International Journal on
ARTIFICIAL INTELLIGENCE TOOLS
Architectures, Languages, Algorithms

Special Issue on FLAIRS 2005:
Knowledge Acquisition and Representation

Guest Editors: **Ingrid Russell, Zdravko Markov and Lawrence B. Holder**

Volume 15 • Number 6 • December 2006

The Design and Testing of a First-Order Logic-Based Stochastic Modeling Language

D. J. Pless, C. Chakrabarti, R. Rammohan and G. F. Luger

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI

International Journal on Artificial Intelligence Tools
Vol. 15, No. 6 (2006) 979–1005
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

THE DESIGN AND TESTING OF A FIRST-ORDER LOGIC-BASED STOCHASTIC MODELING LANGUAGE

DANIEL J. PLESS
Sandia National Laboratories MS 1138
Albuquerque, NM 87185, USA
djpless@sandia.gov

CHAYAN CHAKRABARTI
Computer Science Department, University of New Mexico
Albuquerque, NM 87131, USA
cc@cs.unm.edu

ROSHAN RAMMOHAN
Computer Science Department, University of New Mexico
Albuquerque, NM 87131, USA
roshan@cs.unm.edu

GEORGE F. LUGER
Computer Science Department, University of New Mexico
Albuquerque, NM 87131, USA
luger@cs.unm.edu

We have created a logic-based, Turing-complete language for stochastic modeling. Since 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 networks have limited expressive power, basically constrained to finite domains as in the propositional calculus. Our language contains variables that can capture general classes of situations, events and relationships. A first-order language is also able to reason about potentially infinite classes and situations using constructs such as hidden Markov models(HMMs). Our language uses an Expectation-Maximization (EM) type learning of parameters. This has a natural fit with the Loopy Belief Propagation used for inference since both can be viewed as iterative message passing algorithms. We present the syntax and theoretical foundations for our Loopy Logic language. We then demonstrate three examples of stochastic modeling and diagnosis that explore the representational power of the language. A mechanical fault detection example displays how Loopy Logic can model time-series processes using an HMM variant. A digital circuit example exhibits the probabilistic modeling capabilities, and finally, a parameter fitting example demonstrates the power for learning unknown stochastic values.

Keywords: Stochastic-modeling; loopy belief propagation; parameter-fitting.

1. Introduction

A number of researchers have proposed logic-based representations for stochastic modeling. These first-order extensions to Bayesian networks include probabilistic logic programs[1] and relational probabilistic models.[2,3] The paper by Kersting and De Raedt[4] contains a survey of these logic-based representations. 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[1,4,5] have proposed forms of first-order logic for the representation of probabilistic systems.

Poole[6] gives an early approach which develops an approximate algorithm for another Turing complete probabilistic logic language. In this language, the probabilistic content is expressed as sets of mutually exclusive predicates (facts) annotated with probabilities. The semantics of rules is similar to standard logic. This language is first-order, but hard to use as ensuring that the correct normalization is maintained through the rules is up to the user; this is rather complex even when expressing simple Bayesian networks.

Ngo and Haddawy[1] 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.[4] Both of these papers propose using Bayesian networks for inference.

Kersting and De Raedt[4] 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, $x \leftarrow y,\ z$, they write that x is conditioned on y and z, or, $x \mid y,\ z$. They then annotate these conditional expressions with the appropriate probability distributions. In a 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. Kersting and De Raedt also allow some predicates to range over other sets, for example, `{red, green, blue}`.

Ng and Subrahmanian[5] 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. To be properly Bayesian, one cannot simply exclude certain values (as is done in Ng and Subrahmanian); one must specify the probability of all the allowed values. Now one could specify a flat distribution to reproduce a range of allowed values. However, such a distribution would not remain flat during Bayesian updating. In a Bayesian framework, uncertainty in the value of a probability is handled through higher order probabilities. Ng and Subrahmanian'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 Subrahmanian also show how to prove queries through PROLOG style SLD tree construction.

For the language presented in this paper, we have created a prototype software system was written in OCaml,[7] an object-oriented Caml dialect of ML. That system uses a simple command line interface, and all the "code examples" of this paper can be interpreted by this system; the source code can be found in Ref. 8. The last three authors are currently building a Java version of this stochastic modeling environment that includes a graphical user interface.

We describe our logic-based stochastic modeling language in Section 2. In Section 3, we present our inference scheme based on a form of loopy belief propagation and Markov random fields. Then, in Section 4, we describe several applications of Loopy Logic to diagnostic reasoning. We represent an HMM and extend this capability to model time-series data using a HMM variant, the auto-regressive HMM. This data was collected from sensors located on the failing rotating parts of aircraft engines that were examined by the US Navy. Loopy Logic was used to diagnose faults based on these readings. In Section 5, we detect failures in digital circuits. Finally, in Section 6, we demonstrate parameter-fitting in Loopy Logic using a life-support simulation example. In Section 7, we present conclusions and possible future directions.

2. The Loopy Logic Language

2.1. *Probability models*

Loopy Logic follows Kersting and De Raedt[4] in the basic structure of the language. A sentence in the language is of the form $head|body_1, body_2, ..., body_n = [p_1, p_2, ..., p_m]$. A head is a variable of the system and the bodies are the variables on which the head is conditionally dependent. 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 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 probable to take on either of two values. The underline character (_) is used, as in PROLOG, to denote an anonymous variable. Also, as in PROLOG, the period is used for statement termination. The domain of terms is indicated with set notation. For example, `a` $\in$ `{true, false}` indicates that `a` is either `true` or `false`.

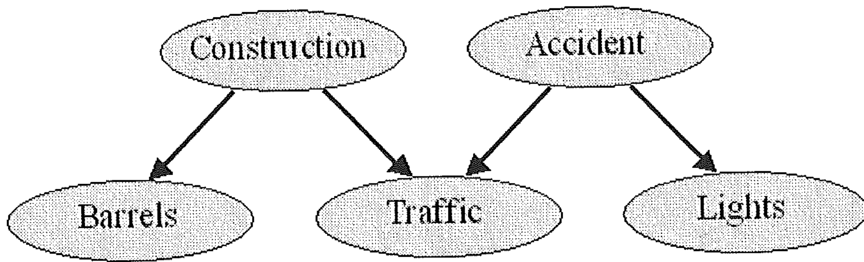

Fig. 1. A Bayesian network.

Figure 1 shows a simple Bayesian network depicting a situation one might encounter while driving. Assume one encounters heavy traffic while driving to work. This could have been caused by either an accident or road construction. If one happens to observe lots of flashing lights ahead, then the cause of heavy traffic is probably an accident. However, if one observes lots of orange barrels ahead, then road construction is more likely to have caused the heavy traffic. This is a classic Bayesian network example and to represent it, we have:

```
c,a,b,t,l <- {true,false}.
c = [0.5, 0.5].
a = [0.01,0.99].
b | c = [[0.9,0.1],[0.001,0.999]].
t | c, a = [...]
l | a = [[0.8,0.2],[0.03,0.97]].
```

For the sake of brevity, we did not include the full distribution for the variable t (this would go in the [...]). If we wanted to use the above network for a number of days simultaneously (the day parameterized by the variable D) we could model the situation with the following code:

```
c,a,b,t,l <- {true,false}.
c = [0.5, 0.5].
a(D) = [0.01,0.99].
b | c = [[0.9,0.1],[0.001,0.999]].
t(D) | c, a(D) = [...]
l(D) | a(D) = [[0.8,0.2],[0.03,0.97]].
```

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[4] allow for general combining functions. Loopy Logic restricts this combining function to one that is simple, useful, and works well with the inference algorithm, the product distribution. Product distribution simply means the product of the different conditional probability tables of the network. 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 the 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. Even for combining in information that is not deterministic, product distributions have been found to be an effective way of representing stochastic models, for example in the domain of handwriting recognition.[9]

Loopy Logic also supports 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)`, 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.

2.2. *Additional syntax*

The syntax described above is sufficient to define stochastic models in Loopy Logic. We augment this syntax to ease construction of these models. To indicate that a variable has a deterministic value, for example, if a is `true`, then one can say `a = true` rather than `a = [1.0, 0.0]`. The language also allows similar shorthand notation within larger structured distributions.

Loopy Logic 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.

2.3. *Learning*

Loopy Logic can also handle parameter fitting, i.e., learning. An example of a statement that indicates a learnable distribution is `a(X) = A`. The upper case "`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, we add 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 d's)x 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)[10] style algorithm for simultaneously inferring distributions and estimating the learnable parameters. We describe this learning algorithm in Section 3.2. Learning can also be applied to conditional probability tables, not just to variables with simple prior distributions. Learnable distributions can also be parameterized with variables just as any other logic term. For example, the rule rain(Day,City) = Rain(City) indicates that the probability distribution for rain varies by city and the value for rain can depend on the day.

Similar to Ngo and Haddawy,[1] we support meta-predicates to allow the automated construction of rules. rain(Day, City) :- climate(City, C) = Rain(C), for example, indicates that the rain in a city is described by the climate for that city. So a non-probabilistic PROLOG term climate(miami, tropical) would indicate that the probability of rain in Miami, e.g. , rain(miami), is a learnable distribution (which would the same for all tropical cities).

3. Inference in Loopy Logic

3.1. *Loopy belief propagation*

One of the simplest possible inference algorithms for Bayesian networks is the message-passing algorithm known as *loopy belief propagation* first proposed by Pearl.[11] This algorithm later had its effectiveness demonstrated by Murphy *et al.*[12] after the connection between loopy belief propagation and Turbo Codes was pointed out by McEliece *et al.*[13] Loopy Logic takes an approach similar to Murphy *et al.*[12] who represent stochastic models as Markov fields rather than Bayesian networks.

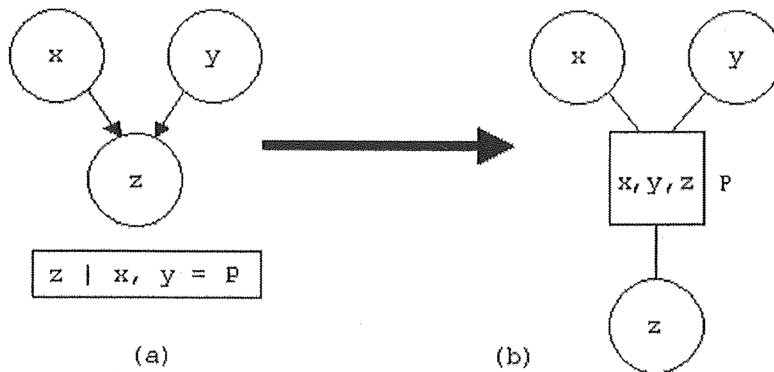

Fig. 2. The transition of a piece of a Bayesian network into an equivalent piece of a Markov random field. Note that this generates a bipartite graph due to the addition of the cluster node, the square node that is annotated with the conditional probability distribution P.

In Kersting and De Raedt's work, inference proceeds by constructing an SLD (Selection rule, Linear resolution, Definite clauses) tree (a selective literal resolution system for definite clauses) and then converting it into a Bayesian network. Loopy Logic follows a similar path, but instead converts the SLD tree to a Markov field. The advantage of this 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. In a logic program representing a Bayesian network, the head of a statement corresponds to a child node, while the clauses in the body correspond to the node's parents as shown in Figure 2. To construct a Markov field, Loopy Logic adds a cluster node between the child and its parents. If more than one rule unifies with the rule head, then the variable node is connected to more than one cluster node, which results in a product distribution, as shown in Figure 3.

As a result of the addition of the cluster nodes, the graphs that are generated for inference are bipartite as shown in Figure 2(b). 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, the message from variable node V to cluster node C is the normalized product of all the messages incoming to V other than the message from C. In the other direction, the message from a cluster node C to a variable node V is the product of the conditional probability table (local potential) at C and all the messages to C except the message from V. This product is marginalized over the variable in V before being sent to V. This process, starting from random messages, and iterating until convergence, has been found to be effective for stochastic inference.[12]

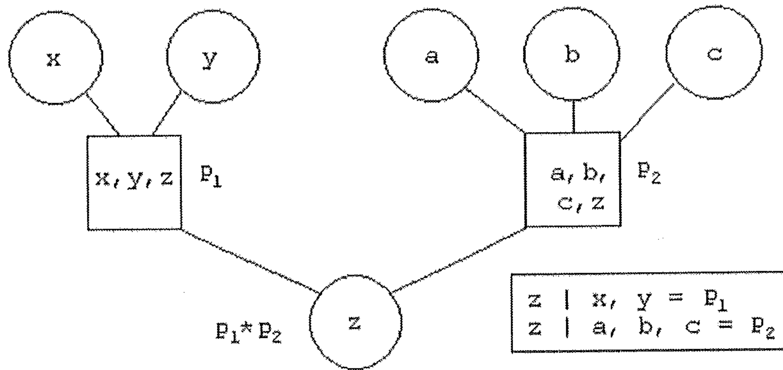

Fig. 3. A product distribution is formed from two rules. This is represented in the Markov network as two cluster nodes attached to a single variable node. We presented a simple numerical example of the product distribution in Section 2.1

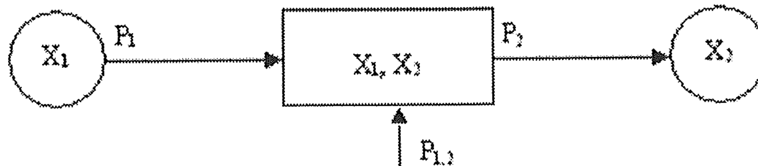

Fig. 4. Example of message passing between variable nodes represented as circles and cluster nodes represented as boxes.

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. All the ground facts must also be included in the network. The resulting bodies are then converted to new goals in the search. Loopy Logic is limited in that goals produced by this search must be ground terms. A fact is a rule with no body just as in PROLOG. An observation is a rule that has a probability distribution in which one of the alternatives has probability one while the remaining alternatives have probability zero. Kersting and De Raedt[4] 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. By contrast, Loopy Logic requires that all entailed goals be ground. This requirement makes for better construction of useful models.

The message passed from a variable node to a cluster node is the normalized product of all the messages incoming to the variable node other than the message from the cluster node. For example, in Figure 4, the message from variable node X_1 to the cluster node is the normalized product of incoming messages to X_1. In the other direction, the message from a cluster node to a variable node is the

product of the conditional probability table (local potential) at the cluster node and all the messages incoming to the cluster node except the message from the variable node. Before passing to the variable node, the message is marginalized based on the variable. In Figure 4, the distribution P_1 over variable X_1 is passed from the variable node X_1 to the cluster node X_1, X_2. The conditional probability table for this cluster node is $P_{1,2}$, which is distributed over X_1 and X_2. These two distributions are multiplied to get a new distribution over X_1 and X_2. The result is marginalized over X_2 to result in P_2 which is then passed to the X_2 variable node.

3.2. *Expectation-maximization for learnable nodes*

The fact that Loopy Logic uses loopy belief propagation offers a natural support for Expectation Maximization learning.[14] Basically, learning is achieved by adding learnable distributions to Kersting and De Raedt's language.[8,15] The learning message passing algorithm is based on the concept of Expectation Maximization (EM) to estimate the learned parameters in the general case of models built in the system.[8] The widespread applicability of the EM algorithm was first discussed by Dempster *et al.*[10] This algorithm estimates learning parameters iteratively, starting with an initial guess. Each iteration of the algorithm consists of an expectation step (E step) and a maximization step (M step). In the expectation step, the distribution for the unobserved variables are based on their known value and the current estimate of the parameters is found. The maximization step re-estimates the parameters. These two steps continue until they reach their maximum likelihood with the assumption that the distribution found in the expectation step is correct. As shown by Ref. 10, each EM iteration increases the likelihood, unless a local maximum has already been reached.

To summarize, EM learning takes the form of parameter fitting. A distribution can be used to estimate the value of the learnable parameter. In the loopy logic algorithm, learning can also be applied to conditional probability tables, not just to variables with simple prior distributions. Learnable distributions can be parameterized with variables just as any other logic term.

To support the EM algorithm, 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 2.3, a small database based on the rule `a(X) = A`, is illustrated in Figure 5.

Loopy Logic performs 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 that 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

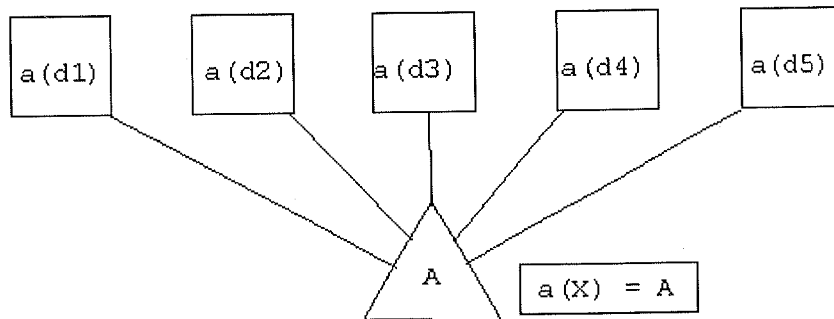

Fig. 5. The learnable node A and the associated cluster nodes that result from a learnable distribution with five datapoints.

node takes the sum of all these cluster messages. The result is then converted to a normalized conditional probability table.

By doing inference (*loopy belief propagation*) on the cluster and variable nodes, Loopy Logic computes 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, all variables are updated 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 learned 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 supports is the *generalized belief propagation* of Yedidia *et al.*[16]

In the next Section we present three examples that emphasize different strengths of our stochastic modeling language, fault detection in a mechanical system using a time series based hidden Markov model, the diagnosis of failures in digital circuits, and learning with parameter fitting in a space-based life support system. All our data, except for that used to diagnose helicopter rotor systems (provided us by the US Navy) is simulated. As mentioned in the introduction, the current system is written in OCaml and all the example code fragments from our examples have been interpreted in that environment.

4. Time-Series Modeling and Analysis

We now demonstrate the representational power of Loopy Logic. We first show how to represent a simple HMM and then extend the idea to a real problem of time-series modeling.

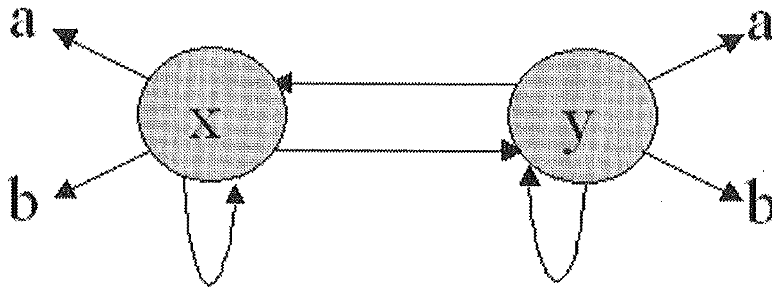

Fig. 6. A HMM with two hidden states (x,y) and two emit symbols (a,b).

4.1. *Example: A hidden Markov model*

First, we present an example of a Hidden Markov Model (HMM)in our stochastic logic language. In this example, there are 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}` as is presented in Figure 6. The conditional distribution for these emissions is also represented in this model by an adjustable distribution.

```
state <- {x,y}.
emit <- {a,b}.
state(s(N)) | state(N) = State.
emit (N) | state(N) = State
```

The Hidden Markov Model works as follows. Each state is represented with an integer that is zero or the successor of another integer. The notation `s()` denotes successor state. An integer *shorthand* is implemented in this 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, because of the representational flexibility of Loopy Logic, the previous four lines of code are sufficient to specify an HMM. The next five lines are included to demonstrate the utility of several of our other extensions. Note, for example, the definition of the and predicate:

```
observed,observe_position,and <- {true,false}.
and(X,Y) | X,Y = [true,false,false,false].
observe_position([ ],N) = true.
observe_position([H|T],N) = and(<emit(N)=H>,observe_position(T,s(N))).
observed(L) = observe_position(L,0).
```

The predicate `observe_position(X, Y)` indicates what the position of the system is `Y` at time `X`. For example, `observe_position([ ], N) = true` indicates that it is true that `N` (nothing) is observed at the start position. Without these last five lines, one must specify an observed sequence by including in the database a separate fact for each emission 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])`.

A product of HMMs is expressed 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, the logic-based stochastic language offers far more generality than is required to represent simple HMMs.

4.2. *Fault detection in a mechanical system*

We investigate the application of Loopy Logic to fault detection and diagnosis in mechanical systems. The time-series data was obtained from sensors monitoring helicopter rotors for the United States Navy. The task was to construct a quantitative model of the whole process and use it to diagnose and predict faults. Various techniques were investigated for preprocessing the data. Methods of modeling the system included simple correlative classification and Hidden Markov Models.[17,18]

The data available to us was sampled from the readings of various sensors monitoring a mechanical process. The data was collected over a period of time during which a fault was seeded in the mechanical process. For example, missing teeth in a gear or a crack in the drive shaft. The sensors were typically thermocouples and vibration meters that are continuous and analog devices and the data was subsequently sampled and available in digital format. Figure 7 shows such a sample.

As we can see, the raw data is intractable, noisy and unsuitable for any sort of mathematical or logical analysis. In order to get a better understanding on the nature of the data, it proved necessary and sufficient to look at its frequency characteristics. The frequency spectrum of the data was calculated using the fast Fourier transform algorithm. The data in this form seemed more tractable as is shown in Figure 8.

To get rid of some artifacts which may be due to noise in the frequency domain representation of the data and to consolidate information over time we computed the mean of several such windows. These processed datasets were considered observations relevant to the consequent modeling process.

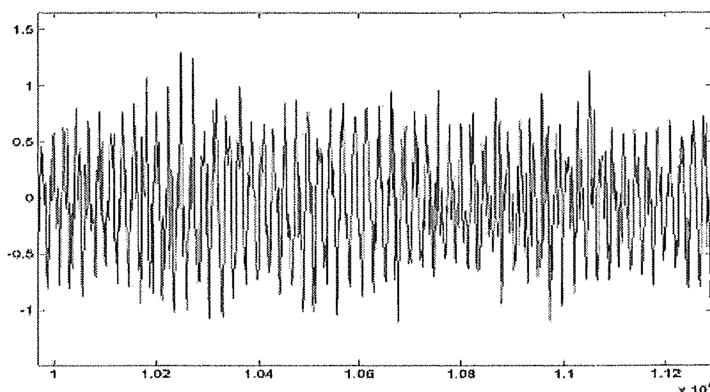

Fig. 7. Raw time-series data obtained directly from the mechanical process.

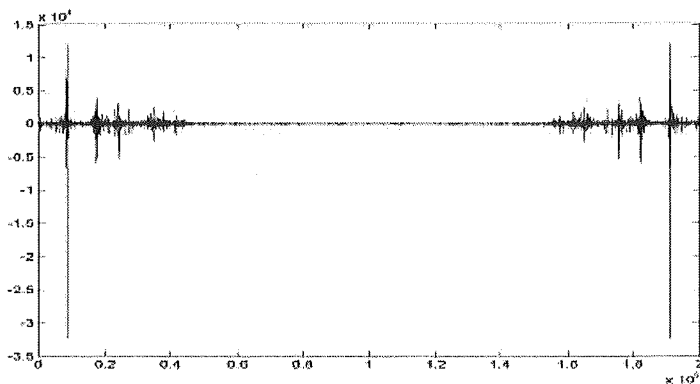

Fig. 8. Data after transformation into frequency domain and smoothing.

The mathematical correlation between observations was used as a metric of distance. Using this metric correlation plots were computed between half the observations that were chosen as training data. A significant and steep drop in correlation was noticed at samples bunched around a particular point in time. This point was around two thirds of the total observation time away from the first sample. Assuming that the center point of this lack of correlation was the point that the fault characteristics peaked, the time-line was split into three regions: `Safe`, `Unsafe` and `Faulty`.

Using these sets of correlation plots as our "learned" model of the data and fault process, the other half of the data, the test data, was correlated with the training dataset. The best fit of these new curves to the training correlation curves were computed using the Least Mean Square metric. With this method the test data was successfully classified as `Safe`, `Unsafe` or `Faulty` 75% of the time.

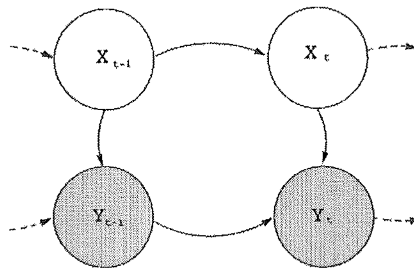

Fig. 9. Auto-regressive HMM.

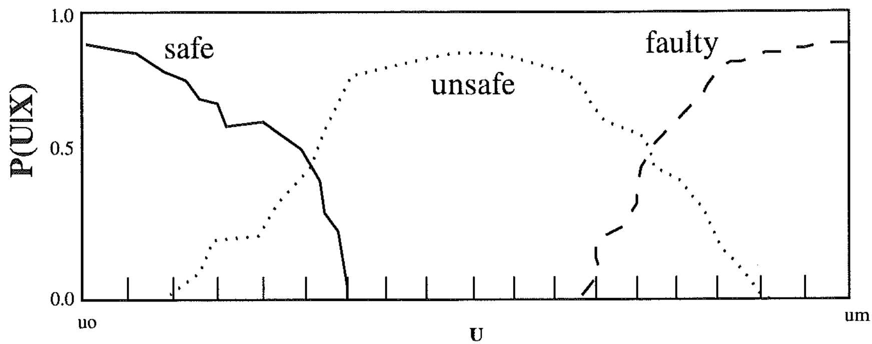

Fig. 10. A model of distributions $P(U|X = i)$ as learned from simulated training data.

For our model, in order to build a more robust, versatile and generic model than the above correlation-classification technique, we decided to explore the use of a variant of the hidden Markov model (HMM). The Auto Regressive Hidden Markov Model (AR-HMM)[19] seemed suitable for this purpose. The AR-HMM incorporates a correlation measure between consequent observations in time rather than just between states and state-observation pairs. Computationally, it provides an additional path of inference from observation to hidden state. Figure 9 shows the correlation link between states and observations at two consecutive instances of time (`t` and `t-1`).

The blank circles, labeled `X` are the hidden states of the system that could be one of `{Safe, Unsafe, Faulty}`. The shaded circles labeled `Y` are the observations. `X(t)` is the state of the system at time `t` and `Y(t)` is the observed signature at time `t`. Before we apply the algorithm to real time data we evaluate the distribution $P(u_j|X)$ of expected frequency signatures corresponding to the states from a state-labeled dataset. Note that $U = u_1, u_2 ... u_k$ is the set of observations that have been recorded while training the system. Say for example, if u_1 through u_k were observed when the system gradually went from `safe` to `faulty` we would expect $P(u_1|X = safe)$ to be much higher than $P(u_k|X = safe)$. See Figure 10 for a graphical representation of this probability.

In this 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. The notation *corrcoef* denotes correlation coefficiant. Correlation coefficient in this case describes the measure of similarity between the observed frequency signatures and the signatures obtained from the training data.

$$P(Y_t = y_t|X_t = i) = max(abs(corrcoef(y_t, u_j))) * P(u_t|X_t = i) . \tag{1}$$

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.

$$P(Y_t = y_t|Y_{t-1} = y_{t-1}) = \frac{abs(corrcoef(y_t, y_{t-1})) * ((\#u_{t-1}\ to\ u_t\ transitions)}{(\#u_{t-1}\ observations))} \tag{2}$$

where,

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

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

```
x <- {safe, unsafe, faulty}.
y(N) | x(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 very small training dataset of one seeded fault occurrence taking the system from `safe` to `faulty`. We obtained a performance accuracy close to 80% on this test data.[17]

5. Fault Diagnosis in Digital Circuits

As an example of the representational power of Loopy Logic, we consider the diagnosis of combinatorial (acyclic) digital circuits. 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 four values. The component is either good or has one of three failures: it is stuck with a value of `one`, stuck at `zero`, or intermittent, where the output of the element is random. 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 isn't working properly, and one or more test cases (values for inputs and outputs), 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. Let `N` be an ID (a tag for identifying the component) for a component of the circuit, and `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:

```
val, and, or, not <- {v0, v1}.
mode <- {good, s0, s1, random}.
```

The values `v0` and `v1` represent the low and high voltage values in a digital circuit. 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 put in 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`.

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

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 (one for `and`, zero 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 component.

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

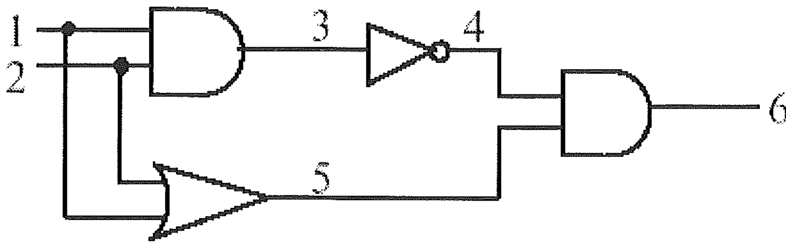

Fig. 11. A Sample Circuit that Implements XOR.

```
or([]) = v0.
or([H|T]) = val(H), or(T) = [[v0,v1],[v1,v1]].

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

The circuit of Figure 11 is described by the following four lines of code.

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

We now introduce failure probabilities into the different components. Above we used a learnable distribution, but for the sake of simplicity, in our tests we shall assume that the failure probabilities are fixed and are the same for all types of components.

```
mode(N) = [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) = v0.
val(2) = v1.
```

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

```
val (6)?
```

We get the following response:

```
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(N) = [0.5,0.3,0.2].
```

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

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

The interpreter response:

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

The interpreter indicates 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 that had a 1% probability of failure. We force the output at component 6 to be wrong.

```
mode(N) = [0.989, 0.01, 0.001].
val(1) = v0.
val(2) = v1.
val(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
s0: 0.325943967278
s1: 0.000680864679517
mode(5)
good: 0.67340708122
```

```
s0: 0.325943967278
s1: 0.000648951502418
mode(6)
good: 0.673729762485
s0: 0.326270237515
s1: 0
```

This response shows that 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) = v0.
val(2) = v1.
val(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
```

In our research, similar diagnostic tests on a dozen different circuits of varying sizes and complexity were performed.[17] 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. 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 above examples show, the representation of circuits and test data is transparent as well.

6. Learning through Parameter-fitting

As described in Section 2.3, parameter fitting (learning) is an important component of Loopy Logic. In order to demonstrate parameter fitting, we consider the simulation of a space station that models a small part of an advanced life support system.[17] The scenario involves the interaction between the power sub-system and the life support system on a remote base station. The power supply is dependent on an unknown external force and fluctuates. Life support has a number of states; `{normal, stressed, critical}`, that depend on power availability, demand, activity and location. The simulation assumes one astronaut. The consumption of life support resources is a function of the astronaut's exertion level and location. Our goal is to learn the model and predict the state of the life support system. Given that life support is dependent on power and consumption, we have a learnable distribution, where `N` is the time step and `LS` is the learnable distribution:

```
life_support(N) | power(N), consumption(N) = LS
```

The state of power can be monitored from voltage output, which can be in either of five states from very high to very low, `{vh,vmh,vm,vml,vl}`. We learn the distribution, `LS`, by first watching emission from life support that will raise alerts, `{ok, warning, danger}`. At some point life support emissions may end, but we still need to know the state of the life support system. We can do this using the learned distribution, `LS`. The life support system can be completely described in Loopy Logic by the following lines of code.[17] The numbers in the following example are constructed for illustrative purposes only. In an actual application they would be derived from statistical observations or standard non-informative priors such as max-entropy estimates derived from known constraints, see Jaynes.[20]

```
life_support <- {normal,stressed,critical}.
ls_emit <- {ok,warning,danger}.
power <- {high,medium,low}.
power_emit <- {vh,vmh,vm,vml,vl}.
person_activity <- {sleep,normal,hi_exert}.
person_location <- {in, out}.
consumption <- {low,med,high}.

consumption(N) | person_activity(N),
person_location(N)=
[[[0.7,0.2,0.1],[0.3,0.5,0.2]],[[0.2,0.5,0.3],
[0.6,0.2,0.2]],[[0.2,0.5,0.3],[0.1,0.2,0.7]]].
```

```
life_support(N) | power(N),consumption(N) = LS.

life_support(N) | ls_emit(N) = [[0.7,0.2,0.1],
[0.2,0.6,0.2],[0.1,0.2,0.7]].

power(N) | power_emit(N) =
[[0.7,0.2,0.1],[0.6,0.3,0.1],[0.2,0.6,0.2],
[0.1,0.3,0.6],[0.1,0.2,0.7]].
```

Here are some observations from the life support systems.

```
ls_emit(1)=danger
ls_emit(2)=danger
ls_emit(3)=danger
ls_emit(4)=warning
ls_emit(5)=ok
ls_emit(6)=ok
ls_emit(7)=ok
ls_emit(8)=ok
ls_emit(9)=ok
ls_emit(10)=warning
power_emit(1)=vml
power_emit(2)=vml
power_emit(3)=vm
power_emit(4)=vmh
power_emit(5)=vmh
power_emit(6)=vh
power_emit(7)=vh
power_emit(8)=vh
power_emit(9)=vh
power_emit(10)=vh
power_emit(11)=vmh
power_emit(12)=vh
power_emit(13)=vh
power_emit(14)=vh

person_activity(1)=hi exert
person_activity(2)=hi exert
person_activity(3)=normal
person_activity(4)=normal
person_activity(5)=normal
person_activity(6)=sleep
person_activity(7)=sleep
person_activity(8)=sleep
```

```
person_activity(9)=normal
person_activity(10)=hi exert
person_activity(11)=hi exert
person_activity(12)=hi exert
person_activity(13)=hi exert
person_activity(14)=hi exert
person_location(1)=out
person_location(2)=out
person_location(3)=in
person_location(4)=in
person_location(5)=in
person_location(6)=in
person_location(7)=in
person_location(8)=in
person_location(9)=in
person_location(10)=out
person_location(11)=out
person_location(12)=out
person_location(13)=out
person_location(14)=out
person_activity(10)=normal.
person_activity(11)=normal.
person_activity(12)=normal.
person_activity(13)=normal.
person_activity(14)=normal.
person_location(10)=in.
person_location(11)=in.
person_location(12)=in.
person_location(13)=in.
person_location(14)=in.
```

We begin the simulation at `time = 1` with life support in `critical` condition, power supply `low`, astronaut `outside` and in a state of `high exertion`. The power supply stabilizes around `time = 6`, and at the same time the astronaut goes to `sleep`. He later wakes up, begins `high exertion` activity and ventures `outside`. The power remains stable, except for a slight dip at `time = 11`. The life support emissions end at `time = 10`. Thereafter, the state of the system must be determined from the learnt distribution, `LS`. Table 1 shows the likelihood of states at each time step. The system determines that the state of life support after time step 10, when the astronaut is `outside` and exhibiting `high exertion`, is more likely to be in state `{stressed}`. This seems a logical inference because when the astronaut was in `high exertion` and the power level was `low`, the state of life support was `{critical}`. The high amount of exertion has likely put the life support system in

Table 1. Probabilities of life support system state at time steps 1,2,...,14.

	Life Support System States		
Time	Normal	Stressed	Critical
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0.77	0.23
5	0.74	0.14	0.12
6	0.92	0.2	0.06
7	0.93	0.01	0.06
8	0.96	0.03	0.04
9	0.95	0	0.05
10	0.11	0.78	0.11
11	0.21	0.49	0.3
12	0.23	0.53	0.24
13	0.24	0.54	0.23
14	0.23	0.53	0.26

Table 2. Probabilities of life support system when *person_activity* is *normal* and *person_location* is *in*.

	Life Support System States		
Time	Normal	Stressed	Critical
10	0.96	0	0.04
11	0.69	0	0.31
12	0.76	0	0.24
13	0.76	0	0.24
14	0.76	0	0.24

a `stressed` state, but since power output is `full`, it is not reaching a `critical` state. Also note that at `time = 11`, when the power output dipped slightly, the likelihood of being in state `critical` was at its highest level since `time = 3`. In contrast, we run another modified program where after the astronaut wakes up, he begins `normal` activity `inside`, as opposed to `high exertion` activity `outside`, with results displayed in Table 2. In this case, the network correctly infers that life support is more likely to be in a `normal` state. These results demonstrate Loopy Logic's ability to determine, learn and reason in uncertain situations. In this case, the uncertainty is which state the life support system is in after life support emissions have stopped.

7. Conclusions and Future Directions

We have created a new first-order and Turing-complete logic-based stochastic modeling language. This language is supported by a well-known and effective inference

algorithm, loopy belief propagation. 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.

The fault diagnosis example was a powerful illustration of the representational power of Loopy Logic. Without a first-order representational system, it would be a non-trivial task to design a system that can diagnose digital circuit failures as well as estimate failure probabilities from a dataset of test cases. With Loopy Logic, the basic circuit model can be constructed using only nine statements. As the four circuit examples show, the representation of circuits and test data is transparent as well.[17]

The experiments on the circuit showed that there are two questions that Loopy Logic can answer. For both questions, it is assumed that the probability distribution across different modes of failures (good, stuck at zero, stuck at one) is known for all component types. In the first question, given that a circuit is not working properly, Loopy Logic can calculate the failure probabilities of individual components of the circuit, i.e., which component in the circuit is most likely to have failed. This is useful in isolating the faulty component in the circuit. In the second question, Loopy Logic can calculate the accuracy of the predicted outputs of the digital circuits given the failure probabilities of individual components of the circuit.[17]

Mathematical analysis of the experimental results verified that Loopy Logic always correctly isolated the faulty component and could correctly calculate the level of uncertainty associated with the circuit outputs. Several interesting trends were noticed. In large circuits, it took several cycles to converge but still yielded consistent results. In larger circuits with a greater number of different gates, even a small failure probability in individual components influenced the accuracy of the result. This is because the small failure probabilities propagated along the gates and cumulatively resulted in a large failure probability. In faulty circuits with a large number of gates, Loopy Logic could isolate the faulty component with a higher degree of accuracy.[17]

The mechanical fault detection application demonstrated time series analysis. Loopy Logic had the ability to handle recursive structures. Hence it could effectively represent time series processes by constructing potentially infinite databases. The time series data from the mechanical system was modeled as an Auto-Regressive Hidden Markov Model. Loopy Logic was used for the inference in the AR-HMM. It was very easy to infer the learnable distributions with an accuracy close to 80%.[17]

The life support simulation was an example of parameter fitting. Loopy Logic's use of loopy belief propagation is a natural support for Expectation Maximization learning.[8]

Loopy Logic demonstrated the expressive power to represent time-series processes and perform time-series type learning and reasoning. The ability of the language to deal with recursion was also observed. It had the capability to handle potentially infinite situations with repetitive structure. This can enable us to build potentially infinite databases that can efficiently represent time-series learning and reasoning as well as various forms of Markov models.

There are several exciting future directions. The most promising area is modifying the language to handle continuous distributions, as the present system is only defined for discrete distributions. However, with the ability to handle continuous distributions as well, we can model systems that have continuous random variables, which are common in many situations. Now, in a first-order language, one could represent real values with infinite strings of Boolean random variables. Such a scheme would be cumbersome and quite inefficient. It would be a significant addition to allow continuous random variables to be modeled directly.

Another direction for extending this languages is as a decision support system. There would be significant enhancement through embedding utility values and decision points within a first-order stochastic modeling language. Using continuous variables might make it possible to support decision theory in the same framework. In addition, it would be very interesting if we could relax the constraint on rules that requires that all goals be ground, thus producing a more expressive language. Furthermore, we could 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.

Finally, the most ambitious extension is qualitative model induction. Currently, loopy logic can learn quantitative information about a domain; that is, it can discover the appropriate probabilities from the data. However, it would be exciting if it could also discover the form of the relations in the data. Alternately, one might use a maximum likelihood approach with a penalty function for complex models with many parameters. This latter approach would then entail a search through the space for the model with the best score.

Several research groups have explored this difficult area. Getoor *et al.*[21] and Segal *et al.*[22] consider model induction in the context of more traditional Bayesian Belief networks and Angelopoulos and Cussens[23] and Cussens[24] in the area of Constraint Logic Programming. Finally, the Inductive Logic Programming community[25,26] has also addressed the learning of structure in declarative stochastic representations. We plan to consider a combination of these three approaches.

Acknowledgments

This work was supported in part by NSF Research Grant IIS-9800929. We also recieved support from the U.S. Navy through an SBIR contract with Management Sciences Inc. of Albuquerque, NM. We thank C. Stern, K. Greene, Y. Ning and

N. Divakar for supporting this research with their hard work. We would also like to thank our reviewers for their insightful comments on earlier drafts of this document.

References

1. Ngo, L. and Haddawy, P. *Answering Queries from Context-Sensitive Knowledge Bases.* (Theoretical Computer Science, 1997), 171, pp. 147–177.
2. Koller, D. and Pfeffer, A. *Probabilistic Frame-Based Systems.* Proceedings of American Association of Artificial Intelligence Conference. (MIT Press, Cambridge, MA, 1998), pp. 580–587.
3. Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. *Learning Probabilistic Relational Models.* Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, (AAAI Press, Menlo Park, CA, 1999), pp. 1300–1307.
4. Kersting, K. and De Raedt, L. *Bayesian Logic Programs.* Proceedings of the Work-in-Progress Track at the Tenth International Conference on Inductive Logic Programming. (AAAI Press, Menlo Park, CA, 2000), pp. 138–155.
5. Ng, R. and Subrahmanian, V. *Probabilistic Logic Programming.* Information and Computation. (1992), 101(2), pp. 150–201.
6. Poole, D. *Logic Programming, Abduction and Probability: A top-down anytime algorithm for estimating prior and posterior probabilities.* New Generation Computing. (1993), 11(3-4), pp. 377–400.
7. The Caml Language "http://caml.inria.fr/about/index.en.html" 26 October, 2005.
8. Pless, D. *First-Order Stochastic Modeling.* Ph.D. Thesis. (Computer Science Department, University of New Mexico, Albuquerque, NM, 2003).
9. Mayraz, G. and Hinton, G. 2000. *Recognizing Hand-written Digits Using Hierarchical Products of Experts.* Advances in Neural Information Processing Systems 13 (Cambridge, MA: MIT Press), pp. 953–959.
10. Dempster, A., Laird, N., and Rubin, D. *Maximum likelihood from incomplete data via the EM algorithm.* Journal of the Royal statistical Society. Series B (Methodological) (1977), pp. 1–38.
11. Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* (Morgan Kaufmann, San Diego, CA, 1988).
12. Murphy, K., Weiss, Y., and Jordan, M. *Loopy Belief Propagation for Approximate Inference: An Empirical Study* Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (Morgan Kaufmann, San Francisco, CA, 1999), pp. 467–475.
13. R. McEliece, D. MacKay, and J. Cheng. *Turbo decoding as an instance of Pearl's "belief propagation algorithm".* IEEE Journal on Selected Areas in Communication. (1998), 16(2), pp. 40–152.
14. Pless, D. and Luger, G. *Loopy Logic.* Technical Report. (Computer Science Dept., University of New Mexico, Albuquerque, NM, 2002). TR-CS-2002-23.
15. Pless, D. and Luger, G. *EM Learning of Product Distributions in a First-Order Stochastic Logic Language.* Proceedings of the International Joint Conference on Artificial Intelligence (2003).
16. Yedidia, J., Freeman, W., and Weiss, Y. *Generalized Belief Propagation.* Advances in Neural Information Processing Systems 13 (MIT Press, Cambridge, MA, 2000), pp. 689–695.
17. Chakrabarti, C. *First-Order Stochastic Systems for Diagnosis and Prognosis.* MS Thesis (Computer Science Department, University of New Mexico, Albuquerque, NM, 2005).
18. Chakrabarti, C., Rammohan, R., Pless, D., and Luger, G. F. *A First-Order Stochas-*

tic Modeling Language for Diagnosis. Proceedings of the Eighteenth International FLAIRS Conference. (AAAI Press, Clearwater Beach, FL, 2005).

19. Juang, B. *On the Hidden Markov Model and Dynamic Time Warping for Speech Recognition: a unified view* Technical Report, vol 63, 1213-1243 AT&T Labs. (1984).
20. Jaynes, E.T., *Probability Theory: The Logic of Science*, Cambridge UK: The University Press, 2003.
21. Getoor, L., Friedman, N., Koller, D., and Pfeffer, A. *Learning Probabilistic Relational Models.* Relational Data Mining. eds: S. Dzeroski and N. Lavorac. (Springer, 2001).
22. Segal, E., Koller, D. and Ormoneit, D. *Probabilistic Abstraction Hierarchies.* Neural Information Processing Systems, MIT Press, Cambridge, MA, 2001).
23. Angelopoulos, N. and Cussens, J. *Markov Chain Monte Carlo Using Tree-Based Priors on Model Structure.* Proceedings of the Seventeenth Annual Conference on Uncertainty in Artificial Intelligence. (Morgan Kaufmann, San Francisco, CA, 2001), pp. 16–23.
24. Cussens, J. *Parameter Estimation in Stochastic Programs.* Machine Learning (2001), 44, pp. 245–271.
25. Muggleton, S. *Bayesian Inductive Logic Programming.* Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory, ed. M Warmuth (ACM Press, New York, NY, 1994), pp. 3–11.
26. Muggleton, S. *Learning Structure and Parameters in Stochastic Logic Programs.* Linkping University Electronic Articles in Computer and Information Science. (2002).