Mix-nets: Factored Mixtures of Gaussians in Bayesian Networks with Mixed Continuous And Discrete Variables

Scott Davies and Andrew Moore

April 2000

CMU-CS-00-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was supported in part by an NSF KDI Award.

Keywords: Bayesian networks, mixture models, machine learning

Abstract

Recently developed techniques have made it possible to quickly learn accurate probability density functions from data in low-dimensional continuous spaces. In particular, mixtures of Gaussians can be fitted to data very quickly using an accelerated EM algorithm that employs multiresolution kd-trees (Moore, 1999). In this paper, we propose a kind of Bayesian network in which low-dimensional mixtures of Gaussians over different subsets of the domain's variables are combined into a coherent joint probability model over the entire domain. The network is also capable of modelling complex dependencies between discrete variables and continuous variables without requiring discretization of the continuous variables. We present efficient heuristic algorithms for automatically learning these networks from data, and perform comparative experiments illustrating how well these networks model real scientific data and synthetic data. We also briefly discuss some possible improvements to the networks, as well as their possible application to anomaly detection, classification, probabilistic inference, and compression.

1 Introduction

Bayesian networks (otherwise known as belief networks) are a popular method for representing joint probability distributions over many variables. (See, e.g., (Pearl, 1988).) A Bayesian network contains a directed acyclic graph G with one vertex V_i in the graph for each variable X_i in the domain. The directed edges in the graph specify a set of independence relationships between the variables. Define $\vec{\Pi}_i$ to be the set of variables whose nodes in the graph are "parents" of V_i. The set of independence relationships specified by a given graph is then as follows: given the values of $\vec{\Pi}_i$ but no other information, X_i is conditionally independent of all variables corresponding to nodes that are not V_i's descendants in the graph. This set of independence relationships allows us to decompose the joint probability distribution $P(\vec{X})$ in the following manner:

$$P(\vec{X}) = \prod_{i=1}^{N} P(X_i|\vec{\Pi}_i),$$

where N is the number of variables in the domain. Thus, if in addition to G we also specify $P(X_i|\vec{\Pi}_i)$ for every variable X_i, then we have specified a valid probability distribution $P(\vec{X})$ over the entire domain.

Bayesian networks are most commonly used in situations where all the variables are discrete, largely because it is difficult to model complex probability densities over continuous variables, and difficult to model interactions between continuous and discrete variables. When Bayesian networks with continuous variables are used, the continuous variables are usually modeled with simple parametric forms such as multidimensional Gaussians. Some researchers have recently investigated the use of more complicated continuous distributions within Bayesian networks; for example, weighted sums of Gaussians have been used to approximate conditional probability density functions (Driver & Morrell, 1995). Unfortunately, such complex distributions over continuous variables are usually quite computationally expensive to learn. If an appropriate Bayesian network structure is known beforehand, then this expense may not be too problematic, since only N conditional distributions must be learned. On the other hand, if the dependencies between variables are not known *a priori* and the structure must be learned from the data, then the number of conditional distributions that must be learned and tested while a structure-learning algorithm searches for a good network can become unmanageably large.

However, very fast algorithms for generating complex joint probability densities over small sets of continuous variables have recently been developed (Moore, 1999). This paper investigates how these algorithms can be employed to learn Bayesian networks over many variables, each of which can be either continuous or discrete. In section 2, we describe the type of parameterizations employed in our networks' nodes, and how they are learned from data given a fixed Bayesian network structure. In section 3, we describe an algorithm for automatically learning the structures of our Bayesian networks from data. In section 4, we provide experimental results illustrating the effectiveness of our methods on two real scientific datasets and two synthetic datasets. In section 5 we discuss possible applications, and finally in section 6 we discuss a few possible lines of future research.

2 Mix-nets

2.1 General methodology

Suppose that we have a very fast, black-box algorithm A geared not towards finding accurate conditional models of the form $P_i(X_i|\vec{\Pi}_i)$, but rather towards finding accurate *joint* probability models $P_i(\vec{S}_i)$ over subsets of variables $\vec{S}_i$, such as $P_i(X_i, \vec{\Pi}_i)$. Furthermore, suppose it is only capable of generating joint models for relatively small subsets of the variables, and that the models returned for different subsets of variables are not necessarily consistent. For example, if we were to ask A for two different models $P_1(X_5, X_{17})$ and $P_2(X_5, X_{24})$, the marginal distributions $P_1(X_5)$ and $P_2(X_5)$ of these models may be inconsistent. Can we still combine many different models generated by A into a valid probability distribution over the entire space?

Fortunately, the answer is yes, as long as the models returned by A can be marginalized exactly. If for any given $P_i(X_i, \vec{\Pi}_i)$ we can compute a marginal distribution $P_i(\vec{\Pi}_i)$ that is consistent with it, then we can employ P_i as a conditional distribution $P_i(X_i|\vec{\Pi}_i)$ trivially as follows:

$$P_i(X_i|\vec{\Pi}_i) = P_i(X_i, \vec{\Pi}_i)/P_i(\vec{\Pi}_i).$$

In this case, given a directed acyclic graph G specifying a Bayesian network structure over N variables, we can simply use A to acquire N models $P_i(X_i, \vec{\Pi}_i)$, marginalize these models, and string them together to form a probability distribution over the entire space:

$$P(\vec{X}) = \prod_{i=1}^{N} P_i(X_i, \vec{\Pi}_i)/P_i(\vec{\Pi}_i)$$

A simple but key observation is that even though the marginals of different P_i's may be inconsistent with each other, the P_i's are only *used* conditionally, and in a manner that prevents these inconsistencies from actually causing the overall model to become inconsistent. Of course, the fact that there are inconsistencies at all — suppressed or not — means that there is a certain amount of redundancy in the overall model. However, if allowing such redundancy lets us employ a particularly fast and effective model-learning algorithm A, it may be worth it.

Previous research has similarly conditionalized joint models over subsets of variables in order to use them within Bayesian networks. For example, the conditional distribution of each variable in the network given its parents can be modeled by conditionalizing another "embedded" Bayesian network that specifies the joint between the variable and its parents (Heckerman & Meek, 1997a). (Some theoretical issues concerning the interdependence of parameters in such models appear in (Heckerman & Meek, 1997a) and (Heckerman & Meek, 1997b).) Joint distributions formed by convolving a Gaussian kernel function with each of the datapoints have also been conditionalized for use in Bayesian networks over continuous variables (Hofmann & Tresp, 1995).

2.2 Handling continuous variables

Suppose for the moment that $\vec{X}$ contains only continuous values. What sorts of models might we want A to return? One powerful type of model for representing probability density functions over small sets of variables is a *Gaussian mixture model* (see e.g. (Duda & Hart, 1973)). Let $\vec{s_j}$ represent the values that the j^{th} datapoint in the dataset D assigns to a variable set of interest $\vec{S}$. In a Gaussian mixture model over $\vec{S}$, we assume that the data are generated independently through the following process: for each $\vec{s_j}$ in turn, nature begins by randomly picking a class, c_k, from a discrete set of classes $c_1, \ldots, c_M$. Then nature draws $\vec{s_j}$ from a multidimensional Gaussian whose mean vector $\vec{\mu_k}$ and covariance matrix Σ_k depend on the class. This produces a distribution of the following mathematical form:

$$P(\vec{S}|\vec{\theta}) = \sum_{k=1}^{M} \alpha_k (2\pi||\Sigma_k||)^{-\frac{1}{2}} exp(-\frac{1}{2}(\vec{S} - \vec{\mu_k})^T \Sigma_k^{-1} (\vec{S} - \vec{\mu_k}))$$

where α_k represents the probability of a point coming from the k^{th} class, and

$$\vec{\theta}^T = \{\alpha_1, \ldots, \alpha_M, \vec{\mu_1}, \ldots, \vec{\mu_M}, \Sigma_1, \ldots, \Sigma_M\}$$

denotes the entire set of the mixture's parameters. It is possible to model any continuous probability distribution with arbitrary accuracy by using a Gaussian mixture with a sufficiently large M.

Given a Gaussian mixture model $P_i(X_i, \vec{\Pi}_i)$, it is easy to compute the marginalization $P_i(\vec{\Pi}_i)$: the marginal mixture has the same number of Gaussians as the original mixture, with the same α's. The means and covariances of the marginal mixture are simply the means and covariances of the original mixture with all elements corresponding to the variable X_i removed. Thus, Gaussian mixture models are suitable for combining into global joint probability density functions using the methodology described in section 2.1, assuming all variables in the domain are continuous. This is the class of models we employ for continuous variables in this paper, although many other classes may be used in an analogous fashion.

Note that while the functional form of each $P_i(X_i|\vec{\Pi}_i)$ is expressible as a mixture of Gaussians over X_i for any *specific* set of values assigned to $\vec{\Pi}_i$, it is not generally expressible as a finite mixture of Gaussians over $X_i \cup \vec{\Pi}_i$. For example, a two-variable mixture $P(X, \Pi)$ composed of two axis-aligned Gaussians is shown in Figure 1, along with the corresponding $P(X|\Pi)$. For any fixed value π of Π, $P(X|\pi)$ is a mixture of two Gaussians, but $P(X|\Pi)$ as a function of both X and Π cannot be expressed as a finite mixture of Gaussians. (To see this, note that each of the two "ridges" in the bottom half of the plot for $P(X|\Pi)$ extends to infinity in one direction — one in the $-\Pi$ direction and one in the $+\Pi$ direction.)

The functional form of the conditional distribution we use is similar to that employed in previous research by conditionalizing a joint distribution formed by convolving a Gaussian kernel function with all the datapoints (Hofmann & Tresp, 1995). The differences are that our distributions use fewer Gaussians, but these Gaussians have varying weights and varying non-diagonal covariance matrices. The use of fewer Gaussians makes our method more suitable for some applications such as compression, and may speed up inference. (Our method may also yield more accurate models in many situations, but we have yet to verify this experimentally.)

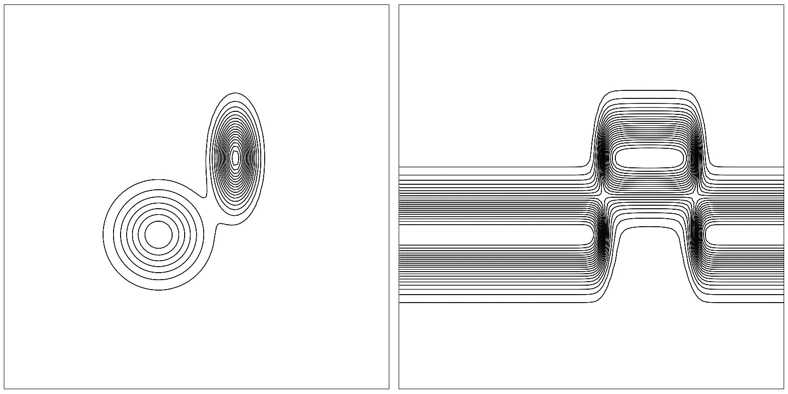

Figure 1: Contour plots for a simple Gaussian mixture $P(X, \Pi)$ (on the left) and the corresponding conditional distribution $P(X|\Pi)$ (on the right). X is the vertical axis and Π is the horizontal axis.

2.2.1 Learning Gaussian mixtures from data

The EM algorithm is a popular method for learning mixture models from data (see, e.g., (Dempster et al., 1977)). The algorithm is an iterative algorithm with two steps per iteration. The Expectation or "E" step calculates the distribution over the unobserved mixture component variables, using the current estimates for the model's parameters. The Maximization or "M" step then re-estimates the model's parameters to maximize the likelihood of both the observed data and the unobserved mixture component variables, assuming the distribution over mixture components calculated in the previous "E" step is correct. For Gaussian mixture models, the steps of the EM algorithm work as follows:

- E step: Given the current network parameters $\vec{\theta}$, for each datapoint $\vec{s_j}$ and each class c_k, calculate the extent w_{jk} to which class c_k "owns" $\vec{s_j}$: $w_{jk} = P(c_k|s_j, \vec{\theta})$.
- M step: Adjust $\vec{\theta}$ as follows:

$$\alpha_k = \frac{sw_k}{R}, \vec{\mu_k} = \frac{1}{sw_k}\sum_{j=1}^{R} w_{jk}\vec{s_j},$$

$$\Sigma_j = \frac{1}{sw_k}\sum_{j=1}^{R} w_{jk}(\vec{s_j} - \vec{\mu_k})(\vec{s_j} - \vec{\mu_k})^T$$

where R is the number of datapoints in the dataset and $sw_k = \sum_{j=1}^{R} w_{jk}$.

Each iteration of the EM algorithm increases the likelihood of the observed data or leaves it unchanged; if it leaves it unchanged, this usually indicates that the likelihood is at a local maximum. Unfortunately, each iteration of the basic algorithm described above is slow, since it requires a entire pass through the data. Instead, we use an accelerated EM algorithm in which multiresolution kd-trees (Moore et al., 1997) are used to dramatically reduce the computational cost of each iteration (Moore, 1999). We refer the interested reader to this previous paper (Moore, 1999) for details.

An important remaining issue is how to choose the appropriate number of Gaussians, M, for the mixture. If we restrict ourselves to too few Gaussians, we will fail to model the data accurately; on the other hand, if we allow ourselves too many, then we may "overfit" the data and our model may generalize poorly. A popular way of dealing with this tradeoff is to choose the model maximizing a scoring function that includes penalty terms related to the number of parameters in the model. We employ the Bayesian Information Criterion (Schwarz, 1978), or BIC, to choose between mixtures with different numbers of Gaussians. The BIC score for a given probability model $P'(\vec{S})$ is as follows:

$$BIC(P') = \log P'(D_S) - \frac{\log R}{2}|P'|$$

where D_S is the dataset D restricted to the variables of interest $\vec{S}$, R is the number of datapoints in the dataset, and $|P'|$ is the number of parameters in P'.

Rather than re-run the EM algorithm to convergence for many different choices of M and choosing the resulting mixture that maximizes the BIC score, we use a heuristic algorithm that starts with a small number of Gaussians and stochastically tries adding or deleting Gaussians as it progresses. Gaussians with high overall probabilities are sometimes each split into two Gaussians, and Gaussians with low overall probabilities are sometimes eliminated. After the number of Gaussians is changed in this fashion, the EM algorithm is run for a few more iterations. If the resulting mixture has a higher BIC score than the BIC score of the mixture with the previous number of Gaussians, then the algorithm continues; otherwise it resets its distribution back to the mixture with the previous number of Gaussians, runs the EM algorithm for a few more iterations, and then continues stochastically from there. The details of this algorithm are described in a separate forthcoming paper (Sand & Moore, 2000).

2.3 Handling discrete variables

Suppose now that a set of variables $\vec{S_i}$ we wish to model includes discrete variables as well as continuous variables. Let $\vec{Q_i}$ be the discrete variables in $\vec{S_i}$, and $\vec{C_i}$ the continuous variables in $\vec{S_i}$. One simple model for $P_i(\vec{Q_i}, \vec{C_i})$ is a lookup table with an entry for each possible set $\vec{q_i}$ of assignments to $\vec{Q_i}$. The entry in the table corresponding to a particular $\vec{q_i}$ contains two things: the marginal probability $P_i(\vec{q_i})$, and a Gaussian mixture modeling the conditional distribution $P_i(\vec{C_i}|\vec{q_i})$. Let us refer to tables of this form as *mixture tables*. We obtain the mixture table's estimate for each $P_i(\vec{q_i})$ directly from the data: it is simply the fraction of the records in the dataset that assigns the values $\vec{q_i}$ to $\vec{Q_i}$. Given an algorithm A for learning Gaussian mixtures from continuous data, we use it to generate each conditional distribution $P_i(\vec{C_i}|\vec{q_i})$ in the mixture table by calling it with the subset of the dataset D that is consistent with the values specified by $\vec{q_i}$.

Suppose now that we are given a Bayesian network structure over the entire set of variables, and for each variable X_i we are given a mixture table for $P_i(\vec{S_i}) = P_i(X_i, \vec{\Pi_i})$. We must now calculate new mixture tables for each of the marginal distributions $P_i(\vec{\Pi_i})$ so that we can use them for the conditional distributions $P_i(X_i|\vec{\Pi_i}) = P_i(X_i, \vec{\Pi_i})/P_i(\vec{\Pi_i})$. Let $\vec{C_i}$ represent the continuous variables in $\{X_i\} \cup \vec{\Pi_i}$; $\vec{Q_i}$ represent the discrete variables in $\{X_i\} \cup \vec{\Pi_i}$; $\vec{C_{\Pi_i}}$ represent the continuous variables in $\vec{\Pi_i}$; and $\vec{Q_{\Pi_i}}$ represent the discrete variables in $\vec{\Pi_i}$. (Either $\vec{Q_{\Pi_i}} = \vec{Q_i}$ or $\vec{C_{\Pi_i}} = \vec{C_i}$, depending on whether X_i is continuous or discrete.)

If X_i is continuous, then the marginalized mixture table for $P_i(\vec{\Pi_i})$ has the same number of entries as the original table for $P_i(X_i, \vec{\Pi_i})$, and the estimates for $P(\vec{Q_i})$ in the marginalized table are the same as in the original table. For each combination of assignments to $\vec{Q_i}$, we simply marginalize the appropriate Gaussian mixture $P_i(\vec{C_i}|\vec{Q_i}) = P_i(\vec{C_i}|\vec{Q_{\Pi_i}})$ in the original table to a new mixture $P_i(\vec{C_{\Pi_i}}|\vec{Q_{\Pi_i}})$, and use this new mixture in the corresponding spot in the marginalized table.

If X_i is discrete, then for each combination of assignments to $\vec{Q_{\Pi_i}}$, we combine several different Gaussian mixtures for various $P_i(\vec{C_{\Pi_i}}|\vec{Q_i})$'s into a new Gaussian mixture for $P_i(\vec{C_{\Pi_i}}|\vec{Q_{\Pi_i}})$. First, the values of $P_i(\vec{Q_{\Pi_i}})$ in the marginalized table are computed trivially from the original table as $P_i(\vec{Q_{\Pi_i}}) = \sum_{X_i} P_i(X_i, \vec{Q_{\Pi_i}})$. $P_i(X_i|\vec{Q_{\Pi_i}})$ is then calculated as $P_i(X_i, \vec{Q_{\Pi_i}})/P_i(\vec{Q_{\Pi_i}})$. Finally, we combine the Gaussian mixtures corresponding to different values of X_i according to the relationship

$$P_i(\vec{C_{\Pi_i}}|\vec{Q_{\Pi_i}}) = \sum_{X_i} P_i(X_i|\vec{Q_{\Pi_i}}) P_i(\vec{C_{\Pi_i}}|\vec{Q_i}).$$

We have now described the steps necessary to use mixture tables in order to parameterize Bayesian networks over domains with both discrete and continuous variables. Note that mixture tables are not particularly well-suited for dealing with discrete variables that can take on many possible values, or for scenarios involving many dependent discrete variables — in such situations, the continuous data will be shattered into many separate Gaussian mixtures, each of which will have little support. Better ways of dealing with discrete variables are undoubtedly possible, but we leave them for future research (see section 6.3). (We will briefly discuss how we currently handle mixture tables' potential problems with sparse data in our experimental results section.)

3 Learning mix-net structures

Given a Bayesian network structure over a domain with both discrete and continuous variables, we now know how to learn mixture tables describing the joint probability of each variable and its parent variables, and how to marginalize these mixture tables to obtain the conditional distributions needed to compute a coherent probability function over the entire domain. But what if we don't know *a priori* what dependencies exist between the variables in the domain — can we learn these dependencies automatically and find the best Bayesian network structure on our own, or at least find a "good" network structure?

In general, finding the optimal Bayesian network structure with which to model a given dataset is NP-complete (Chickering, 1996), even when all the data is discrete and there are no missing values or hidden variables. A popular heuristic approach to finding networks that model discrete data well is to hillclimb over network structures, using a scoring function such as the BIC as the criterion to maximize. (See, e.g., (Heckerman et al., 1995).) Unfortunately, hillclimbing usually requires scoring a very large number of networks. While our algorithm for learning Gaussian mixtures from data is comparatively fast for the complex task it performs, it is still too expensive to re-run on the hundreds of thousands of different variable subsets that would be necessary to parameterize all the networks tested over an extensive hillclimbing run. (Such a hillclimbing algorithm has previously been used to find Bayesian networks suitable for modeling continuous data with complex distributions (Hofmann & Tresp, 1995), but in practice this method is restricted to datasets with relatively small numbers of variables and datapoints.)

However, there are other heuristic algorithms that often find networks very close in quality to those found by hillclimbing but with much less computation. A frequently used class of algorithms involves measuring all pairwise interactions between the variables, and then constructing a network that models the strongest of these pairwise interactions (e.g. (Chow & Liu, 1968), (Sahami, 1996), (Friedman et al., 1999)). We use such an algorithm in this paper to automatically learn the structures of our Bayesian networks.

In order to measure the pairwise interactions between the variables, we start with an empty Bayesian network B_ϵ in which there are no arcs — i.e., in which all variables are assumed to be independent. We use our mixture-learning algorithm to calculate the parameters in this empty network, and then calculate this network's BIC score. The BIC score of a given Bayesian network B is simply the log-likelihood of the dataset D given the network, minus a penalty term proportional to the number of parameters in the network:

$$BIC(B) = \log P(D|B) - \frac{\log R}{2}|B|,$$

where $|B|$ is the number of parameters in the entire network B. The number of parameters in the network is equal to the sum of the parameters in each network node, where the parameters of a node for variable X_i are the parameters of $P_i(X_i, \Pi)$.

Once we have calculated the BIC score of the empty network B_ϵ, we calculate the BIC score of every possible Bayesian network containing exactly one arc. With N variables, there are $\binom{N}{2}$ or $O(N^2)$ such networks. Let B_{ij} denote the network with a single arc from X_i to X_j. Note that to compute the BIC score of B_{ij}, we need not recompute the mixture tables for any variable other than X_j, since the others can simply be copied from B_ϵ. Now, define $I(X_i, X_j)$, the "importance" of the dependency between variable X_i

and X_j, as follows:

$$I(X_i, X_j) = BIC(B_{ij}) - BIC(B_\epsilon).$$

After computing all the $I(X_i, X_j)$'s, we initialize our current working network B to the empty network B_ϵ, and then greedily add arcs to B using the $I(X_i, X_j)$'s as hints for what arcs to try adding next. At any given point in the algorithm, the set of variables is split into two mutually exclusive subsets, *DONE* and *PENDING*. All variables begin in the *PENDING* set. Our algorithm proceeds by selecting a variable in the *PENDING* set, adding arcs to that variable from other variables in the *DONE* set, moving the variable to the *DONE* set, and repeating until all variables are in *DONE*. High-level pseudo-code for the algorithm appears in Figure 2.

The algorithm generates and tests $O(N^2)$ mixture tables containing two variables each in order to calculate all the pairwise dependency strengths $I(X_i, X_j)$, and then $O(N * K)$ more tables containing $MAXPARS + 1$ or fewer variables each during the greedy network construction. K is a user-defined parameter that determines the maximum number of potential parents evaluated for each variable during the greedy network construction.

Note that as the algorithm is described above, the step in the algorithm labeled with a "†" might appear to take $O(N^2)$ time, thus bumping the overall time of the algorithm up to $O(N^3)$. By caching information between iterations, the cost of this step per iteration could be reduced to $O(N \log K)$, for a total cost of $O(N^2 \log K)$. However, this savings is largely irrelevant; the real cost of the structure-learning algorithm lies in the $O(N^2)$ calls to the mixture-table learning algorithm. Each of these calls typically takes at least $O(R)$ time, where R is the number of records in the dataset, and R is typically much larger than N.

If *MAXPARS* is set to 1 and $I(X_i, X_j)$ is symmetric, then our heuristic algorithm reduces to a maximum spanning tree algorithm (or to a maximum-weight forest algorithm if some of the I's are negative). Out of all possible Bayesian networks in which each variable has at most one parent, this maximum spanning tree is the Bayesian network B^1_{opt} that maximizes the scoring function. (This is a trivial generalization of the well-known algorithm (Chow & Liu, 1968) for the case where the unpenalized log-likelihood is the objective criteria being maximized.) If *MAXPARS* is set above 1, our heuristic algorithm will always model a superset of the dependencies in B^1_{opt}, and will always find a network with at least as high a BIC score as B^1_{opt}'s.

- $B := B_\epsilon$, *PENDING* := the set of all variables, *DONE* := {}
- While there are still variables in *PENDING*:
  - Consider all pairs of variables X_d and X_p such that X_d is in *DONE* and X_p is in $PENDING$.† Of these, let X_d^{max} and X_p^{max} be the pair of variables that maximizes $I(X_d, X_p)$. Our algorithm selects X_p^{max} as the next variable to consider adding arcs to. (Ties are handled arbitrarily, as is the case where *DONE* is currently empty.)
  - Let $K' = \min(K, |DONE|)$, where K is a user-defined parameter. Let $X_d^1, X_d^2, \ldots X_d^{K'}$ denote the K' variables in *DONE* with the highest values of $I(X_d^i, X_p^{max})$, in descending order of $I(X_d^i, X_p^{max})$.
  - For $i = 1$ to K':
    * If X_p^{max} now has *MAXPARS* parents in B, or if $I(X_d^i, X_p^{max})$ is less than zero, break out of the for loop over i and do not consider adding any more parents to X_p^{max}.
    * Let B' be a network identical to B except with an additional arc from X_d^i to X_p^{max}. Call our mixture-learning algorithm to update the parameters for X_p^{max}'s node in B', and compute $BIC(B')$.
    * If $BIC(B') > BIC(B), B := B'$.
  - Move X_p^{max} from *PENDING* to *DONE*.

Figure 2: Our greedy network structure learning algorithm.

There are a few details that prevent our $I(X_i, X_j)$'s from being perfectly symmetric. Because the mixtures we use have redundant parameters, the number of parameters in B_{ij} and B_{ji} are not necessarily equal, and so the two networks' BIC scores may be different even if the distributions they model are identical. Furthermore, the distributions modeled by the two networks will not generally be identical, since our mixture-learning algorithm is stochastic and will not usually find distributions with the truly highest possible likelihoods. Also, even in scenarios in which all the variables are discrete, the two distributions may not be identical because of the slight adjustments we make in our models' parameters in order to handle sparse data (as described in the experimental results section). In practice, however, I is close enough to symmetric that it's often worth pretending that it is symmetric, since this cuts down the number of calls we need to make to our mixture-learning algorithm in order to calculate the $I(X_i, X_j)$'s by roughly a factor of 2.

Since learning joint distributions involving real variables is expensive, calling our mixture table generator even just $O(N^2)$ times to measure all of the $I(X_i, X_j)$'s can take a prohibitive amount of time. We note that the $I(X_i, X_j)$'s are only used to choose the order in which the algorithm selects variables to move from *PENDING* to *DONE*, and to select which arcs to try adding to the graph. The actual values of $I(X_i, X_j)$ are irrelevant — the only things that matter are their ranks and whether they are greater than zero. Thus, in order to reduce the expense of computing the $I(X_i, X_j)$'s, we can try computing them on a *discretized* version of the dataset rather than the original dataset that includes continuous values. The resulting ranks of $I(X_i, X_j)$ will not generally be the same as they would if they were computed from the original dataset, but we would expect them to be highly correlated in many practical circumstances.

Our structure-learning algorithm is similar to the "Limited Dependence Bayesian Classifiers" previously employed to learn networks for classification (Sahami, 1996), except that our networks have no special target variable, and we add the potential parents to a given node one at a time to ensure that each actually increases the network's score. Alternatively, our learning algorithm can be viewed as a restriction of the "Sparse Candidate" algorithm (Friedman et al., 1999) in which only one set of candidate parents is generated for each node, and in which the search over network structures restricted to those candidate parents is performed greedily. (We have also previously used a very similar algorithm for learning networks with which to compress discrete datasets (Davies & Moore, 1999).)

4 Experiments

In this section, we compare the performance of the network-learning algorithm described above to the performance of four other algorithms. Each of the four other algorithms is designed to be similar to our network-learning algorithm except in one important respect. First we describe a few details about how our primary network-learning algorithm is used in our experiments, and then we describe the four alternative algorithms.

4.1 Algorithms

4.1.1 Mix-net learner

This is our primary network-learning algorithm. For our experiments on both datasets, we set $MAXPARS$ to 3 and K to 6. When generating any given Gaussian mixture, we give our accelerated EM algorithm thirty seconds to find the best mixture it can. In order to make the most of these thirty-second intervals, we also limit our overall training algorithm to using a sample of at most 10,000 datapoints from the training set. Rather than computing the $I(X_i, X_j)$'s with the original dataset, we compute them with a version of the dataset in which each continuous variable has been discretized to 16 different values. The boundaries of the 16 bins for each variable's discretization are chosen so that the number of datapoints in each bin is approximately equal.

Mixture tables containing many discrete variables (or a few discrete variables each of which can take on many values) can severely overfit data, since some combinations of the discrete variables may occur rarely in the data. For now, we attempt to address this problem as follows:

- The estimates for the distribution $P_i(\vec{Q}_i)$ over the discrete variables in any given mixture table are smoothed by adding half a datapoint's worth of probability mass to each possible combination and renormalizing accordingly.
- In addition to the Gaussian components, each mixture over continuous variables contains a uniform component. This uniform component represents a constant density over a hypervolume bounding the entire

dataset. We fix this uniform component's total probability mass at half a datapoint's worth, and renormalize the distribution accordingly. If there are too few datapoints in the mixture to fit even a single Gaussian, then the mixture contains only this uniform component, which is assigned a total probability mass of one in this special case.

Whenever Gaussian mixtures are learned, there is a possibility that a Gaussian will become ill-conditioned and further mathematical operations will fail due to roundoff error. Even worse, a Gaussian may shrink to an arbitrarily small size around a single datapoint and thus contribute an arbitrarily large amount to the log-likelihood of the training data. We help prevent these conditions from occurring by adding a small constant to the diagonal elements of all Gaussians' covariance matrices. (A more principled but slightly more complex approach would be to use a prior over the Gaussians' parameters, such as a normal-Wishart distribution.)

4.1.2 Independent Mixtures

This algorithm will help us illustrate how much leverage our mix-net learning algorithm gets by modeling any dependencies between variables at all. It is identical to our mix-net learning algorithm in almost all respects; the main difference is that here the *MAXPARS* parameter has been set to zero, thus forcing all variables to be modeled independently. We also give this algorithm more time to learn each individual Gaussian mixture, so that it is given a total amount of computational time at least as great as that used by our mix-net learning algorithm.

4.1.3 Trees

This algorithm will help us illustrate how much leverage our mix-net learning algorithm gets by generating models more complex than the optimal tree-shaped (or forest-shaped) network. It is identical to our primary network-learning algorithm in all respects except that the *MAXPARS* parameter has been set to one, and we give it more time to learn each individual Gaussian mixture (as we did for the Independent Mixtures algorithm).

4.1.4 Single-Gaussian Mixtures

This algorithm will help us illustrate how much leverage our mix-net learning algorithm gets by using mixtures containing multiple Gaussians. It is identical to our primary network-learning algorithm except for the following differences. When learning a given Gaussian mixture $P_i(\vec{C_i}|\vec{Q_i})$, we use a single multidimensional Gaussian rather than a mixture. (Note, however, that some of the marginal distributions $P_i(\vec{C_{\Pi_i}}|\vec{Q_{\Pi_i}})$ may contain multiple Gaussians when the variable marginalized away is discrete.) Since single Gaussians are much easier to learn in high-dimensional spaces than mixtures are, we allow this single-Gaussian algorithm much more freedom in creating large mixtures. We set both $MAXPARS$ and K to the total number of variables in the domain minus one. We also allow the algorithm to use all datapoints in the training set rather than restrict it to a sample of 10,000. Finally, we use the original real-valued dataset rather than a discretized version of the dataset when computing each pairwise interaction $I(X_i, X_j)$.

Disclaimer: when modeling a set of N continuous variables (and no discrete variables) with this type of mix-net, the overall distribution modeled is simply a N-dimensional Gaussian. Unfortunately, a multidimensional Gaussian with a full covariance matrix would take $O(N^3)$ (mostly redundant) parameters to model with a mix-net rather than the usual $O(N^2)$. (See section 6.1 for one possible partial workaround to this problem.) On the other hand, mix-nets do have the added flexibility of being able to model *some* important dependencies between N continuous variables without modeling all $O(N^2)$ of them, and of being able to model interactions between discrete and continuous variables.

4.1.5 Pseudo-Discrete Bayesian Networks

This algorithm is similar to our primary network-learning algorithm in that it uses the same sort of greedy algorithm to select which arcs to try adding to the network. However, the networks this algorithm produces do not employ Gaussian mixtures. Instead, the distributions it uses are closely related to the distributions that would be modeled by a Bayesian network for a completely discretized version of the dataset. For each continuous variable X_i in the domain, we break X_i's range into F buckets. The boundaries of the buckets are chosen so that the number of datapoints lying within each bucket is approximately equal. The conditional distribution for X_i is modeled with a table containing one entry for every combination of its parent variables, where each continuous parent variable's value is discretized according to the F buckets we have selected for that parent variable. Each entry in the table contains a histogram for X_i recording the conditional probability that X_i's value lies within the boundaries of each of X_i's F buckets. We then translate the conditional probability associated with each bucket into a conditional probability density spread uniformly throughout the range of that bucket. (Discrete variables are handled in a similar manner, except the translation from conditional probabilities to conditional probability densities is not performed.)

When performing experiments with this algorithm, we re-run it for several different choices of F: 2, 4, 8, 16, 32, and 64. Of the resulting networks, we pick the one that maximizes the BIC. When the algorithm uses a particular value for F, the variable interactions $I(X_i, X_j)$ are computed using a version of the dataset that has been discretized accordingly, and then arcs are added greedily as in our mix-net learning algorithm. The networks produced by this algorithm do not have redundant parameters as our mix-nets do, as each node contains only a model of its variable's conditional distribution given its parents rather than a joint distribution.

Disclaimer: much research has been performed on better ways of discretizing real variables in Bayesian networks (e.g. (Kozlov & Koller, 1997), (Monti & Cooper, 1998a)). The simple discretization algorithm discussed here and currently implemented for our experiments is certainly not state-of-the-art.

4.2 Datasets and results

We tested the previously described algorithms on two different datasets taken from real scientific experiments. The "Bio" dataset contains data from a high-throughput biological cell assay. There are 12,671 records and 31 variables. 26 of the variables are continuous; the other five are discrete. Each discrete variable can take on either two or three different possible values.

The "Astro" dataset contains data taken from the Sloan Digital Sky Survey, an extensive astronomical survey currently in progress. This dataset contains 111,456 records and 68 variables. 65 of the variables are continuous; the other three are discrete, with arities ranging from three to 81.

Two minor adjustments are made to each of the original datasets before handing them to any of our learning algorithms. First, all continuous variables are scaled so that all values lie within $[0, 1]$. This helps put the log-

	Bio	Astro
Independent Mixtures	33300 +/- 500	2746000 +/- 5000
Single-Gaussian Mixtures	65700 +/- 200	2436000 +/- 5000
Pseudo-Discrete	59100 +/- 100	3010000 +/- 1000
Tree	74600 +/- 300	3280000 +/- 8000
Mix-Net	80900 +/- 300	3329000 +/- 5000

Figure 3: Mean log-likelihoods (and the standard deviations of the means) of test sets in a 10-fold cross-validation.

likelihoods we report in context, and possibly helps prevent problems with limited machine floating-point representation. Second, the value of each continuous value in the dataset is randomly perturbed by adding to it a value uniformly selected from [-.0005, .0005]. This noise is added to eliminate any deterministic relationships or delta functions in the data. The log-likelihood of a continuous dataset exhibiting even a single deterministic relationship between two variables is infinite when given the correct model; in such a situation, it is not clear how meaningful log-likelihood comparisons between competing learning algorithms would be. We add uniform noise rather than Gaussian noise in order to prevent the introduction of a bias that favors Gaussian mixtures.

For each dataset and each algorithm, we performed ten-fold cross-validation, and recorded the log-likelihoods of the test sets given the resulting models. Figure 3 shows the mean log-likelihoods of the test sets according to models generated by our five network-learning algorithms, as well as the standard deviation of the means. (Note that the log-likelihoods are positive since most of the variables are continuous and bounded within $[0, 1]$, which implies that the models usually assign probability densities greater than one to regions of the space containing most of the datapoints. The probability distributions modeled by the networks are properly normalized, however.)

On the Bio dataset, our primary mix-net learner achieved significantly higher log-likelihood scores than the other four model learners. The fact that it significantly outperformed the independent mixture algorithm and the tree-learning algorithm indicates that it is effectively utilizing relationships between variables, and that it includes useful relationships more complex than mere pairwise dependencies. The fact that its networks outperformed the pseudo-discrete networks and the single-Gaussian networks indicates that the Gaussian mixture models used for the network nodes' parameterizations helped the network achieve much better prediction than possible with simpler parameterizations. Our primary mix-net learning algorithm took about an hour and a half of CPU time on a 400 MHz Pentium II to generate its model for each of the ten cross-validation splits for this dataset.

The mix-net learner similarly outperformed the other algorithms on the Astro dataset. The algorithm took about three hours of CPU time to generate its model for each of the cross-validation splits for this dataset.

As additional tests of the mix-nets' robustness, we constructed two synthetic datasets from the Bio dataset. For the first synthetic dataset, all real values in the original dataset were discretized in a manner identical to the manner in which the pseudo-discrete networks discretized them, with 16 buckets per variable. (Out of the many different numbers of buckets we tried with the pseudo-discrete networks, 16 was the number that worked best on the Bio dataset.) Each discretized value was then translated back into a real value by sampling it uniformly from the corresponding bucket's range. The resulting synthetic dataset is similar in many respects to the original dataset, but its probability densities are now composed of piecewise constant axis-aligned hyperboxes — precisely the kind of distributions that the pseudo-discrete networks model. This synthetic dataset causes the pseudo-discrete network learning algorithm to learn a network identical to the network it learns from the original dataset; the pseudo-discrete network's test-set log-likelihood performance on this synthetic dataset is also identical to its test-set log-likelihood performance on the original data. However, we might expect mix-nets to perform much worse than the pseudo-discrete networks on this synthetic dataset, since the synthetic dataset's distributions may be much harder to represent with mixtures of Gaussians. As it turns out, the test-set performance of mix-nets on this synthetic dataset is worse than the performance of pseudo-discrete networks, but not dramatically so: the mix-net's average test-set log-likelihood on the synthetic drops down to 57600+/-200. This is significantly worse than the pseudo-discrete networks' log-likelihood, which stayed at 59100+/-100, but this difference in scores is not nearly as large as the difference on the original dataset, where the mix-nets clearly dominated.

For the second synthetic dataset, we generated 12,671 samples from the network learned by the Independent Mixtures algorithm during one of it cross-validation runs on the Bio dataset. The test-set log-likelihood of the models learned by the Independent Mixtures algorithm on this dataset is 32580 +/- 60, while our primary mix-net learning algorithm scored a slightly worse 31960 +/- 80. However, the networks learned by the mix-net learning algorithm did not actually model any spurious dependencies between variables. The networks learned by the Independent Mixtures algorithm were better only because the Independent Mixtures algorithm was given more time to learn each of its Gaussian mixtures.

5 Possible applications for Mix-Nets

5.1 Classification

So far, we have only discussed learning mix-nets in situations where our objective is to find a network that accurately models the distribution over the entire set of variables. What if our goal is to accurately predict the distribution of one discrete target variable given the values of all the other variables in the domain? A network learned by an algorithm optimized to accurately model the distribution over all the variables is not likely to fare well compared to networks learned by algorithms that take the specific prediction task at hand into consideration.

A simple, popular and effective type of classifier, the Naive Bayes classifier, assumes that the non-target variables are all independent of each other given the value of the target variable. This corresponds to using a Bayesian network in which there is an arc from the target variable to each non-target variable, but no arcs between the non-target variables. The non-target variables are usually assumed to be discrete; however, continuous variables have been handled in the past by using Gaussians or kernel density estimators for the conditional distributions of continuous variables (e.g., (John & Langley, 1995)).

A recently developed type of classifier, Tree Augmented Naive Bayes (TAN) (Friedman et al., 1997), augments the network structure of Naive Bayes with additional arcs between the non-target variables, where each non-target variable is conditioned on at most one other non-target variable. This classifier has been extended to handle continuous variables by representing each continuous variable in the network twice: once in a discretized form, and once in a simple conditional parametric form(Friedman et al., 1998).

Our greedy network-learning algorithm can easily be modified to learn mix-net classifiers similar in structure to TAN classifiers. By raising our algorithm's $MAXPARS$ parameter, it can also be used to learn classifiers with more complicated network structures. The network structure-learning algorithm would be very similar to the previously developed "Limited Dependence Bayesian Classifiers" algorithm (Sahami, 1996). The mix-nets' more flexible parameterizations would allow these classifiers to model complex interactions between continuous and discrete variables without requiring discretization of the continuous variables. Furthermore, since mix-nets can have discrete variables conditioned on continuous variables, the same network-learning algorithm can be used to learn networks for predicting the conditional probability density of a continuous variable given the values of all the other continuous and discrete variables in the domain. (Using these models may be somewhat computationally expensive, however, since the conditional distribution over the target variable is not obviously expressible in closed form and one may have to resort to numeric integration, for example.)

5.2 Anomaly detection

One obvious application for accurate joint probability models over large numbers of discrete and continuous variables is anomaly detection. The models can be used online to help detect the presence of abnormally low-probability situations. Alternatively, they can be used offline on the same datasets from which they are learned in order to rank the datapoints by their log-likelihoods. If the learned models are accurate, the datapoints assigned low log-likelihoods are probably unusual in reality as well. For example, we are currently exploring the use of networks learned from astronomical survey data to automatically select unusual astronomical objects for further inspection by human investigators.

5.3 Inference

While it is possible to perform exact inference in some kinds of networks modeling continuous values (e.g. (Driver & Morrell, 1995), (Alag, 1996)), exact inference in arbitrarily-structured mix-nets with continuous variables may not be possible. However, inference in these networks can be performed via stochastic sampling methods. If we are given a mixture table modeling $P(\vec{Y}, \vec{X})$ and specific values $\vec{x}$ for $\vec{X}$, it is possible to compute a conditional mixture table $P(\vec{Y}|\vec{x})$. This conditional mixture table can then be sampled straightforwardly. Thus, given a mix-net, we can easily employ likelihood weighting to generate a set of weighted datapoints representing a sample from any conditional distribution we desire. Whether likelihood weighting or other sampling methods will yield acceptably accurate inference results in a reasonable amount of time remains to be seen. Other approximate inference methods such as variational inference or discretization-based inference are also worth investigating.

5.4 Data compression

Many popular and powerful methods for data compression such as arithmetic coding (see, e.g., (Witten et al., 1987)) rely on explicit probabilistic models of the data they are compressing. Recent research ((Frey, 1998), (Davies & Moore, 1999)) has shown that using automatically learned Bayesian networks for these models can result in compression ratios dramatically better than those achievable by `gzip` or `bzip2`, while maintaining megabyte per second decoding speeds (Davies & Moore, 1999). Can this approach be extended to real-valued data?

In order to compress real-valued data, some loss of accuracy must usually be accepted — after the first few significant figures, real values typically become impossible to model as anything other than incompressible random noise. Thus, the question is: how much can the data be compressed if we are willing to accept some given average loss of accuracy in the reconstruction? Lossily compressing values using a Gaussian model is a well-studied problem (see, e.g. (Sayood, 1996)). How do we lossily compress values coming from a mixture of Gaussians? One obvious approach would be to encode each point as follows. First, we calculate the likelihood with which it came from each Gaussian in the mixture. Suppose the maximum likelihood Gaussian is G_m. We then encode in our compressed dataset the fact that the next datapoint is generated by G_m, and then encode the datapoint using G_m as our model distribution.

Unfortunately, this method of coding would be suboptimal when the Gaussians overlap. However, it is possible for an algorithm to effectively recover the bits wasted in this manner by using a clever "bits-back" method to encode some extra "side information" in the choice of which Gaussian gets used for the encoding (Frey, 1998). For example, if two Gaussians are almost equally likely to have generated the data, then we can effectively transmit about one bit's worth of information (about some other datapoint, for example) "for free" in our choice of which of the two Gaussians we use, rather than always simply picking the Gaussian with the slightly higher likelihood.

Automatically learned mix-nets may be an effective model class with which to compress large datasets containing both continuous and discrete values. We are currently investigating this possibility.

6 Conclusions and future research

We have described a practical method for learning Bayesian networks capable of modeling complex interactions between many continuous and discrete variables, and have provided experimental results showing that the method is both feasible and effective on scientific data with dozens of variables. The networks learned by this algorithm and related algorithms show considerable potential for many important applications. However, there are many ways in which our method can be improved upon. We now briefly discuss a few of the more obvious possibilities for improvement.

6.1 Variable grouping

The mixture tables in our network include a certain degree of redundancy, since the mixture table for each variable models the joint probability of that variable with its parents rather than just the conditional probability of that variable given its parents. For example, consider a completely connected network containing N continuous variables in which the joint probability of each variable and its parents is modeled as a single multidimensional Gaussian. As mentioned in Section 4.1.4, in this case our network will have $O(N^3)$ parameters, despite the fact that the overall distribution modeled by the network is actually just a single multidimensional Gaussian representable with $O(N^2)$ parameters. This wastes memory and computational time. Perhaps more importantly, the larger number of parameters may cause a network-learning algorithm to favor a simpler model with fewer parameters, even if there is enough data to justify the $O(N^2)$ parameters that would be used by a single multidimensional Gaussian. Naturally, it is possible to eliminate this redundancy in the special case of single-Gaussian mixtures by falling back to a representation in which each variable is modeled as a linear function of its parent variables plus Gaussian noise. Some other techniques have also been developed for computing nonredundant parameterizations of Bayesian networks with embedded joint distributions (Heckerman & Meek, 1997a). However, we know of none that are obviously practically applicable to the

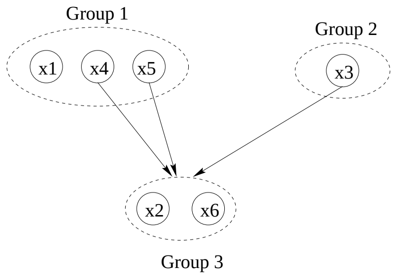

Figure 4: An example mix-net in which six variables are represented by a graph with three groups.

type of model employed in this paper.

One possible approach for reducing the amount of redundancy in the network is to allow each node to represent a *group* of variables. Each variable must be represented in exactly one group. A group G_i representing multiple variables $\vec{X}_i$ simply contains a mixture table modeling $P_i(\vec{X}_i, \vec{\Pi}_i)$, where $\vec{\Pi}_i$ is the set of G_i's parent variables. This mixture table can be marginalized to obtain $P_i(\vec{\Pi}_i)$ (and thus $P_i(\vec{X}_i|\vec{\Pi}_i)$) just as we did in the case where each node represented only one variable.

Suppose X_p is a parent variable of group G_i, and that X_p is represented in some other group G_j that also includes some other variable X_q. It is interesting to note that it is *not* necessary to include X_q in the distribution modeled at group G_i. For example, Figure 4 shows a mix-net with three nodes and six variables. Group 3 contains a mixture table modeling $P(X_2, X_3, X_4, X_5, X_6)$, but X_1 is not included in this model even though the two other variables in Group 1 are included. This example network decomposes the probability distribution over $X_1, \ldots, X_6$ as follows:

$$P(X_1, \ldots, X_6) = P_1(X_1, X_4, X_5)P_2(X_3)\frac{P_3(X_2, X_3, X_4, X_5, X_6)}{P_3(X_3, X_4, X_5)}$$

where P_1, P_2, and P_3 are computed from three different mixture tables that are not necessarily consistent with each other.

Grouping variables together in such a manner allows us to reduce the number of parameters in the network. For example, a single multidimensional Gaussian over N variables can now be represented with $O(N^2)$ parameters by simply placing them all in a single group. One possible line of research would be to design mix-net learning algorithms that automatically group variables together in order to reduce the amount of redundancy.

6.2 Alternative structure-learners

So far we have only developed and experimented with variations of one particular network structure-learning algorithm. There is a wide variety of structure-learning algorithms for discrete Bayesian networks (see, e.g., (Cooper & Herskovits, 1992), (Lam & Bacchus, 1994), (Heckerman et al., 1995), and (Friedman et al., 1999)), many of which could be employed when learning mix-nets. The quicker and dirtier of these algorithms might be applicable directly to learning mix-net structures. The more time-consuming algorithms such as hillclimbing can be used to learn Bayesian networks on discretized versions of the datasets; the resulting networks may then be used as hints for which sets of dependencies might be worth trying in a mix-net. Such approaches have been previously been shown to work well on real datasets (Monti & Cooper, 1998b).

6.3 Alternative parameter-learners

While the accelerated EM algorithm we use to learn Gaussians mixtures is very fast for low-dimensional mixtures and comes up with fairly accurate models, its effectiveness decreases dramatically as the number of variables in the mixture increases. This is the primary reason we have not yet attempted to learn mixture networks with more than four variables per mixture. Further research is currently being conducted on alternate data structures and algorithms which with to accelerate EM in the hopes that they will scale more gracefully to higher dimensions (Moore, 2000). In the meantime, it would be trivial to allow the use of some high-dimensional single-Gaussian mixtures within mix-nets as we do for the "competing" algorithm described in Section 4.1.4.

Other methods for accelerating EM have also been developed in the past, some of which might be used in our Bayesian network-learning algorithm instead of or in addition to the accelerated EM algorithm employed in this paper. The EM algorithm can be viewed as maximizing a single function whose local maxima correspond to local maxima of the likelihood function; the E step increases this function by adjusting the datapoints' estimated class distributions, and the M step increases it by adjusting the model parameters. This view justifies many variants of EM that may provide faster convergence (Neal & Hinton, 1998).

Another approach to accelerating the EM algorithm for Gaussian mixture models is to take a single pass through the dataset while heuristically maintaining in memory a limited-size buffer of datapoints whose class memberships are independently uncertain, and a set of summary statistics for the other datapoints (Bradley et al., 1998). This method would not provide the same drastic speed improvements provided by our currently employed acceleration method if used on low-dimensional datasets that fit completely in memory. However, it may scale more gracefully to very large high-dimensional datasets. Exploiting this alternative acceleration method might allow us to learn mix-nets with more parents per variable. (This alternative acceleration method could also simply be used to learn a Gaussian mixture over the entire set of continuous variables. We suspect that simple Gaussian mixtures in very large-dimensional spaces will frequently not perform as well as factorized models such as the ones employed here. However, comparative experiments testing this hypothesis on real datasets would be useful.)

Our current method of handling discrete variables does not deal very well with discrete variables that can take on many possible values, or with combinations involving many discrete variables. Better methods of dealing with these situations are also grounds for further research. One possibility would be to use mixture models in which the hidden class variable determining which Gaussian each datapoint's continuous values come from also determines distributions over the datapoint's discrete values, where each discrete value is assumed to be conditionally independent of the others given the class variable. Such an approach has been used previously in AutoClass (Cheeseman & Stutz, 1996). The EM acceleration algorithm exploited in this paper would have to be generalized to handle this class of models, however. Another possibility would be to use decision trees over the discrete variables rather than full lookup tables, a technique previously explored for Bayesian networks over discrete domains (Friedman & Goldszmidt, 1996).

The Gaussian mixture learning algorithm we currently employ attempts to find a mixture maximizing the joint likelihood of all the variables in the mixture rather than a conditional likelihood. Since the mixtures are actually used to compute conditional probabilities, some of their representational power may be used inefficiently. The EM algorithm has recently been generalized to learn joint distributions specifically optimized for being used conditionally (Jebara & Pentland, 1999). If this modified EM algorithm can be accelerated in a manner similar to our current accelerated EM algorithm, it may result in significantly more accurate networks.

Finally, further comparisons with alternative methods for modeling distributions over continuous variables (such as the Gaussian kernel functions used in (Hofmann & Tresp, 1995)) are warranted.

Acknowledgements

The authors would like to thank the anonymous reviewers of a previous version of this paper for their comments.

References

Alag, S. (1996). Inference Using Message Propogation and Topology Transformation in Vector Gaussian Continuous Networks. *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI96).*

Bradley, P. S., Fayyad, U., & Reina, C. A. (1998). *Scaling EM (Expectation-Maximization) Clustering to Large Databases* (Technical Report MSR-TR-98-35). Microsoft Research, Redmond, WA.

Cheeseman, P., & Stutz, J. (1996). Bayesian classification (AutoClass): Theory and results. *Advances in Knowledge Discovery and Data Mining.* MIT Press.

Chickering, D. (1996). Learning Bayesian networks is NP-complete. *Learning from Data: Artificial Intelligence and Statistics V* (pp. 121–130). Springer-Verlag.

Chow, C. K., & Liu, C. N. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, *IT-14*, 462–467.

Cooper, G. F., & Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, *9*, 309–347.

Davies, S., & Moore, A. (1999). Bayesian Networks for Lossless Dataset Compression. *Conference on Knowledge Discovery in Databases (KDD-99).*

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, *B 39*, 1–39.

Driver, E., & Morrell, D. (1995). Implementation of Continous Bayesian Networks Using Sums of Weighted Gaussians. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI95).*

Duda, R., & Hart, P. (1973). *Pattern Classification and Scene Analysis.* John Wiley & Sons.

Frey, B. J. (1998). *Graphical Models for Machine Learning and Digital Communication.* MIT Press.

Friedman, N., Geiger, D., & Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, *29*, 131–163.

Friedman, N., Godszmidt, M., & Lee, T. J. (1998). Bayesian Network Classification with Continuous Attributes: Getting the Best of Both Discretization and Parametric Fitting. *Proceedings of the Fifteenth International Conference on Machine Learning (ICML).*

Friedman, N., & Goldszmidt, M. (1996). Learning Bayesian Networks with Local Structure. *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI96).*

Friedman, N., Nachman, I., & Peér, D. (1999). Learning Bayesian Network Structures from Massive Datasets: The Sparse Candidate Algorithm. *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI99)* (pp. 206–215).

Heckerman, D., Geiger, D., & Chickering, D. M. (1995). Learning Bayesian networks: the combination of knowledge and statistical data. *Machine Learning*, *20*, 197–243.

Heckerman, D., & Meek, C. (1997a). *Embedded Bayesian network classifiers* (Technical Report MSR-TR-97-06). Microsoft Research, Redmond, WA.

Heckerman, D., & Meek, C. (1997b). Models and selection criteria for regression and classification. *Proceedings of Thirteenth Conference of Uncertainty in AI (UAI97)* (pp. 223–228). Providence, RI: Morgan Kaufmann.

Hofmann, R., & Tresp, V. (1995). Discovering Structure in Continuous Variables Using Bayesian Networks. *Advances in Neural Information Processing Systems 8.* MIT Press.

Jebara, T., & Pentland, A. (1999). The Generalized CEM Algorithm. *Advances in Neural Information Processing Systems 12*. MIT Press.

John, G., & Langley, P. (1995). Estimating Continuous Distributions in Bayesian Classifiers. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI95)*.

Kozlov, A., & Koller, D. (1997). Nonuniform dynamic discretization in hybrid networks. *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI97)*.

Lam, W., & Bacchus, F. (1994). Learning Bayesian belief networks: an approach based on the MDL principle. *Computational Intelligence*, *10*, 269–293.

Monti, S., & Cooper, G. F. (1998a). A Multivariate Discretization Method for Learning Bayesian Networks from Mixed Data. *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI98)*.

Monti, S., & Cooper, G. F. (1998b). Learning Hybrid Bayesian Networks from Data. *Learning in Graphical Models*. Kluwer Academic Publishers.

Moore, A. W. (1999). Very Fast EM-based Mixture Model Clustering using Multiresolution kd-trees. *Advances in Neural Processing Systems 12*. MIT Press.

Moore, A. W. (2000). The Anchors Hierarchy: Using the triangle inequality to survive high dimensional data. To appear in UAI-2000.

Moore, A. W., Schneider, J., & Deng, K. (1997). Efficient Locally Weighted Polynomial Regression Predictions. *Proceedings of the 1997 International Machine Learning Conference*. Morgan Kaufmann.

Neal, R. M., & Hinton, G. E. (1998). A view of the EM algorithm that justifies incremental, sparse, and other variants. *Learning in Graphical Models*. Kluwer Academic Publishers.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan-Kaufmann.

Sahami, M. (1996). Learning Limited Dependence Bayesian Classifiers. *KDD-96: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (pp. 335–338). AAAI Press.

Sand, P., & Moore, A. W. (2000). Fast Structure Search for Gaussian Mixture Models. Submitted to Knowledge Discovery and Data Mining 2000.

Sayood, K. (1996). *Introduction to Data Compression.* Morgan Kaufmann.

Schwarz, G. (1978). Estimating the dimension of a model. *Annals of Statistics*, *6*, 461–464.

Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the Association for Computing Machinery*, *30*, 520–540.