,1,6)  = ?
```

The Loopy Logic interpreter gives the following response.

```
val(6)
v0:  0.59
v1:  0.41
```

The Loopy Logic interpreter tells us that in this model the output is more likely to be wrong. This is because we have introduced a higher (50%) probability of failure.

Now, consider the second problem. We know that a fault has occurred and we want to find the likely causes for the fault, i.e., which of the gates in the circuit might be faulty. We again consider the initial model which had a 1% probability of failure. We force the output at component 6 to be wrong.

```
mode=[0.989,0.01,0.001].
val(1,1,1)=v0.
val(1,1,2)=v1.
val(1,1,6)=v0.
mode(3),mode(4),mode(5),mode(6)?
```

As shown, we have set the output component to be `v0` when in fact the correct output should be `v1`. We now want to find the probability of failure of each component in the circuit. This is done by the query on the fifth line, above. We get the following response.

```
mode(3)
good:  0.957753489171
s0:  0.00968405954672
s1:  0.0325624512826
mode(4)
good:  0.673375168043
stuck_at_0:  0.325943967278
stuck_at_1:  0.000680864679517
mode(5)
good:  0.67340708122
stuck_at_0:  0.325943967278
stuck_at_1:  0.000648951502418
mode(6)
good:  0.673729762485
stuck_at_0:  0.326270237515
stuck_at_1:  0
```

This response shows the failure probabilities of each component. It tells us that component 3 is `good` with a 95.77% probability. Component 4, 5 and 6 are `good` with 67.33% probability. Further, it also tells us that component 4 is `stuck_at_0` with 32.59% probability. Mathematical analysis shows that this inference is correct.

Next, we repeat the diagnostic test where the third input value, `val(1,1,6)=v0,` is incorrect:

```
val(1,1,1)=v0.
val(1,1,2)=v1.
val(1,1,6)=v0.
mode(3),mode(4),mode(5),mode(6)?
```

We get the following response:

```
mode(3)
good:  0.470338983051
s0:  0.282203389831
s1:  0.247457627119
mode(4)
good:  0.423728813559
s0:  0.406779661017
s1:  0.169491525424
mode(5)
good:  0.440677966102
s0:  0.406779661017
s1:  0.152542372881
mode(6)
good:  0.491525423729
s0:  0.508474576271
s1:  0
```

Once again, we observe by analysis that the results obtained from Loopy Logic are valid. In our research, similar diagnostic tests on a dozen different circuits of varying sizes and complexity were performed. The smallest circuit had 6 components and the largest circuit had 10,700 components. Some circuits had loops in them as well. The results provided by Loopy Logic were found to be accurate in all cases (Chakrabarti 2005). The largest circuit converged in less than 15 minutes on a standard linux cluster. Without a powerful stochastic modeling tool, it is a non-trivial task to design a system that can diagnose digital circuit failures as well as estimate failure probabilities from a data set of test cases. With our system, the basic model can be constructed using only nine

statements. As the example shows, the representation of circuits and test data is transparent as well. Please refer to Chakrabarti (2005) for further details.

3.2 Example: Fault Detection in a Mechanical System

In the final example, we predict a future event, namely a breakdown of a mechanical system due to a fault (Chakrabarti 2005). Data from various analog sensors are available to us as observations from the time of start of a test. The time domain representation of the data is unwieldy and intractable for computation. So we deal with the data in the frequency domain by computing the Fourier transform of the time-series data. Further, we smooth the data by averaging the frequency domains of each set of M consecutive preliminary observations. It is this domain of converted and smoothed data that makes up our observation Y_t.

Dynamic Bayesian Networks (DBN's) (Dagum et al., 1992) can be used as a tool to model dynamic systems. More expressive than Hidden Markov Models (HMM) and Kalman Filter Models (KFM), they can be used to represent other stochastic graphical models in AI and machine learning.

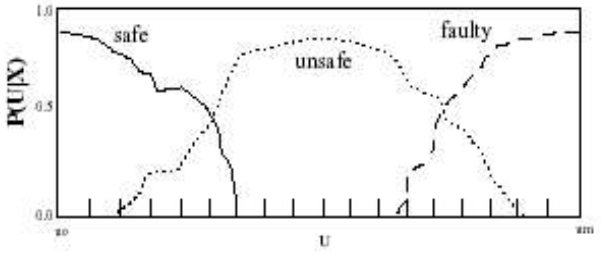

Figure 2. A model of distributions P(U | X = i) as learned from the training data.

It is reasonable to assume that the observation at current time slice Y_t is related to the observation at the previous time slice, Y_{t-1}, i.e., the observations are temporally correlated. In fact we use correlation as a metric of distance between observations. A lack of correlation between observations in consecutive time slices is probably an indication of anomalous behavior. For example, when the system changes state from $X_{t-1}=safe$ to $X_t=unsafe$ we expect the corresponding observations Y_{t-1} and Y_t to show lower levels of correlation. This understanding of the data leads us to consider the use of the AR-HMM (auto-regressive HMM) (Juang, 1984) to model the system. We chose to model the AR-HMM on three hidden states, `{safe,unsafe,faulty}`. We infer the probability distribution of the system state at time *t*, $P(X_t)$.

In the AR-HMM, (see Figure 3) the customary HMM assumption that Y_t does not have a direct causal relationship with Y_{t-1} is relaxed. Before we apply the algorithm to real time data we evaluate the probability distribution $P(u_j|X)$ of expected frequency signatures corresponding to the states from a state-labeled dataset. Note that $U=u_1,u_2,\ldots,u_k$ is the set of observations that have been recorded while training the system. Say for example, if observation u_1 through observation u_k were recorded when the system gradually went from `safe` to `faulty` we would expect $P(u_1|X=safe)$ to be significantly higher than $P(u_k|X=safe)$.

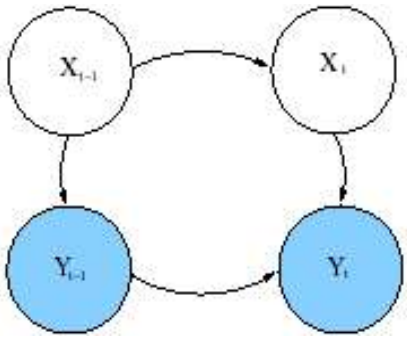

Figure 3. An Auto-Regressive Hidden Markov Model

From the causality expressed in the AR-HMM we know,

(1)

$$P(Y_t=y_t|X_t=i,Y_{t-1}=y_{t-1}) = P(Y_t=y_t|X_t=i)*P(Y_t=y_t|Y_{t-1}=y_{t-1})$$

In our design, the probability of an observation given a state is the probability of observing the discrete prior that is closest to the current observation, penalized by the distance between the current observation and the prior.

$$P(Y_t=y_t|X_t=i) = max(|(corrcoef(y_t,u_j))|)*P(u_t|X_t=i) \quad (2)$$

Further, the probability of an observation at time *t* given another particular observation at time *t-1* is the probability of the most similar transition among the priors penalized by the distance between the current observation and the observation of the previous time step. Note that y_t is a continuous variable and potentially infinite in range but we limit it to a tractable set of finite signatures, U by replacing it by the u_j with which it best correlates.

$$P(Y_t=y_t|Y_{t-1}=y_{t-1}) = |(corrcoef(y_t,y_{t-1}))| * \frac{(number\ of\ u_{t-1}-u_t\ transitions)}{(number\ of\ u_{t-1}\ observations)}$$

where $u_t=argmax_{u_j}(|(corrcoef(y_t,u_j))|)$

The relationship governing the learnable distributions is expressed in Loopy Logic as follows:

```
x <- {safe,unsafe,faulty}.
y(s(N))|x(s(N))=LD1.
y(s(N))|y(N))=LD2.
```

Preprocessing the data and computing the correlation coefficients off-line, we tested the above technique on a large training dataset of several seeded fault occurrences taking the system from `safe` to `faulty` . We obtained a performance accuracy close to 80% on this test data. Please refer to Chakrabarti (2005) for details.

4. Conclusions and Further Research

We have created a new first-order Turing-complete logic-based stochastic modeling language. A well-known and effective inference algorithm, loopy belief propagation, supports this language. Our combination rule for complex goal support is the product distribution. Finally, a form of EM parameter learning is supported naturally within this framework. From a larger perspective, each type of logic (deductive, abductive, and inductive) can be mapped to elements of our declarative stochastic 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, and learning through fitting parameters to known data sets, is a form of induction.

A future direction for research is to extend Loopy Logic to include continuous random variables. We also plan to extend learning from parameter fitting to full model induction. Getoor et al. (2001) and Segal et al. (2001) consider model induction in the context of more traditional Bayesian Belief Networks and Angelopoulos and Cussens (2001) and Cussens (2001) in the area of Constraint Logic Programming. Finally, the Inductive Logic Programming community (Muggleton, 1994) also addressed the learning of structure with declarative stochastic representations. We plan on taking a combination of these approaches.

Acknowledgments

This research was supported by NSF (115-9 800929, INT-9900485), and a NAVAIR STTR (N0421-03-C-0041). The development of Loopy Logic was based on Dan Pless's PhD research at the University of New Mexico. (Pless and Luger 2001, 2003)

References

Angelopoulos, N., and Cussens, J. 2001. Markov Chain Monte Carlo Using Tree-Based Priors on Model Structure. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, San Francisco.: Morgan Kaufmann.

Chakrabarti, C. 2005. First-Order Stochastic Systems for Diagnosis and Prognosis, Masters Thesis, Dept. of Computer Science, University of New Mexico.

Cussens, J. 2001. Parameter Estimation in Stochastic Logic Programs, *Machine Learning* 44:245-271.

Dagum, P., Galper, A., and Horowitz, E. 1992. Dynamic Network Models for Forecasting. In *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, 41-48. Morgan Kaufmann.

Getoor, L., Friedman, N., Koller, D., and Pfeffer, A. 2001. Learning Probabilistic Relational Models. *Relational Data Mining*, S. Dzeroski and N. Lavorac (eds).: Springer.

Juang, B. 1984. On the Hidden Markov Model and Dynamic Time Warping for Speech Recognition: a unified view, Technical Report, vol 63, 1213-1243 AT&T Labs

Kersting, K. and De Raedt, L. 2000. Bayesian Logic Programs. In *AAAI-2000 Workshop on Learning Statistical Models from Relational Data.* Menlo Park, CA.: AAAI Press.

Koller, D., and Pfeffer, A. 1998. Probabilistic Frame-Based Systems. In *Proceedings of the Fifteenth National Conference on AI*, 580-587. Cambridge, MA.: MIT Press.

Mayraz, G., and Hinton, G. 2000. Recognizing Hand-Written Digits using Hierarchical Products of Experts. *Advances in Neural Information Processing Systems 13:* 953-959, 2000.

Muggleton, S. 1994. Bayesian Inductive Logic Programming. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, 3-11. New York.: ACM Press.

Ng, R. and Subrahmanian, V. 1992. Probabilistic Logic Programming. *Information and Computation*: 101-102

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

Pearl, P. 1988. Probabilistic Reasoning in Intelligent Systems: *Networks of Plausible Inference.* San Francisco CA.: Morgan Kaufmann.

Pless, D., and Luger, G.F. 2001. Toward General Analysis of Recursive Probability Models. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, San Francisco.: Morgan Kaufmann.

Pless, D., and Luger, G.F. 2003. EM Learning of Product Distributions in a First-Order stochastic Logic Language. *IASTED Conference*, Zurich.: IASTED/ ACTA Press.

Segal, E., Koller, D. and Ormoneit, D. 2001. Probabilistic Abstraction Hierarchies. *Neural Information Processing Systems*, Cambridge, MA.: MIT Press.