Fast Factored Density Estimation and Compression with Bayesian Networks

Scott Davies

May 2002

CMU-CS-02-138

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Andrew Moore, Chair
Christos Faloutsos
John Lafferty
Tom Mitchell
Nir Friedman, Hebrew University

This research was supported by NSF grants DMS-9873442 and ACI-0121671.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Carnegie Mellon University or the National Science Foundation.

Keywords: Machine learning, density estimation, Bayesian networks, graphical models, Gaussian mixture models, compression, interpolating density trees, conditional density estimation

To my family — especially my father, Donald.

Abstract

Many important data analysis tasks can be addressed by formulating them as probability estimation problems. For example, a popular general approach to automatic classification problems is to learn a probabilistic model of each class from data in which the classes are known, and then use Bayes's rule with these models to predict the correct classes of other data for which they are not known. Anomaly detection and scientific discovery tasks can often be addressed by learning probability models over possible events and then looking for events to which these models assign low probabilities. Many data compression algorithms such as Huffman coding and arithmetic coding rely on probabilistic models of the data stream in order achieve high compression rates.

In this thesis we examine several aspects of probability estimation algorithms. In particular, we focus on the automatic learning and use of probability models based on Bayesian networks, a convenient formalism in which the probability estimation task is split into many simpler subtasks. We also emphasize computational efficiency. First, we provide Bayesian network-based algorithms for losslessly compressing large discrete datasets. We show that these algorithms can produce compression ratios dramatically higher than those achieved by popular compression programs such as `gzip` or `bzip2`, yet still maintain megabyte-per-second decoding speeds on well-aged conventional PCs. Next, we provide algorithms for quickly learning Bayesian network-based probability models over domains with both discrete and continuous variables. We show how recently developed methods for quickly learning Gaussian mixture models from data [Moo99] can be used to learn Bayesian networks modeling complex nonlinear relationships over dozens of variables from thousands of datapoints in a practical amount of time. Finally we explore a large space of tree-based density learning algorithms, and show that they can be used to quickly learn Bayesian networks that can provide accurate density estimates and that are fast to evaluate.

Acknowledgements

I'd like to thank my advisor, Andrew Moore, for the guidance and encouragement he consistently provided throughout my stay at Carnegie Mellon — his insight, wit, and kindness greatly increased the quality of my research, and of my time at CMU in general. I'd also like to thank my thesis committee members — Andrew, Nir Friedman, John Lafferty, Tom Mitchell, and Christos Faloutsos — for providing valuable comments and feedback on this thesis. Much lively and useful discussion was also provided by various members of Andrew's research group (the "Auton Lab"), for which I am grateful. Many thanks are also due to Shumeet Baluja for our research collaborations during my earlier years at CMU; while these collaborations were performed before the material in this thesis were developed, they helped provide the initial interest and experience in probabilistic modeling that lead indirectly to this thesis.

I am deeply indebted to many friends at Carnegie Mellon for helping to make my time there enjoyable. In particular, I'd like to thank David Rochberg, Andrew Willmott, Robert O' Callahan, Ted Wong, Herbie Lee, and David Maltz — my housemates throughout various parts of my stay — for all the interesting conversation, home-cooked meals, and moral support. Periodic contact with old California friends, particularly Brad Williams and Carrie King, also helped ward off the occasional Pittsburgh gloom.

Finally, I'd like to thank my family for all their love and support over the years: my mother Henriette; my brother Chris; my sister Juliette; my stepfamily Rae, Tania, and Vanessa; and especially my father Donald, whose encouragement has been instrumental in forming my technical interests and a major contributing factor to my well-being.

Contents

Chapter 1

Introduction

1.1 Background: Bayesian networks

Suppose we have a domain consisting of a set of N variables $\vec{X} = (X_1, X_2, \ldots X_N)$. For the moment, assume that each variable X_i is discrete — that is, that it can take on some finite possible set of values. (A binary variable that can only take on the values "true" or "false" is an example of such a variable.) Now suppose we wish to model the statistical relationships between these variables with a probability distribution $P(X_1, X_2, \ldots, X_N)$ that allows us to calculate the probability that any given event j will assign the values $\vec{x^j} = (x_1^j, x_2^j, \ldots, x_N^j)$ to $\vec{X}$. What sort of models might we employ?

The most general possible model to use in this situation would be a lookup table containing one probability for every possible combination of values that could be assigned to $\vec{X}$. However, when there are many variables, this table can be extremely large — for example, if the domain consists of 30 binary variables, then this table would contain 2^{30} (over a billion) probabilities. This can be impractical for multiple reasons. First, even if the domain is known well enough that such a table could be specified with perfect accuracy, there may be situations in which we wish to be able to answer questions such as "What is the probability that X_2 is false and X_5 is true?" without specifying the values of all the other variables. To answer such questions with a lookup table would require summing over all the table entries consistent with the specified variable values, and the number of such entries grows exponentially with the number of variables in the domain. Second, we will often need to *learn* the model

$P(\vec{X})$ from a finite set of previously observed events. In such scenarios, the number of events we need to observe in order to acquire accurate estimates for the probabilities in the lookup table also grows exponentially with the number of variables.

Bayesian networks (otherwise known as belief networks) are a popular method for representing joint probability distributions over many variables. (See, e.g., [Pea88].) 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 factor 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 $P(X_i|\vec{\Pi_i})$ is the conditional probability distribution of X_i given $\vec{\Pi_i}$.

For example, Figure 1.1 shows the structure of a Bayesian network for a somewhat facetious medical domain with six binary variables. The network represents the following factorization of the joint distribution:

$$P(V, C, M, R, H, S) = P(V) \cdot P(C) \cdot P(M \mid V) \cdot P(R \mid C, M) \cdot P(H \mid C, M) \cdot P(S|M)$$

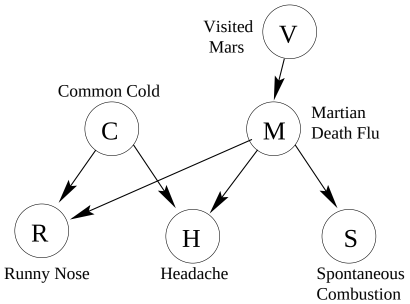


Figure 1.1: An example Bayesian network structure.

In addition to its graph structure, a Bayesian network also needs a set of tables specifying how each variable's probability distribution depends on the values of its parent variables in the graph. For example, for R, we might have the following table:

C	M	$P(R=0 \mid C,M)$	$P(R=1 \mid C,M)$
0	0	0.95	0.05
0	1	0.50	0.50
1	0	0.10	0.90
1	1	0.02	0.98

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.

Any joint probability distribution $P(\vec{X})$ can be represented with a Bayesian network. In the case where no independencies between variables exist, the joint distribution can be modelled with a fully connected Bayesian network in which each variable has all previous variables as its parents, where some arbitrary ordering of the variables is used to determine precedence. For example, if this ordering is $X_1, \ldots, X_N$, then the fully connected Bayesian network with respect to this ordering corresponds to the equation

$$P(\vec{X}) = \prod_{i=1}^{N} P(X_i|X_1, \ldots, X_{i-1}).$$

The total number of independent parameters required for such a network would be identical to the number of independent parameters in the lookup-table representation of $P(\vec{X})$ — for example, $2^N - 1$ in the case of N binary variables. However, when independencies exist between variables and the Bayesian network therefore has fewer parents per variable, the Bayesian network requires many fewer parameters to specify the joint distribution. For example, the Bayesian network in Figure 1.1 requires $1+1+2+4+4+2 = 14$ independent parameters, as opposed to the 63 that would be required with a fully connected network. Thus, a sparsely connected Bayesian network structure essentially provides a method for breaking the problem of estimating a joint distribution $P(\vec{X})$ into a set of conditional probability estimation problems $P(X_i|\vec{\Pi}_i)$, each of which involves only a relatively small number of variables. When we are attempting to learn $P(\vec{X})$ from a finite set of datapoints, the fact that these conditional distributions require many fewer total parameters means that the joint distribution can be learned more accurately using the Bayesian network representation than with the lookup-table respresentation. This can be true even when the Bayesian network structure effectively assumes independencies that are not actually present in the domain, as long as the most important dependencies *are* modelled.

There are other ways in which joint probability distributions can be split into products of factors each of which involves only a few variables. For example, Markov random fields (see [KS80] for a tutorial), also known as Markov networks (e.g. [Pea88]), are *undirected* graphs whose structures specify factorizations of the form

$$P(\vec{X}) = \alpha \prod_{c=1}^{C} \psi_c(\vec{S_c}),$$

where $c \in \{1, 2, \ldots, C\}$ denotes a particular clique in the undirected network, $\vec{S_c} \subset \vec{X}$ denotes the set of variables associated with the set of vertices in clique c, $\psi_c(\vec{S_c})$ denotes a function over the variables in $\vec{S_c}$, and α is constant guaranteeing that the probability distribution is normalized to 1. Markov networks and Bayesian networks are both special cases of chain graphs (see e.g. [Lau96]) in which directed arcs connect various subgraphs, each of which is internally connected with undirected arcs; the directed graph over these undirected components must be acyclic. There are certain advantages to using graphical models that allow undirected arcs, such as Markov networks, but there are notable disadvantages as well. In general, if we must learn the appropriate clique functions $\psi_c(\vec{S_c})$ from data, then computing the appropriate normalization constant α can become computationally intractable. One major exception is when the cliques are of reasonably small size and the model is *decomposable*; in Markov networks, decomposability corresponds to the undirected graph being *triangulated* (see e.g. [Ber73]). However, as it turns out, all Markov networks that are decomposable can also be modelled as Bayesian networks, although not all Bayesian networks correspond to decomposable models. Decomposable models with reasonably small clique sizes can be used to efficiently perform arbitrary probability *inference* exactly — that is, such joint models $P(\vec{X})$ can be used to efficiently compute conditional probabilities $P(\vec{H}|\vec{E})$ for arbitrary sets of variables $\vec{H} \subset \vec{X}$ and $\vec{E} \subset \vec{X}$. However, throughout this thesis we will focus primarily on applications for which arbitrary inference is not necessary, in which case our models need not be decomposable. Finally, there are situations in which Bayesian networks lend themselves more naturally to certain computational operations such as compression. Therefore we will restrict our attention in this thesis to Bayesian networks.

1.2 Thesis Overview

In Chapter 2, we provide Bayesian network-based algorithms for losslessly compressing large discrete datasets. First, we examine the use of Bayesian networks in conjunction with arithmetic coding. We use ordinary Bayesian networks to compress datapoints on an individual basis, and dynamic Bayesian networks to compress sequences of datapoints in which adjacent datapoints are highly correlated. We also examine modified Bayesian networks in which variables are automatically grouped together in order to improve the compression rates achievable with Huffman coding, which is significantly more computationally efficient than arithmetic coding. We show that these algorithms can produce compression ratios dramatically higher than are achieved by popular compression programs such as `gzip` or `bzip2` while maintaining megabyte-per-second decoding speeds on well-aged conventional PCs.

The rest of the thesis concentrates on practical learning algorithms for Bayesian networks that model both discrete and continuous variables. In Chapter 3, we show how recently developed algorithms for quickly learning accurate low-dimensional Gaussian mixture models from data [Moo99] can be used to learn joint distributions over dozens of continuous and discrete variables. We do so by using automatically learned Bayesian network structures to combine mixtures learned from different subsets of variables and datapoints. Finally, in Chapter 4, we explore and develop a large space of tree-based models for conditional density estimation, and algorithms with which to learn them. As in Chapter 3, these density estimators are then used in automatically learned Bayesian networks to model joint distributions over many continuous and discrete variables. While the models in Chapters 3 and 4 have not yet been applied to compression problems, they were designed partially with that potential application in mind. In particular, the models developed in Chapter 4 are quite fast to learn and evaluate, and have other properties that make them appealing for compression.

Chapter 2

Bayesian Networks for Discrete Dataset Compression

The recent explosion in research on probabilistic data mining algorithms such as Bayesian networks has been focused primarily on their use in diagnostics, prediction and efficient inference. In this chapter, we examine the use of Bayesian networks for a different purpose: lossless compression of large datasets. We present methods for automatically learning Bayesian networks and dynamic Bayesian networks to use with arithmetic coding, as well as modified Bayesian networks to use with Huffman coding. These algorithms often achieve significantly better compression ratios than achieved with popular compression algorithms such as those used by `gzip` and `bzip2`.

2.1 Background: Compression Techniques

In this section we provide a very brief introduction to common compression techniques. For more comprehensive descriptions, see an introductory textbook on compression (e.g. [Say96], [WMB99]).

Dictionary Techniques

Perhaps the most commonly used class of compression algorithms is the set of "dictionary techniques" used in general-purpose compression programs such as `gzip`. Dictionary-based algorithms maintain dictionaries containing sequences of source symbols. Whenever the source contains a symbol sequence that appears in the dictionary, that sequence's position in the dictionary is encoded rather than the individual symbols themselves. For example, the LZ77 algorithm [ZL77] and its derivatives maintain a sliding window of the source symbols encoded in the immediate past; when a new source symbol sequence is encountered that matches a sequence contained in the window, the sequence's position in the window and its length are encoded. The LZ78 algorithm [ZL78] and its derivatives such as LZW [Wel84] maintain tables of previously seen sequences, and encode source sequences via their positions in the table.

These algorithms can be shown to achieve asympotically optimal compression rates [Ziv78]; however, they may require the use of unmanageably large dictionaries in order to do so.

Arithmetic Coding

Arithmetic coding (developed by Rissanen [Ris76] and Pasco [Pas76]; see Witten, Neal, and Cleary [WNC87] for a tutorial) allows sequences of symbols to be encoded nearly optimally without requiring the enumeration of all possible source code sequences of length k. Arithmetic coding effectively maps an entire sequence of source symbols to a real number between 0 and 1. The arithmetic encoder begins with a range $R = [0, 1)$. As each symbol in the source sequence is encoded, the current range R is subdivided into a partitions, where a is the number of possible values the symbol could have taken on; the size of each of these partitions is proportional to the probability of symbol taking on the corresponding value. The current range is then restricted to the partition corresponding to the source symbol being encoded.

For example, suppose we have a source sequence where each source symbol can take on one of three values, s_1, s_2, and s_3, with probabilities .1, .6, and .3, respectively. The range $R = [0, 1)$ is initially split into the subranges $[0, .1), [.1, .7)$, and $[.7, 1)$. If the first symbol to encode is s_2, then the encoder restricts R to $[.1, .7)$. This range is then further subdivided into the three subranges $[.1, .16), [.16, .52)$, and $[.52, .7)$. If the second symbol to encode is s_3, then R is set to $[.52, 7)$, and so forth. (See Figure 2.1 for a diagram of this process.) When all the symbols in the source sequence have been coded, the encoder outputs the binary representation of a number within its current range R to a sufficient precision to disambiguate it from all numbers outside of R. If the sequence encoded is $s^1, \ldots, s^k$, then the number of bits required for this disambiguation is

$$-\log P(s^1, \ldots, s^k),$$

plus one or a few more depending on the particulars of the implementation. Taking the expectation of this quantity over all possible source symbol sequences tells us that the expected number of bits required to encode k symbols is approximately

$$\sum_{s^1,\ldots,s^k} -P(s^1,\ldots,s^k)\log P(s^1,\ldots,s^k) = H(s^1,\ldots,s^k)$$

where $H(s^1,\ldots,s^k)$ is the *entropy* of $P(s^1,\ldots,s^k)$. Since this is the information-theoretic minimum average number of bits required to encode a source sequence with distribution $P(s^1,\ldots,s^k)$ (see, e.g., [CT91]), arithmetic encoding approaches optimality in the limit as k approaches infinity.

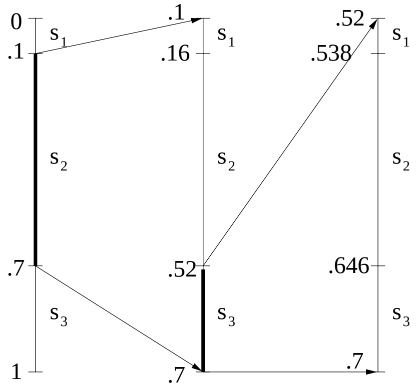


Figure 2.1: Restriction of the range R during arithmetic encoding while encoding the symbol s_2 followed by s_3

As described above, arithmetic coding appears to require the use of arbitrary-precision arithmetic operations in order to manipulate the current range R. However, it is possible to use limited-precision integer arithmetic to approximate "perfect" arithmetic coding. This typically results in a negligible increase in the number of bits required for the encoding, and drastically reduces the computational expense of compression and decompression.

One important advantage of arithmetic coding is that it may be used in conjunction with any algorithm for modelling probability distributions over source symbols, although some models can be used more practically than others. Models that more accurately predict the probability distribution of the next symbol as a function of previously coded symbols achieve better compression ratios. The problem of compressing a dataset thus reduces to the problem learning an accurate probabilistic model of that data.

Huffman Coding

Given a small discrete set of source symbols and their associated probabilities, a simple greedy algorithm developed by David Huffman [Huf51] can be used to find an optimal code with which to encode source symbols on an individual basis. The algorithm is based on the insight that there must exist an optimal code such that the two least likely source symbols are encoded with bit patterns that are of equal length and differ only in their last bit. The algorithm grows a binary tree in a "bottom-up" fashion by consecutively merging pairs of subtrees corresponding to groups of source symbols; at any point the two groups with the lowest total probability are merged. For example, consider the coding problem in Figure 2.2. Assume we have five possible source symbols with the probabilities shown in part (1) of the figure. The two symbols with the least probability are B and D, with probabilities of .05 and .10 respectively. These two symbols are merged into a group with a total probability of .15; this group corresponds to the depth-1 subtree at the bottom right of part (2) of the figure. Now, the two groups with the least total probability are the group for B and D (total probability .15) and the group for C (probability .20), so these two groups are merged; and so forth. The resulting sequence of group merges produces the tree shown in part (2) of the figure. The binary Huffman code assigned to a given source symbol is then determined by the sequence of left/right branch decisions in this tree required to reach the symbol, as shown in part (3) of the figure. The average number of bits required to encode a single source symbol with this Huffman code, assuming the symbol is drawn from a probability distribution according to part (1) of the figure, is

$$2 \cdot .32 + 3 \cdot .05 + 2 \cdot .20 + 3 \cdot .10 + 2 \cdot .33 = 2.15.$$

This is the lowest possible number of bits if each source symbol must be encoded independently with an integral number of bits.

Unfortunately, if the probability for one particular source symbol is very high, Huffman coding can be inefficient, as the code requires at least one bit for each source symbol encoded (unlike arithmetic coding). For example, if there are only two possible source symbols, each will require one bit to encode, even if one value is vastly more likely than the other. However, one can alleviate this problem by grouping sequences of source symbols together in blocks of k symbols and using Huffman coding on these blocks rather than the individual source symbols; as k increased, Huffman encoding will approach optimality, although care must taken to prevent unmanageably large codebooks from resulting. In Section 2.3.1, we will describe a modified Bayesian network learning algorithm that will automatically find good groups of variables to encode as blocks in order to achieve compression performance that is usually very

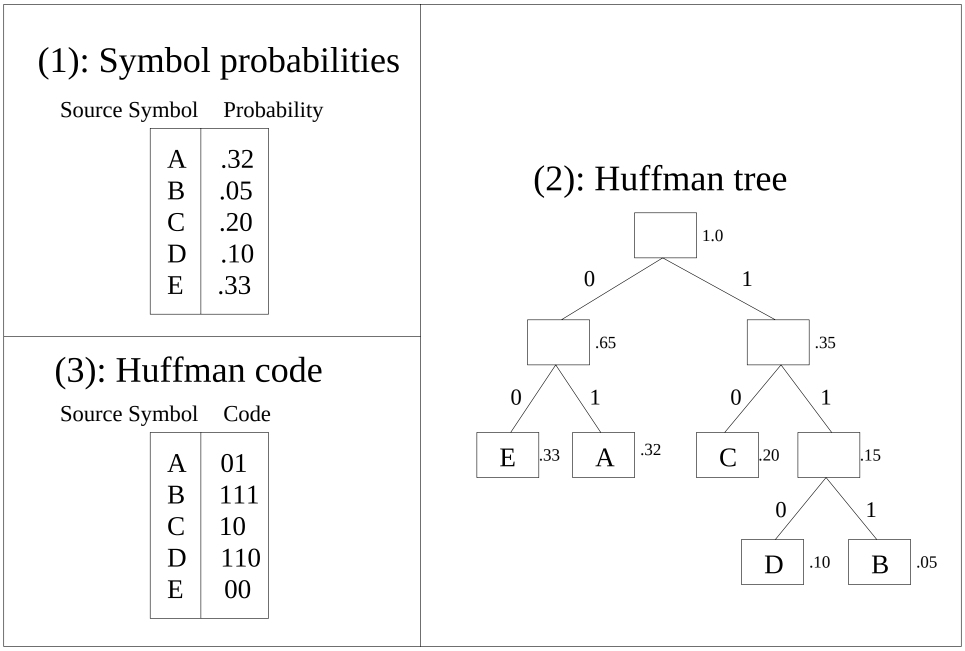


Figure 2.2: An example of Huffman coding.

close to that achieved by arithmetic coding.

2.2 Using Bayesian networks for data compression

Bayesian networks are straightforward to use with arithmetic coding. To encode a record j of the dataset with a Bayesian network B, one treats each of the variable values in j as an individual "source symbol". These values are passed to the arithmetic encoder in an order consistent with a topological sort of B's vertices. This way, the decoder will have already decoded the values of any given variable X_i's parent variables by the time it needs to decode the value of X_i, and can use the appropriate entry in X_i's probability table to determine the probability distribution of values for X_i.

Automatically-learned Bayesian networks have been used previously in conjunction with arithmetic encoding in recent research by Frey [Fre98]. In Frey's work, fixed network structures are employed in which each node has many parents; the probability of each node given its parents is paramaterized using logistic regression [MN83]. In order to capture complex nonlinear dependencies between variables, Frey uses networks with many hidden variables — that is, nodes that do not correspond to any observed values in the dataset being compressed. This creates several problems. First, finding the correct probabilities for the tables in networks with hidden variables is more difficult than it is in situations where all variables are always observed — one must resort to iterative procedures such as the Expectation Maximization algorithm [DLR77], or "EM", in order to obtain good estimates. Second, even once these parameters have been set and we wish to use the resulting network for compression or decompression, we need to be able to estimate the probability distribution of the hidden variables given the observed variables, or vice versa. When there are many unobserved variables, this problem is generally intractable.

Frey addresses these two problems by using the iterative "wake-sleep" algorithm [HDFN95] to adjust the parameters of a *Helmholtz machine* [DHNZ95]. A Helmholtz machine consists of a pair of Bayesian networks. The first of these networks, the *generative* network, has the observable variables conditioned on the hidden variables; the second, the *recognition* network, has the hidden variables conditioned on the observable variables. When encoding an instance, the recognition network is used to induce a distribution over the hidden variables; the values of these hidden variables are then chosen. The generative network is then used in conjunction with arithmetic encoding to encode both the hidden and observed variables. By itself, this method required too many bits to code each instance to be useful on the datasets used in Frey's experiments; the resulting compression rate was significantly worse than `gzip`'s. However, it is possible for the encoder to convey "side information" through its particular choice of hidden variable values using a technique called *bits-back coding* [HZ94]. By encoding part of the dataset through this "side information" channel, it is possible to obtain compression rates significantly better than `gzip`'s.

Unfortunately, Frey's approach has several notable disadvantages. Its computational costs are prohibitive in situations where fast decompression is desired. Even if the Helmholtz machines' parameters are not adjusted "on the fly" during both decompression and compression as they are in Frey's work, many expensive floating-point mathematical operations must be performed for every node in the networks employed. Furthermore, the bits-back coding scheme required to achieve adequate compression ratios is inherently block-oriented, which makes it unsuitable for situations in which random access to dataset items is required.

We take a different approach to using Bayesian networks for compression. Rather than using densely connected networks with fixed structures and hidden variables, we employ sparsely connected Bayesian networks with no hidden variables. Once a suitable such network has been found for the data, it can perform compression and decompression quickly with no floating-point operations. We now turn our attention to the task of automatically learning such networks.

2.2.1 Learning Bayesian Networks from Complete Data

Given a dataset D, we would like to automatically learn a Bayesian network B that accurately models the probability distributions in D with a small number of network parameters (i.e., entries in the probability tables associated with the variables). If there are no missing values or hidden variables in D — that is, if the data is "complete" — and if we are given B's structure, then it is trivial to fill in B's probability tables to maximize the log-likelihood of the data: namely, we simply use the empirical distributions appearing in D. However, even with complete data, the problem of finding the best network structure is NP-hard [Chi96]. Learning a Bayesian network is thus typically done by using a search procedure to find a network B that maximizes (or at least hopefully comes close to maximizing) a scoring function $C(B, D)$. A popular scoring function is the *Bayesian Information Criterion* (BIC) [Sch78],

$$C(B, D) = \log P(D \mid B) - |B| * 0.5 \log R$$

where $|B|$ is the number of independent parameters (probabilities) stored in the net and R is the number of records in the dataset. Maximizing BIC corresponds directly to minimizing the number of bits required to store both (1) the parameters of the network B to a reasonable level of precision and (2) an efficient encoding (such as arithmetic encoding) of D using the probability distribution entailed by B. Thus, the BIC is naturally suited for finding Bayesian networks that are good for compression. This "minimum description length" (or MDL) approach has also been used for learning Bayesian networks in cases where compression is not necessarily the primary objective [LB94].

For the experimental results in the next section, two algorithms for learning Bayesian networks were used. The first algorithm uses a form of stochastic hill-climbing over possible network structures using the Bayesian Information Criterion as its scoring function. AD-Trees [ML98] are used to speed up this search by decreasing the amount of time necessary to calculate the dataset statistics required for the search. (See [ML98] for details of the search algorithm.)

The second algorithm, which we designed for very large datasets, takes two sweeps through the dataset. In the first sweep, the algorithm collects the dataset statistics required to measure the increase in BIC score that would be achieved by adding any single arc to an empty Bayesian network structure. This process requires $O(N^2 * (R + a^2))$ time, where N is the number of variables, R is the number of records, and a is the maximum arity of the variables. Let $I(X_i, X_j)$ denote the increase in BIC score achieved by adding an arc from X_i to X_j. This is proportional to the *mutual information* between X_i and X_j (see e.g. [CT91]), minus a penalty term proportional to the number of added parameters, and is thus symmetric, i.e. $I(X_i, X_j) = I(X_j, X_i)$. We then greedily grow a network structure in which each node has at most c parents, where c is a user-defined parameter. This growth occurs without referring to the dataset; the greedy algorithm naively assumes, for example, that if X_j and X_k are the best candidate parents for X_i based on $I(X_i, X_j)$ and $I(X_i, X_k)$, then $\{X_j, X_k\}$ is the best parent set of size 2 for X_i. We omit the details here, but the algorithm is very similar to the greedy network-learning algorithm described later in Section 3.3, as well as to an algorithm previously developed to learn Bayesian networks for classification [Sah96]. In the special case where c is 1, this algorithm reduces to a penalized version of Chow and Liu's dependency-tree algorithm [CL68], and finds a network with the optimal BIC score out of all networks in which each variable has at most one parent. Once the network structure has been determined, a second sweep is then made over the dataset to fill in the probability tables of the resulting network; this takes $O(N * (R + a^c))$ time.

Since the algorithm in this section was originally developed, a more sensible and general approach called the Sparse Candidate Algorithm [FNP99] has been developed for quickly learning good network structures over discrete variables with few passes through the dataset. While the Sparse Candidate algorithm does not apply directly to situations in which the conditional distributions used in the network cannot be computed quickly from sets of sufficient statistics — such as the conditional distributions that will be used in Chapter 3 and Chapter 4 — it is directly applicable to the network-learning task at hand in this section, and would make a good replacement for the algorithm employed here.

2.2.2 Experimental Results

In this section, we examine the effectiveness of using automatically learned Bayesian networks in conjunction with arithmetic coding in order to perform compression.

Census dataset

Each record in this dataset corresponds to a person; variables represent such things as the person's sex, occupation, income, etc. The dataset consists of 142,521 records, each of which has twelve symbolic values. The number of possible values each variable can assume varies between two and twelve.

A verbose ASCII version of this dataset requires about 21 MB of disk space; a more frugal binary representation takes up 536 KB. `gzip`, a popular UNIX compression utility employing the LZ77 algorithm, reduces this dataset to 294 KB when used in its "best-compression" mode. `bzip2`, a compression utility using the Burrows-Wheeler block-sorting algorithm [BW94], can compress a version of the file down to 220 KB (also when in "best-compression" mode). As their inputs, `gzip` and `bzip2` are given the dataset as a binary file in which each variable value is encoded in its own byte; because these two programs are byte-oriented, this results in better compression than when the dataset is given to them in the more compact bit-oriented 536 KB representation. This byte-oriented representation is used in the other experiments in this chapter as well, since it also helps the compression ratios `gzip` and `bzip2` on those experiments.

In conjunction with the two Bayesian network learning algorithms discussed above, we use a limited-precision arithmetic coding library written by Carpinelli *et al.* [CMN+95] based on a paper by Moffat *et al.* [MNW95]. We modified the library so it could encode to and decode from RAM when desired rather than only to or from disk; all of the decoding speed results we will show later are based on decompressing from RAM. The network-learning algorithm employing stochastic hillclimbing compressed the census dataset to 169 KB. This includes the space required to encode the learned network structure and all the corresponding conditional probability tables. The algorithm employs one of four different encoding methods for each table, depending on the number of non-zero entries and their relative probabilities. (We omit the tedious details.) With c, the maximum number of parents per node, set to two, the two-pass greedy algorithm compressed it to 171 KB. For comparison, using arithmetic encoding in conjunction with an empty Bayesian network (that is, one in which no dependencies between variables are modelled) produces a 231 KB file.

Banking dataset

This dataset, a set of customer profiles from a Pennsylvania bank, consists of 6372 records, each of which contains 142 values. Real-valued variables were quantized into symbolic variables taking on 16 values; however, some of the naturally symbolic variables were very high-arity (up to about 100 possible values), and these values were not changed. This quantized dataset took up 416 KB in raw binary form. `gzip` compressed the dataset down to 345 KB; bzip2, to 273 KB.

Using a network learned by the greedy two-pass method in which each node had at most one parent, arithmetic encoding was able to represent the file in 166 KB. (With an empty Bayesian network, the file was compressed to 240 KB — significantly worse than with one parent per node, but still better than `gzip` or `bzip2`.)

EDSGC dataset

This dataset consists of 900,000 records with 27 variables; each record represents an galaxy from the Edinburgh/Durham Southern Galaxy Catalogue Survey (EDSGC) [HDCM89]. Variables include the galaxy's position, magnitude, geometry, and so forth. All variables were quantized to sixteen values. A raw binary file containing this quantized data requires 11.8 MB. `gzip` compresses the quantized dataset down to 6.9 MB; `bzip2` to 5.6 MB.

Since there were many records but a reasonably small number of variables, a random sample of 50,000 records was selected and used to learn a Bayesian network using stochastic hillclimbing; once the network was learned from this sample, a final pass through the entire dataset was used to fill in the network's probability tables. Allowing the stochastic hillclimbing to progress for 10,000 iterations resulted in a network that was able to compress the dataset to 4.8 MB. Using the greedy two-pass algorithm on the entire dataset to learn a network in which each node had at most three parents resulted in a 4.2 MB file. (For comparison, with an empty Bayesian network, arithmetic encoding produced a 9.0 MB file.)

Sloan dataset

This dataset is taken from a larger astronomical survey currently in progress. It contains approximately 3,080,000 records with 49 variables; all continuous variables are quantized to sixteen values. The raw binary form of the quantized data takes 53.1 MB. `gzip` compresses the data to 35.6 MB; `bzip2` to 27.9 MB. Using the greedy two-pass algorithm on the entire dataset to learn a network with at most three parents per variable results in a 23.9 MB file.

2.3 Data reordering and Dynamic Bayesian Networks

Once the algorithms described above learn a Bayesian network modeling a dataset, they use the network to compress each item in the dataset independently of all the others. Essentially, the algorithms are assuming that the items in the dataset are independently and identically distributed (or *i.i.d.*). In reality, datasets frequently violate this assumption. Probability distributions exhibited in real-life data may shift over time, either gradually or suddenly. Furthermore, it is quite often the case that the order in which items happen to appear in the dataset is irrelevant. (Hence the term "data*set*".) It may be possible to significantly improve compression performance in such cases by reordering the data. One possible approach would be to lexicographically sort one set of datapoints, and then encode the bits of other set of datapoints "for free" by using them to permute the previously sorted set before it is encoded. The decoder could then reconstruct the bits in the second set of datapoints after decoding the first set by reconstructing the permutation that must have been applied to its sorted version. While potentially interesting, such an approach would be complex to implement, and would not be able to exploit any preexisting dependencies between neighboring datapoints in the original dataset. We do not explore this avenue further in this thesis.

Another method is to use *adaptive* coding in which the probabilistic model of the data gradually shifts as data is processed. We will discuss this approach further in Section 2.5; for now, we simply note that adaptive coding has disadvantages that make it unsuitable for some applications. In particular, updating the probabilistic models during decompression may be undesirably time-consuming. Furthermore, in some situations we may wish to maintain coarse-granularity random access to the data — for example, we may wish to be able to decompress all datapoints stored in a specific disk block without having to decompress any others. Adaptive coding is difficult to apply effectively in such situations.

Another approach is to explicitly model correlations between consecutive datapoints. Even if such correlations are not present in the original dataset, they can be created by sorting the dataset. It may not be practical to completely sort very large datasets, particularly datasets too large to fit in main memory, merely for the purposes of increased compression performance. However, it may be possible to get some benefit from the tricks mentioned above with much less computational expense by instead only sorting within relatively small blocks of the dataset, or by radix sorting only on the values of a few variables.

In this section we examine the use of dynamic Bayesian networks [DK88] to represent dependencies between consecutive datapoints in order to increase compression performance. Dynamic Bayesian networks are Bayesian networks that represent how a system evolves from one time step to another. A dynamic Bayesian network consists of two Bayesian networks. The first Bayesian network, the *prior network*, specifies a distribution over the system's possible starting values. The second network, the *transition network*, specifies the distribution over the system's variables in the next time step given the values of the variables in the current time slice. For example, Figure 2.3 shows a dynamic Bayesian network for a system with four variables. Part (a) shows the prior network, and part (b) shows the transition network. The top four nodes X_1 through X_4 in the transition network correspond to the variables' values at some time t, while the bottom four nodes X'_1 through X'_4 correspond to the variables at some time $t+1$.

When faced with datasets that are not *i.i.d.* — either because there are trends in the data that change over time, or because a formerly *i.i.d.* dataset has been re-ordered to improve compression efficiency — one natural approach is to treat the dataset as a time series and to learn a dynamic Bayesian network that models this series. A greedy heuristic algorithm for learning dynamic Bayesian networks in a manner similar to the two-pass algorithm described in Section 2.2.1 was implemented. The algorithm learns a transition network in which each node representing a variable in datapoint $j+1$ is conditioned on at most c parent nodes. Each of these parent nodes can be either a node representing a variable in the most recent dataset item that

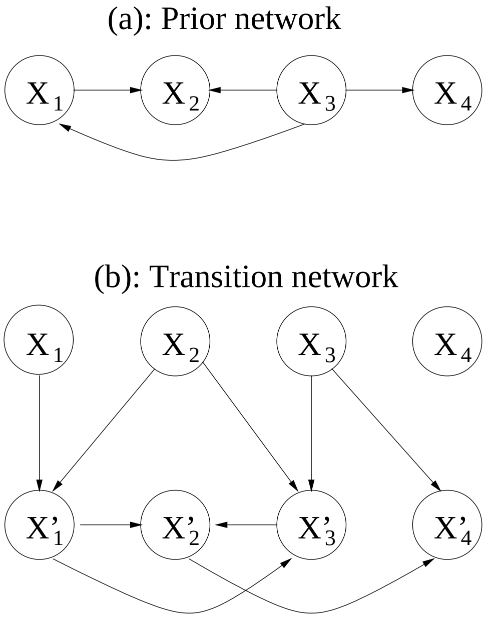


Figure 2.3: An example dynamic Bayesian network, consisting of a prior network and a transition network.

has been completely coded (that is datapoint j), or a node representing a previously coded variable in the same dataset item we are currently coding (datapoint $j + 1$). The algorithm does not bother learning any dependencies in the prior network, since they would be used only for coding the very first dataset item. Again, we omit the details; similar previous research exists on automatically learning dynamic Bayesian structures from data [FMR98].

We used this automatic Dynamic Bayesian network learning algorithm to compress versions of the previously mentioned datasets in which the records were either left in their initial positions or sorted lexicographically.

2.3.1 Experimental results

Census dataset

Learning a dynamic Bayesian network for this dataset in its natural ordering and then using it for compression did no better than the analagous algorithm that used non-dynamic Bayesian networks: with c set to a maximum of two parents per node, both algorithms produced a 171 KB file. It appears that the data is *i.i.d.*, or at least that a given dataset item has little influence over the very next dataset item. However, if the dataset is sorted and then a dynamic Bayesian network is learned on this sorted dataset instead, the resulting compressed file is only 18.6 KB in size. Since there are 142,521 items in the dataset, this works out to an average of *only 1.04 bits per item*, including the cost of encoding the network. As it turns out, there are only roughly 14,000 unique items in the dataset — most items in the dataset are duplicated many times. A special-purpose algorithm for dealing with exact duplicates or a delta-coding scheme might fare somewhat better in this case, but the dynamic Bayesian network technique appears to handle it quite well without any such special-casing. (For purposes of comparison, `gzip` was able to compress the sorted dataset down to 36.2 KB, while `bzip2` compressed it to 58.2 KB.)

Banking dataset

As in the census dataset, using an automatically learned dynamic Bayesian network on this dataset in its natural ordering did not improve compression performance over using the analagous non-dynamic network. Sorting the dataset and then modeling it with a dynamic Bayesian network performed only very slightly better, reducing the resulting file size from 166 KB to 163 KB. This dataset is much more "sparse" than the census dataset in that it has fewer items and many more variables; some of its variables are also very high-arity. As a result, sorting only created significant inter-item dependencies in the first few variables used for the sort, and did not make it any easier to model the others. (Sorting did not significantly improve `gzip`'s or `bzip2`'s performance in this case either.)

EDSGC dataset

Using an automatically learned dynamic Bayesian network on this dataset dramatically improved compression performance over its non-dynamic counterpart: 2.6 MB rather than 4.2 MB. Thus, this dataset is clearly not *i.i.d.* even its natural form. As it turns out, each datapoint includes variables that encode a position in the sky, and the dataset was largely ordered by these position variables, so the position values of adjacent dataset items are highly correlated.

Sorting this dataset using its natural variable ordering improves compression slightly (2.5 MB rather than 2.6 MB), and slightly improves the performance of `gzip`

(6.5 MB vs. 6.9 MB) and bzip2 (5.5 MB vs. 5.6 MB) as well.

Sloan dataset

As in the EDSGC dataset, using an automatically learned dynamic Bayesian network on the Sloan dataset significantly improved compression performance: 16.1 MB, as opposed to 23.9 MB for the non-dynamic Bayesian network. In this case, however, sorting the dataset lexicographically actually caused the dynamic Bayesian networks to perform slightly worse (17.1 MB), perhaps because the variables that were most strongly correlated between adjacent dataset items in the unsorted version were not the first variables used for the lexicographic sort. Sorting did slightly increase the performance of gzip (34.0 MB vs. 35.6 MB) and bzip2 (27.7 MB vs. 27.9 MB).

Summary of experiments with Bayesian network-based arithmetic coding

The results of this section are summarized in Table 2.1 (along with the results from Section 2.2.2 for comparison). Depending on which dataset is being compressed and whether this dataset has been sorted, compression based on using dynamic Bayesian networks in conjunction with arithmetic encoding was able to produce files that were 40-60% smaller than produced by gzip, and 20-60% smaller than produced by bzip2. Sorting the datasets sometimes increased compression performance — dramatically so in the case of the Census dataset.

2.4 Compression With Network-Based Huffman Coding

The algorithms described above have proven useful for long-term file compression tasks where decompression speed and random access requirements are not crucial. However, if we hope to use compression in more speed-critical applications, such as speeding up data analysis by compressing data into memory rather than leaving it on disk, we need fast decompression of random dataset items. Arithmetic coding is somewhat computationally expensive; furthermore, no random access is possible within a sequence of bits encoded with a single application of arithmetic coding, since there is no well-defined bit position where the encoding of one value ends and another

	Census	Banking	EDSGC	Sloan
# dataset items	142500	6370	900000	3.08 M
# variables	12	142	27	49
variable arity	2-12	2-100	2-16	2-16
Uncomp. binary	536 KB	416 KB	11.8 MB	53.1 MB
gzip	294 KB	345 KB	6.9 MB	35.6 MB
bzip2	220 KB	273 KB	5.6 MB	27.9 MB
Bayes Net	169 KB	166 KB	4.2 MB	23.9 MB
Dyn. Bayes Net	171 KB	166 KB	2.6 MB	16.1 MB
Sort + gzip	36 KB	343 KB	6.5 MB	34.0 MB
Sort + bzip2	58 KB	272 KB	5.5 MB	27.7 MB
Sort + Dyn. BN	19 KB	163 KB	2.5 MB	17.1 MB

Table 2.1: Compression with Bayesian networks and arithmetic coding: experimental results summary

begins. It is possible to separate records or variables into independently coded blocks, but since arithmetic coding requires an extra one or two bits at the end of each block, this causes arithmetic coding to lose some of its compresssion performance (although not too much).

In contrast, Huffman coding uses relatively inexpensive table lookups to perform encoding and decoding, and each coded value naturally has a well-defined start and end position in the resulting bitstream. This makes Huffman-based coding attractive for applications requiring fast decompression and/or random access. However, as mentioned previously, Huffman-based decoding provides poor compression performance when applied to probability distributions in which some values are very probable. It is possible to group variables together to overcome this problem, but then the tables required for encoding and decoding can become prohibitively large if too many variables are placed in one group. Additionally, if one variable is highly correlated with many other variables, it may be advantageous to have the value of that variable change the coding schemes associated with several variable groups, *without* that variable's value actually being coded in the compressed representations of all of the groups it influences.

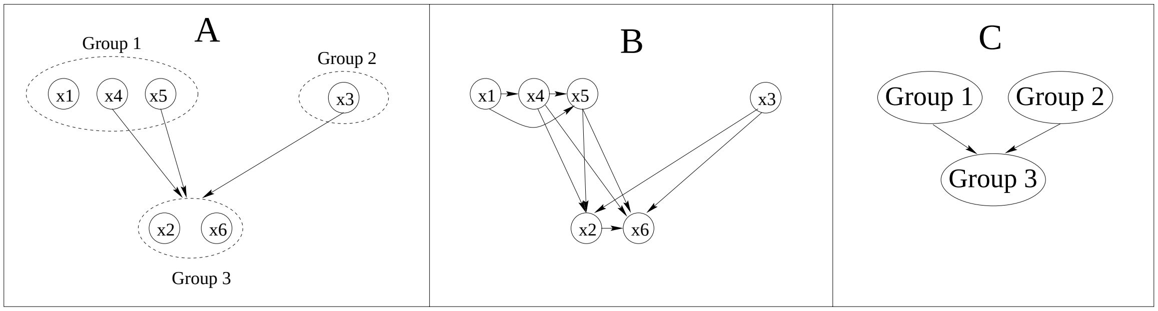


Figure 2.4: An example Huffman network (A), along with its corresponding variable-based (B) and group-based (C) Bayesian networks.

2.4.1 Huffman Networks

We address these issues by using a modified Bayesian network — referred to hereafter as a *Huffman network* for convenience — in which each node actually models a *group* of variables in the dataset rather than an individual variable. Each group of variables is Huffman coded as a single unit. For example, if a group contains three binary variables, then that group is Huffman coded as if it were a single variable taking on eight possible values; each of these eight values is assigned a probability equal to the joint probability of the corresponding combination of values for the original three binary variables.

In order to take into consideration dependencies between variables residing in different groups, we allow the probability distribution over the possible values for each group to be conditioned on the values of other variables. For example, in Figure 2.4A, six variables have been placed into three groups. The joint probability distribution of all the variables in Group 3 (namely, variables x_2 and x_6) is conditioned on the values of variables x_3, x_4, and x_5. This conditioning is represented in the graph by arcs from x_3, x_4, and x_5 to Group 3. Assuming all the variables are binary, this means that Group 3 requires eight Huffman tables — one for each possible combination of values to x_3, x_4 and x_5. Each of these tables then has four entries — one for each possible combination of x_2 and x_6. Note, however, that Group 3 is *not* conditioned on the value of x_1, despite the fact that x_1 is in the same group as x_4 and x_5. This added flexibility can help in certain situtations — for example, if x_2 and x_6 are independent of x_1 given x_4 and x_5, then conditioning Group 3 on the value of x_1 would double the number of Huffmann tables required by Group 3 without increasing Group 3's coding efficiency.

The Huffman network can be thought of as a Bayesian network over the original variables in which all variables in the same group are completely connected (e.g., Figure 2.4B). This representation tells us what dependencies between variables are being modeled by the coding scheme associated with the Huffman network. At the same time, the Huffman network can be thought of as a Bayesian network over the groups themselves (e.g., Figure 2.4C), where an arc exists from group G to group G' if and only if an arc exists from at least one variable in G to the group G' in the Huffman network. This view summarizes how the coding groups in the Huffman network are connected, thus telling us which groups of variables need to be decoded before other groups can be decoded.

We use a given Huffman network to perform compression as follows. First, we take one pass through the dataset to compute contingency tables for each of the groups in the network. The contingency table for a given group with a set of variables V and set of conditioning variables P counts how many times each possible combination of values for $V \bigcup P$ occurs in the dataset. These contingency tables are represented sparsely so that combinations that never actually occur in the dataset are never explicitly represented.

Once these contingency tables have been calculated, we build the Huffman tables for all of the groups in the network. If we're compressing to a file, we encode these tables at the beginning of the file so they can be extracted later for decompression. (We omit the details.) Next, we encode all of the records. To encode a record, we encode the variable groups in some order consistent with a topological sort of the groups in the Huffman network. When encoding any given group, we use the values of the group's conditioning variables to select the appropriate Huffman table, and then output the bits in the entry corresponding to the values of the group's variables. Decoding is performed in an analogous manner.

Compression and Decompression Performance

We discuss how to automatically learn good Huffman networks in the next section. But first, let us briefly discuss the compression performance of manually specified networks on a few synthetic datasets in order to illustrate situtations where grouping variables together makes sense versus situations where adding arcs between groups makes sense.

Dataset 1 contains 100,000 records with 32 binary variables. The variables are all independent of each other, and each variable has a probability of .2 of being set to zero and .8 of being set to one. In such a dataset, a Huffman network with each variable in its own individual group will not compress the data at all, regardless of what arcs are in the network. However, if variables are grouped together, then some compression can be achieved. Simply pairing the variables together in groups of two allows the file size to be reduced by 25 percent; similar results are achieved by using groups of four or eight variables. If we try putting too many variables in a group, on the other hand, such as sixteen, the Huffman tables begin requiring too many bits to specify, and the file size increases. The following table shows the compression performance of four different Huffman networks on this dataset in which the variables are place in groups of 2, 4, 8, or 16. We also provide results for `gzip` and `bzip2` for purposes of comparison.

Dataset 1	Uncompressed	gzip	bzip2	Huffman(2)	Huffman(4)	Huffman(8)	Huffman(16)
	401 KB	310 KB	331 KB	313 KB	297 KB	295 KB	396 KB

Dataset 2 contains 100,000 records with 32 variables, each of which can take on 32 values. The first variable randomly takes on one of its 32 possible values with uniform probability. Each other variable is then independently set at random to either be indentical to the first variable (with probability .5) or to be different from the first variable (with probability .5). In the latter case, its value is chosen with uniform probability from the remaining possible values. The first variable has a strong correlation with all other variables in the dataset, so it helps to have these variables either in the same group as the first variable or in groups conditioned on the first variable. However, only a limited number of variables can be put into the same group as the first variable without making the Huffman table for that group prohibitively expensive, so most of the variables must lie in other groups. Therefore, the Huffman network we use for this dataset simply has every variable in its own group, with all groups conditioned on the first variable.

Dataset 2	Uncompressed	gzip	bzip2	Huffman
	2.00 MB	1.92 MB	1.87 MB	1.42 MB

Finally, dataset 3 contains 100,000 records with 40 variables, each of which can take on 4 values. These variables have been randomly arranged into families of four variables. Within each family, the first variable is chosen at random; each other variable in the family is assigned a value identical to the value of the first variable in the family with probability .9, and to a random different value with probability .1.

If we compress this dataset with a Huffman network in which each variable is in its own group and in which the dependencies between the variables in the dataset are accurately modeled with arcs between these variables, the network compresses the data by about 30 percent. On the other hand, if we use a Huffman network in which the variables are simply put in groups of 4 corresponding to the families of connected variables in the first network, then we can compress the dataset by over a factor of 2. Even though the first network in a sense more accurately reflects how the data was generated, it suffers from the fact that each of the individual variables requires at least 1 bit to encode even if that variable is highly predictable given its parent variable. It is interesting to note that `gzip` and `bzip2` fail to compress this dataset at all.

Dataset 3	Uncompressed	gzip	bzip2	Huffman(arcs)	Huffman(groups)
	1.00 MB	1.00 MB	1.01 MB	685 KB	486 KB

2.4.2 Learning Huffman Networks

The problem of automatically finding effective Huffman networks to use for compression is very similar to the problem of finding maximum-BIC Bayesian networks, and is almost certainly at least as difficult. Therefore, as in the case of learning Bayesian networks, we must rely on heuristic search techniques. We have not extensively explored possible search algorithms for finding good Huffman networks; however, we have implemented a relatively simple multiple-restart stochastic hillclimbing algorithm. At each step during a hillclimbing run, the search algorithm considers making one of the following changes to its current Huffman network:

- Add an arc from a randomly selected variable to a randomly selected group, or remove the arc if one already exists

- Move a variable from its current group to a randomly selected group

If the change under consideration would create a cycle in the Huffman network, then it is immediately rejected and another change is randomly considered. Otherwise, the algorithm evaluates the resulting network and compares its estimated compression performance to the estimated compression performance of the current working network. A good network minimizes the total number of bits required to: (1) encode the network itself, and (2) encode the data with the network. Both of

	Arithmetic coding	Huffman coding (no groups)	Huffman network w/groups
Census	169 KB, 0.54 MB/sec	232 KB, 1.0 MB/sec	171 KB, 1.9 MB/sec
Banking	166 KB, 0.49 MB/sec	222 KB, 0.7 MB/sec	179 KB, 1.1 MB/sec
Astro1	4.2 MB, 0.50 MB/sec	4.8 MB, 1.0 MB/sec	4.3 MB, 2.2 MB/sec
Astro2	23.9 MB, 0.31 MB/sec	28.1 MB, 0.43 MB/sec	24.1 MB, 1.3 MB/sec

Table 2.2: Experimental results for arithmetic coding vs. Huffman networks

these terms can be estimated accurately from the Huffman trees associated with the group nodes once we have computed them. An important point to note about the evaluation of the network is that it is *local*: that is, if we change one part of the Huffman network, we do not need to recalculate the contributions of the other parts, since they remain the same. When the algorithm tries more than some user-specified number of changes to the network structure in a row with no improvement, the algorithm restarts another hillclimbing run with another initial network structure. As currently implemented, this algorithm requires many passes through the dataset; for large datasets, we use a randomly selected sample of datapoints rather than the entire dataset in order to maintain reasonable speed.

2.4.3 Experimental Results

We use multiple-restart hillclimbing over Huffman networks in order to find good coding networks for the four datasets previously examined. The starting point of the hillclimbing algorithm is a Huffman network with each variable in its own group; this initial Huffman network may either be empty (i.e., contain no arcs) or contain arcs corresponding to those learned by the Bayesian network-learning algorithms used in Section 2.2.2. Once the algorithm settles on a "good" network, we measure its performance both in terms of compressed file size (as we did in Section 2.2.2) and in terms of how fast it is able to perform decompression on encoded representations of the data in memory. Speed is measured in terms of the number of bits of uncompressed data decoded per second on a 450 MHz Pentium II. We compare the performance of Huffman networks learned via our stochastic hillclimbing procedure with (1) the performance of the Bayesian network-based arithmetic coding approach from Section 2.2.2, and (2) the performance of Huffman networks that employs the same network structures as the corresponding Bayesian network-based arithmetic coders, with each variable coded in its own group. The results are shown in Table 2.2.

The results show that naively using Huffman coding with the Bayesian networks learned in Section 2.2.2 results in significantly worse compression rates than arithmetic coding – in some cases, worse than gzip or bzip2 (see Table 2.1). Furthermore, it is often not too much faster than arithmetic coding. However, when variables are grouped together in Huffman network coding groups, the compression rates come much closer to those achieved with arithmetic coding — to within 1%-8%. Furthermore, as a side effect, decoding speeds become signficiantly faster when the variables are placed in coding groups since there is less execution overhead per variable. The resulting speeds were 2-4 times faster than arithmetic coding.

2.5 Conclusions, Related Work, and Possible Extensions

So far, we have only compared the algorithms developed in this chapter against gzip and bzip2, and the primary metric for comparison used is the resulting compression rate. While these results are encouraging, more thorough comparisons versus dictionary-based compression algorithms are warranted, particularly in (1) observing the effects of changing the dictionary size and coding granularity of the dictionary-based algorithms, and (2) comparing the relative speeds of the algorithms. Other "black box" compression algorithms worth comparing against include prediction by partial matching (PPM) [CW84], which uses a variable-length context over previous bytes to probabilistically predict the next byte, and dynamic Markov compression (DMC) [CH87] which automatically learns finite-state models of the datastream.

Huffman encoding allowed us to achieve much faster decompression speeds than with arithmetic coding at nearly the same compression rates, but it is still significantly slower than gzip or bzip2. The Huffman coder used in the experiments throughout this chapter was a naive implementation that performed decoding on a bit-by-bit basis using binary trees. More efficient algorithms for Huffman coding exist, such as canonical Huffman coding [HL90]. We have performed preliminary tests with a simple implementation of canonical Huffman coding; decoding speed was indeed increased by a further 10-20%. This isn't terribly dramatic, but additional optimization may still be possible; a fair amount of other previous research has been performed on making Huffman coding efficient ([CKP85], [Sie88], [MT97]).

All the experimental results for the Bayesian network-based compression algorithms presented in this chapter have required at least two passes through the dataset. Additionally, once a network was learned from the dataset, it was kept fixed throughout the subsequent compression of that dataset — that is, the model used for compression was *static*. There are advantages to this approach: namely, it is possible to maintain random access to small blocks of data, and the computational cost of compressing or decompressing any given dataset item is relatively small once the model has been learned and fixed. However, there are some situations in which it may be desirable to take a single pass through the data, such as when it is stored on media that requires long access times. Furthermore, if the probability distributions exhibited by the data change throughout the dataset, adaptive compression algorithms can achieve significantly better compression rates that static algorithms. In such situations, it may be better to use an *adaptive* compression algorithm that dynamically adjusts its model of the data during compression.

Frey [FHD96] breaks the dataset into blocks; after each block is encoded or decoded, the parameters of the network are adjusted using the wake-sleep algorithm. When attempting to learn a network with more complicated conditional probability distributions and an unknown structure, however, things become somewhat more complicated. How do we maintain the statistics used by the Bayesian network while its structure is being changed on the fly? If the current network does not model any dependencies between variables X and Y, for example, how will we ever notice that such a dependency exists in the data?

Friedman and Goldszmidt [FG97] address this problem by maintaining a set of *frontier* networks that each differ from the current network by one arc. When it is determined that one of these frontier networks is better than the current network, the current network is replaced with that frontier network, and a new set of frontier networks are generated. The statistics of these new frontier networks are then updated as more data is processed, and so forth. Relatively simple adjustments to the Bayesian Information Criteria scoring functions are made to account for the fact that not all statistics have been derived from the same number of data points. This technique could easily be applied to adaptive compression with sparsely connected Bayesian networks.

Extending such adaptive methods to work with dynamic Bayesian networks would be straightforward. This combination may be particularly useful for handling data containing both short-term and long-term variations in its distributions. The gradual changes in the networks' parameters and structure would allow them to better capture the long-term changes; meanwhile, the explicit modelling of dependencies between immediately adjacent datapoints may allow the model to track short-term correlations more effectively than possible with adaptive coding over nondynamic Bayesian networks.

There are a wide variety of ways in which Bayesian network structures can be learned, and in which the conditional probability distributions at their nodes can be expressed. For example, Frey's work [Fre98] uses highly connected networks in which each node has a fairly restricted conditional probability distribution. On the other hand, the algorithms used in this chapter generate networks with very sparse connectivity, but in which the nodes have unrestricted conditional probability distributions respresented in full tabular form — that is, the probability distribution over each variable's possible values is recorded seperately for every possible combination of values that variable's parent variables can take on. It would be interesting to compare the relative effectiveness of these two approaches, both in terms of compression rates and computational feasability.

It is possible to compromise between the unrestricted conditional probability distribution tables used here and the finite-parameter distributions used in work such as Frey's — namely, by learning "local structure" within the conditional probability distribution for each node [FG96b]. For example, one can use a decision tree for each variable representing how that variable's distribution depends on particular combinations of its parent variables' values, without exhaustively enumerating *all* possible combinations of the parent variables' values. This may allow us to condition some variables' distributions on the values of many other variables in a compact manner while still capturing some of the complexities in how these other variables' effects combine. (Essentially any supervised machine learning method that is capable of returning a probability distribution over a "class" variable's value when given the values of other predictive features can be used in this context in place of decision trees.) In Chapter 4, we will explore more general tree-based representations of conditional probability distributions. These representations could easily be used for the compression tasks addressed in this chapter.

So far, we have restricted our attention to datasets in which all variables are discrete. Of course, many datasets have real-valued variables instead, or a mixture of real-valued and symbolic variables. How should we deal with compressing the real values in such datasets? Since it is impossible to represent arbitrary real values perfectly in any finite number of bits, we must settle for an approximate representation. We might imagine attempting to compress real values losslessly up to the limits of a given machine's native floating-point format; however, if the least significant bits in these numbers are largely noise, they will be (1) hard to compress effectively, and (2) useless for most applications anyway. Thus, datasets with real-valued variables typically necessitate the use of *lossy* compression techniques that are not guaranteed to perfectly reconstruct the original uncompressed data.

Throughout the next two chapters, we will examine algorithms that learn Bayesian networks modeling probability distributions over both discrete and continuous variables. While we have not yet applied these algorithms to the lossy compression of real-valued datasets, they were designed partially with this application in mind. This is particularly the case with the Bayesian network-based models described in Chapter 4, which can be evaluated quickly and have other properties that make them convenient for compression.

A system called SPARTAN [BGR01] was recently developed for lossily compressing datasets by using networks of CART-like decision and regression trees. (SPARTAN was developed after the material in this chapter and Chapter 3 was published ([DM99], [DM00]) and largely concurrently with the material developed in Chapter 4.) There are considerable differences between SPARTAN's approach and the approach to compression that would most naturally result from the material in the next two chapters, however; see Section 4.9 for details.

Chapter 3

Mix-Nets

3.1 Introduction

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 typically 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 [DM95]. 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. In particular, mixtures of Gaussians can be fitted to data very quickly using an accelerated EM algorithm that employs multiresolution kd-trees [Moo99]. In this chapter, 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. In Section 3.2, 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 3.3, we describe an efficient heuristic algorithm for automatically learning the structures of our Bayesian networks from data. In section 3.4, we provide experimental results illustrating the effectiveness of our methods on two real scientific datasets and two synthetic datasets. In Section 3.5 we discuss possible applications, and finally in Section 3.6 we discuss related work and possible lines of further research.

First, a quick note about notation. When modelling probability distributions over continuous variables, the functions used usually provide estimated probability *densities* at the specified points — that is, in order to compute the actual probability that a continuous variable X will take on a value in some specified range close to some specific value x, it is necessary to integrate the value of the density function over that range. Discrete probability distributions are usually specified by functions that represent probability *masses*. Since the models discussed in this chapter and Chapter 4 are intended to model distributions of discrete variables and continuous variables simultaneously, we will often simply write $P(\vec{X})$ where $\vec{X}$ is a set of variables, some of which may be continuous and others of which may be discrete; converting this to an actual probability would, of course, require integrating over a volume in the space of continuous variables. For simplicity, we may also sometimes use notation such as

$$\int P(\vec{X})d\vec{Y}$$

in situations where $\vec{Y} \subset \vec{X}$ may contain both discrete and continuous variables. This is to be understood as shorthand for integrating over the continuous variables in $\vec{Y}$ and summing over the discrete variables.

3.2 Mix-nets

3.2.1 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,

$$P_i(\vec{\Pi}_i) = \int P_i(X_i, \vec{\Pi}_i) dX_i,$$

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.

Joint models over subsets of variables have been similarly conditionalized in previous research 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 distribution between the variable and its parents [HM97a]. (Some theoretical issues concerning the interdependence of parameters in such models are discussed in [HM97a] and [HM97b].) 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 [HT95].

3.2.2 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. [DH73]). 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:

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

where α_k represents the probability of a point coming from the k^{th} class, d is the number of dimensions, 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

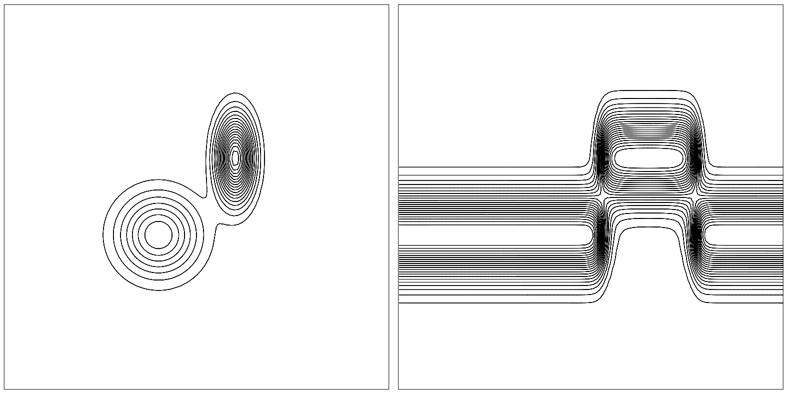


Figure 3.1: Contour plots for a simple Gaussian mixture $P(X, \Pi)$ (on the left) and the corresponding conditional distribution $P(X|\Pi)$ (on the right). X is the vertical axis and Π is the horizontal axis.

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 3.2.1, assuming all variables in the domain are continuous. This is the class of models we employ for continuous variables in this chapter, although many other classes may be used in an analogous fashion.

While $P_i(X_i|\vec{\pi}_i)$ is expressible as a mixture of Gaussians over X_i for any *specific* set of values $\vec{\pi}_i$ 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 3.1, 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 [HT95]. The differences are that our distributions use fewer Gaussians, but these Gaussians have varying weights and varying non-diagonal covariance matrices; we also employ a different learning algorithm to tune the model's parameters. 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.)

Learning Gaussian mixtures from data

The EM algorithm is a popular method for learning mixture models from data (see, e.g., [DLR77]). 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 [MSD97] are used to dramatically reduce the computational cost of each iteration [Moo99]. We refer the interested reader to this previous paper [Moo99] 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 [Sch78] previously discussed in Section 2.2.1 to choose between mixtures with different numbers of Gaussians. 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 [SM00][1]. 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.

3.2.3 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 corresponding to 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 $P_i(\vec{S}_i) =$

[1] Thanks to Andrew Moore and Peter Sand for providing the C code for this algorithm.

$P_i(X_i, \vec{\Pi}_i)$. 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 3.6). The models we will discuss in Chapter 4 naturally handle discrete variables in a much more graceful manner. (We will briefly discuss how we currently handle mixture tables' potential problems with sparse data in our experimental results section.)

3.3 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?

As mentioned in Section 2.2.1, finding the optimal Bayesian network structure with which to model a given dataset is NP-complete [Chi96], 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. 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 [HT95], 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. [CL68], [Sah96], [FNP99], and the second algorithm used in Section 2.2.1). We use such an algorithm in this chapter 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 (see Section 2.2.1). 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 3.2.

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 "†" in Figure 3.2 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

- $B := B_\epsilon$, *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 3.2: The greedy network structure learning algorithm employed in this chapter.

the well-known algorithm [CL68] 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.

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 be if they were computed from the original dataset, but we would expect them to be highly correlated in many practical circumstances.

Much like the structure-learning algorithm employed in Chapter 2, the structure-learning algorithm used here is similar to the "Limited Dependence Bayesian Classifiers" previously employed to learn networks for classification [Sah96], 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.

The learning algorithm is also somewhat similar in spirit to the "Sparse Candidate" algorithm [FNP99]. We will generalize the algorithm further in Section 4.6.

3.4 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.

3.4.1 Algorithms

Mix-net learner

This is our primary network-learning algorithm, as described in Figure 3.2. 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.)

Independent Mixtures

This algorithm will help us illustrate how much leverage our mix-net learning algorithm gets by modeling any dependencies at all between variables. 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.

Trees

This algorithm will help us illustrate how much leverage our mix-net learning algorithm gets by generating models more complex than tree-shaped (or forest-shaped) networks. 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).

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: as implemented for these experiments, this algorithm overcounts the number of parameters truly required to represent the distributions being modelled. When a joint distribution $P(X_i, \vec{\Pi_i})$ composed of a single Gaussian is used conditionally, the resulting conditional distribution $P(X_i|\vec{\Pi_i})$ is the same as would be provided with linear regression, which requires only $O(|\vec{\Pi_i}|)$ parameters. (See Section 4.2.2 for details.) Better methods for learning Bayesian networks of this form have been researched in the past (e.g. [GH94]). However, the accuracy of the implementation here is probably not too much worse than the performance that would be achieved with these other methods given the large number of datapoints used in our experiments.

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. [FG96a], [MC98a], [MC99]). The simple discretization algorithm discussed here and currently implemented for our experiments is certainly not state-of-the-art.

3.4.2 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 were made to each of the original datasets before handing

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

them to any of our learning algorithms. First, all continuous variables were scaled so that all values lie within $[0, 1]$. This helps put the log-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 were randomly perturbed by adding to it a value uniformly selected from [-.0005, .0005]. This noise was 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. (See Section 4.3 for further discussion on this topic.) We added 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 3.3 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 dataset 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.

3.5 Possible applications for Mix-Nets

3.5.1 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., [JL95]).

A recently developed type of classifier, Tree Augmented Naive Bayes (TAN) [FGG97], 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 [FGL98].

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 MAX-PARS parameter higher than 1, 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 [Sah96]. 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 sampling or numeric integration.)

3.5.2 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. 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 [NCC$^+$01].

3.5.3 Inference

While it is possible to perform exact inference in some kinds of networks modeling continuous values (e.g. [DM95], [Ala96]), 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 $P(X_i, \vec{\Pi}_i)$ and specific values $\vec{\pi}_i$ for $\vec{\Pi}_i$, it is possible to compute a conditional mixture table $P(X_i|\vec{\pi}_i)$. 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 discretization-based inference (e.g. [KK97]) or variational inference (see e.g. [JGJS98]) are also worth investigating.

3.5.4 Data compression

As discussed in Chapter 2, many popular and powerful methods for data compression such as arithmetic coding rely on explicit probabilistic models of the data they are compressing; 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. 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. [Say96]). 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 is 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 [Fre98]. 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 a reasonably effective model class with which to compress large datasets containing both continuous and discrete values. However, in Chapter 4, we will explore a different set of models that appear even more suitable.

3.6 Conclusions, Related Work, and Possible Extensions

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.

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. 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 [HM97a]. However, we know of none that are obviously practically applicable to the type of model employed in this chapter. Another possible approach is to simply drop the use of parameter-counting score metrics and instead rely on other methods such as cross-validation in order to control the complexity of the model. This is the approach we will take in Chapter 4.

Throughout this chapter 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., [CH92], [LB94], [HGC95], and [FNP99]), 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 previously been shown to work well on real datasets [MC98b]. In Chapter 4 we will explore this issue further, albeit in conjunction with different types of conditional distributions than the ones employed in this chapter.

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 (e.g. [Moo00]).

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 chapter. 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 [NH98].

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 [BFR98]. 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. Some preliminary experiments in Section 4.8.7 are performed in which global mixture models are compared to the Bayesian network-based models described in the next chapter.

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 [CS96]. The EM acceleration algorithm exploited in this chapter 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 [FG96b]. In Chapter 4, we will examine related approaches that handle continuous variables as well.

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 [JP99]. 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 in Bayesian networks are warranted (e.g. [HT95], [FN00]).

Chapter 4

Interpolating Conditional Density Trees

4.1 Introduction

While the Gaussian mixture-based algorithms in the previous chapter appear fairly effective at learning complex conditional distributions in a reasonable amount of time, the learning algorithm is still quite time-consuming on large datasets with many variables. Furthermore, evaluating the resulting distributions at specified points is also time-consuming, since each point requires the evaluation of many Gaussians. Approximations similar to those used by the accelerated EM algorithm used to learn the models [Moo99] might conceivably allow us to cut down on the number of Gaussians evaluated per point, but these approximations themselves are expensive to compute on a datapoint-by-datapoint basis.

Tree-based models of conditional probability distributions have historically been very popular within the machine learning community for classification and regression tasks (e.g. [Qui86], [BFOS84]). They can be reasonably quick to learn, are quick to evaluate, and can be fairly accurate as well. In both classification and regression trees, a given tree is used to predict the value of some output (or "child") variable X_i given a set of input (or "parent") variables $\vec{\Pi}_i$; in classification trees X_i is a discrete variable, while in regression trees X_i is continuous. In both kinds of trees, each branch node corresponds to a test applied to one or more variables in $\vec{\Pi}_i$, and each of the branch's child nodes corresponds to one of the mutually exclusive results of this test.

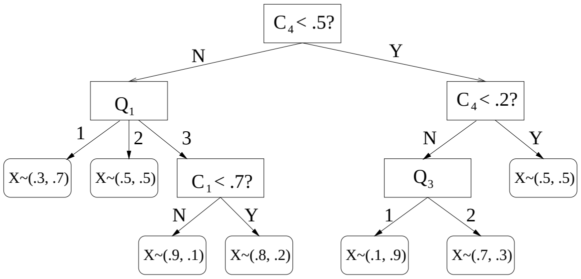


Figure 4.1: An example of a conditional density tree (or classifcation tree) for predicting the distribution of a binary variable X as a function of several other variables.

For example, one branch node might test a discrete variable $X_d \in \vec{\Pi}_i$ and have one child for each possible value of X_d; another branch in the same tree might test a continuous variable $X_c \in \vec{\Pi}_i$ and have one child corresponding to $X_c \leq b$ and another child for $X_c > b$ for some threshold b. Each leaf l of the tree contains a prediction for the value of X_i; depending on the task, the leaf's prediction may simply be the most likely value of X_i, or it may be a probability distribution over X_i. In the latter case, if X_i is discrete, this distribution is typically a multinomial model with one probability for each of X_i's possible values. ($\vec{\Pi}_i$ are ignored once the leaf is reached.) If X_i is continuous, the conditional distribution within a leaf is typically assumed to be a Gaussian whose mean is a either a linear function of $\vec{\Pi}_i$ or simply a constant.

Figure 4.1 shows an example classification tree in which the distribution of a binary variable X is predicted as a function of several other variables, some of which are discrete (the Q's) and some of which are continuous (the C's). To find the distribution of X, the prediction algorithm simply starts at the root of the tree (shown at the top of our diagram) and follows a path down the tree's branches according to the values of the other variables until it reaches a leaf. For example, if the continuous variable C_4 is less than .5, and the ternary discrete variable Q_1 has a value of 1, then the algorithm would predict that X has a 30% chance of taking on its first possible value and a 70% chance of its second.

Throughout the following discussion, we will often refer to the *constraints* associated with a given node of the tree. These constraints are simply the set of preconditions imposed on the node by all its ancestors in the tree. For example, consider the leaf in Figure 4.1 in which X's estimated distribution is $(.1, .9)$. The set of constraints associated with this leaf is $\{C_4 < .2, Q_3 = 1\}$. Similarly, the branch node testing Q_1 has the constraint set $\{C_4 < .5\}$.

In addition to conventional classification and regression tasks, tree-based conditional density estimators have also been used for the conditional distributions within Bayesian networks for discrete variables [FG96b]. Tree-based approximations of previously known *joint* probability distributions over sets of variables $\vec{S_i}$ (some variables of which may be continuous) have also been used in the past in order to perform inference in graphical models [KK97]. In such trees, the density $P_l(\vec{S_i})$ modelled within each leaf l is a constant.

In this chapter we examine several aspects of tree-based density estimators, with an emphasis on using them to obtain the conditional distributions required for Bayesian networks. First, we discuss learning algorithms for density trees modelling joint distributions $P(\vec{S_i})$. Section 4.2 describes several possible types of distributions to use in the leaves of these density trees. In Section 4.3 we discuss the criteria we use to evaluate density trees with different branching structures, and in Section 4.4 we discuss algorithms for attempting to grow trees maximizing this criteria, including methods for choosing branch variables (Section 4.4.1), choosing the threshold values for branches on continuous variables (Section 4.4.2), and choosing when to stop growing the tree (Section 4.4.3). We also discuss the simple parameter-smoothing method we use to prevent poor performance on previously unseen data (Section 4.4.4).

Section 4.5 describes learning algorithms for density trees modelling conditional distributions $P(X_i|\vec{\Pi}_i)$. Section 4.5.1 describes stratified conditional density trees, in which the desired conditional distribution is learned directly. These density trees are computationally expensive to learn, but fast to evaluate. In Section 4.5.2 we discuss how to take density trees learned to model joint distributions $P(X_i, \vec{\Pi}_i)$ and use them to compute conditional probabilities $P(X_i|\vec{\Pi}_i)$. Evaluating these conditional probabilities can be somewhat computationally expensive; however, joint density trees are easier to learn than stratified conditional density trees, and — somehat surprisingly — are often more accurate for conditional density estimation than stratified conditional density trees are, despite the fact that they are optimized for modelling the joint distribution rather than the conditional distribution. We then describe a way to transform joint density trees and evaluate them approximately. The resulting density estimator effectively combines the fast, accurate learning of joint density trees with the fast evaluation of stratified conditional density trees.

In Section 4.6 we describe a structure-learning algorithm for Bayesian networks generalizing the algorithm used in Section 3.3. Section 4.7 describes a method for improving the performance of density tree algorithms on distributions with sharp features in their marginal distributions. In Section 4.8 we perform an extensive set of experiments evaluating the algorithms proposed throughout previous sections. Finally, in Section 4.9 we discuss related work and directions for further possible research.

4.2 Joint density estimators for density tree leaves

First, we describe several different types of density estimators $P_l(\vec{S_i})$ for use within the leaves of tree-based joint density models. We concentrate primarily on varying methods of handling the continuous variables $\vec{C_i} \subset \vec{S_i}$. The discrete variables $\vec{Q_i} \subset \vec{S_i}$ are handled identically throughout all cases examined here. Namely, within a given leaf l, each discrete variable is assumed to be independent of all other discrete and continuous variables:

$$P_l(\vec{S_i}) = P_l(\vec{C_i}) \prod_{Q_k \in \vec{Q_i}} P_l(Q_k).$$

In joint density trees where any given set of assignments $\vec{S_i} = \vec{s_i}$ is consistent with a single leaf l, $P_l(\vec{S_i})$ may be rewritten as follows:

$$P_l(\vec{S_i}) = \sum_{l'} P(l')P_{l'}(\vec{S_i}|l') = P(l)P_l(\vec{S_i}|l)$$

where the sum collapses because $P(\vec{S_i}|l') = 0$ for all l' not equal to l, the unique leaf consistent with the particular values of $\vec{S_i}$; $P(l)$ is an estimate of the probability that any particular datapoint will be consistent with all the constraints imposed by the ancestor branches of l; and $P_l(\vec{S_i}|l)$ is a conditional distribution over $\vec{S_i}$ given that $\vec{S_i}$ is consistent with l's constraints. This means that in order to learn the density estimator $P_l(\vec{S_i})$ that is used for all datapoints $\vec{S_i} = \vec{s_i}$ consistent with a given leaf l's constraints, we can simply learn a density model $P_l(\vec{S_i}|l)$ over the space of possibilities consitent with l by estimating it from all data consistent with l, and then scaling the probabilities returned by this model by our estimate of $P(l)$.

Thus, all leaf distributions we examine in this thesis may be written as:

$$P_l(\vec{S_i}) = P(l)P_l(\vec{C_i}|l) \prod_{Q_k \in \vec{Q_i}} P_l(Q_k|l).$$

We will also restrict our attention to trees in which each branch node tests exactly one variable. If the variable tested is discrete, the branch simply has one child node for every possible value of that variable. If the variable X tested is continuous, the branch specifies a threshold value b and has two children corresponding to $X \leq b$ and $X > b$. Furthermore, we assume that we have some *a priori* bounds on the minimum and maximum possible values of all continuous variables. (This assumption will be discussed in more detail shortly.) That is, we assume all continuous values are restricted within some known hypercube. Together with the previous constraint on the form of branches allowed on continuous variables, this implies that the space of possible values for C_i consistent with any given leaf in the tree is also a hypercube.

We now discuss several possible estimators for $P_l(\vec{C}_i|l)$. In addition to describing how these joint distributions are learned and evaluated, we will also describe how to use them conditionally (that is, how to calculate $P_l(X_i|\vec{\Pi}_i, l)$) in the cases where the variables in $\vec{C}_i$ are not modelled independently within each leaf.

4.2.1 Constant leaf densities

This density estimator for $P_l(\vec{C}_i|l)$ is very straightforward: namely, we assume a constant density

$$P_l(\vec{C}_i|l) = \frac{1}{Volume_{\vec{C}_i}(l)}$$

where $Volume_{\vec{C}_i}(l)$ is the volume of l's bounding box over the continuous variables $\vec{C}_i$, as determined by the constraints imposed upon l by its ancestors in the tree. Since this box is simply an axis-aligned hypercube, its volume is trivial to compute.

Density trees using constant-leaf densities are fast to learn; however, as our experimental results will show, density trees employing other leaf distributions are usually more accurate.

4.2.2 Gaussian leaf densities

Axis-aligned (diagonal covariance)

This density estimator for $P_l(\vec{C}_i|l)$ assumes that the distribution of each variable $X_k \in \vec{C}_i$ is proportional to a Gaussian and independent of all other variables. We renormalize the distribution of each variable so its integral over the range $[x^0, x^1]$ of

l is 1:

$$P_l(\vec{C}_i|l) = \prod_{X_k \in \vec{C}_i} \alpha_k \exp\left\{-\frac{(X_k - \mu_k)^2}{2\sigma_k^2}\right\}$$

where

$$\alpha_k = \frac{1}{\int_{x^0}^{x^1} \exp\left\{-\frac{(x_k-\mu_k)^2}{2\sigma_k^2}\right\} dx_k}.$$

When learning a leaf distribution, each parameter μ_k and σ_k is simply set to the maximum-likelihood value for an untruncated Gaussian:

$$\mu_k = \frac{1}{R}\sum_{j=1}^{R} x_k^j, \qquad \sigma_k = \frac{1}{R}\sum_{j=1}^{R}(x_k^j - \mu_k)^2$$

where x_k^j is the value that datapoint j assigns to X_k and R is the number of datapoints. The α_k's are then computed using routines for evaluating the error function (see, e.g. [PTVF92]).

A caveat: this is not the same as fitting the best possible truncated and renormalized Gaussian to the data. For example, a uniform distribution could be fitted perfectly in the limit by letting the variance go to infinity and setting the renormalizing constant to correspondingly smaller and smaller numbers. The procedure outlined above will obviously fail to fit this distribution correctly, since any covariance computed from the data will necessarily be finite. Unfortunately, fitting the optimal truncated and renormalized Gaussian presents a more complicated (albeit still only two-dimensional) optimization problem.

Full covariance / linear regression

This density estimator for $P_l(\vec{C}_i|l)$ assumes $\vec{C}_i$ is distributed according to a multidimensional Gaussian with a full covariance matrix:

$$P_l(\vec{C}_i|l) = \frac{1}{(2\pi)^{\frac{d}{2}}|\mathbf{\Sigma}|^{\frac{1}{2}}} \exp\left\{-\frac{1}{2}(\vec{C}_i - \vec{\mu})^T \mathbf{\Sigma}^{-1}(\vec{C}_i - \vec{\mu})\right\}$$

where d is the number of variables in $\vec{C}_i$, μ_i is the vector of means and $\mathbf{\Sigma}$ is the covariance matrix. As in the axis-aligned case, the mean and covariance are set to their maximum-likelihood values

$$\vec{\mu} = \frac{1}{R}\sum_{j=1}^{R} \vec{c^j}, \qquad \mathbf{\Sigma} = \frac{1}{R}\sum_{j=1}^{R}(\vec{c^j} - \mu)(\vec{c^j} - \mu)^T.$$

In cases where we wish to estimate a conditional distribution $P_l(X_i|\vec{C_{\Pi_i}}, l)$ from the joint $P_l(\vec{C_i}, |l)$, we can use the following relationship. Assume $\mathbf{\Sigma}$ is nonsingular; let $\mathbf{K}$ denote its inverse $\mathbf{\Sigma}^{-1}$. Without loss of generality, assume the mean vector, the covariance matrix, and the inverse of the covariance matrix are partitioned as follows:

$$\vec{\mu} = \left(\begin{array}{c} \mu_{X_i} \\ \mu_{\vec{\Pi_i}} \end{array} \right), \mathbf{\Sigma} = \left(\begin{array}{c} \mathbf{\Sigma}_{X_i X_i} \mathbf{\Sigma}_{X_i \vec{\Pi_i}} \\ \mathbf{\Sigma}_{\vec{\Pi_i} X_i} \mathbf{\Sigma}_{\vec{\Pi_i} \vec{\Pi_i}} \end{array} \right), \mathbf{K} = \left(\begin{array}{c} \mathbf{K}_{X_i X_i} \mathbf{K}_{X_i \vec{\Pi_i}} \\ \mathbf{K}_{\vec{\Pi_i} X_i} \mathbf{K}_{\vec{\Pi_i} \vec{\Pi_i}} \end{array} \right)$$

(To keep the notation from getting overly complex, we temporarily assume there are no discrete parent variables and thus $\vec{\Pi_i} = \vec{C_{\Pi_i}}$. Since discrete variables are modelled independently of the continuous variables, they have no effect on the conditional distributions over the continuous variables within a leaf.) We can then break the joint Gaussian into three parts: a Gaussian distribution over the parent variables, a linear transformation that maps an assignment of the parent variables' values to a conditional mean on the child variables, and a covariance matrix for the conditional distribution of the child variables. The Gaussian distribution over the parent variables simply has mean $\mu_{\vec{\Pi_i}}$ and covariance $\mathbf{\Sigma}_{\vec{\Pi_i}}$. The conditional distribution of X_i given $\vec{\Pi_i}$ has the following mean $\mu_{X_i|\vec{\Pi_i}}$ and covariance $\mathbf{\Sigma}_{X_i|\vec{\Pi_i}}$:

$$\mu_{X_i|\vec{\Pi_i}} = \mu_{X_i} + \mathbf{\Sigma}_{X_i \vec{\Pi_i}} (\mathbf{\Sigma}_{\vec{\Pi_i} \vec{\Pi_i}})^{-1} (\vec{\Pi_i} - \mu_{\vec{\Pi_i}}), \qquad \mathbf{\Sigma}_{X_i|\vec{\Pi_i}} = (\mathbf{K}_{X_i X_i})^{-1}.$$

(This is a well-known result; see e.g. [Lau96] for a sketch of the derivation.) Note that when the joint Gaussian's parameters are set to their maximum-likelihood values, the conditional distribution obtained from the above equations is identical to the conditional distribution we would have obtained with linear regression.

Unlike the density estimator we use in the axis-aligned case, we do not guarantee that the estimated joint distribution integrates to 1 over the bounds of the leaf, since evaluating this integral is difficult. (Maximizing the log-likelihood of the distribution while taking truncation and renormalization directly into account would be even more difficult.) However, when estimating $P_l(X_i|\vec{\pi_i}, l)$ for a single target variable X_i and a specific π_i using the equation above, we renormalize the resulting Gaussian distribution over X_i so it does integrate to 1 over X_i's range in l. When using a density tree that models a joint distribution $P(X_i, \vec{\Pi_i})$ in order to compute a conditional distribution $P(X_i|\vec{\Pi_i})$, as we will discuss in Section 4.5.2, we also need $P(\vec{\Pi_i}|l)$ in order to compute $P(l|\vec{\Pi_i})$. While we do not guarantee in this case that $P(\vec{\Pi_i}|l)$ is normalized, it is still easy to guarantee that the resulting estimates for $P(L|\vec{\Pi_i})$ are normalized across the set of leaves L:

$$P(l|\vec{\Pi}_i) = \frac{P(l)P(\vec{\Pi}_i|l)}{\sum_{l'} P(l')P(\vec{\Pi}|l')}.$$

Since the final conditional distribution $P(X_i|\vec{\Pi}_i)$ is then a normalized weighted sum of normalized conditional distributions $P_l(X_i|\vec{\pi}_i, l)$, the final conditional distribution is normalized as well.

Gaussians (with either diagonal or nondiagonal covariance matrices) are one of the most commonly used parametric models for continuous probability distributions. They make a sensible choice for the leaf distributions of simple CART-like trees that do not branch on the variable being modeled. However, in trees that branch on the variables being modeled, they can be less effective than other types of leaf distributions. The distribution modeled in every Gaussian leaf will have a "bump" in it at the Gaussian's mean, which must lie somewhere inside the leaf's boundaries (assuming a more complicated fitting mechanism that takes truncation into account is not being employed). Increasing the resolution of the tree increases the number of bumps, making it impossible to accurately model arbitrary smooth distributions even in the limit of infinite data. The other leaf distributions discussed in this thesis are capable of representing uniform distributions as a special case, and therefore do not suffer from this problem.

4.2.3 Exponential leaf densities

In a leaf of this type, each continuous variable is modeled independently with an exponential distribution that is truncated to the leaf's range and renormalized. Let X be one such variable currently in consideration. If X's range in a leaf l is $[x^l, x^r]$, then $P_l(x|l) = be^{ax}$, where

$$b = \frac{1}{\int_{x^l}^{x^r} e^{ax} dx} = \frac{a}{e^{ax^r} - e^{ax^l}}.$$

Given the data D falling in leaf l, we wish to set a to maximize the total log-likelihood of the data. For simplicity of exposition, we assume without loss of generality that $[x^l, x^r] = [0, 1]$. If x^j is the value that datapoint j assigns to X, then the total log-likelihood of the data (restricted to variable X) is

$$LL(D) \;\; = \;\; \sum_j \log\left(\frac{a}{e^a - 1} e^{ax^j}\right)$$

$$= \quad R(a\bar{x} + \log a - \log(e^a - 1)),$$

where R is the total number of datapoints and $\bar{x}$ is the mean value of X according to the data. Setting the derivative of this log-likelihood to zero gives us

$$(e^a - 1)(1 + \bar{x}a) - ae^a = 0.$$

This equation can be solved for $\bar{x}$ in closed form:

$$\begin{aligned}\bar{x} &= \frac{ae^a - e^a + 1}{ae^a - a} \\ &= -\frac{1}{a} + \frac{1}{2}\coth\left(\frac{a}{2}\right) + \frac{1}{2},\end{aligned}$$

where coth is the hyperbolic cotangent. (The latter form makes it slightly more clear that the function is antisymmetric around $(0, \frac{1}{2})$.) Unfortunately, this relationship between $\bar{x}$ and a is not easily invertible. However, for a given value of $\bar{x}$, we can use Newton's method to find the a for which the derivative of the log-likelihood is zero. Arbitrary initial choices for a can cause Newton's method to diverge on this problem. For $\bar{x}$ close to 0, the correct value for a is approximately $-\frac{1}{\bar{x}}$; in practice, choosing $-\frac{1}{\bar{x}}$ as an initial guess for Newton's method appears to work for the range $0 < \bar{x} \leq .5$. To handle $\bar{x} > .5$, we use the relationship $a(\bar{x}) = -a(1 - \bar{x})$.

This tells us how to find the maximum-likelihood estimate for a when the leaf's range is $[0, 1]$. To find it for a different range $[x^l, x^r]$, we simply rescale the leaf's range to $[0, 1]$, rescale $\bar{x}$ similarly, find the appropriate value for a in this range, and then divide this a by $x^r - x^l$.

The fact that $\bar{x}$ is all that is required to optimally fit the truncated and renormalized exponential distribution makes it significantly simpler and faster to learn than the linearly interpolated probability densities discussed in section 4.2.4. However, while the estimated density within a leaf is nearly linear when $\bar{x}$ is close to the center of the leaf, it is extremely nonlinear when $\bar{x}$ is very close to one of the leaf's boundaries, as seen in Figure 4.2. Our experimental results will reveal that density trees employing exponential-distribution leaves can sometimes be even less accurate than ones using constant-distribution leaves due to certain properties the exponential distribution exhibits in these extreme cases.

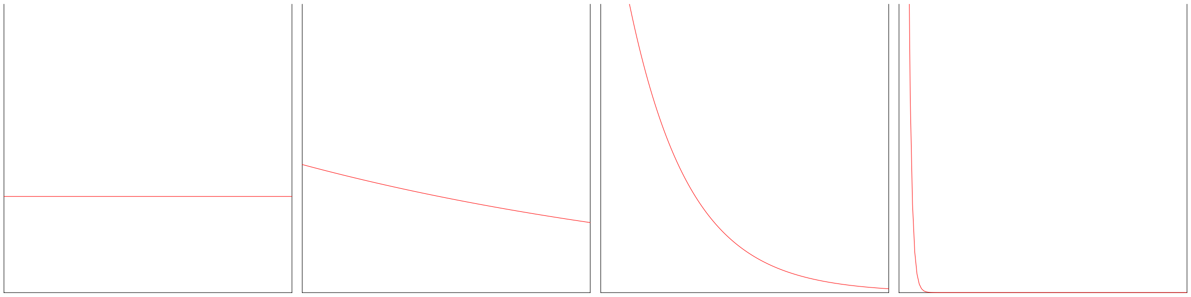


Figure 4.2: Truncated and renormalized exponential distributions for (from left to right) $\bar{x} = .5, \bar{x} = .45, \bar{x} = .2$, and $\bar{x} = .01$. Each distribution's range is [0, 1]; the density for $\bar{x} = .5$ is the constant 1.

4.2.4 Linear leaf densities

In a leaf l of this type, each continuous variable is modeled independently with a density that changes linearly across the bounds of the leaf. Let X be one such variable under consideration. Without loss of generality, assume X's range over l is $[0, 1]$. (Other ranges can be handled by scaling all data to $[0, 1]$ before parameter estimation and adjusting the resulting estimated parameters straightforwardly.) Then $P_l(x|l) = (1 - x)a_0 + xa_1$, where a_0 and a_1 are the estimated densities at $X = 0$ and $X = 1$ respectively. The integral of this distribution over $[0, 1]$ is $\frac{a_0+a_1}{2}$, so a_0 is constrained to $2 - a_1$.

Given the data D falling in leaf l, we wish to set a_1 to maximize the total log-likelihood of the data:

$$LL(D) = \sum_j \log(a_1 x^j + (2 - a_1)(1 - x^j)).$$

Calculating the derivative with respect to a_1 and setting it to zero gives us

$$\sum_j \frac{2x^j - 1}{(2x^j - 1)a_1 + (2 - 2x^j)} = 0,$$

or equivalently

$$\sum_j (2x^j - 1) \prod_{k:k \neq j} ((2x^k - 1)a_1 + (2 - 2x^j)) = 0.$$

Solving this equation directly would apparently involve finding the roots of an R^{th}-degree polynomial, where R is the number of datapoints in D. In some cases, it

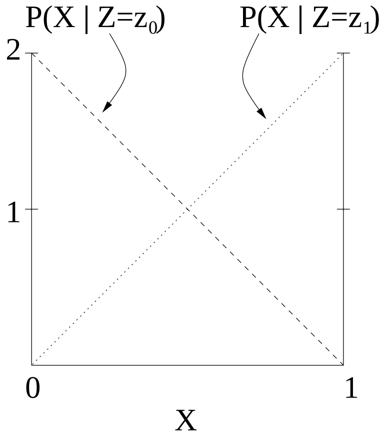


Figure 4.3: The distributions $P(X|Z)$ of the two unobserved classes $Z = z_0$ and $Z = z_1$ used to model linear interpolation.

may have no real solution, since the above equation does not take into account the constraint that a_0 and a_1 must both be nonnegative; in these situations, the optimum estimator is obtained by setting a_0 to 0 and a_1 to 2, or vice-versa. (For example, this occurs any time all of the training data lies in the region $X < .5$.)

Another way of viewing this density estimator is to assume the existence of an unobserved "class" variable Z that determines which of two distributions $P(X|Z = z_0)$ and $P(X|Z = z_1)$ each datapoint is generated from, where $P(x|z_0)$ is simply $2-2x$ and $P(x|z_1)$ is $2x$. (See Figure 4.3.) Maximizing the log-likelihood then boils down to finding the distribution $P(Z)$ that maximizes

$$LL(D) = \sum_j \log \sum_z P(Z = z)P(x^j|Z = z).$$

It is easy to prove that this log-likelihood has at most one distinct local maximum with respect to $P(Z)$ by using the fact that the logarithmic function is concave. Any local optimization routine capable of handling the constraints on $P(Z)$ (namely, that $\sum_z P(z) = 1$ and each $P(z)$ must be in $[0,1]$) can be used for this optimization problem. For this particular density estimator, the optimization is only one-dimensional, since there are only two possible values for the class variable, so we could use any of a wide variety of line-search methods. However, in section 4.2.5, we will consider similar density estimators in which the hidden class variable can take on many more values, thus requiring optimizations over higher-dimensional spaces.

In such higher-dimensional spaces, the *Expectation Maximization* or *EM algo-*

rithm [DLR77] is a simple method for finding distribution parameters that optimize the log-likelihood of data in which some variables are not always observed. The algorithm is an iterative algorithm with two steps per iteration. The Expectation or "E" step calculates an expected distribution over the unobserved variables given the observed variables and the current estimates for the distribution's parameters. The Maximization or "M" step then re-estimates the distribution parameters to maximize the likelihood of both the observed data and the unobserved variables, assuming the unobserved variables are distributed according to the expected values calculated in the previous E step.

For the optimization problem under consideration here, we start with an initial guess $P_0(Z)$ and iteratively generate better estimates $P_1(Z), P_2(Z), \ldots$ as follows:

- E step: for each datapoint j and each possible hidden variable value z^k for that datapoint, calculate $P_t(z^k|x^j) = \alpha_j P(x^j|z^k)P_t(z^k)$, where $\alpha_j = \sum_{z^k} P(x^j|z^k)P_t(z^k)$.
- M step: for each possible value z^k assigned to Z, calculate $P_{t+1}(z^k) = \sum_{j=1}^{R} P_t(z^k|x^j)$.

Since the log-likelihood function has only one local maxima, we expect the choice of $P_0(Z)$ to have little effect on the final outcome. A natural choice is the uniform distribution. The algorithm can be terminated when the increase in log-likelihood between iterations becomes lower than a specified threshold, or terminated after some fixed number of iterations.

It is not difficult to prove that each iteration of the EM algorithm increases the log-likelihood of the data, or at least does not decrease it [DLR77]. Proving that the algorithm actually converges to a local maxima of the log-likelihood function is more involved (again, see [DLR77] for descriptions of the necessary conditions), but in practice it is rare for the algorithm to do otherwise.

Since all the $P(x^j|z^k)$'s in this particular class of density estimator are fixed, they can be precomputed and cached before EM iterations are started. Furthermore, for any given leaf in the density tree, there is only one parameter that needs to be estimated for each continuous variable. In the early stages of the tree-growing process, there will typically be many more datapoints per leaf than are necessary to estimate this parameter to a reasonable level of accuracy, and the cost per EM iteration scales linearly with the number of datapoints used. Therefore, if there are more than some number $R_{\max}$ of datapoints mapped to the leaf, we randomly sample $R_{\max}$ of them (without replacement) and use only those datapoints while determining the leaf's distribution parameters.

Density trees using independent linear interpolations take somewhat longer to learn than those using exponential distributions, but each leaf still only requires one independent parameter per continuous variable being modeled. Furthermore, our experimental results will show that they are typically more accurate than trees employing exponential distributions.

4.2.5 Multilinear leaf densities

In a leaf l of this type, the density of all continuous variables $\vec{C_i}$ is modeled jointly rather than independently. Similarly to how each individual variable was handled in the previous section, the joint distribution over $\vec{C_i}$ can be expressed as a mixture model with hidden class variable Z:

$$P_l(\vec{C_i}) = \sum_{z^k} P(z^k)P(\vec{C_i}|z^k)$$

where each class distribution $P(\vec{C_i}|z^k)$ is fixed. Now, however, there are 2^d different possible values for the class variable, where d is the number of variables in $\vec{C_i}$. Each of these values corresponds to one of the 2^d corners of the d-dimensional hypercube representing the leaf's bounds. As before, for the purposes of exposition we assume without loss of generality that these bounds are $[0,1]^d$. For a given value z^k of Z for which the corresponding coordinates in $\{0,1\}^d$ are $(y_1^k, y_2^k, \ldots, y_d^k)^T$,

$$P(\vec{C_i} = (c_1, c_2, \ldots, c_d)^T|z^k) = 2^d \prod_{j=1}^{d}(1 - |y_j^k - c_j|).$$

Figure 4.4 shows an example calculation of such a $P(\vec{C_i}|Z)$.

As in the previous section, we precompute all $P(x^j|z^k)$'s and then use the EM algorithm to adjust $P(Z)$ towards the distribution maximizing the likelihood of the data.

It might appear at first glance that evaluating $P_l(\vec{C_i})$ takes $\Theta(d2^d)$ time, since it involves a sum over 2^d addends each of which is product of d multiplicands. However, with a bit of additional programming complexity, we compute each product in amortized constant time by reusing the product of the multiplicands it has in common with the previously computed product. This reduces the evaluation time to $\Theta(2^d)$.

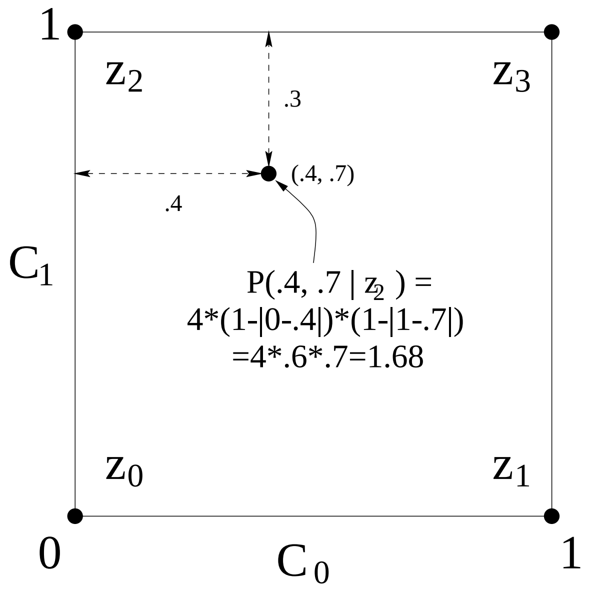


Figure 4.4: An example calculation of $P(C_0 = .4, C_1 = .7|Z = z_2)$, where z_2 is the hidden class value corresponding to the corner $(C_0 = 0, C_1 = 1)$.

In order to evaluate a conditional probability $P_l(X_i|\vec{\Pi}_i)$ (where we once again ignore the possibile existence of discrete variables in $\vec{\Pi}_i$ in order to simplify the notation), we may use the equation

$$P_l(x_i|\vec{\Pi}_i) = \frac{P_l(X = x_i, \vec{\Pi}_i)}{P_l(\vec{\Pi}_i)} = \frac{P_l(X = x_i, \vec{\Pi}_i)}{P_l(X_i = 0.5, \vec{\Pi}_i)}$$

where the latter equation holds because integrating out X_i results in a multilinear interpolation over $\vec{\Pi}_i$ in which each corner's density is an average of the densities of the two corners in the original interpolation that have the same coordinates for $\vec{\Pi}_i$. This same averaging can be achieved by simply setting $X_i = .5$ in the original joint distribution.

This is not necessarily the most efficient way to compute $P_l(X_i|\vec{\Pi}_i)$, but it does lead to a potentially interesting observation. Suppose that rather than maximizing the log-likelihood of the joint $P_l(X_i, \vec{\Pi}_i)$, we wished to maximizine the conditional log-likelihood $P_l(X_i|\vec{\Pi}_i)$. This conditional log-likelihood can be written as the joint log-likelihood of the data minus the joint log-likelihood of a phantom dataset in which each value for X_i is replaced with 0.5. Thus, many optimization algorithms one might use to maximize the joint log-likelihood can be applied straightforwardly to maximizing the conditional log-likelihood, as long as the algorithms are capable of handling datapoints with "negative weight." EM would probably fail if used in this manner, since it is unclear what would keep it from assigning negative values to some $P(Z)$'s. Other optimization algorithms that have been adjusted to take $P(Z)$'s constraints into account (for example, gradient-based methods employing "softmax" changes of variables [Bri90]) might be usable.

As in section 4.2.4, when fitting $P(Z)$ for a given leaf, we restrict the number of datapoints used for the fit in order to increase computational efficiency. However, since the number of parameters required for multilinear interpolation scales with 2^d, we scale the number of datapoints used with 2^d as well.

Our experimental results will show that multlinearly interpolated leaf distributions typically provide the most accurate density estimation. However, the accuracy is not too much greater than that provided by using one independent linear interpolation per variable per leaf, and it does come at considerable additional computational expense.

4.3 Tree evaluation criteria

Now that we have discussed several possible types of density estimators we might wish to use in the leaves of density trees, we move on to discussing different methods for determining density tree structures. This immediately raises the question of how we will evaluate two different density trees or subtrees in order to determine which is "better". Even if we know the exact distribution $P(\vec{X})$ from which the finite dataset given to the density estimator had been generated, there are many possible criteria we could use to measure the quality of the resulting estimated distribution $\hat{P}$. For example, in the statistical literature it is common to use the integrated squared error

$$\int [\hat{P}(\vec{x}) - P(\vec{x})]^2 d\vec{x}.$$

Other possibilities include the L_∞ norm

$$\sup_{\vec{x}} |\hat{P}(\vec{x}) - P(\vec{x})|,$$

the L_1 norm

$$\int |\hat{P}(\vec{x}) - P(\vec{x})| d\vec{x},$$

and the Kullback-Leibler divergence

$$D(P||\hat{P}) \quad = \quad \int P(\vec{x}) \log \frac{P(\vec{x})}{\hat{P}(\vec{x})} d\vec{x}$$

$$= \quad -H(P(\vec{X})) - \int P(\vec{x}) \log \hat{P}(\vec{x}) d\vec{x},$$

where

$$H(P(\vec{X})) = -\int P(\vec{x}) \log P(\vec{x}) d\vec{x}$$

is the *entropy* of the true distrubution $P(\vec{X})$. Since $H(P(\vec{X}))$ is constant when comparing different estimates $\hat{P}(\vec{X})$, minimizing the Kullback-Leibler divergence between $\hat{P}$ and P is the same as maximizing

$$\int P(\vec{x}) \log \hat{P}(\vec{x}) d\vec{x},$$

which is simply the average log-likelihood we would expect $\hat{P}(\vec{X})$ to assign to a point randomly generated from the true distribution $P(\vec{X})$.

Because we will generally not know the true distribution $P(\vec{X})$ from which the original data was generated, we approximate this average log-likelihood by evaluting it over a finite set of "holdout" datapoints that were not used to fit the model $\hat{P}(\vec{X})$ under consideration. In the limit as the size of the holdout set approaches infinity, the density estimator selected via this average log-likelihood criterion is the most likely hypothesis as to $P(\vec{X})$'s form out of all the forms evaluated. Furthermore, it also has the property that it is the best model with which to compress data generated randomly from $P(\vec{X})$ in the limit that an infinite amount of precision is required for each coded value of $\vec{X}$. Since this thesis is focussed partially on potential applications to compression problems, this makes it a natural criterion for us to use.

There are a few caveats, however. Many real-life probability densities are infinite at certain points. For example, a supposedly "continuous" value might actually be quantized so that it always exactly takes on one of 1024 values. The density at each of these points is then a delta function; at all other points it is zero. Alternatively, a sensor may "clip" data so that all input past a certain range is mapped precisely to some maximum representable output; the output variable's density at this maximum will also be a delta function. In such situations, log-likelihood is largely a meaningless measure. The relative log-likelihoods of two different density estimators will be determined almost wholly by exactly how they handle the data lying in these regions of infinite density. Fair comparisons between different density estimation methods on such data would require ensuring that they handled these points in an essentially identical fashion — a tedious and rather difficult task. When using density estimates to compress real-valued data, we normally only care about a finite level of precision and disregard all but the first few significant figures of the data; thus, we could conceivably alter all the density estimators examined to take this finite precision into account and then compare the number of bits required to encode a given level of accuracy using the resulting models. However, again, doing so and ensuring that it's done fairly would be tedious and difficult, since most density learning algorithms have not been designed with that particular task in mind. Therefore, for the purposes of evaluating different density learning algorithms in this thesis, we assume all datasets are generated from distributions in which all densities are finite. To ensure this is the case, random noise is added to all datapoints in all the real-world datasets. The resulting evaluations can be seen as crude approximations to how well the density estimates would perform if used to compress real data to a precision corresponding to the magnitude of the noise.

Another reason the log-likelihood criterion is not quite correct for compression applications is that the cost of encoding the density estimator itself is not taken into account. In many compression applications it might be more appropriate to use a scoring metric that uses the log-likelihood of the training data (rather than of an independent hold-out set) minus a penalty term that scales with the number of parameters required for the model, such as the Bayesian Information Criterion (BIC) [Sch78]. However, when compressing very large datasets it is often computationally infeasible to use the entire dataset while learning a density model, and a relatively small random sample must be used instead. In such situations, the total number of bits required for compression will be determined primarily by the learned model's accuracy on data that was never presented to the learning algorithm, and the number of bits required to encode the model will only be of secondary importance — and therefore the log-likelihood criterion employed here may in fact be more suitable than criteria employing penalized training-set log-likelihoods such as the BIC.

4.4 Tree-growing algorithms

Now that we have discussed the evaluation of candidate density trees, we examine algorithms for growing these candidate trees. In this thesis we will restrict our attention primarily to "top-down" learning algorithms of the following general form:

- Either decide to model the data as a leaf, or decide to branch. If not branching, learn a leaf distribution (of one of the distribution types describe in section 4.2) and return it. Otherwise:

- Decide what variable to branch on; if this variable is continuous, also decide upon a threshold value. Create a branch node corresponding to this branch variable (and threshold value). In the case of a discrete-variable branch, the branch node has one child pointer corresponding to each possible value of the variable. In the case of continuous branch variables, the branch node has two child pointers: one for cases in which the branch variable is less than or equal to the threshold, and another for cases in which the branch variable is greater than the threshold.
- Set each child pointer of the branch node to the result obtained by recursively calling the tree-learning algorithm on the subset of the data satisfying the constraints associated with that particular child.

In addition to the data, the tree learning algorithm is supplied with a set of constraints. Each continuous variable X_i has a constraint in the set of the form $a_i \leq X_i \leq b_i$. Each discrete variable Q_i either has no constraints or has a constraint of the form $Q_i = q_i$ for some value q_i. The algorithm is initially supplied with constraints over the continuous variables corresponding to *a priori* known bounds on their values and no constraints over the discrete variables. When calling itself recursively to learn a branch node's child, the algorithm adds the appropriate discrete-variable constraint or makes the appropriate continuous-variable constraint more specific accordingly.

We now discuss different methods for making the decisions required by the above general algorithm.

4.4.1 Branch variable selection strategies

If we have decided to model the current data subset with a branching density tree rather than a simple leaf distribution, then we need to decide which variable to branch on. One simple possibility is to have the variables "take turns" according to some arbitrary variable ordering as the tree's depth increases, with the exception that each discrete variable can only ever be branched on once. For example, in a joint density tree over two continuous variables, we might arrange the tree so that the root node can only split on variable C_1; all nodes directly below the root node can only split on variable C_2; all nodes two levels below the root node can only split on varible C_1

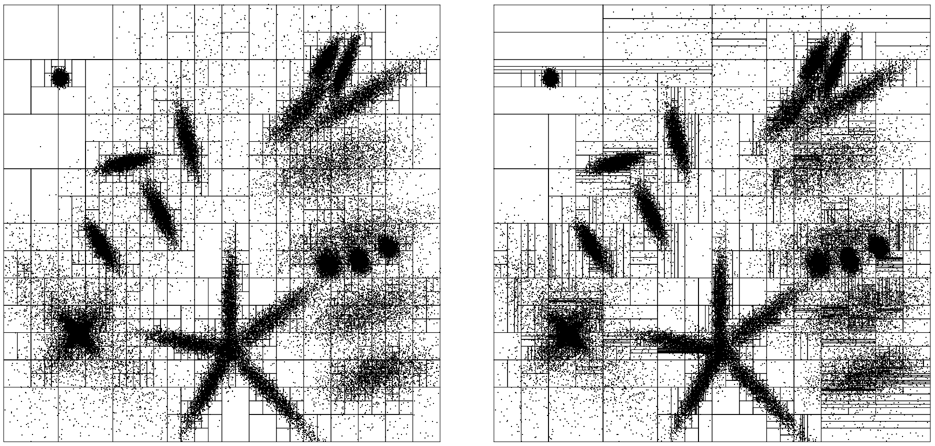


Figure 4.5: Example density trees learned using the turn-based branching criteria (left) and the greedy branching criteria (right) on a synthetic dataset.

again; and so forth. If the split threshold chosen for each branch node is always the midpoint of the range of its branching variable, and the tree is of constant depth, then this imposes a grid structure over (C_1, C_2). If the depth of the tree is allowed to vary instead, this results in a partitioning in which all leaves are either squares or rectangles with aspect ratios of 1:2; an example of such a tree is shown in the left half of Figure 4.5. (When the bounding box over the domain is a hypercube and split points are always in the middle of the branching variable's current range, having the variables "take turns" splitting achieves the same effect as always splitting on the variable with the widest current range.)

Another branch variable selection method used more commonly in decision and regression tree learning algorithms (e.g. [Qui86] and [BFOS84]) is *greedy* selection. When employing this variable selection strategy, a "density stump" of depth one is grown for each possible branch variable. Each of the stumps is evaluated; the best stump is chosen, its children leaves are thrown away, and the learning algorithm is called recursively to learn subtrees to replace the old child leaves.[1] An example of a tree learned using this greedy strategy is shown in the right half of Figure 4.5.

The greedy branch variable selection method is obviously more computationally expensive than the "taking turns" approach, but sometimes lead to significantly more

[1] Actually, for computational efficiency we pass the best stump's children leaves to the recursively called subtree learners rather than throw them away immediately, so the subtree learners don't have to relearn the leaves when deciding whether to prune.

accurate density estimators, as shown in supplemental experimental results in Appendix A.1.

4.4.2 Split point selection

When a branch node tests a continuous variable X_i, we must choose the threshold value t it employs for its test. If the currently valid range of X_i is $[a, b]$, one simple choice is the midpoint $t = (a+b)/2$. In addition to being computationally inexpensive, this choice has a few other advantages. If the density tree is being used for compression, this means the value of t does not need to be encoded in the model, which saves a few bits. Furthermore, splits in different parts of the density tree will have a greater tendency to "line up" with each other by employing the same thresholds. This can reduce the complexity of the tree that results when we "conditionalize" the original density tree as described in section 4.5.4.

If none of these advantages are of particular concern, another possible split point selection algorithm is as follows:

- Select a set D' of up to, for example, 500 datapoints at random. Sort them according to the values they assign to X_i.
- Generate a set of candidate split thresholds $t^1, \ldots, t^K$ based on these sorted datapoints. For example, we might consider all thresholds that lie halfway between two distinct adjacent values in the sorted list of X_i values.
- Pick the candidate split point t^j that would maximize the log-likelihood of D' if we used a stump with t^j as a split point and constant-density child leaves. If there are l^j datapoints in D' less than t^j and r^j datapoints greater than t^j, this log-likelihood is a constant plus

  $$l^j \log \frac{l^j}{t^j - a} + r^j \log \frac{r^j}{b - t^j}.$$

  These evaluations can be performed efficiently by walking through the sorted list of X_i values and generating the candidate split points t_j from these values "on the fly".

Naturally, we would expect this split-point choosing algorithm to help the most when the density tree is actually employing leaves with constant densities (as described in Section 4.2.1). An analogous algorithm tuned for leaves employing exponential or Gaussian densities (as in Section 4.2.3) would also be feasible, but we refrain from examining this possibility further in this thesis. An analogous algorithm tuned for leaves employing linear or multilinear interpolation would likely be too computationally expensive, however, since these leaf densities cannot be fit using small sets of sufficient statistics that can be updated quickly while scanning through the sorted list of X_i values.

One potential pitfall with this split point criterion is that it tends to favor "end-cut" splits — that is, splits near the boundaries of leaves. This phenomenon has been noted before in algorithms for tree-based classification and regression (see, e.g., [MM73] and [BFOS84]). As a crude way of dealing with this problem, we refrain from using any split point such that one of the two leaves would account for less than 10 of the datapoints. (If there are fewer than 20 datapoints, we refrain from using this split point criterion entirely and simply use the midpoint of the branch variable's current range.) Informal experiments not described further in this thesis have shown this can significantly increase the accuracy of density trees learned while using this split point criterion.

Most of our experiments in this thesis will use the simpler midpoint threshold method. Supplemental experiments comparing this method with the more complicated method described above are described in Appendix A.1.

4.4.3 Pruning strategies

Assuming we have some method for choosing variables on which to branch, we must still decide whether *any* branch will result in a density tree that will perform as accurately on unseen data as a simple leaf distribution would perform. One possible method, which we refer to as *stopping*, is to have the learning algorithm return a leaf whenever it determines (via evaluation on a holdout set or some other method) that a leaf models the current data subset more accurately than any single-level density stump it has generated and tested. Another possible method, referred to as *post-pruning*, is for the learner to learn a subtree potentially much deeper than one level and then compare the estimated accuracy of this entire subtree to the estimated accuracy of a leaf. When learning this deeper subtree, some other ad-hoc stopping criterion is used, such as requiring a minimum number of datapoints before allowing a branch to be considered.

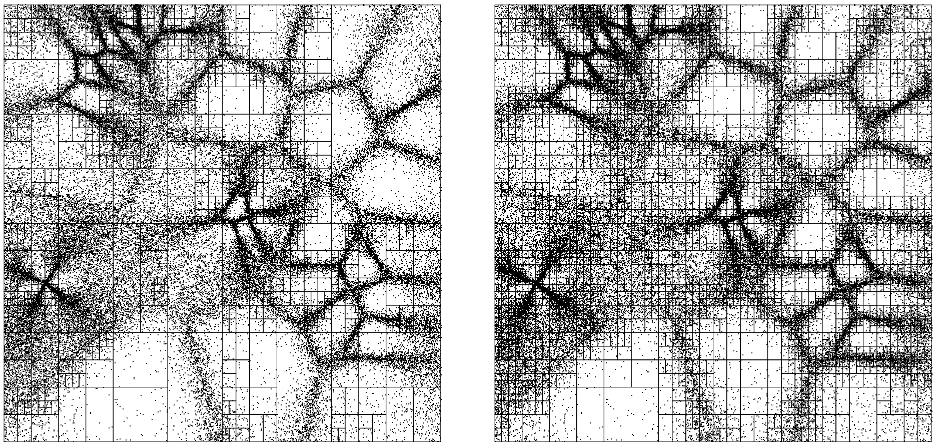


Figure 4.6: Example density trees learned on a synthetic dataset using stopping (left) and post-pruning (right) with constant-density leaves.

Whether stopping or post-pruning generates more accurate results depends on other aspects of the tree learning algorithm. If very simple density estimators are used in the leaves, and if few different combinations of branching variables and branching thresholds are evaluated, then the stopping algorithm will often terminate with a leaf in situations where a density tree of depth two or more would have done much better. For example, Figure 4.6 shows two density trees learned on a synthetic dataset where the leaves are of constant density and only one branch variable/branch treshold combination is attempted. The density tree on the left, which was learned with stopping, has a large region towards the upper-right that is clearly not of constant density but that is modeled with a single leaf. This leaf was not split because it has roughly as many datapoints in its upper half as in its bottom half; by chance, the holdout set had a slightly lower log-likelihood on the candidate density stump employing this top vs. bottom split than it did on the leaf covering the entire area, so the stopping criterion terminated with a leaf prematurely. The density tree on the right learned with post-pruning does not suffer from this obvious problem.

On the other hand, if the density estimators used in the leaves are more flexible, and many candidate branch variable / branch threshold combinations are tried, then it becomes less likely that the stopping algorithm will stop much too early. Furthermore, in such situations it becomes increasingly likely that any finite holdout set used to evaluate different choices of branch variables / branch threshold will happen to have an inaccurately high estimated log-likelihood for one of those choices, and overfitting will result. When overfitting becomes a more pressing issue than underfitting, trees learned with post-pruning often perform slightly worse than trees learned with stopping. This effect can be ameliorated by using one holdout set to evaluate different branch variable / branch threshold combinations and a second separate holdout set to decide whether to use the best of these combinations or to use a leaf instead. However, even with an independent holdout set for pruning, post-pruning can still peform worse than than stopping. Furthermore, learning trees with post-pruning is more computationally expensive, both in terms of time and memory requirements.

Despite these issues, in most of our experiments we will use post-pruning rather than stopping. While post-pruning is often slightly less effective on average, the quality of density trees learned with stopping has a higher variance and is more sensitive to other aspects of the density tree learning algorithm. Supplemental experiments comparing post-pruning versus stopping are included in Appendix A.1.

Another approach previously used in classification and regression trees (see, e.g. [BFOS84]) is to use the holdout set not to directly determine which nodes of the tree to prune, but instead use it to find a good value for a single complexity penalty coefficient that is then used across the entire density tree to determine which branches to prune. This approach might result in more accurate trees than the ones we have produced using the holdout sets more directly, but we leave comparisons along these lines for future research.

4.4.4 Parameter smoothing

Throughout the discussion so far, we have been assuming the use of maximum-likelihood estimates for $P(L = l)$ (the probability distribution over which leaf l a given datapoint is consistent with) and for $P_l(\vec{S_i}|l)$ (the conditional probability density over a set of variables $\vec{S_i}$ given that the datapoint is consistent with the constraints associated with a given leaf l). However, if we are using log-likelihood as our criterion for density estimator quality, such maximum-likelihood estimates can perform arbitrarily poorly on data not seen during the training process. For example, it may be the case that in one of the tree's branches on a discrete variable X, none of the datapoints consistent with the constraints of that branch's ancestors in the tree has X set to some particular value x. In such a situation, a maximum-likelihood tree-learning algorithm would set the branch's corresponding child node to a leaf l and assign $P(l) = 0$. However, there is still a chance that a datapoint consistent with

l's constraints will be seen later; the log-likelihood of such a point would be $-\infty$, thus making it irrelevant how well the density tree did on any other datapoints being evaluated. Alternatively, a leaf l may assign a probability $P_l(\vec{S_i}|l) = 0$ to some combinations of values for $\vec{S_i}$. For example, in the case of linear or multilinear interpolation, it is often the case that the density at some of the leaf's edges or corners will converge to zero.

One theoretical way to address this problem would be to use a Bayesian analysis in which the set of parameters $\vec{\Theta}$ in a density tree with a fixed structure T are given a prior distribution $P_T(\vec{\Theta})$. The data D could then be used to find these parameters' posterior distribution $P_T(\vec{\Theta}|D)$, and then the probabilility of any given datapoint $P_T(\vec{s_i}|D)$ could be calculated by integrating over $P_T(\vec{\Theta}|D)$:

$$P_T(\vec{s_i}|D) = \int P_T(\vec{\theta}|D) \cdot P_T(\vec{s_i}|\vec{\theta}) d\vec{\theta}.$$

When the distributions $P_T(\vec{s_i}|\vec{\theta})$ and priors $P_T(\vec{\Theta})$ are of certain forms, the above integral can be calculated in closed form. For example, for a single discrete variable Q, the integral can be evaluated in closed form if $P_T(Q|\Theta)$ is a multinomial distribution and the prior over its parameters $P_T(\vec{\Theta})$ is a Dirichlet distribution. It can similarly be evaluated if $P_T(\vec{\Theta})$ is a Gaussian distribution and the priors over its parameters is a normal-Wishart distribution. However, it is not clear whether some of the leaf density estimators examined here (such as the linear and multilinear density estimators) are amenable to this form of analysis. Instead, we rely on a commonly used and much simpler technique for working around the problems with maximum-likelihood estimation: namely, we adjust the distribution slightly towards the uniform distribution in an *ad hoc* manner.

One possible smoothing method is to simply learn a maximum-likelihood density tree $P_T(\vec{S_i})$ on the training data and then let the final estimated distribution $P'(\vec{S_i})$ be a mixture model

$$P'(\vec{S_i}) = (1-\alpha)P_T(\vec{S_i}) + \alpha P_U(\vec{S_i})$$

where $P_U(\vec{S_i})$ is a "slack" distribution that assigns nonzero probabilities to all possible values of $\vec{S_i}$. If a bounding box is known *a priori* for the continuous variables $\vec{C_i} \in \vec{S_i}$, then $P_U(\vec{C_i})$ can be a uniform distribution assigning equal probability densities to all points lying within that bounding box. In our experiments, will we assume such bounding boxes are known; when peforming comparative experiments with real-life datasets, we will "cheat" and generate these bounding boxes using *all* the data rather than just the training set. However, this is only done for convenience; one could always model $P_U(\vec{C}_i)$ with a wide Gaussian or Cauchy distribution instead, where the scale for each variable could be set according to the range that variable's values take on in the training data. In our experiments we will generally set α to $\frac{1}{2|D'|}$, where $|D'|$ is the number of datapoints used to train the density tree. The performance of the density estimators appears fairly insensitive to α as long as α is set within an order of magnitude or so of this heuristically chosen value.

Another possible smoothing method is to smooth $P(L)$ and each $P_l(\vec{S}_i|l)$. The maximum-likelihood estimate for $P(l)$ is the fraction of datapoints consistent with l's constraints. Let $a^j(l)$ denote the j^{th} ancestor node of l in the density tree — that is, $a^0(l) = l$, $a^1(l)$ the immediate parent of l, $a^2(l)$ the parent of the parent of l, and so forth, up to $a^d(l)$, where d is l's depth in the tree. Then $P(l)$ can also be expressed as

$$P(l) = \prod_{j=0}^{d-1} \frac{|D_{a^{j+1}(l)}|}{|D_{a^j(l)}|}$$

where $|D_n|$ is the number of datapoints consistent with the constraints associated with a node n in the tree. We can smooth $P(L)$ by smoothing each of the fractions in this product:

$$P(l) = \prod_{j=0}^{d-1} \frac{|D_{a^{j+1}(l)}| + \alpha}{|D_{a^j(l)}| + \alpha\kappa(a^{j+1}(l))}$$

where $\kappa(n)$ is the number of children of a given branch node n. That is, we essentially pretend that at each branch node in the tree, some small additional number α of "phantom datapoints" are consistent with each of the node's children. (In our experiments we generally set α to 0.5; again, the actual value used appears to have little impact on the performance of the resulting density estimators as long as it is within an order of magnitude or so 0.5.) The method for smoothing $P_l(\vec{S}_i|l)$ depends on the particular density estimator being used. For a discrete variable Q, we can simply smooth the maximum-likelihood distribution by assuming the existence of α "phantom datapoints" consistent with each possible value of Q. Constant-density continuous distributions do not need to be smoothed. Exponential and linear densities can be smoothed by adding "phantom datapoints" located at the center of the leaf's bounding box. A Gaussian distribution can be smoothed by averaging into its mean vector and covariance matrix the effects of "phantom datapoints" distributed according to a Gaussian with a standard deviation proportional to the leaf's width and a mean lying in the center of the leaf.

There is no theoretically compelling reason to smooth $P(L)$ and the $P_l(\vec{S}_i|l)$'s if the mixture-model method of smoothing is also already being employed, or vice versa; informal experiments seem to indicate that it makes little difference which is used, as long as one or both are. However, smoothing $P(L)$ and the $P_l(\vec{S}_i|l)$'s does have the pleasant side effect of removing the need for many annoying special-case checks in the implementation. We will generally employ both methods of smoothing for the experiments in this thesis.

4.5 Conditional density trees

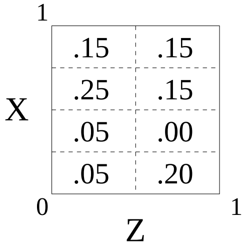
Now that we have discussed in detail how density trees can be learned and used for joint probability distributions $P(\vec{S}_i)$, we move on to discuss learning and using density trees for *conditional* distributions $P(X_i|\vec{\Pi}_i)$.

One might attempt to alter the algorithms discussed in the previous section so that only conditional distributions $P_l(X_i|\vec{\Pi}_i, l)$ are modeled in the tree's leaves, and so that the conditional log-likelihood of the datapoints is used as the criteria for determining the structure of the tree. However, this immediately raises the question of whether such density trees should be allowed to contain branch nodes that test the value of X_i. If such branches are not allowed, then the resulting density tree may not be able to represent the conditional distribution accurately, assuming the leaf distributions are restricted to simple parametric forms. On the other hand, if such branches are allowed at arbitrary points in the tree, then learning a tree that represents a valid conditional probability distribution becomes difficult. In order for the density tree to represent a valid conditional probability distribution, it must be the case that

$$\int P(X_i|\vec{\pi_i})dX_i = 1$$

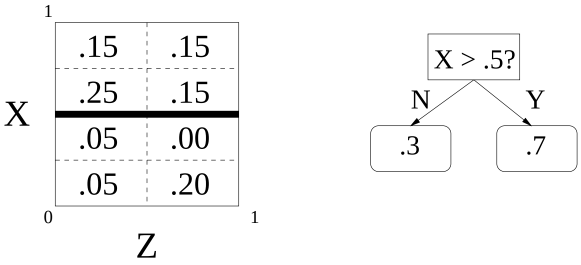
for all possible values of $\vec{\pi_i}$. Unfortunately, it appears that ensuring this constraint is satisfied requires us either to impose severe restrictions on the accuracy of the density estimator or to reason about the structures of different subtrees simultaneously, thus destroying the divide-and-conquer nature of the learning algorithm.

To see this, consider the following example in which we attempt to learn a conditional density tree $P(X|Z)$ where X and Z are both real-valued with values between 0 and 1. For simplicity, assume we only consider branches that split on $X \in \{.25, .5, .75\}$ or $Z = .5$, and that the leaves are of constant density. The finest possible density tree obeying these constraints is a simple discretization of the $X \cdot Z$ space into 8 buckets. Suppose the training data had the following joint distribution over these 8 buckets:



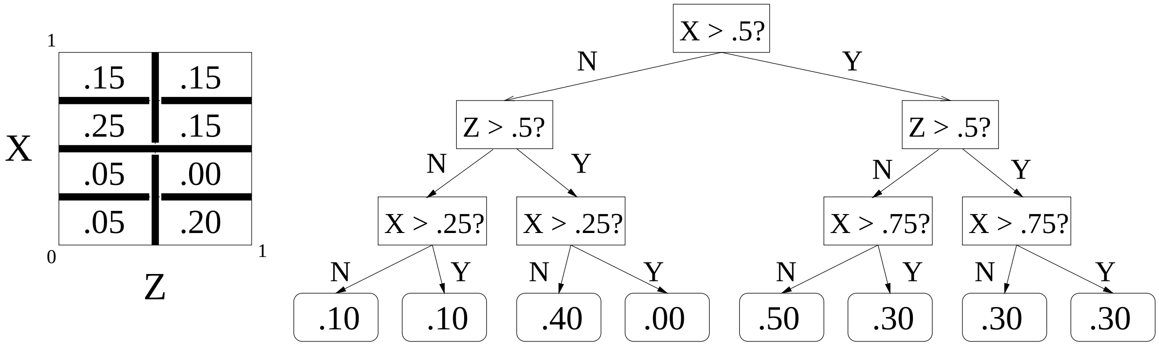
.15	.15
.25	.15
.05	.00
.05	.20

The density tree with a root node splitting on $X = .5$ represents a partitioning of these 8 buckets into two sets of four. This partitioning is shown to the left, and the structure of the corresponding maximum-likelihood conditional density tree is shown to the right:



Here in each of the density tree's leaves we have written not the conditional probability density $P(X|Z)$ but the total conditional probability *mass* contained in the leaf: namely, the integral of the conditional probability density $P(X|Z)$ over the range of X in the leaf.

Now suppose we refine each branch of this tree by splitting on $Z = .5$, and then split each of the resulting new nodes on $X = .25$ or $X = .75$. The corresponding partitioning and maximum-likelihood conditional density tree would look like this:

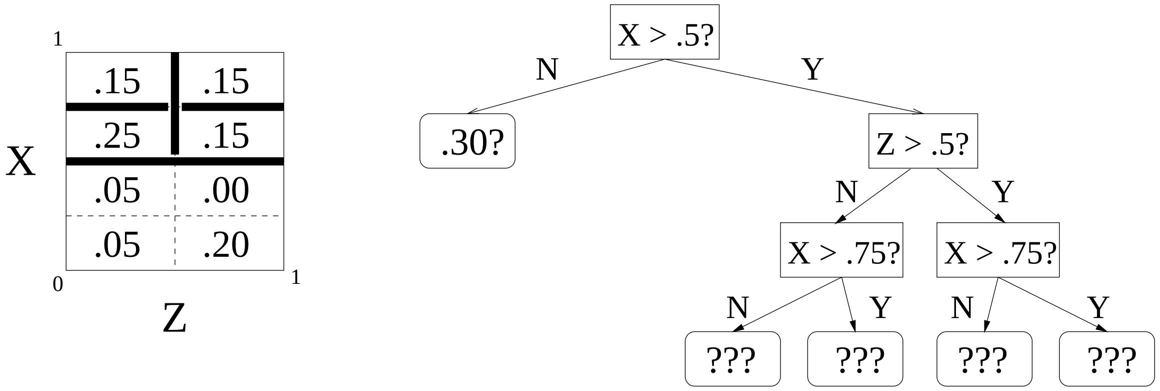


Here each leaf's *conditional* probability mass is two times the fraction of the data lying in the corresponding bucket, since half of the data happens to have $X > .5$ and half of the data does not. So, for example,

$$P(.5 < X < .75|0 < Z < .5) = \frac{P(.5 < X < .75, 0 < Z < .5)}{P(0 < Z < .5)}$$

$$= \frac{.25}{.05 + .05 + .25 + .15} = \frac{.25}{.5} = .5.$$

However, suppose that after evaluating this refined tree we decide we don't actually have enough data to justify the $Z > .5$ split in the $X < .5$ half of the tree, and instead use a density tree structure like this:



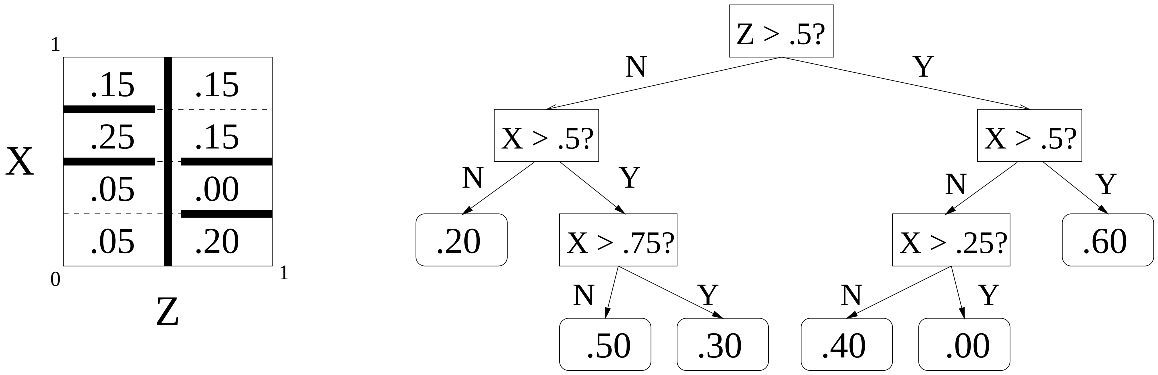
In such a situation, how would we compute, say, $P(.5 < X < .75|0 < Z < .5)$ and $P(.75 < X < 1|0 < Z < .5)$? If we naively compute them using the same equation as before while leaving the leaf for $P(X < .5)$ at 0.3 , then the resulting density tree will not model a valid conditional distribution, since

$$P(X < .5|0 < Z < 1)+P(.5 < X < .75|0 < Z < .5)+P(.75 < X < 1|0 < Z < .5) \neq 1.$$

Thus, changing the structure of the left-hand half of the tree would require us to alter the leaf values in the right-hand half of the tree as well: the divide-and-conquer algorithm that worked for joint density trees does not work here for conditional density trees. We could attempt to regain the divide-and-conquer nature of the algorithm by noticing the .3 / .7 probability mass ratio at the root-level split and requiring that $P(0 < X < .5|Z)$ must be .3 and $P(.5 < X < 1|Z)$ must be .7 for all values of Z regardless of further subtree refinements; however, this approach would obviously cause most conditional probability distributions to be unrepresentable regardless of how much data was used during the learning process.

4.5.1 Stratified conditional trees

Most common tree-based learning algorithms such as CART [BFOS84] and ID3 [Qui86] only test the parent (or "input") variables at their branches, and so the problem raised in the previous section is not an issue for them. Each leaf in such trees generally contains a simple parametric distribution of the child (or "output") variables, or even just a point estimate of the child variables in the case of regression. However, there is no reason in principle to stop at a simple parametric distribution for the child variable once the branching on parent variables has finished. Instead, one can employ a *stratified* tree in which any path from the root of the tree to a leaf first passes through a sequence of branch nodes that only test the parent variables, and then through another sequence of branch nodes that only test the child variables. A stratified conditional density tree for the example problem discussed in the previous section might look like this:



where for clarity we have again listed the conditional probability *masses* inside the leaves rather than the conditional probability densities. When the density tree structure is restricted in this fashion, it is simple to account for the constraint that $P(X_i|\vec{\Pi}_i)$ integrated over X must equal 1 for all $\vec{\Pi}_i$, since all the conditional probability mass for any value of $\vec{\Pi}_i$ lies in a single subtree.

As our experimental results will show, allowing such branches on the output variable can result in conditional density trees that are much more accurate than conditional density trees with branches only on the input variables and simple parametric distributions at the leaves. Unfortunately, the problem of searching for good stratified conditional trees is more difficult than the problem of searching for good joint density trees. When learning joint density trees, the performance of the trees is somewhat insensitive to the exact order in which different variables are used in branches; if the wrong variable happens to branched upon at one level, at least it can still be branched upon at the next. This relative insensitivity is what makes greedy algorithms feasible. Stratified conditional density trees, on the other hand, have the constraint that once the output variable is tested in a branch, the input variables can never be tested again in any further branches below that branch. Furthermore, if the density estimators being used in the leaves are particularly simple, such as constant densities, testing a proposed branch by learning a one-level "stump" is a poor approach. For example, suppose we wish to test whether branching on a parent variable Z at the root of the density tree is a good idea. If we grew a one-level stump with constant-density leaves, each of these leaves would still have a *conditional* probability mass of 1, and the conditional log-likelihood of the data would be precisely the same as if we had not branched on Z at all. It is only after further branching on X_i that the usefulness of branching or not branching on Z at the top of the tree can be gauged with any accuracy.

Therefore, when learning stratified conditional density trees, we take the following approach. We use a recursive greedy top-down learning algorithm that is the same as the learning algorithm used for joint density trees, except:

- It is only allowed to test the "parent" or input variables in its branch nodes.
- Wherever the joint density tree learner would call a subroutine to learn a leaf, the stratified conditional density tree learner instead calls a subtree learner. This subtree learner is restricted to trees that branch only on the output variable X_i; the leaves of this tree contain conditional probability densities rather than joint probability densities.
- Conditional log-likelihoods rather than joint log-likelihoods are used when judging the quality of proposed subtrees.

Because entire subtrees are generated by the stratified tree learner where the joint density tree learner only had to generate leaves, the stratified tree learner is significantly more computationally expensive. There may be other less expensive algorithms for learning stratified trees that are nearly as accurate; however, the algorithm presented here is designed primarily to provide a rough experimental upper bound for how accurate we might expect stratified density trees to be.

An example of a conditional density tree learned on a synthetic two-dimensional dataset is shown in Figure 4.7.

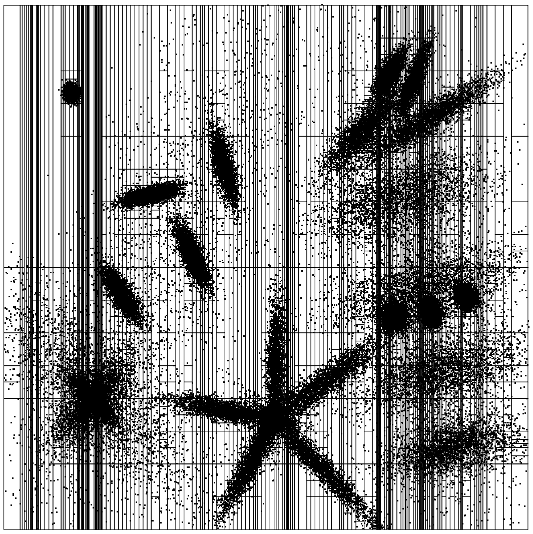


Figure 4.7: An example stratified tree learned to model the conditional distribution of the vertical-axis variable given the horizontal-axis variable.

When the output variable is discrete, it makes no real difference whether the output variable is ever tested in a branch node. The leaf density estimators we use in this thesis all use multinomial distributions for the discrete variables, and each branch on a discrete variables has one child for every possible value of that variable. Replacing a leaf with a branch on the output variable would therefore merely move the information previously contained in the old leaf's multinomial distribution to the conditional probability masses recorded for the new branch's new child leaves. In this case, the stratified conditional density tree learning algorithm becomes very similar to classical decision tree learning algorithms such as ID3 [Qui86]. ID3 uses the *information gain* between each candidate branch test and the class variable to greedily learn the tree structure. This information gain is directly proportional to the increase in the conditional log-likelihood of the training data that would be achieved by performing the same split. Our algorithm also uses an increase in log-likelihood as its criterion, although it may evaluate this increase only on a subset of the training data that was not used to fit the leaves' parameters.

4.5.2 Using joint density trees conditionally

While the stratified conditional density trees discussed in the previous section can model conditional density trees much more accurately than CART-like single-level conditional density trees, they are computationally expensive to learn. Furthermore, as we shall see later in the experimental results, their accuracy can still be improved on significantly.

In this section we discuss the use of density trees modeling joint distributions $P(\vec{S}_i)$ to obtain conditional density estimates $P(X_i|\vec{\Pi}_i)$. Assuming we have a density tree for $P(\vec{S}_i)$, we can obtain an estimate for a particular $P(x_i|\vec{\pi}_i)$ as follows:

$$\begin{aligned} P(x_i|\vec{\pi}_i) &= \sum_l P(l|\vec{\pi}_i) \cdot P(x_i|\vec{\pi}_i, l) \\ &= \sum_l \frac{P(l) \cdot P(\vec{\pi}_i|l)}{\sum_{l'} P(l') \cdot P(\vec{\pi}_i|l)} \cdot P(x_i|\vec{\pi}_i, l) \\ &= \frac{P(l_c) \cdot P(\vec{\pi}_i|l_c) \cdot P(x_i|\vec{\pi}_i, l_c)}{\sum_{l'} P(l') \cdot P(\vec{\pi}_i|l')} \end{aligned}$$

where the summation over l collapses to a single leaf l_c consistent with both x_i and $\vec{\pi}_i$, since all other leaves l have either $P(\vec{\pi}_i|l)$ or $P(x_i|\vec{\pi}_i, l)$ equal to zero. This equation gives us a simple way of calculating conditional distributions $P(X_i|\vec{\Pi}_i)$ from trees modeling joint distributions $P(X_i, \vec{\Pi}_i)$, assuming the distribution $P(X_i, \vec{\Pi}_i|l)$ within each leaf l can be marginalized to compute $P(\vec{\Pi}_i|L)$ and conditionalized to compute $P(X_i|\vec{\Pi}_i, L)$.

Joint density trees are trivially capable of representing Bayes classifiers when used in this manner. In particular, since each leaf in the density trees employed in this thesis models all discrete variables independently, a Naive Bayes classifier for discrete variables is obtained in the special case where the density tree is a one-level density stump with a root node branching on the variable to be predicted. Such Naive Bayes classifiers have previously been used to model the conditional distributions within Bayesian networks [HM97a]. A commonly used Bayes classifier for continuous variables is to model each class distribution with a Gaussian; this classifier is obtained simply with a density stump branching on the class variable with leaves employing Gaussian distributions over the continuous variables, as discussed in section 4.2.2. More generally, suppose a joint density tree over discrete variables has a branch structure similar to the branch structure of a two-way conditional density tree (as discussed in section 4.5.1): that is, once the output variable is tested in a branch node, no further tests can be performed on the input variables in subsequent levels of the tree. When this joint density tree is used to estimate conditional distributions for the output variable, it is similar in form and function to a hybrid decision tree / Naive Bayes classifier also developed in previous research [Koh96]. In the most general case when the tree has an arbitrary branch structure (and the variables are not necessarily discrete), the algorithm for computing conditional distributions essentially creates a Bayes classifier "on the fly" across different parts of the tree to determine which of the leaves consistent with $\vec{\pi_i}$ the datapoint probably came from.

For any given $\vec{\pi_i}$, most leaves l' in the tree will impose constraints on $\vec{\Pi_i}$ that $\vec{\pi_i}$ fails to meet. Thus, most of the terms in the denominator of the last equation above are zero, and the corresponding leaves can be omitted from the summation. This can be accomplished by performing the summation during a depth-first traversal of the tree in which subtrees that impose constraints inconsistent with $\vec{\pi_i}$ are ignored. However, it may still be the case that many leaves in the tree are consistent with $\vec{\pi_i}$. This problem tends to be worst when there are *few* parent variables, since these trees have a larger fracion of branches on the child variable, and the summation algorithm must recurse on all of the children of such branch nodes rather than just the single child consistent with $\vec{\pi_i}$.

If the class of density functions used in the leaves is closed under addition and scalar multiplication, then we can take a density tree modeling $P(X_i, \vec{\Pi_i})$ and precompute a *marginalized* density tree $P(\vec{\Pi_i})$. Such a marginalization algorithm for density trees with constant-density leaves has been used in previous work by Kozlov and Koller on message-passing algorithms for inference in continuous-variable graphical models [KK97]. Once this tree is computed, we can compute the conditional distribution simply as

$$P(X_i|\vec{\Pi_i}) = \frac{P(X_i, \vec{\Pi_i})}{P(\vec{\Pi_i})},$$

where computing the numerator and evaluating the denominator each require locating and evaluating only one leaf distribution in the appropriate tree. Unfortunately, many types of leaf density estimators examined in this thesis are not closed under addition, including the factorized distributions for multiple discrete variables and the exponential, Gaussian, and factorized linear distributions for continuous variables. Furthermore, for some operations we might wish to perform with the density trees, such as sampling or compression, being able to compute $P(X_i|\vec{\Pi_i})$ as a quotient of two black-box functions is not particularly helpful; such operations are much more naturally computed in terms of leaf probabilities $P(L|\vec{\Pi}_i)$ and leaf-dependent conditional probabilities $P(X_i|L, \vec{\Pi}_i)$.

4.5.3 Speeding up the conditional evaluation of joint density trees

However, we can still speed up the evaluation of conditional probabilities a bit by generating an auxiliary "skeleton" marginalized tree from the original density tree. All branches in this skeleton marginalized tree are on the parent variables $\vec{\Pi}_i$. There is one leaf in this tree for every possible distinct *combination* of leaves in the original joint density tree that can be simultaneously consistent with any fixed value $\vec{\pi}_i$. Each leaf in the skeleton marginalized tree contains a vector of pointers to all the original leaves that are consistent with its constraints on $\vec{\Pi}_i$. (If the original leaves do not contain explicit records of the constraints over X_i imposed on them by branch nodes above them in the tree, these constraint sets are recorded in the marginalized tree leaf's vector along with the corresponding pointers to the original leaves.) This vector of pointers allows us to compute the necessary conditional distribution more quickly by preventing us from having to traverse the original density tree in order to find all the necessary leaves. An example of a skeleton marginalized tree is shown in Figure 4.8.

The branch structure of this skeleton marginalized tree can be created with an algorithm similar to that used previously in the marginalization of density trees with constant-density leaves [KK97]. We first define a *fracturing* procedure that destructively refines one tree structure T_t so that no leaf in T_t simultaneously intersects more than one distinct leaf in some other tree structure T_s, where two nodes are said to intersect if there exists some $(x_i, \vec{\pi}_i)$ that is consistent with the constraints associated with both nodes. (No two leaves in the same density tree intersect.) The recursive FRACTURE routine described in Figure 4.9 takes the root nodes n_t and n_s of two trees T_t and T_s and returns the root of the destructively refined T_t. All "leaves" in T_t are simply placeholders that contain no information other than the constraints imposed on them by their ancestors in the tree; these placeholder leaves will be replaced later. Since T_t is only refined — that is, no two of its leaves are ever joined — it is also the case that no leaf in the result intersects more than one leaf of the original T_t. Thus, FRACTURE(n_t, n_s) is symmetric in that it returns a tree with the same set of leaves that FRACTURE(n_s, n_t) would; our particular implementation of

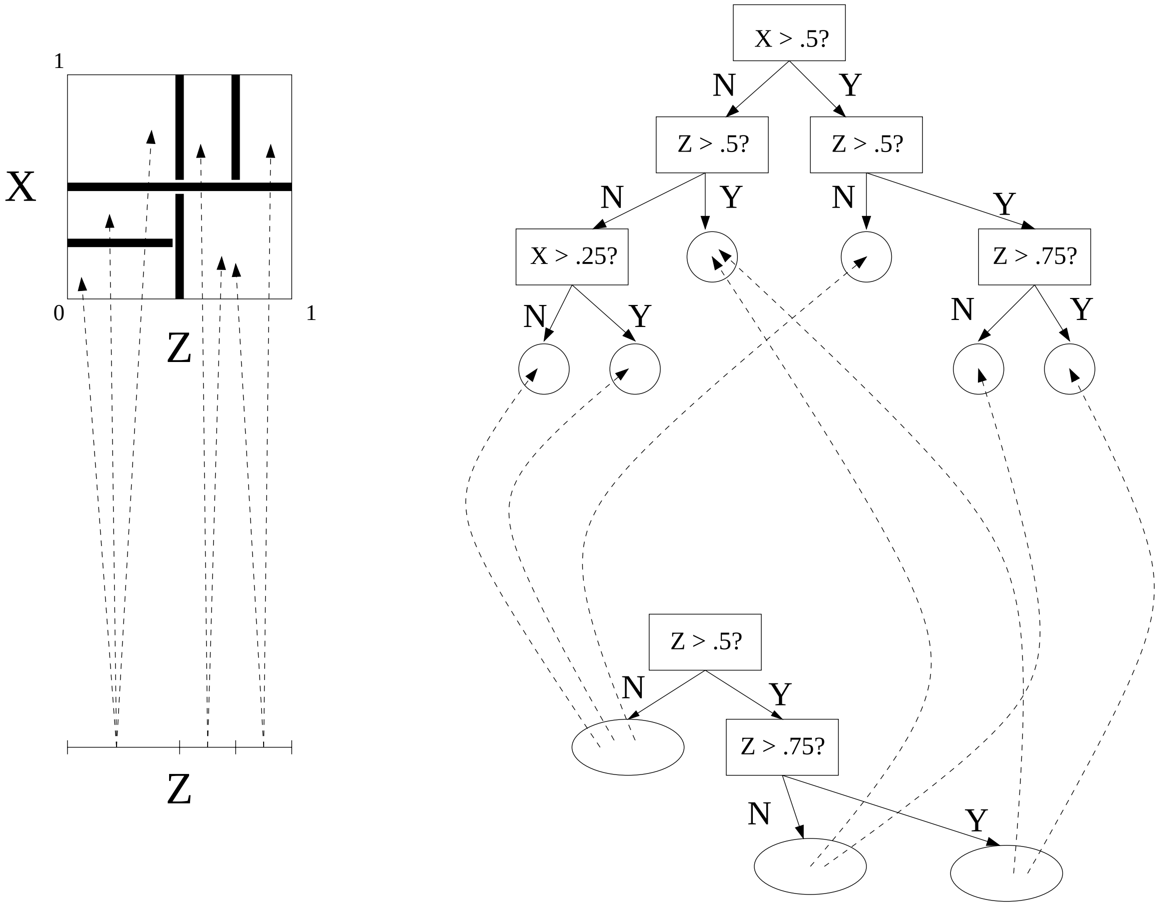


Figure 4.8: An example of a density tree and its skeleton marginalized density tree. Geometrical representations of the trees are shown to the left; to the right are their tree-based representations. The top half of each representation shows the original density tree; the bottom half show the corresponding marginalized density tree. The leaves of the marginalized density tree contains pointers back up to the leaves in the original density tree.

FRACTURE merely happens to destroy one of its two argument trees because the algorithm is slightly more straightforward to describe and implement that way.

With FRACTURE defined, we can now define COLLAPSE, a recursive procedure that takes the root of a density tree T and a variable X_i as arguments, and returns a new tree structure representing the marginalization of X_i out of T. As in FRACTURE, the leaves in this new tree are merely placeholders to be filled in later; COLLAPSE merely generates the new tree's branch structure. Pseudocode for COLLAPSE appears in Figure 4.10.

Once the structure of the skeleton marginalized tree has been generated, the leaves are filled in so that each leaf contains an array of pointers to all the leaves in the original tree that are consistent with its constraints on $\vec{\Pi}_i$. We omit the details; in the next section we will describe a related algorithm where this array is replaced with a subtree with branches on X_i.

4.5.4 Approximate conditional evaluation of joint trees

Unfortunately, while the skeleton marginalized trees described above can speed up evaluation by providing a convenient set of pointers to all the necessary leaves, evaluating the sum $\sum_{l'} P(l')P(\vec{\pi_i}|l')$ may still involve an expensively large number of terms, particularly when there are few parent variables. We can speed up the conditional evaluation of joint density trees further by introducing an approximation. Within the context of any given leaf l_s of the skeleton marginalized tree, we can approximate the conditional distribution $P(\vec{\Pi}_i|l')$ over each original density tree leaf l' as a constant $\hat{P}_s(\vec{\Pi}_i|l')$ specific to l_s. We compute this constant distribution once and store it within l_s; each $\hat{P}_s(\vec{\Pi}_i|l_t)$ is the average of $P(\vec{\Pi}_i|l_t)$ over all datapoints consistent with l_s's constraints. The conditional density can then be be computed approximately as

$$P(x_i|\vec{\pi_i}) = \frac{P(l_c) \cdot \hat{P}_s(\vec{\pi_i}|l_c)}{\sum_{l'} P(l') \cdot \hat{P}_s(\vec{\pi_i}|l')} \cdot P(x_i|\vec{\pi_i}, l_c) = \alpha_c^s P(x_i|\vec{\pi_i}, l_c)$$

where α_c^s is a constant.

Now that we no longer have to look at any leaves other than l_c in order to compute the conditional density, we need a faster way of finding l_c from l_s than walking through a linear array trying to find the one consistent with a given x_i. A natural choice is to create a subtree with branches testing X_i. This subtree structure can be generated by creating a placeholder leaf with the same constraints as l_s and then replacing it

```
FRACTURE($n_t$, $n_s$):

• If $n_s$ is not a branch node, return $n_t$.
• Otherwise, if $n_t$ is a branch, destructively set each child $n_c$ of $n_t$ to FRACTURE($n_c$, $n_s$) and return $n_t$.
• Otherwise (in the case that $n_t$ is a leaf and $n_s$ is a branch), count the number of $n_s$'s children that intersect $n_t$. (This number will be nonzero.)
    – If there is exactly one child $n_c$ of $n_s$ that intersects $n_t$, then return FRACTURE($n_t$, $n_c$).
    – Otherwise, create a new branch node $n_b$ employing the same branch test as $n_s$. Let $K$ denote the number of $n_s$'s children.
    – For every $i$ between 1 and $K$:
        ∗ Create an empty placeholder leaf whose constraints are the constraints associated with $n_t$ plus the additional constraint imposed by the $i^{th}$ possible result of $n_b$'s branch test. Let this placeholder leaf be denoted $n_i$.
        ∗ Destructively set $n_b$'s $i^{th}$ child to FRACTURE($n_i$, the $i^{th}$ child of $n_s$).
    – Return $n_b$.
```

Figure 4.9: Pseudocode for the FRACTURE procedure. The tree with root n_t is destructively modified so that none of its leaves intersect more than one leaf in n_s, and this modified tree's root is returned.

```
COLLAPSE($n_t$, $X_i$):
  • If $n_t$ is not a branch node, return a placeholder leaf.
  • Otherwise:
      – Let $K$ denote the number of $n_t$'s children.
      – For all $i$ from 1 to $K$:
          ∗ Let $m_i$ be COLLAPSE($n_t$'s $i^{th}$ child, $X_i$).
      – If $n_t$ branches on a variable other than $X_i$, return a new branch node
        employing the same branch test as $n_t$, but with $m_1, \ldots, m_K$ as its children.
      – Otherwise:
          ∗ For all $i$ from 2 to $K$:
              · Destructively set $m_1$ to FRACTURE($m_1$, $m_i$).
              · Delete $m_i$.
        Return $m_1$.
```

Figure 4.10: Pseudocode for the COLLAPSE procedure. The routine returns a new version of the tree rooted at n_t in which all branches on X_i have been marginalized away. The resulting tree has one leaf for every distinct possible combination of leaves in the original tree that can be consistent with a fixed $\vec{\pi_i}$.

with FRACTURE(l_s, n_t), where n_t is the root of the original density tree. Each leaf of this subtree is then assigned a pointer to the single leaf in the original density tree consistent with its constraints. When this operation has been performed for each leaf in the skeleton marginalized tree, the result is very similar in structure to the stratified conditional tree described in section 4.5.1. We refer to this new kind of tree as a *conditionalized joint density tree.* An example is shown in figure 4.11. These trees can be used either as "skeleton marginalized trees" to slightly speed up the computation of the *exact* conditional distribution from the original joint density tree

$$P(x_i|\vec{\pi_i}) = \frac{P(l_c) \cdot P(\vec{\pi_i}|l_c) \cdot P(x_i|\vec{\pi_i}, l_c)}{\sum_{l'} P(l') \cdot P(\vec{\pi_i}|l')}$$

by organizing pointers to all the relevant nodes in the original joint density tree, or to compute the even faster approximate conditional distribution

$$P(x_i|\vec{\pi_i}) = \alpha_c^s P(x_i|\vec{\pi_i}, l_c),$$

as time constraints require.

There are several notable differences between stratified conditional trees and conditionalized joint density trees, however. The structures of stratified conditional density trees are optimized directly for maximimizing the total conditional log-likelihood of the data rather than the joint. Conceptually, one would expect this to make stratified conditional density trees more accurate at estimating conditional densities. However, searching for a good stratified conditional density tree is more computationally expensive for the reasons described in section 4.5.1. Furthermore, the structures of joint density trees are more flexible, allowing them to conform faithfully to the regions in which there are many training datapoints without breaking other low-density regions into too many leaves. For example, consider figure 4.11. In order for a stratified conditional tree to create a split on $Z > .75$ in the $(X > .5, Z > .5)$ region where there might be plenty of data, this same split must be applied across *all* values of X, including the potentially much lower-density region $(X \leq .5, Z > .5)$. The joint density tree is not as inflexible in this respect. While the structure of a conditionalized joint tree is similar to that of a stratified conditonal tree, each of its leaves is a pointer to a leaf in the joint density tree that may be have trained on a significantly larger set of data than a corresponding leaf of a stratified conditional tree would have been. For example, the conditionalized joint tree leaf corresponding to the region $(Z > .75, X < .5)$ is a pointer to the joint tree leaf which was trained on all data in the larger region $(Z > .5, X < .5)$. This added flexibility may help conditional joint density trees compensate for the fact that they are optimized to model joint distributions rather than conditional ones.

Finally, if each leaf of the original joint tree employs a nonuniform distribution over the parent variables, then obtaining the conditional distribution $P(X_i|\vec{\pi_i})$ from a joint tree using the relationship

$$P(x_i|\vec{\pi_i}) = \sum_l P(l|\vec{\pi_i}) \cdot P(x_i|\vec{\pi_i}, l)$$

can result in more accurate density estimation than would be possible by simply using the conditional distribution of a single stratified density tree leaf, *even if the joint tree is structured like a stratified conditional density tree* — that is, with all branching on $\vec{\Pi_i}$ performed before any branching on X_i. Intuitively, by combining the distributions learned in different leaves using this relationship, we have essentially created a *"soft branch"* over $\vec{\Pi_i}$ that helps us to more accurately predict X_i as a function of $\vec{\Pi_i}$ without actually splitting the dataset further into completely disjoint subsets.

4.6 Structure-learning algorithm for Bayesian Networks using conditional density trees

Most previous algorithms for learning Bayesian networks over continuous variables have taken one of the following approaches:

1. Employ simple parametric distributions such as Gaussians that have easily computable sufficient statistics; search directly over Bayesian networks employing these continuous distributions (e.g. [HG95]). This approach has the obvious drawback that the networks learned may be inaccurate when the data does not obey the assumptions behind the model's parametric forms.

2. Search for a network structure that accurately models a version of the dataset in which each variable is independently quantized; then, use this same structure for a Bayesian network modeling the original continuous variables (e.g. [MC98b]). This approach has the disadvantage that the discretization process may cause some intervariable dependencies in the continuous data to be lost, and may add spurious dependencies. Furthermore, the structure-learning procedure does not take into account the representational power of the particular continuous

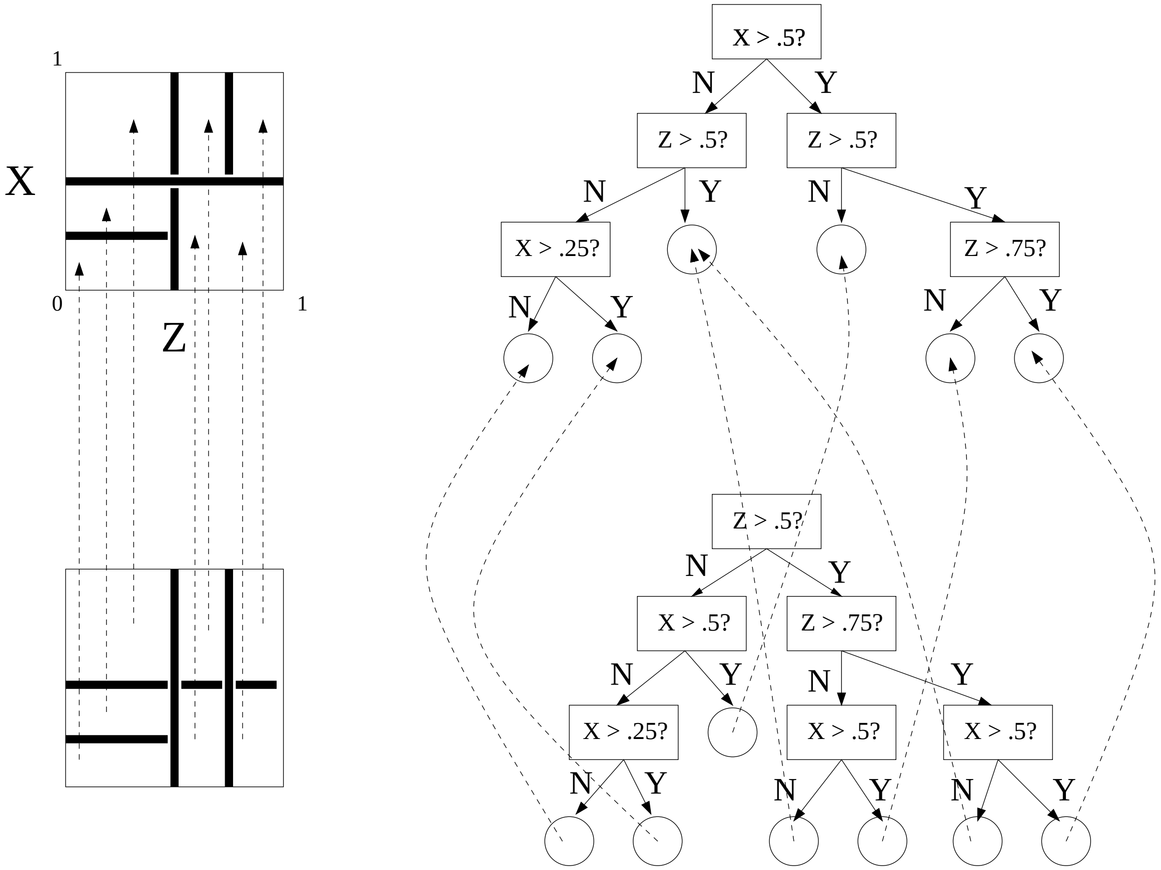


Figure 4.11: An example of a conditionalized joint density tree. A geometrical representation of the tree is shown to the left; to the right is its tree-based representation. The top half of each representation shows the original density tree; the bottom half show the auxiliary tree used to evaluate conditional densities. Each leaf of the auxiliary tree contains a pointer back up a single leaf in the original density tree.

distributions that will be used in the final network — for example, how many parent variables can be used before too many datapoints would be required to learn the continuous distribution.

3. Independently quantize the variables as in approach 2, but optimize the quantization so that the quantized variables predict the hidden class variable of a mixture model learned on the continuous variables [MC99]. This approach ameliorates some of the disadvantages of approach 2, but at the computational cost of learning a joint mixture model over all the variables.

4. Perform a simultaneous search over discretization policies and networks that model the corresponding discretized variables ([MC98a], [FG96a]). Because the discretization policy takes into account the particular variable interactions being modeled in the network, fewer dependencies in the original data are lost and fewer spurious dependencies are generated. However, this problem does not completely go away; additionally, as in the previous approach, this approach does not take into account the complexity of the particular models one might have in mind for the final network over the original continuous variables.

5. Perform a simple greedy structure search over networks that employ complex continuous distributions, as in section 3.3. This approach has the disadvantage that the greedy search may be inadequate to find a good network in some domains, particularly those in which networks employing incorrect variable orderings require many more arcs than networks with correct variable orderings.

6. Perform an extensive structure search directly over networks that employ complex continuous distributions ([HT95], [FN00]). This method is computationally tractable only in domains with relatively small numbers of variables and/or datapoints when each continuous distribution required during the search is time-consuming to learn.

In this section we use the speed with which conditional density trees can be learned to examine hybrid structure-learning algorithms that attempt to combine the best aspects of some of these approaches. We generalize the greedy learning algorithm described in section 3.3 in several ways:

- The greedy algorithm may be started from an arbitrary network structure B_0 rather than an empty network structure B_ϵ. In particular, it may be useful to start it from a network structure that was learned by a more extensive search procedure on a discretized version of the dataset. If the more extensive search is able to identify roughly the right order for the variables in the network and/or good parent sets that are difficult to find greedily, then this may be a significantly better starting point for the greedy algorithm to begin finding better continuous-distribution networks. Alternatively, it may be started with the result of a previous iteration of the same greedy algorithm. When run in this fashion, the overall algorithm is similar in spirit to the Sparse Candidate algorithm previously developed for discrete domains [FNP99].

- The pairwise score improvements $I(X_i, X_j)$ used in section 3.3 were measured with respect to B_ϵ, and were always scores for arc additions. The pairwise score improvements we use now are with respect to B_0. These pairwise score improvements will generally not be near-symmetric as they were before. Some of these improvements will be for arc deletions rather than additions, and some arc additions may be invalid because they would create cycles in the graph. The pairwise improvements are stored in a list sorted in order of decreasing estimated score improvement; the algorithm runs down this list and attempts the corresponding network structure changes in order.

- The conditional distributions used during the greedy algorithm's search over network structures may not be of the same form as the distributions used in the final network. For example, the greedy search could be performed on a completely discretized dataset, or with density trees that use constant-density leaves; after the search over structures has been completed, the resulting network stucture can be used in conjunction with more complex distributions, such as density trees with leaves employing multilinear interpolation.

We will examine the effects of these generalizations in the experimental results section. Pseudocode for the greedy algorithm we employ throughout this chapter is shown in Figure 4.12.

At the beginning of the greedy algorithm, one function $S_f(X_i, \vec{\Pi}_i)$ is used to come up with a (possibly crude) *ranking* of all possible single-arc changes to B_0. These changes are then attempted in order from most to least promising according to this ranking, subject to the constraints that no variable can have more than MAXPARENTS parents and no more than MAXCHANGES changes to any given variable's

- Given:
  - B_0, an initial network structure.
  - $S_s(X_i, \vec{\Pi}_i)$, a function returning the estimated contribution to network quality that would be achieved by using $\vec{\Pi}_i$ as X_i's parents in a network. This function will generally learn a conditional distribution $P(X_i|\vec{\Pi}_i)$ and estimate its predictive power, usually by evaluating the conditional log-likelihood of a holdout set.
  - $S_f(X_i, \vec{\Pi}_i)$, another function similar to $S_s(X_i, \vec{\Pi}_i)$ but that may potentially learn and evaluate simpler distributions than S_s and thus require less computational time. (S_s is "slow"; S_f is "fast".)
  - MAXCHANGES, a maximum number of changes to attempt on any single variable's parent set.
  - MAXPARS, a maximum number of parents any variable may have.
- Let L be a list in which each element l^u contains a child variable X_c^u, a parent variable X_p^u, and a score s^u. Initialize L to the empty list.
- For each pair of variables X_c and $X_p \neq X_c$:
  - Let $\vec{\Pi}_c(B_0)$ denote the set of X_c's parents in B_0.
  - If $X_p \in \vec{\Pi}_c(B_0)$, let $\vec{\Pi}_c{}' = \vec{\Pi}_c(B_0) - \{X_p\}$; otherwise let $\vec{\Pi}_c{}' = \vec{\Pi}_c(B_0) \cup \{X_p\}$.
  - If changing X_c's parent set in B_0 to $\vec{\Pi}_c{}'$ would not result in a cycle, add an entry l^u to L with $X_c^u = X_c$, $X_p^u = X_p$, and $s^u = S_f(X_c, \vec{\Pi}_c{}') - S_f(X_c, \vec{\Pi}_c)$.
- Sort L according to the scores s_u in descending order.
- Let $B = B_0$. For each variable X_i, compute $S_s(X_i, \vec{\Pi}_i(B))$, where $\vec{\Pi}_i(B)$ denotes the set of X_i's parents in the network structure B, and set CHANGETRIES(X_i) to zero.
- For u from 1 to $|L|$, the length of L:
  - Let X_c and X_p denote the child and parent variables recorded in l^u. If CHANGETRIES(X_c) $>$ MAXCHANGES, skip to the next value of u. Otherwise:
  - Let $\vec{\Pi}_c{}' = \vec{\Pi}_c(B) \cup \{X_p\}$ if B contains no arc from X_p to X_c, or $\vec{\Pi}_c(B) - \{X_p\}$ if B already contains such an arc. If using $\vec{\Pi}_c{}'$ as X_p's parent set in B would create a cycle, or $\vec{\Pi}_c$ has more than MAXPARS variables, skip to the next value of u. Otherwise:
  - Increment CHANGETRIES(X_c) by one. Evaluate $S_s(X_c, \vec{\Pi}_c{}')$; if it is greater than $S_s(X_c, \vec{\Pi}_c(B))$, change X_c's parent set in B to $\vec{\Pi}_c{}'$ (and store $S(X_c, \vec{\Pi}_c{}')$ for future reference).
- Return B.

Figure 4.12: The general form of the greedy structure-learning algorithm employed in this section.

parent set are attempted. A second function $S_s(X_i, \vec{\Pi}_i)$ is used to estimate the quality of these attempted arc changes; this function may be more accurate and more computationally expensive than S_f. As in the similar algorithm used in Section 3.3, the scales of the quality estimates returned by S_s and S_f may be totally different; the only thing that matters is that the rankings of different parent sets as evaluated by S_s should be highly correlated with the rankings of parent sets as evaluated by S_f.

The restriction on the number of parents per variable is used largely for computational reasons. In particular, the amount of time required to learn density tree leaves that use multilinear interpolation grows exponentially with the number of variables, and (as our experiments will show) these are often the most accurate trees to use. Likewise, the rationale for continuing to use the rankings provided by S_f even after the variables' parent sets have been changed from what they were in B_0 is also one of computational efficiency: it may be too computationally expensive to reevaluate an average of $O(N)$ possible further parent-set changes every time an arc is added to or removed from the network. Rather than perform these reevaluations immediately, the algorithm optimistically assumes that the parent-set changes that were most promising in B_0 are still promising even after some changes have been made to the network.

This is the same heuristic motivating the Sparse Candidate algorithm [FNP99]. However, the Sparse Candidate algorithm uses this heuristic to precompute sufficient statistics for promising parent sets and then restricts the network structure search to these parent sets. Once these sufficient statistics are computed, it is possible to quickly perform (for example) an exact steepest-ascent hillclimbing search among all network structures employing those promising parent sets. In this thesis, our networks usually employ nonparametric continuous-distribution density estimators rather than discrete contingency tables. (While each individual density tree *leaf* employs a parametric distribution, the number of the leaves can theoretically grow unboundedly with the size of the dataset.) There are therefore no simple sufficient statistics that can be computed; performing exact steepest-ascent hillclimbing in this setting would be just as expensive with a fixed parent set as it is with a more flexible one. The greedy algorithm presented here can be seen as an approximation of steepest-ascent hillclimbing in which approximate and sometimes "out-of-date" estimates are used for which directions in the network-structure search space are steepest.

When the initial network structure B_0 has no arcs, MAXPARENTS is set to 1, and S_s is consistent with S_f, then the greedy algorithm degenerates to a maximum spanning forest algorithm and generates the optimal network structure in which each variable has at most one parent. The greedy algorithm previously described in section 3.3 also has this property; the difference between the two algorithms in this case is that the previously described algorithm is closely related to Prim's algorithm for finding minimum spanning trees, while the algorithm in Figure 4.12 is more closely related to Kruskal's algorithm instead (see, e.g. [CLR90]).

As described in Figure 4.12, the greedy algorithm may try evaluating parent sets with S_s even when the corresponding estimated quality improvements from S_f are negative. If S_s and S_f are identical or extremely similar, it may be more practical to simply skip such cases, since S_f is also likely to indicate that these parent sets are poor. In our experiments in this chapter, however, the algorithm tries such candidate parent sets anyways.

In these experimental results (Section 4.8.5), we will examine the speed and effectiveness of several different methods for ranking single-arc changes (S_f), estimating network quality during the actual greedy search (S_s), and computing final conditional distributions for the network structures learned. As we will see, the ability to use different methods for these different tasks can allow the greedy algorithm to find accurate networks quickly, particularly when the greedy algorithm is applied iteratively in a fashion similar to the Sparse Candidate Algorithm.

4.7 Marginal distribution flattening

In real-world datasets, the marginal distributions of some continuous variables can be quite complex and exhibit sharp features. Modeling the marginal distribution of each of these variables individually is relatively easy if one has enough data. Unfortunately, when several variables are modeled jointly in the same density tree, it becomes difficult to model all variables' marginal distributions accurately at the same time, since each branch on one variable reduces the amount of data from which the distributions of all the other variables are learned in each of the branch's subtrees. In this section we describe a simple data preprocessing trick that can sometimes help alleviate this problem.

Suppose we wish to model a probability density $P(X_k)$ over a one-dimensional continuous variable X_k. Suppose we also have a strictly monotonic function $Y_k(X_k)$. Rather than learn a model of X_k's distribution directly, we can learn a model of $P(Y_k)$. Then, by the fundamental transformation law of probabilities, we can compute

$$P(X_k) = P(Y_k)\left|\frac{dY_k}{dX_k}\right|.$$

If we have a vector of continuous variables $\vec{S}$ we wish to model jointly, and a vector of transformation functions $\vec{Y}(\vec{S})$ with one element for every $X_k \in \vec{S}$, this generalizes to

$$P(\vec{S}) = P(\vec{Y}(\vec{S}))\prod_{k:X_k\in\vec{S}}\left|\frac{dY_k}{dX_k}\right|.$$

If we can learn a vector of transformation functions $\vec{Y}(\vec{S})$ such that modeling $P(\vec{Y})$ is easier than modeling $P(\vec{S})$ directly, and all the individual derivatives $\frac{dY_k}{dX_k}$ are easy to evaluate, then this relationship will allow us to model $P(\vec{S})$ more accurately.

Most of the types of continuous distributions we use in tree leaves as discussed above (namely constant densities, exponential densities, and linearly interpolated densities, but not Gaussian densities) can be used to trivially model constant densities. If we can find a transformation $Y_k(X_k)$ that "flattens" the marginal distribution of each variable — that is, makes it nearly constant — then we might expect this to make modeling joint distributions between the transformed variables easier, since the joint density approximator can then spend less of its representational power learning the variables' marginal distributions and more on the interesting relationships *between* variables.

Fortunately, such a transformation is easy to find. Namely, we need only learn a model marginal distribution $P_{marg}(X_k)$ for each variable X_k, and then let $Y_k(X_k)$ be its *cumulative* distribution

$$Y_k(x_k) = \int_{-\infty}^{x_k} P_{marg}(x'_k)dx'_k.$$

This choice of $Y_k(X_k)$ makes the marginal distribution of Y_k constant to the extent that $P_{marg}(X_k)$ accurately models the data. Furthermore, $\frac{dY_k}{dX_k}$ is simply $P_{marg}(X_k)$.

Note that if all we cared about were the marginal cumulative distributions, we could simply learn them directly rather than learning the marginal densities and then integrating. For example, one trivial learning algorithm for the cumulative distribution $Y_k(x_k)$ would be the fraction of the observed datapoints with $X_k < x_k$, or the *empirical cumulative distribution function.* In addition to being unbiased, the empirical cumulative distribution function also has the minimum possible variance (see, e.g., [Sco92]). However, this estimator would be useless for our purposes: in the end we want a valid probability distribution over $\vec{X}$, and this requires sensible estimates for the marginal probability densities $\frac{dY_k}{dX_k}$. This unbiased choice for estimating $Y_k(x_k)$ would give us an unbiased but extremely high-variance marginal probability density estimator that would be zero everywhere except where the actual datapoints lie, at which points the estimated density would be infinite. Unfortunately, unlike cumulative distributions, there is no single unbiased estimator for density functions that has the minimum possible variance regardless of the distribution being learned [Ros56].

For our marginal density estimates, we use a tree-based density estimator such as described in previous sections to learn each marginal distribution $P_{marg}(X_k)$. Assuming the type of distribution used in each leaf can be analytically integrated, it is simple to transform each one-dimensional density tree $P_{marg}(X_k)$ into an identically structured tree representing the corresponding cumulative distribution $Y_k(X_k)$. To do so, we simply perform a depth-first traversal of the original density tree, making sure branches corresponding to smaller values of X_k are traversed first. Each leaf l of the original density tree representing the function $P^l_{marg}(X_k)$ over the leaf's range $[a_l, b_l]$ becomes a new leaf representing the function

$$Y^l_k(x_k) = C + \int_{a_l}^{x_k} P^l_{marg}(x'_k)dx'_k$$

where C is the sum of the integrated probabilities of all leaves already traversed. If the density tree used to represent $P_{marg}(X_k)$ also has a uniform global "slack" distribution added as described in section 4.4.4, a corresponding global linear term is added to the density tree representing $Y_k(X_k)$.

Figure 4.13 shows a rather pathological two-dimensional distribution exhibited by two variables in the Bio dataset, along with the two-dimensional distribution resulting when these variables are transformed so they have approximately uniform marginal distributions. The original data exhibits strong periodic spikes in the marginal distribution of the variable corresponding to the plot's Y axis, possibly due to a quanitization artifact of some sort that only affects part of the data. These spikes have effectively been removed from the marginal distributions of the transformed version of the data. There are still strong discontinuities in the transformed joint distribution, but these discontinuities exist where the *relationship* between the two variables changes in an interesting fashion. The job of modeling the spikes in the marginal

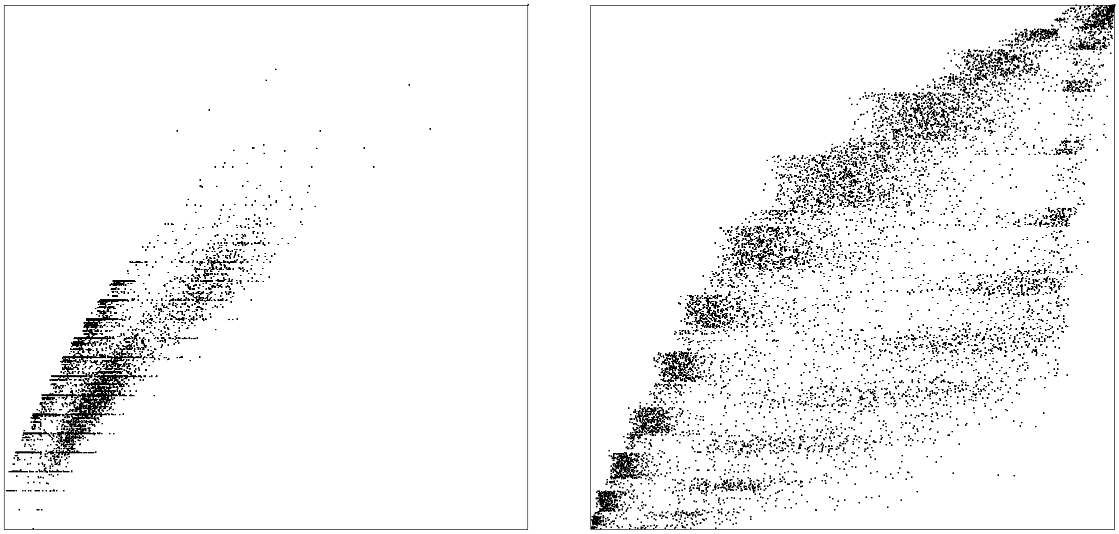


Figure 4.13: A two-dimensional distribution from the Bio dataset: original version (left) and transformed version in which both marginal distributions are approximately uniform (right).

distributions of the variables has largely been taken over by the separate marginal models, allowing the learner of the joint model to concentrate on this relationship.

When using joint density trees conditionally as described in section 4.5, we need to calculate $P(X_i|\vec{\Pi}_i)$. Given the transformation functions for X_i and $\vec{\Pi}_i$, and a joint density tree representing $P(Y_i, \vec{Y}_{\Pi_i})$ (where $\vec{Y}_{\Pi_i}$ is the vector in which each variable $X_k \in \vec{\Pi}_i$ is replaced with $Y_k(X_k)$), this can be done as follows:

$$\begin{aligned} P(X_i|\vec{\Pi}_i) &= \frac{P(X_i, \vec{\Pi}_i)}{P(\vec{\Pi}_i)} \\ &= \frac{P(Y_i, \vec{Y}_{\Pi_i}) \displaystyle\prod_{k:X_k \in \{X_i\} \cup \vec{\Pi}_i} \left|\tfrac{dY_k}{dX_k}\right|}{P(\vec{Y}_{\Pi_i}) \displaystyle\prod_{k:X_k \in \vec{\Pi}_i} \left|\tfrac{dY_k}{dX_k}\right|} \\ &= P(Y_i|\vec{Y}_{\Pi_i}) \cdot \left|\frac{dY_i}{dX_i}\right| \end{aligned}$$

where $P(Y_i|\vec{Y}_{\Pi_i})$ is computed with the density tree representing $P(Y_i, \vec{Y}_{\Pi_i})$ as described in section 4.5.

Note that it is unnecessary to evaluate any of the original density trees $P_{marg}(X_k) =$

$\frac{dY_k}{dX_k}$ for the parent variables $X_k \in \vec{\Pi}_i$. In addition to being computationally convenient, this means inaccuracies in the parent variables' learned marginal distributions do not directly affect the final estimates of $P(X_i|\vec{\Pi}_i)$: only those of $P_{marg}(X_i)$ do. Inaccuracies in the parent variables' learned marginals only harm the accuracy of the final conditional distribution by making the transformed data's marginals imperfectly flat, which might make learning the joint distribution over the transformed variables slightly more difficult to learn than it would be if the marginals were perfectly flat. This is in contrast with what would happen if we managed to completely botch the modeling of $P_{marg}(X_i)$, in which case the final conditional distribution $P(X_i|\vec{\Pi}_i)$ would probably be inaccurate no matter how accurate the joint distribution $P(Y_i|\vec{Y_{\Pi_i}})$ were subsequently modeled.

Now suppose we are searching for a good Bayesian network structure with which to model the data, where each conditional distribution in the network will be modeled using a conditional density tree over data transformed in the manner described above. The probability density the network models over the original N variables will be

$$P(\vec{X}) = \prod_{i=1}^{N} P(Y_i(X_i)|\vec{Y_{\Pi_i}}(\vec{\Pi}_i)) \cdot P_{marg}(X_i).$$

Thus, each datapoint's contribution to the log-likelihood of a given network will be

$$\sum_{i=1}^{N} \log P(Y_i(X_i)|\vec{Y_{\Pi_i}}(\vec{\Pi}_i)) + \sum_{i=1}^{N} \log P_{marg}(X_i).$$

The latter term in this sum is independent of Π_i, i.e. of the network structure. This means that when searching for the best network structure with which to model the original data, we can simply:

- Learn one marginal distribution $P_{marg}(X_i)$ for each continuous variable X_i;
- generate the corresponding cumulative distributions $Y_i(X_i)$ and use them to transform all the data in one pass; then
- learn a network modeling the transformed data, without referring back to the transformations (or the learned marginal distributions that generated them).

When using learning joint density trees and then using them conditionally, the marginal-flattening method described here helps prevent the joint density tree learner from needlessly spending its representational power modeling changes in each parent variables' marginal distribution that would then have no effect on the estimated conditional distributions. Note, however, that the joint density tree learner may still waste some of its representational power modeling relationships between two or more of the parent variables even when these relationships have no bearing on the desired conditional distribution. One possible interesting avenue for future research along similar lines would be to learn transformation functions that approximately flatten the *joint* distributions of the parent variables (rather than just their marginals). These transformed parent variables could then be used in place of the originals when learning a joint density between another variable and the parent variables.

For example, in order to flatten a joint distribution between two variables, we could first flatten each of their marginal distributions using the technique described above. Then, we could learn a two-dimensional density tree over both (transformed, marginally flattened) variables in which all the tree's leaves represent constant densities. We could then use this density tree to generate an additional transformation function in which different subranges of one variable result in different transformations applied to the other variable. The cumulative distributions required for these transformations can be obtained relatively simply from a stratified reconstruction of the two-dimensional density tree generated using the algorithm described in section 4.5.4.

This line of reasoning brings up the possibility of using a *flattening network* in conjunction with the primary Bayesian network. The flattening network would be a directed acyclic graph with a variable ordering consistent with that of the primary Bayesian network. For any variable X_i in the Bayesian network, the job of all the nodes preceding $X_i' s$ node in the flattening network would be to approximately remove as many dependencies as possible from the variables they model by transforming the variables appropriately. Each density tree used in the primary Bayesian network could then be learned from data in which all dependencies between the parent variables have been approximately removed. The learner of these density trees would then be free to spend more of its representational power on interesting relationships between the child variable and the parent variables.

In this thesis, however, we will only implement and test transformations in which each variable's marginal distribution is flattened independently of all the others. (This corresponds to using a "flattening network" containing no arcs.) The investigation of more complicated transformation functions is left for future research.

4.8 Experimental results

In this section we perform an extensive set of experiments on a variety of tree-based density estimators. After describing the datasets and default parameters used throughout the tests (Section 4.8.1), the following comparisons are made:

- In Section 4.8.2 we compare the performance of stratified conditional density trees (see Section 4.5.1) versus CART-like trees that are only allowed to branch on input variables.
- In Section 4.8.3 we compare the performance of stratified conditional density trees versus joint density trees that are evaluated conditionally (see Section 4.5.2).
- In Section 4.8.4 we examine the effects of the approximate conditionalizing of joint trees (see Section 4.5.4).
- In Section 4.8.5 we evaluate several different variations of the greedy Bayesian network structure-learning algorithm discussed in Section 4.6.
- In Section 4.8.6 we evaluate the marginal distribution flattening algorithm discussed in Section 4.7.
- Finally, in Section 4.8.7 we compare our density-tree-based Bayesian network models with global mixture models.

4.8.1 Datasets and default parameters

Datasets

The Bio and Astro datasets previously discussed in Section 3.4.2 had a small amount of uniform noise added to each continuous variable value. In addition to these versions of the Bio and Astro datasets, we also employ versions in which the added noise is Gaussian instead. For the Bio dataset, we perform tests with two different versions with Gaussian noise: one in which the added noise has a standard deviation of .001, and another in which the noise has a standard deviation of .02. Comparing the results on all three different Bio dataset versions will help us partially discern how the relative accuracies of the various algorithms being evaluated depend on the fine-grained features of the distributions being modeled. Since the Astro dataset is significantly larger and therefore more time-consuming to deal with, most of our experiments will only be performed on the previously used version in which the added noise is of uniform density; however, we will sometimes perform experiments on another version in which the added noise is Gaussian with a standard deviation of .001, as the situation warrants.

We also use four synthetic datasets, each containing two continuous variables and 80,000 datapoints. The "Connected" and "Separate" datasets were both generated by sampling from a mixture of Gaussians; the primary difference between the two is that the Gaussians in the Separate dataset overlap less than in the Connected dataset. The "Voronoi" dataset was generated by sampling datapoints near a set of line segments that form a mesh over the space similar to the boundaries in a Voronoi tesselation of the space. The "Squiggles" dataset was generated by sampling datapoints near a set of sinusoidal one-dimensional strings.[2]

Default parameters

The following defaults for the density-tree learning algorithms will be used in our experiments except where we specify otherwise:

- The greedy branching variable selection strategy described in Section 4.4.1 is used.
- A branch on a continuous variable is always performed on the midpoint of the current bounding box (see Section 4.4.2).
- Post-pruning is used rather than stopping (see Section 4.4.3). 25% of the training data is held out for pruning. At least 10 datapoints must satisfy a given leaf's constraint set for a branch to be considered.
- 25% of the remaining training data is held out for evaluating different choices of branching variables.
- All Gaussian and linear-regression leaves are renormalized as described in section 4.2.2.

[2] Thanks to Andrew Moore for generating these datasets.

- The EM algorithm employed for fitting linearly (Section 4.2.4) and multilinearly (Section 4.2.5) interpolated leaves is initialized at the uniform distribution and is run for 10 iterations. A maximum of $25 * 2^d$ randomly selected datapoints are used to fit any single d-dimensional multilinear interpolation; a maximum of $25 * 2 * d$ datapoints are used to fit any single linear interpolation in which each variable is modeled independently. (Informal experiments not described further in this thesis indicated that running EM for 20 iterations and using all datapoints rather than a sample of this size did not result in significantly more accurate trees, and incurred considerable additional computational expense. We have not yet attempted experiments in which the number of iterations or sample sizes are smaller; it is possible these algorithms could be sped up even further without significant loss of accuracy.

- The global uniform "slack" distribution (see Section 4.4.4) is assigned a probability mass of 10 datapoints' worth, i.e. $\alpha = \frac{10}{10+R}$, where R is the number of datapoints in the training set. Half a datapoint's worth of mass was used for per-branch smoothing of $P(L)$ and $P_l(\vec{S_i}|l)$. These values were chosen without careful study; informal experimentation has suggested that their values are largely irrelevant as long as they're within roughly the same order of magnitude. (Assuming pruning of some sort is employed, as it is in our experiments — other informal experiments in which fixed-depth trees were learned exhibited much greater sensitivity to the exact amount of smoothing employed.)

- Marginal distribution flattening (sect 4.7) is *not* performed.

4.8.2 Conditional density trees: one-level (CART-style) vs. stratified

In this section, we compare the performance of two CART-like tree-based estimators with the performance of three different stratified tree-based estimators as described in Section 4.5.1. For a continuous child variable, the leaves of the single-level trees contain either (1) a Gaussian distribution over the child variable independent of the parent variables' values, or (2) a Gaussian distribution over the child variable whose mean is a linear function of the parent variables, as determined by linear regression. (See Section 4.2.2.) The leaves of the stratified trees can contain either of the types of Gaussian distributions used in the single-level trees, or uniform distributions over the ranges of the child variable's values that are consistent with the leaves' constraints.

For the two-dimensional synthetic datasets, the task is to learn the conditional distribution $P(X_2|X_1)$. For the "real" datasets (Bio and Astro), the task is to learn a joint distribution over all variables by learning all the conditional distributions required by a Bayesian network with a fixed structure. (Experiments in which the structure is learned will be performed later in Section 4.8.5.) These structures were taken from previous structure-learning experiments performed for Section 3.3. The Bio dataset's network has 50 arcs between the 31 variables, with the number of parents for any particular variable varying from zero to four. The Astro dataset's network has 107 arcs between the 68 variables, with the number of parents for any particular variable varying from zero to three. For the Bio dataset, we perform multiple experiments in which different types and magnitudes of noise are added to the data: uniform noise with a width of .001, Gaussian noise with a standard deviation of .001, and Gaussian noise with a standard deviation of .02.

Figures 4.14 and 4.15 summarize the results. For each dataset/model combination, the mean log-likelihood of the test set in a ten-fold cross-validation is reported, as well as the empirically estimated 95% confidence interval of this mean. The algorithm with the highest mean for a given dataset is shown in bold italics, as are all other algorithms whose means are not lower than it with a statistical significance of at least 95% (according to a Student's t-test). The mean time required to learn each model on each dataset is also shown. (The time listed is the mean for one of the cross-validation folds, not for all ten. We omit confidence intervals on these means since any speedup factor of, say, two or more was definitely consistent, and any speedup factor of much less than that is of dubious practical significance and is undoubtedly very implementation-dependent.) The machines used for the experiments were otherwise unloaded Pentium-class machines with clock cycle speeds ranging from 400 to 500 MHz. All necessary I/O was performed outside of the timing loops, and all tests involving any given dataset were always performed on the same machine.

On all datasets, it is clear that the stratified trees provide much more accurate density estimation than single-level CART-style trees, regardless of whether each leaf of the single-level trees uses linear regression or a simple Gaussian distribution over the child variable. This comes as no surprise for the synthethic datasets where the conditional distributions are obviously multimodal, but the results on the real datasets are worth noting. For example, the difference in test-set log-likelihood be-

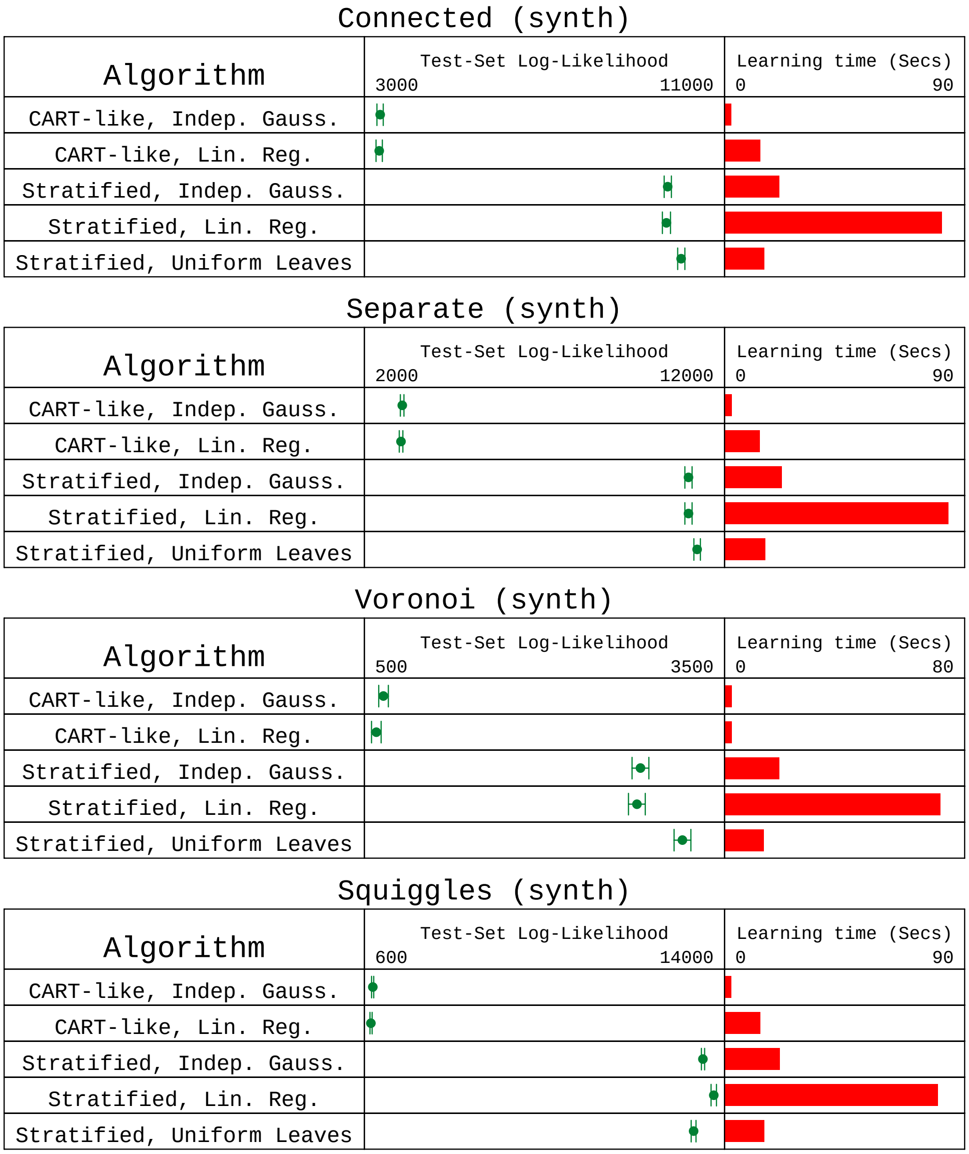


Figure 4.14: Experimental comparison of CART-like vs. stratified conditional density trees on synthetic datasets

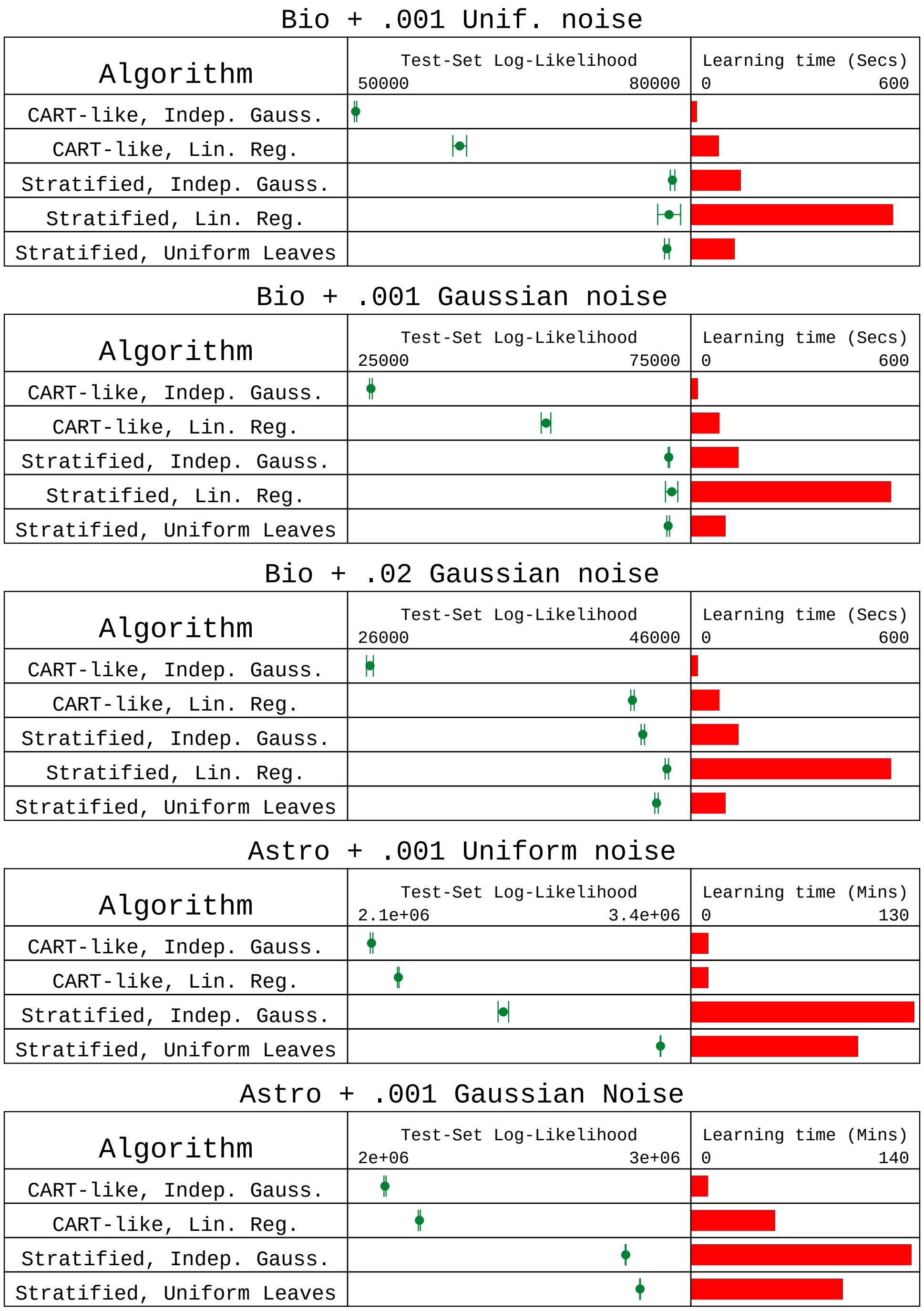


Figure 4.15: Experimental comparison of CART-like vs. stratified conditional density trees on scientific datasets

tween “Single-level Linear Regression” and “Stratified Linear Regression” on the “Bio .001 Gaussian” dataset is approximately 18300. In other words, according to the test-set data, the stratified model appears more likely than the one-level model by a factor of approximately e^{18300} in the absence of any prior information. More realistically, since there are approximately 1267 items in each test set, this means that each individual datapoint in the test set was more likely to have been generated by the stratified model than the one-level model by a factor of approximately $e^{14.4} \approx 2,000,000$. If we further divide by the number of variables (31), we see that, on average, each variable value was more likely under the stratified model than the one-level model by a factor of $e^{.47} \approx 1.6$. Similarly, the difference in the log-likelihoods of “Stratified Indep. Gaussian” vs. “Single-level Linear Regression” on the “Astro .001 Uniform” dataset works out to an average factor of $e^{35.7} \approx 3 * 10^{15}$ per datapoint, or $e^{.525} \approx 1.7$ per variable value.

Using linear regression within the tree leaves appears to perform no better than simple Gaussian distributions on the synthetic datasets, except perhaps on the Squiggles dataset when they were used with stratified density trees. It does appear to help significantly on the various versions of the Bio dataset. At first, linear-regression leaves did not work at all on the Astro dataset: sometimes the predicted conditional mean for a datapoint in the test set was so far away from the correct leaf’s bounding box that the estimated integral of the conditional Gaussian within that leaf was 0, causing the renormalization to fail. In order to address this problem, we modified our code so that it switches the conditional distribution of the leaf to a uniform distribution in such pathological cases. This allowed us to acquire a result showing that linear-regression leaves can provide better log-likelihoods than simple Gaussian-distribution leaves on the Astro dataset, at least in the case of single-level CART-style trees. However, the corresponding experiment for stratified trees was aborted after it was determined that it would take several CPU-days to complete.

Stratified trees with uniform-distribution leaves were significantly more accurate than those with Gaussian or linear-regression leaves on most of the synthetic datasets; learning them was slightly faster than learning trees with Gaussian leaves, and much faster (by a factor of 4 or so) than learning trees with linear-regression leaves. However, uniform-distribution leaves performed significantly worse on the Squiggles dataset and in some instances of the Bio dataset.

The fact that Gaussian leaves can be less accurate than simple uniform-density leaves in stratified trees may be partially due to the fact that the conditional density estimated for any combination of parent values will have a "bump" for every leaf in the tree, as discussed in Section 4.2.2. This problem might be fixed by using a more complicated optimization routine that fits the Gaussian while taking truncation and renormalization into account, as hinted in Section 4.2.2. Unfortunately, this would probably slow the learning process down considerably, and it is not clear whether it could feasibly be generalized to handle the case of linear-regression leaves.

Stratified density trees can require almost an order of magnitude more computational time to learn due to the more complicated nature of the tree structure search space. It may be possible to develop effective stratified density tree algorithms that do not take as much time to learn. However, as we will see in the next section, it is already much faster to learn joint density trees and then use them conditionally (as described in Section 4.5.2); surprisingly, this can result in more accurate density estimation as well.

4.8.3 Conditional density estimation: stratified trees vs. joint trees

In this section we compare the performance of the stratified conditional density trees described in Section 4.5.1 with the performance of joint density trees used conditionally as described in Section 4.5.2. As noted in Section 4.5.4, there are several differences in how these two different kinds of density trees are learned that would cause us to expect their accuracies to be different:

- The structure of stratified trees is optimized for the specific conditional distribution for which the tree will be used, while the structure of joint trees is not. We might expect this to cause stratified trees to be more accurate than joint trees.
- The structure of stratified trees is less flexible than the structure of joint trees, as discussed previously in section 4.5.4. We might expect this to cause stratified trees to be less accurate than joint trees.
- Joint trees employing leaves with nonuniform distributions over the parent variables are in a sense using "soft branches" that help them to predict X_i as a function of $\vec{\Pi}_i$ more flexibly without actually splitting the data into completely

disjoint subsets according to $\vec{\Pi}_i$. We might expect this to cause stratified trees to be less accurate than joint trees.

We attempt to gauge the impact of each these differences separately by testing eight different conditional density-tree algorithms:

- Stratified density trees employing uniform-distribution leaves. This algorithm is listed as “Stratified Cond Uniform” in Figures 4.16 and 4.17, and is the same as the “Stratified, Uniform” algorithm employed in Figures 4.14 and 4.15.

- Joint density trees employing uniform-distribution leaves. These joint trees are then used conditionally as described at the beginning of Section 4.5.2. This algorithm is listed as “Joint Uniform” in Figures 4.16 and 4.17.

- Stratified density trees in which each leaf has a linearly interpolated distribution over X_i that is independent of $\vec{\Pi}_i$. This algorithm is listed as “Stratified Cond Linear” in Figures 4.16 and 4.17.

- Joint density trees in which each leaf models each variable independently with a linearly interpolated density as described in Section 4.2.4. This algorithm is listed as “Joint Indep. Linear”.

- Joint density trees in which each leaf models the continuous variables jointly using multilinear interpolation as described in Section 4.2.5. This algorithm is listed as “Joint Multilinear”.

- Stratified density trees in which each leaf models the continuous variables jointly using multilinear interpolation. The distribution within each leaf is learned using the same algorithm as would be used in the analogous joint density tree. However, each leaf l’s joint distribution $P(X_i, \vec{\Pi}_i | l)$ is then only used to compute the conditional distributions $P(X_i | \vec{\Pi}_i, l)$ required for the stratified tree’s conditional density estimation algorithm. This algorithm is listed as “Stratified Cond Multilinear”.

- Density trees that are identical to the previously listed “Joint Uniform” trees except they are *structured* like stratified density trees, i.e., with all branches on $\vec{\Pi}_i$ before any branches on X_i. However, while the tree has this restriction, it structure is still being optimized for joint log-likelihood rather than conditional log-likelihood. This algorithm is listed as “Stratified Joint Uniform”.

- Density trees that are identical to the previously listed "Joint Multilinear", except they are *structured* like stratified density trees. This algorithm is listed as "Stratified Joint Multilinear".

Figures 4.16 and 4.17 show how these algorithms performed on most of the same learning tasks used in the previous set of results (Figures 4.14 and 4.15). (Due to the extreme amounts of computational time taken on the Astro datasets by the Stratified algorithms, we restrict ourselves here to one of the Astro dataset versions, and have not yet tested the "Stratified Joint Uniform" or "Stratified Cond Multilinear" algorithms.)

Several notable patterns can be seen in this set of results. Overall, however, the most important thing to note is that the joint density trees were always much faster to learn than stratified conditional density trees, and this additional speed came with little loss of predictive accuracy — in fact, when the trees' leaves employed nonuniform distributions, joint density trees were *more* accurate than stratified density conditional trees.

The differences in the accuracy of the various algorithms were relatively small on the two-dimensional Connected, Separate, and Voronoi datasets — at most five or six times the standard deviations of the test-set log-likelihoods' estimated means. (The listed uncertainties are for 95% confidence intervals, or two standard deviations in each direction.) However, the differences become more significant on the real datasets, where higher-dimensional density trees are being employed and the distributions being modeled have sharper features. Whether these differences are of actual practical significance depends on the application. For example, the difference in the test-set probabilities of the Joint Uniform vs. Joint Multilinear algorithms on the "Bio .001 uniform" dataset works out to a factor of approximately 97 per datapoint, or 1.16 per variable value. The difference in the test-set probabilities of the Joint Multilinear vs. Conditional Uniform algorithms on the "Astro .001 uniform" dataset works out to roughly a factor of 127 per datapoint, or 1.07 per variable value.

Comparing the results of the Stratified Cond Uniform and Stratified Joint Uniform alogrithms shows that when the leaf distributions are uniform, Stratified conditional density trees do in fact appear slightly more accurate than joint density trees that are restricted to the same Stratified structure. This is to be expected, since the only real difference between these two algorithms is that the Stratified Cond Uniform tree-learning algorithm is optimizing for the appropriate conditional distribution, whereas

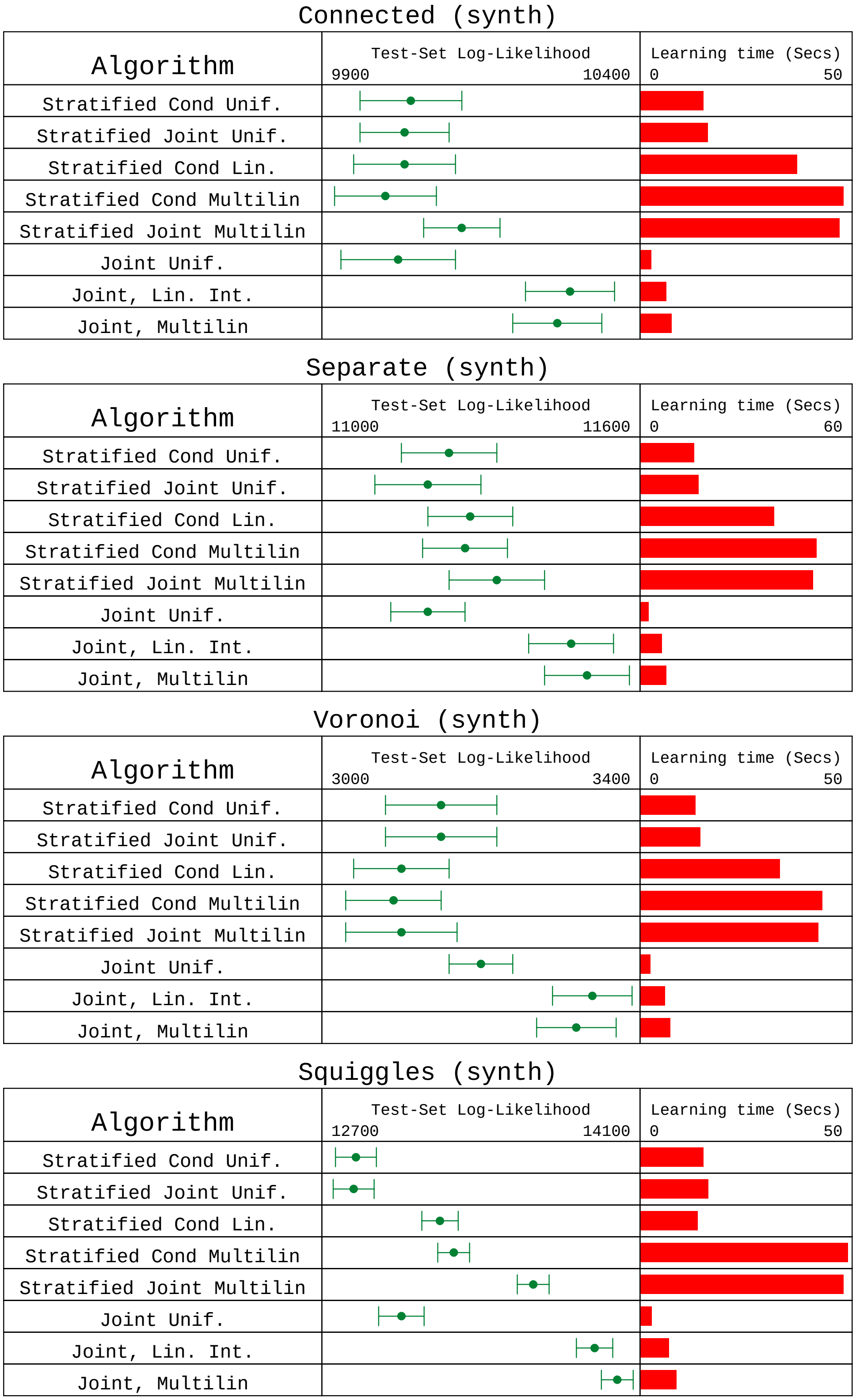


Figure 4.16: Experimental comparison of stratified conditional density trees vs. joint density trees on synthetic datasets

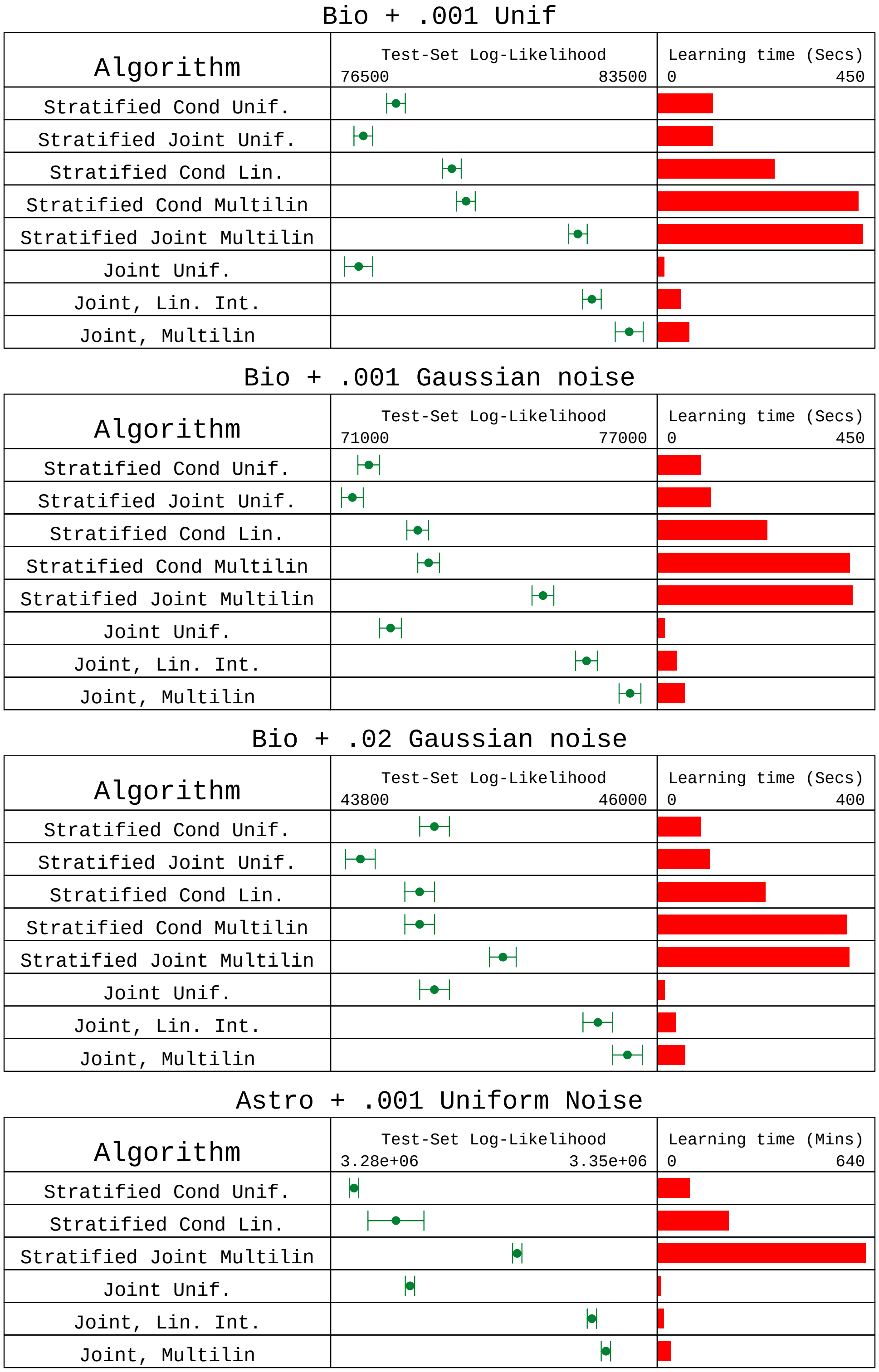


Figure 4.17: Experimental comparison of stratified conditional density trees vs. joint density trees on scientific datasets

the Stratified Joint Uniform algorithm is optimizing for the joint.[3] However, these differences are small, particularly on the two-dimensional synthetic datasets.

Comparing the results of the Stratified Cond Uniform and Joint Uniform algorithms suggests that when the uniform-leaf joint density trees are freed from the structural restrictions of stratified trees, this added flexibility can occasionally make up for the fact that they are optimized for the wrong distribution (i.e. joint rather than conditional), as it appears to do in the Squiggles, Gaussian-noise Bio, and Astro datasets. However, on many of the other datasets it appears to make no significant difference.

The picture changes when nonuniform leaf distributions are employed. The joint density trees employing nonuniform leaf distributions (Joint Indep. Linear and Joint Multilinear) consistently and significantly outperform all stratified conditional density trees (Stratified Cond Uniform, Stratified Cond Linear, and Stratified Cond Multilinear) both in terms of learning speed and prediction accuracy. This increased prediction accuracy occurs despite the fact that they are optimized for joint distributions rather than the conditional distributions for which they are subsequently used.

Comparing the Stratified Cond Multilinear and Stratified Joint Multilinear algorithms allows us to specifically test the "soft branching" hypothesis. Even when joint density trees are restricted to have the same structure as conditional density trees, the fact that they learn the parent variables' distributions in the leaves allows them to predict the output variables more accurately than the corresponding stratified conditional density trees in which the parent variables' distributions are not modeled in the leaves. The results on the Squiggles, Bio, and Astro datasets all lend support to this hypothesis. (The results on the other synthetic datasets are also positive, but only slightly so.) By themselves, these results do not exclude the possibility that the differences in accuracy were due entirely to subtle differences in tree structure caused by the different optimization criteria (conditional log-likelihood vs. joint log-likelihood); however, other experimental results in Appendix A.2 show this is not the case.

In all experiments with joint density trees, trees employing nonuniform leaf distributions were significantly more accurate than those employing uniform leaves. Trees

[3] Actually, this is not quite true, since the "Uniform" leaves in our trees still have non-constant distributions over any discrete variables they model, so some "soft branching" may still occur due to these discrete variables. However, there are no discrete variables in the synthetic datasets, and only a few in the Bio and Astro datasets. Furthermore, further supplemental experiments in Appendix A.2 control for this difference.

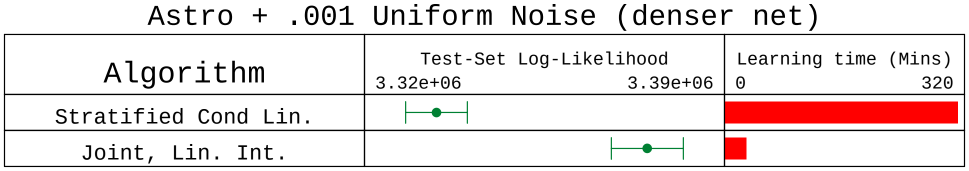


Figure 4.18: Experimental comparison of stratified conditional density trees vs. joint density trees: higher network connectivity

using multilinear interpolation were more accurate than those employing independent linear interpolations, and this increased accuracy is statistically significant; however, this difference is not dramatic, and comes at significant additional computational expense.

One might worry that the particular network structures used in the above experiments happen to be particularly favorable to the density-tree algorithms that learn joint distributions that are then used conditionally, as opposed to the stratified conditional density trees. After all, the network structures used here were generated during previous experiments with Mix-nets (Chapter 3), which also learned joint distributions that were used conditionally; furthermore, the network structures used here are rather sparse. As a followup experiment, we used the greedy network-learning algorithm described in Section 4.6 on a version of the Astro dataset discretized with 4 buckets per variable, using stratified conditional density trees to model the discretized data. No *a priori* restriction on the number of parents per variable was enforced. This produced networks with an average of approximately three parents per variable, or roughly twice the number of parents per variable in the network employed in the tests above. We compared the performance of "Joint Lin. Int." with "Stratified Cond Lin" density trees using these network structures, using a much faster machine than the machines employed for the other experiments in this thesis. The results, shown in Figure 4.18, show that "Joint Lin. Int." is still significantly more accurate.

4.8.4 Approximate conditionalizing of joint trees for fast evaluation

In this section we evaluate the algorithm proposed in Section 4.5.4 for "conditionalizing" joint density trees so conditional probabilities can be computed quickly.

We compare four algorithms. The first two are the "Stratified Cond Linear" and

"Joint Lin. Int." algorithms described in the previous section. The third, "Joint LI Conditionalized", is the same as "Joint Lin. Int.", except the resulting joint density tree is supplemented with a conditionalized joint density tree (as described in Section 4.5.4) which is used to speed up the *exact* evaluation of conditional densities from the joint density tree by providing pointers directly to the relevant nodes in the original tree. The fourth algorithm, "Joint LI Approx. Cond.", is the same as the third, but the conditionalized density tree is evaluated *approximately* as

$$P(x_i|\vec{\pi_i}) = \alpha_c^s P(x_i|\vec{\pi_i}, l_c)$$

where l_c is the joint density tree leaf consistent with both x_i and $\vec{\pi_i}$, and the α_c^s's are computed as described in Section 4.5.4. Figures 4.19 and 4.20 summarize the results. As before, the "Learn time" listed is the average training time per cross-validation fold. The evaluation time listed is the average time per cross-validation fold required to evaluate the conditional log-likelihoods of all modeled variables in the *entire* dataset (that is, both the training and test sets).

The results clearly show that approximately conditionalized joint density trees can be used to calculate conditional probability much more quickly than nonconditionalized joint density trees. The speedup factor ranges roughly from 7 to 25 in these particular experiments; it is greatest on the synthetic datasets since two-dimensional problems tend to be the most expensive case for the conditional evaluation of joint density trees (see Section 4.5.2). Some of this speedup — roughly a factor of 2 in all cases — was due to the quicker access to the leaves of the original joint density tree provided by the conditionalized tree's structure. The remainder of the speedup was due to the approximate evaluation procedure in which only the single leaf consistent with both the child variable value x_i and parent variable values $\vec{\pi_i}$ is evaluated.

This approximate evaluation causes a noticable amount of accuracy to be lost on problems in which the distributions have sharp features, such as the Squiggles and small-noise Bio datasets. On many other problems, however, including the Astro dataset, no significant accuracy is lost. Furthermore, conditionalized joint density trees are still significantly more accurate than stratified conditional density trees, and can be learned much faster (by a factor of roughly 3 to 6 in our experiments). *Thus, conditionalized joint density trees represent a useful compromise between the learning speed and accuracy of joint density trees and the evaluation speed of conditional density trees.*

One possible way to improve the accuracy of conditionalized joint trees would

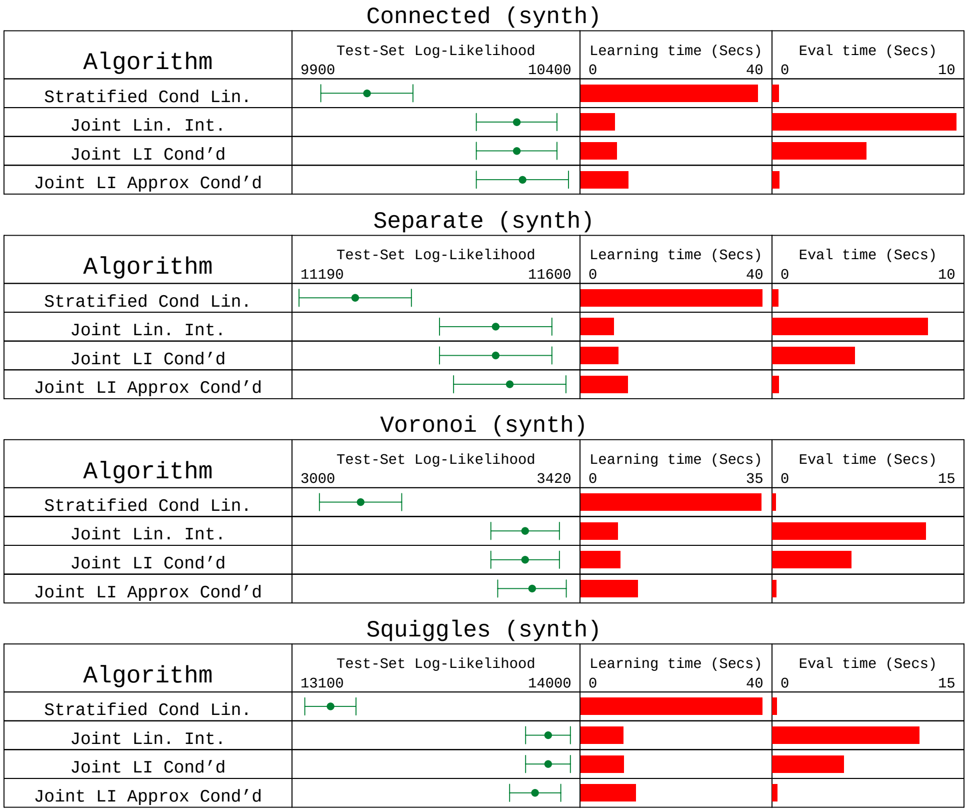


Figure 4.19: Experimental results for approximate conditionalizing on synthetic datasets

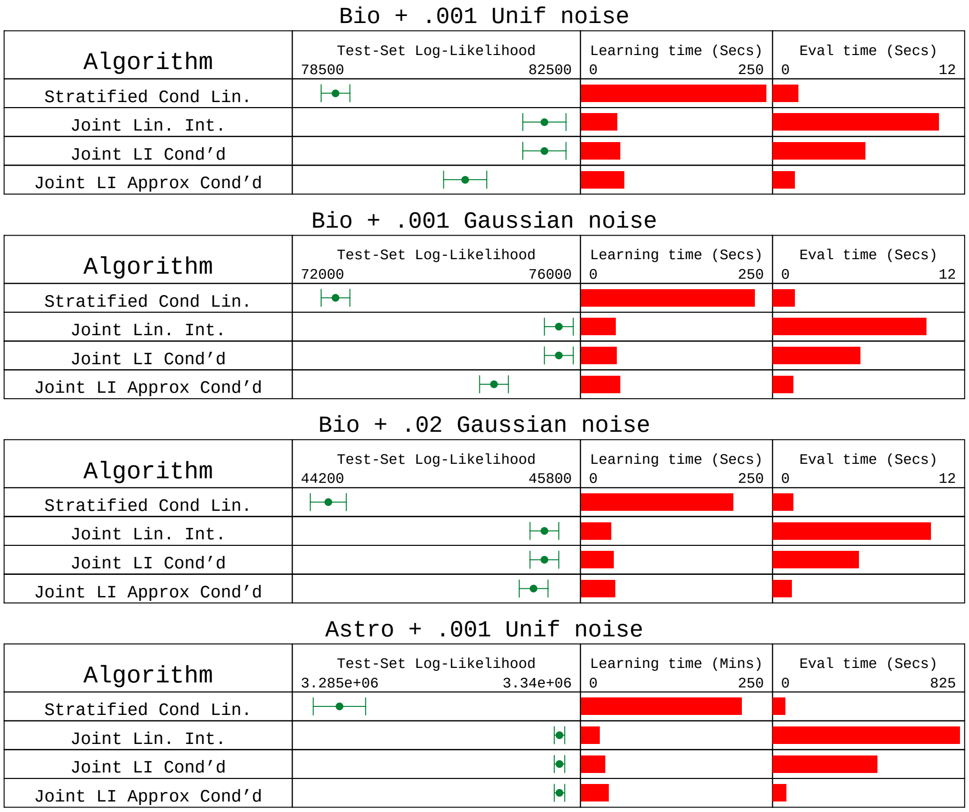


Figure 4.20: Experimental results for approximate conditionalizing on scientific datasets

be to refine the subtrees over the input variables $\vec{\Pi}_i$ further before branching on X_i begins. Multiple refined subtrees corresponding to different values of $\vec{\Pi}_i$ would then have identical subtree structures over X_i, and would point to identical sets of leaves in the original joint tree; the only differences between them would be in the interpolation coefficients α_c^s they used to approximate $P(l_c|\vec{\Pi}_i)$. Determining whether to refine a given subtree over $\vec{\Pi}_i$ would be a matter of explicitly trading off the additional memory and evaluation-time computational costs versus the resulting increased accuracy; we do not investigate this issue further in this thesis.

4.8.5 Network structure-learning algorithms

In this section we evaluate the speed and accuracy of several variations of the greedy network-learning algorithm discussed in Section 4.6.

Our first set of experiments consists of several different algorithms applied to the version of the Bio dataset in which uniform noise of magnitude .001 has been added. Throughout all the experiments, we set MAXPARS (the maximum number of parents any given variable can have) to 5 and MAXCHANGES (the maximum number of parent-set changes to attempt on any one variable during any single iteration of the greedy algorithm) to 10. MAXPARS was tuned to this value by observing that density trees with greater numbers of parent variables never provided much additional prediction accuracy, and were very computationally expensive to learn; this value of MAXCHANGES was simply the first we tried. (See Appendix A.2 for an experiment in which a value of 1 was also tried for MAXCHANGES.) Both $S_f(X_i, \vec{\Pi}_i)$ and $S_s(X_i, \vec{\Pi}_i)$ estimate the goodness of a given parent set $\vec{\Pi}_i$ for a given variable X_i by learning a density tree of some sort using 75% of the training data and then evaluating the conditional log-likelihood of the remaining 25%; however, the types of density trees they employ may be different. In these experiments we do not conditionalize the joint density trees.

Each of the algorithms is run for several iterations. Each iteration of a given algorithm uses the network returned by the previous iteration of the same algorithm for its initial network B_0. The first iteration of each algorithm is provided the empty network $B_0 = B_\epsilon$, except one algorithm that is instead initially provided with the best network structure found by a stochastic search algorithm on a discretized version of the data.

After each iteration, we measure the quality of the resulting network structure with respect to another density tree learning algorithm, which may be different from any of the density tree learning algorithms used during the actual network structure-learning procedure. In most experiments, the density trees used for this measurement will be joint density trees employing multilinearly interpolated leaves, since these tend to provide better final density estimates than those employing other types of leaves.

First, we examine the impact on speed and accuracy of using completely discretized versions of the dataset during different phases of the greedy network-learning algorithm. We compare the following variations:

- Versions where a discretized version of the dataset is used for both S_f (the learning algorithm used to evaluate all possible arc changes at the beginning of any given iteration) and S_s (the learning algorithm used to evaluate the quality of a candidate parent set throughout the greedy network-learning process). We try two different discretizations of the dataset: one in which each variable has been quantized into 4 bins, and another in which each variable has been quantized into 8 bins. Each variable is quantized independently of all the others; the boundaries of the bins are selected so that roughly the same number of datapoints lie in each bin. Density trees are still used to model the distributions of these discretized variables, but the particular density tree learning algorithm used makes the resulting models very similar to the contingency tables typically used in discrete-variable Bayesian networks: namely, we employ stratified conditional density trees (Section 4.5.1) that use the "taking turns" algorithm for selecting branch variables (Section 4.4.1), with pruning disabled. This effectively implements a "sparse array" representation of a contingency table. These versions of the algorithm are labeled "Disc4→ML" and "Disc8→ML" according to the number of discretization bins used per variable.

- Versions where a discretized version of the dataset is used for S_f, but S_s uses joint density trees with uniform-density leaves to model the original continuous data. These versions of the algorithm are labeled "Const w/Disc4 Arc Scores→ML" and "Const w/Disc8 Arc Scores→ML" according to the number of discretization bins S_f uses per variable.

- A version of the algorithm where joint density trees with constant-density leaves are learned on the original continuous data for both S_f and S_s. This version is labeled "Const→ML".

Figure 4.21 summarizes the results. The plot has one line for each of the five algorithm variations; each point on each line represents one iteration of that algorithm. The time associated with the n^{th} point for a given algorithm includes the time required for iterations 1 through n of the algorithm, plus the time required to learn joint multilinear density trees for the variable combinations occuring in the network structure returned by the n^{th} iteration. The log-likelihood associated with the n^{th} point is the mean test-set log-likelihood of the Bayesian network with the structure learned by the n^{th} iteration and the conditional distributions determined by the subsequently learned joint multilinear density trees. (These means are over 10-fold cross-validations. The vertical error bars are the 95% empirically estimated confidence intervals of these means.) The results of every algorithm's first iteration does appear in the plot; the lines coming up from the bottom of the plot are coming from "Iteration 0" of all the algorithms, which corresponds to using an empty network.

Unsurprisingly, the Disc4→ML and Disc8→ML algorithms were the fastest per iteration. However, the resulting network structures were not particularly useful for the final parameterizations over the original continuous data. The performance was also quite sensitive to the discretization level used — Disc4 performed much worse than Disc8. The network structures returned by the Disc4 algorithm's early iterations also caused the subsequent multilinear density tree learning to take much more computation. (The curve for Disc4 doubles back on itself because it was faster to perform two iterations of the greedy algorithm using the discrete data and then reparameterize the network with multilinear density trees than to only perform one iteration of the greedy algorithm before reparameterizing.)

The algorithms ("Const w/Disc4 Arc Scores→ML" and "Const w/Disc8 Arc Scores→ML) that used discretized data for S_f but uniform-leaf density trees over the original data for S_s found significantly better networks in almost as little time as the algorithms which also use discretized data for S_s. They were also much less sensitive to the particular level of discretization used.

The algorithm ("Const→ML") that employs uniform-leaf density trees for both S_f and S_s takes significantly more time per iteration than any of the others. After a few iterations, it does find networks that are more accurate, with statistical significance; however, this requires three or four time-consuming iterations, and the difference in accuracy is still relatively small.

Next we compare the effect of using different kinds of density trees for the reparam-

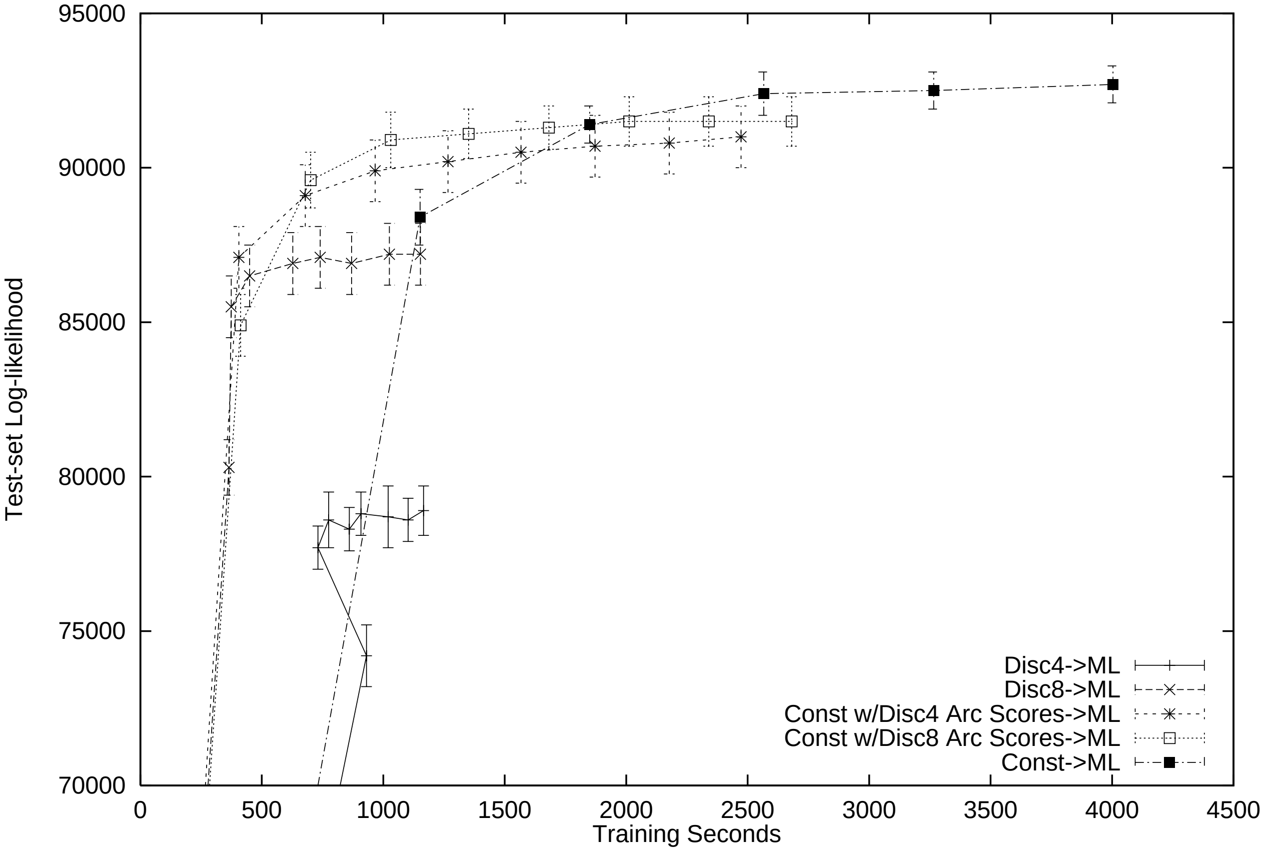


Figure 4.21: Bio dataset structure-learning experiments: effects of using discretized distributions for quality estimates.

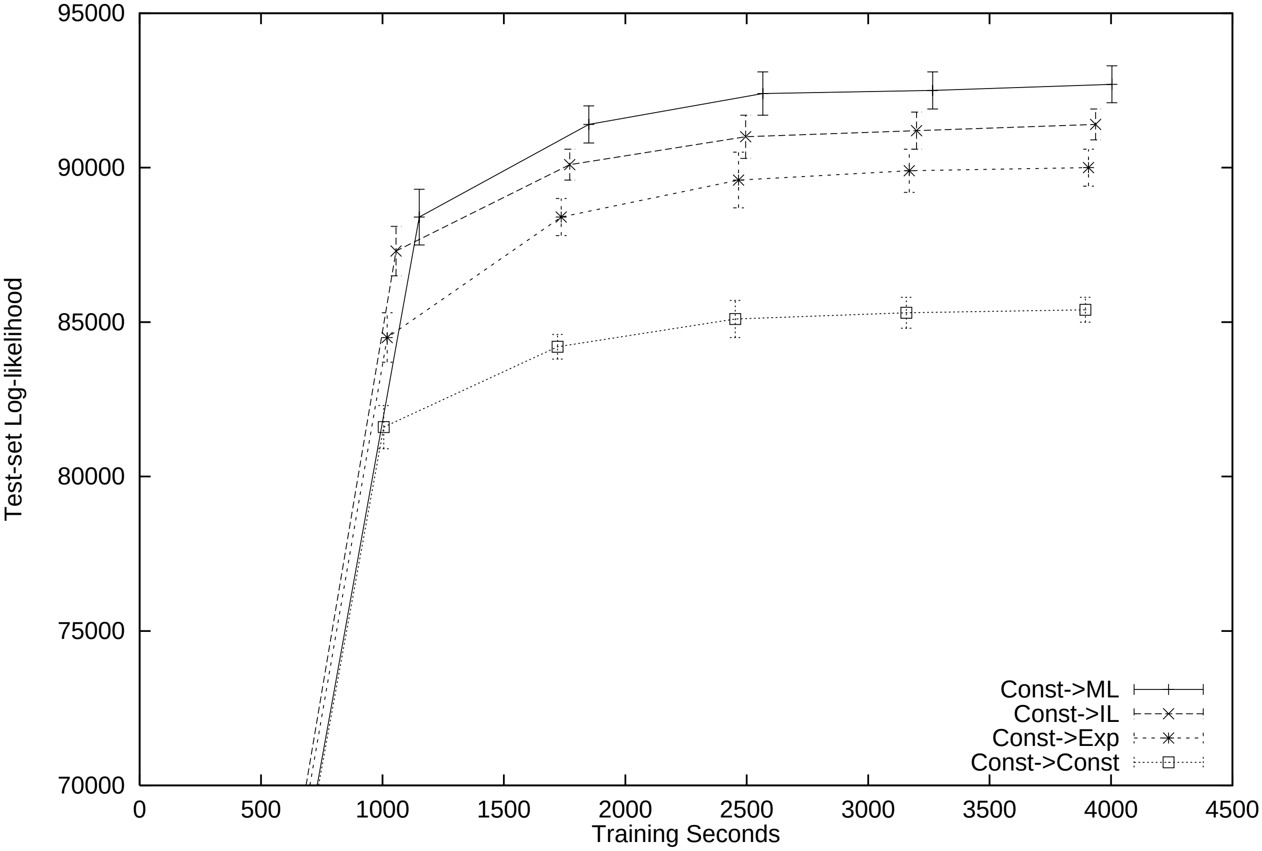


Figure 4.22: Bio dataset structure-learning experiments: effects of different reparameterizations of the same network structures.

eterization of the networks after each iteration of the greedy algorithm. The greedy algorithm used in all the following variations uses density trees with constant-density leaves for both S_s and S_f. (Discretized versions of the dataset are never used.) After each iteration, we test the effectiveness of using four different kinds of density trees to parameterize the resulting network structure. These four density tree algorithms are identical except for the kinds of leaf distributions they employ: constant, exponential, independent linear, or multilinear. (See Section 4.2.) The algorithms are labelled "Const→Const", "Const→Exp", "Const→IL", and "Const→ML", accordingly. Figure 4.22 shows the results.

Despite the fact that the greedy structure-learning algorithm is optimizing the structure while using density trees with constant-density leaves, all the other density tree types work better than constant-leaf density trees on the resulting network structures. The differences in the accuracies of the four tree types is quite consistent throughout multiple iterations of the structure-learning algorithm. Multilinear interpolation produced the most accurate density estimation, followed by independent linear interpolation, exponential distributions, and constant distributions, in that order. Further note that the final accuracy of the "Const→Const" algorithm is significantly worse than accuracies of any of the previously evaluated learning algorithms that employ multilinear density trees for their final distributions, except for the "Disc4" algorithm.

Finally, we compare the previous "Const→ML" network learning algorithm with two others. The first learner, "ML→ML", uses density trees with multilinearly interpolated leaves throughout the entire learning process — that is, for S_f, S_s, and the final networks. The second learner is identical to the "Const→ML" algorithm, except it is initialized with a non-empty network structure. This structure was learned using a stochastic search procedure on a discretized version of the dataset (with 8 discretization bins per variable). AD-Trees ([ML98]; see this reference for the description of the stochastic searh algorithm as well) were used to speed up the search. The best network found during 100,000 iterations of the search was used for this second variation of the greedy algorithm, which we label "Disc Search→Const→ML". The results are shown in Figure 4.23.

Only two iterations of the "ML→ML" could be run due to the large amount of time required per iteration. Furthermore, the result at the end of each these two iterations was no better than the result of the corresponding iteration of "Const→ML", which ran many times faster. This suggests that using density trees with constant-density leaves is a more effective strategy during the network structure search, despite the fact that multilinear density trees are much better candidates for the final network parameterizations.

The results of "Disc Search→Const→ML" were similarly unimpressive. The 100,000-iteration stochastic search over network structures provided a starting network structure that was significantly less useful for modeling the original continuous data than the structure found by a single iteration of the greedy algorithm intialized from the empty network, which required much less time.[4] When the greedy algorithms are run for three iterations from their starting points, the greedy algorithm that had

[4] Note that an "iteration" of our greedy algorithm involves much more work than an "iteration" of the stochastic search procedure; comparing numbers of iterations between the two algorithms would be meaningless.

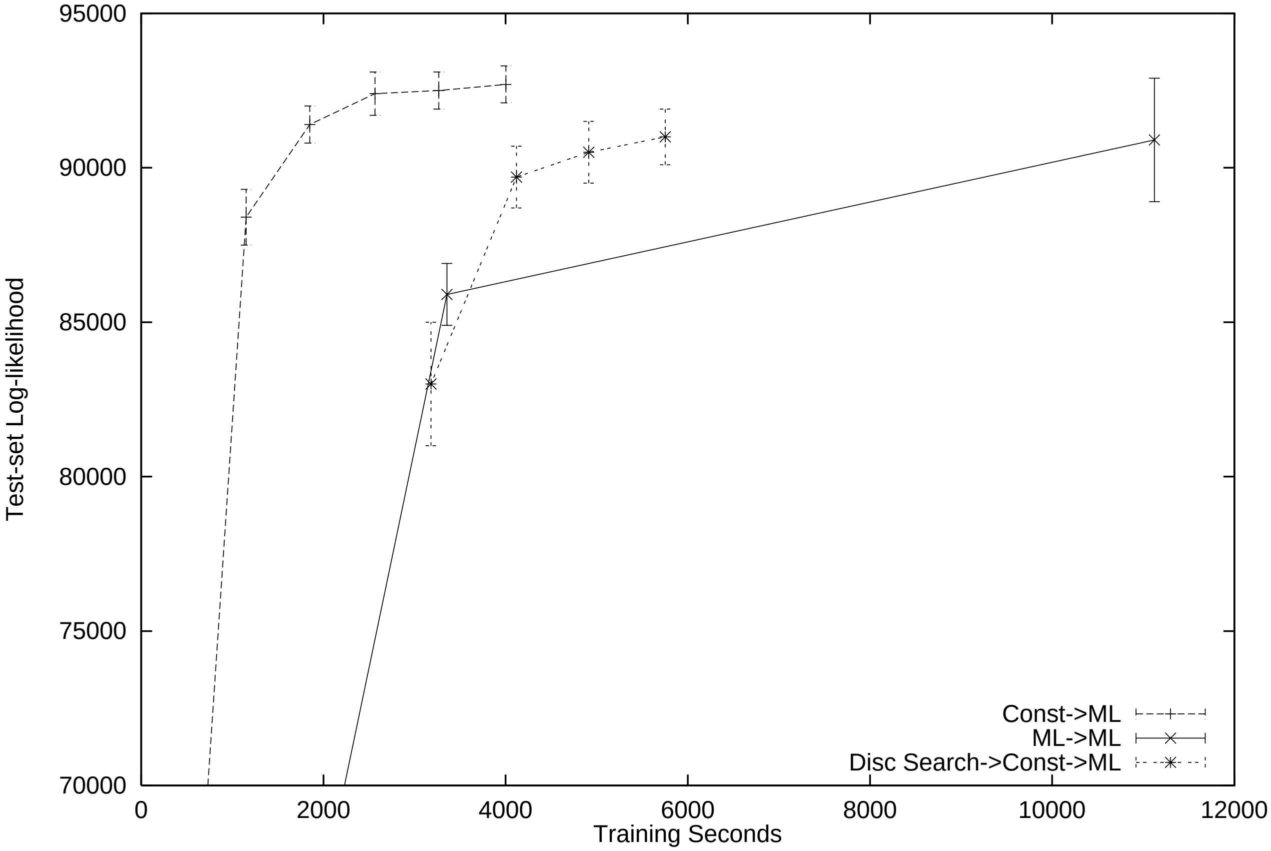


Figure 4.23: Bio dataset structure-learning experiments: comparing vs. more expensive algorithms.

been initialized with the network found by the stochastic search procedure returns a network that is *less* accurate than that found by an identical greedy algorithm started from the empty network.

Now we turn our attention to the Astro dataset. (Again, we will only use the version in which uniform noise of magnitude .001 has been added.) In the following experiments we will use the same learning parameters as in the Bio dataset; however, in order to keep the computational time tractable, we restrict S_s and S_f to use a maximum of 10000 training points and 2500 evaluation points. After each iteration of the greedy network-learning algorithm, the quality of the resulting network structure is evaluated by reparameterizing the network with joint multilinear density trees learned with the entire training set.

Due to the amount of CPU time required for structure-learning experiments on this dataset, we try a smaller set of variations of the greedy network-learning algorithm: "Disc4→ML", "Disc8→ML", "Const w/Disc8 Arc Scores→ML", and "Const→ML". We also include an experiment in which the previously mentioned stochastic search algorithm is employed for 100,000 iterations on a version of the dataset that has been discretized to 8 values per variable and restricted to a randomly sampled 10,000 training datapoints. (We also tried using 10,000 iterations on 100,000 datapoints instead; this took about the same amount of time but the learned networks were slightly less accurate.) This final algorithm is labelled "100000-it Disc8 Search→ML".

The results are shown in Figure 4.24. The Disc4→ML algorithm is not shown on the plot because it performed extremely poorly: attempting to reparameterize the network found by the first iteration with multilinear density trees took over four hours (per cross-validation fold), and the resulting networks were less accurate than those found by the first iteration of any of the other greedy learning algorithms. On the other hand, the Disc8→ML algorithm performed very well on this dataset, finding networks about as accurate as those found by any of the other algorithms but in significantly less time. Const w/Disc8 Arc Scores→ML found networks of about the same quality, but in somehat more time. Const→ML had still not quite found networks of the same quality after over twice as much time as Const w/Disc8 Arc Scores→ML. Using the stochastic search procedure on the discretized data ("100000-it Disc8 Search→ML") produced network structures that were less accurate than any found by any versions of the greedy algorithm.

While we do not supply a graph here similar to Figure 4.22 comparing the use of

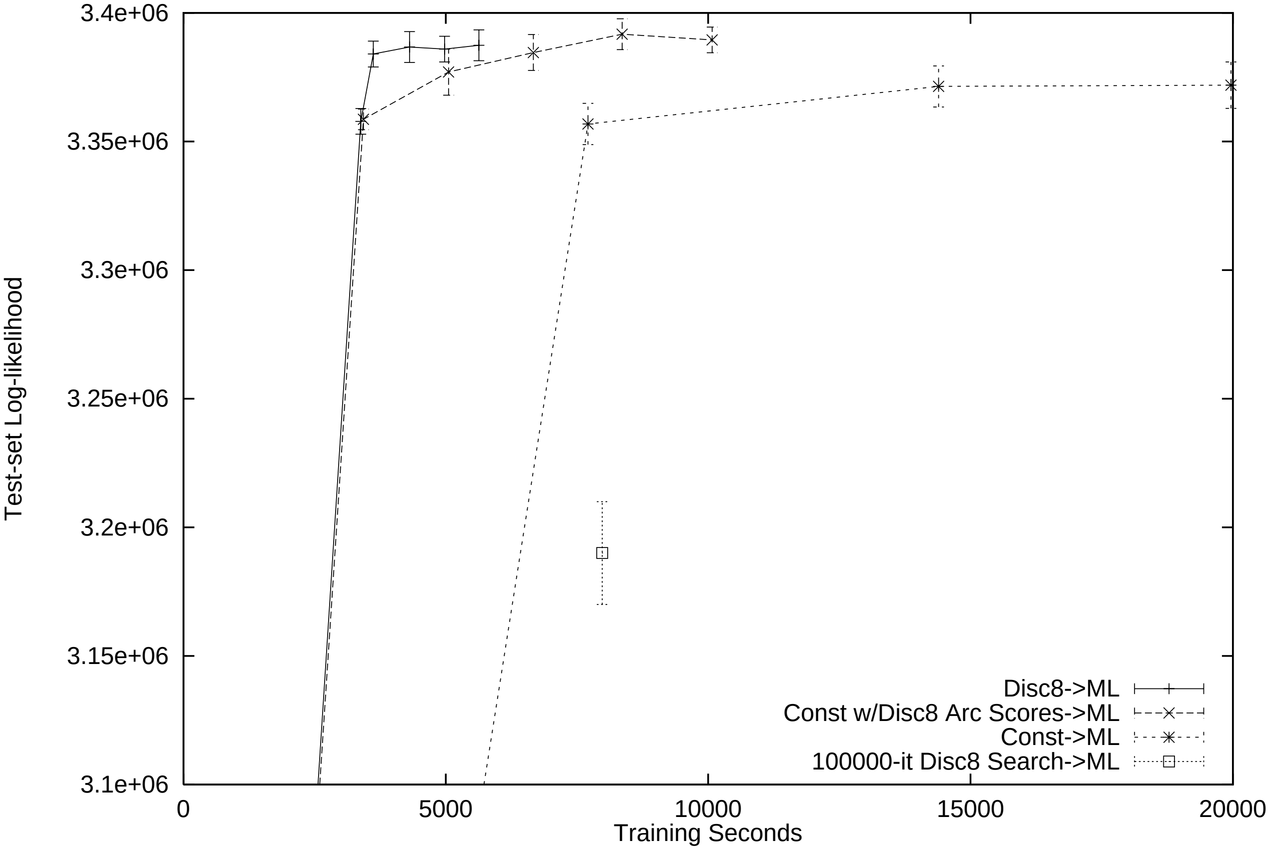


Figure 4.24: Astro dataset structure-learning experiments.

different kinds of density trees for the final network parameterizations, Section 4.8.6 includes results for such a comparison. As in the Bio dataset, trees employing multilinearly interpolation appear to result in the most accurate final density estimators.

The structure-learning results on the Astro and Bio dataset suggest that being able to use different types of density trees for different stages of the network-learning algorithm can be very useful for finding accurate networks in a reasonable amount of time. In particular, using simple contingency-table-like density trees over discretized data for S_f appears desirable to maintain reasonable speed, but using them for S_s as well can cause the algorithm to be very sensitive to the discretization level used and can sometimes lead to poor accuracy. Using density trees with multilinearly interpolated leaves appears to be the most accurate choice for the final parameterization of the networks.

The results also suggest that our greedy network-learning algorithm is capable of finding accurate networks quite quickly compared to the stochastic search procedure we tested against. However, it may be that the particular stochastic search procedure used here was not particularly efficient; further study is warranted.

4.8.6 Marginal distribution flattening

In this section we examine the effects of the marginal distribution flattening algorithm discussed in Section 4.7 on density trees employing constant, exponential, independent linear, or multilinear leaves. Experiments are performed on both the Bio and Astro datasets, with multiple types and magnitudes of noise added to them. In addition to gauging the usefulness of marginal distribution flattening, these experiments also serve to compare the effectiveness of the different leaf distributions.

For each dataset, the same network structures are used across all algorithms. The network structures used were found via running four iterations of the "Const w/Disc8 Arc Scores" version of the greedy structure-learning algorithm (see the previous section for details). The network structure used during a particular fold of the cross-validation was learned using only the training data for that fold. (These results are thus not directly comparable with those in sections previous to Section 4.8.5.) Figures 4.25 and 4.26 summarize the results. (The times required to learn the network structures are not included in the listed learning times.)

The results show that using the marginal distribution flattening algorithm can

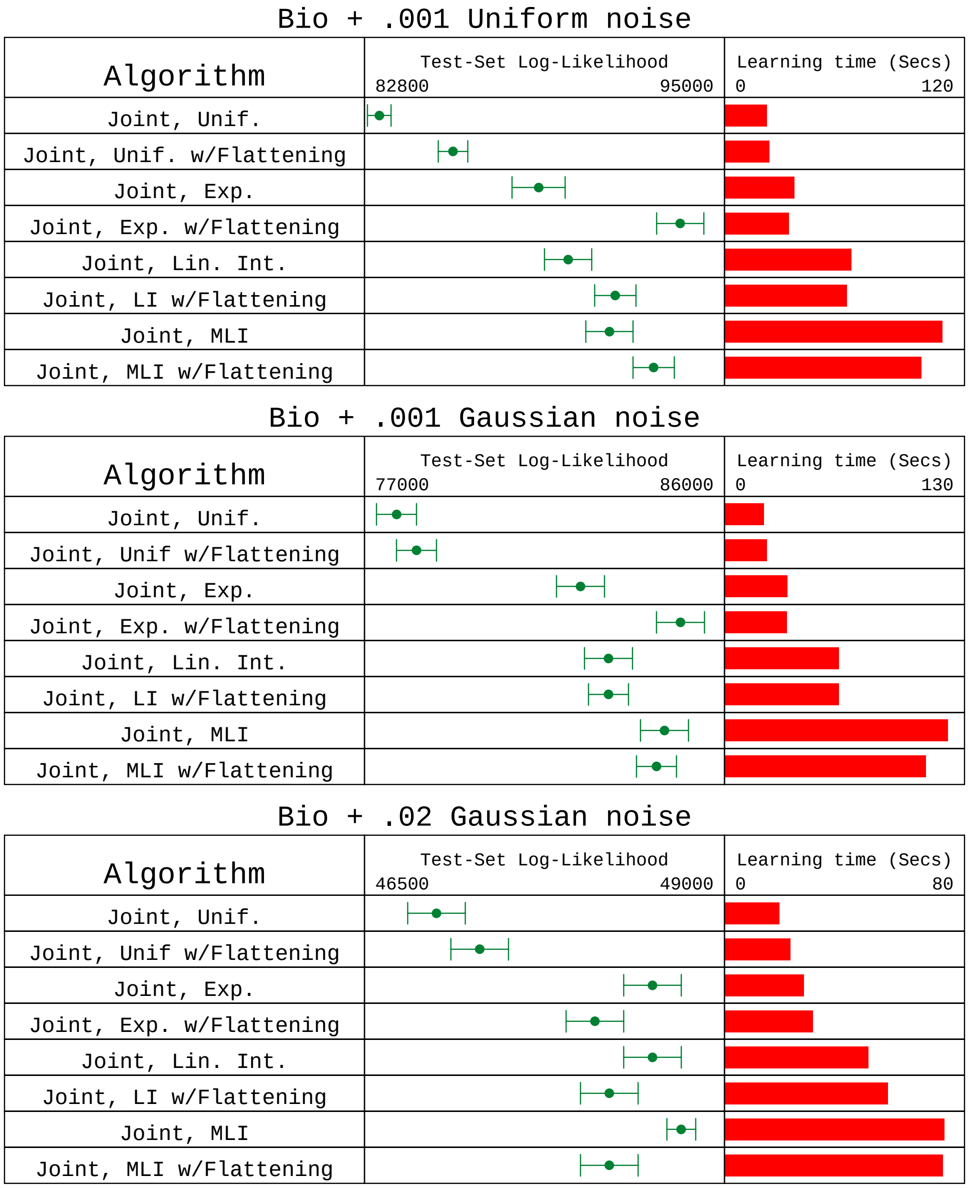


Figure 4.25: Experimental results for marginal distribution flattening on Bio datasets

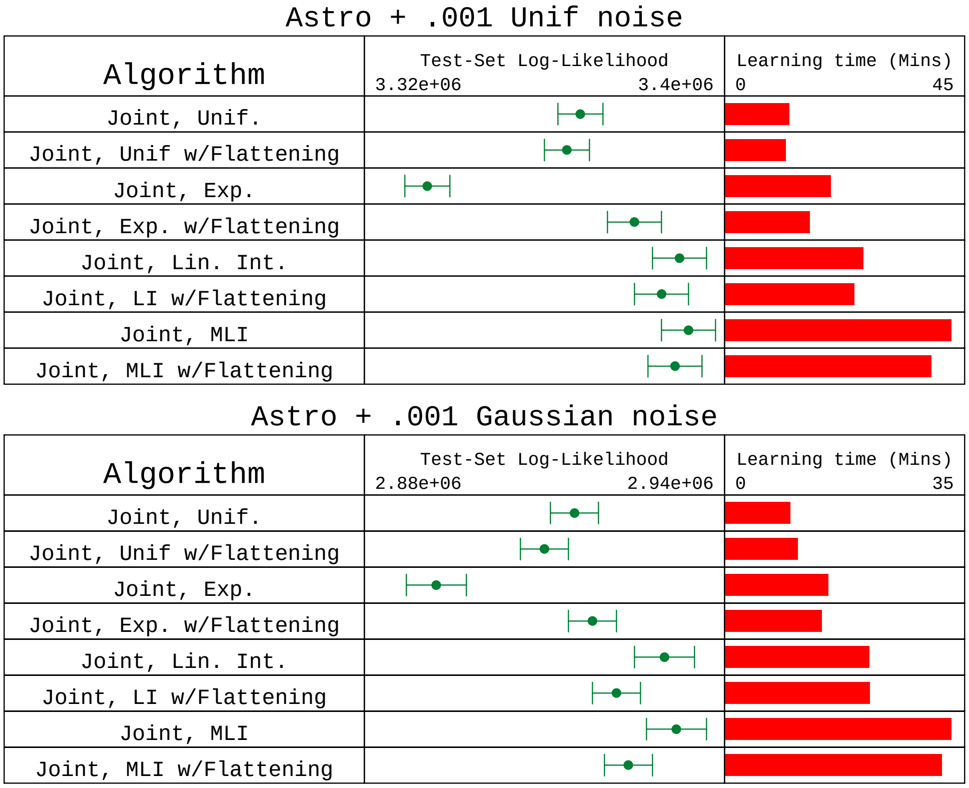


Figure 4.26: Experimental results for marginal distribution flattening on Astro datasets

significantly increase accuracy in some situations where the marginal distributions exhibit sharp features, although it also decreases accuracy in other situations. In particular, it significantly helped all of the different density tree types on the Bio .001 Uniform dataset, and the uniform-leaf and exponential-leaf trees on the Bio .001 Gaussian dataset. (It also helped the exponential-leaf trees on both the Astro .001 Uniform and Astro .001 Gaussian datasets, but as we will discuss shortly this is probably mostly due to a peculiarity of the exponential distribution that causes it to interact poorly with the greedy variable selection method employed by the tree structure-learning algorithm.) This suggests that marginal distribution flattening can be a useful tool in cases where the variables are known to have complicated marginal distributions. However, it should not be applied blindly.

The learning times listed for the algorithms with marginal distribution flattening include the extra times required to learn all the one-dimensional density trees for the flattening process. Notably, the total learning times with flattening were nevertheless often shorter than the learning times without. This is probably due to the fact that marginal distribution flattening tends to result in density trees of more even depth when the midpoint branch threshold method is used. Density trees of even depth are faster to learn than skewed ones because on average each datapoint is involved in fewer leaf-learning attempts.

Throughout these results, multilinear interpolation almost always resulted in the most accurate density estimation, the exceptions being on the low-noise Bio datasets when marginal distribution flattening was employed, in which case exponential leaves worked better. However, it was also the most computationally expensive, and the difference in accuracy between multilinear interpolation and independent interpolation was statistically insignificant on the Astro datasets. Both multilinear and independent interpolation resulted in significantly greater accuracy than constant-density leaves in all tests.

The performance of the exponential-distribution trees was notably inconsistent — the best of all the estimators on the Bio .001 Uniform distribution when marginal distribution flattening was used, but the worst of all on the Astro datasets when it was not used. An examination of the Astro datasets and the properties of the truncated exponential distribution reveals one possible explanation. The Astro dataset has several variables in which the marginal distributions' means are very close to zero. As it turns out, the truncated exponential distribution has the following property: when a variable's distribution is concentrated close to one side of a leaf, replacing that leaf with a one-level density stump branching on that variable with a threshold anywhere near the center of the old leaf causes only a very small change in the log-likelihood of the data. This causes the greedy branch variable selection mechanism to prefer branching on *other* variables when exponential leaves are being used, whereas the other leaf distribution types we examine will tend to branch on the variable with the skewed distribution, which tends to eventually lead to more accurate trees. The fact that the marginal distribution flattening algorithm helps exponential-leaf density trees more than it helps others is probably due to the fact that the flattened distributions trigger this pathological behavior less frequently. Supplemental experiments in Appendix A.2 provide further evidence that the problems with exponential-distribution leaves are indeed caused by poor interactions with the greedy variable selection algorithm. It may be possible to improve the accuracy of exponential-leaf density trees by special-casing this situation or using better branch threshold selection algorithms; since exponential distributions are faster to fit than the other non-uniform distributions, further research along these lines would be useful.

4.8.7 Density trees vs. global mixture models

Throughout our experiments so far, all the probability models we have compared have been based on sparsely connected Bayesian networks in which no hidden variables are employed. While density trees appear to be good candidates to use for the conditional distributions of such networks, the question remains whether sparsely connected Bayesian networks are capable of accurately modeling real-world data, particularly when an appropriate network structure is not known beforehand. Is it possible to perform the required combinatorial search through network structures *and* learn all the necessary conditional distributions in less time than would be required to learn the parameters of a single unfactored joint model for the entire distribution, and have the resulting Bayesian network still be a more accurate density estimator than the unfactored model? In this section we provide experimental results suggesting that the answer is "yes", at least in some cases.

We compare our density-tree-based Bayesian network learning algorithm with AutoClass [CS96], an unsupervised learning algorithm for mixture models that employs an approximately Bayesian version of EM. In our experiments, AutoClass models each mixture component with a full-covariance Gaussian over the continuous variables and an independent multinomial distribution for each discrete variable. (Informal experiments with diagonal-covariance Gaussians rather than full-covariance ones resulted in worse density estimation.) For speed, we use the publicly available C implementation rather than the LISP implementation.

AutoClass is started off with numbers of mixture components that were close to the best numbers found in informal preliminary testing; this is to ensure we do not cripple the algorithm needlessly by having it waste too much time optimizing the parameters of distributions with far too many or too few mixture components. While AutoClass is an "anytime algorithm" in that it has no definite termination criterion and will supposedly find better solutions the longer it is run, in our experiments the amount of time we gave it was largely irrelevant to the accuracy of the resulting density estimators as long as it tried a nonzero number of mixtures with approximately the right number of components. In practice we would not normally know the correct number of mixture components to use ahead of time, but a roughly correct number can be found in a reasonable amount of time by trying mixtures with 1 component, 2 components, 4 components, 8 components, and so forth, until the performance starts dropping, or AutoClass begins returning mixtures with significantly fewer components than the mixtures are initialized with. (AutoClass apparently has no mechanism for adding components to a mixture "on the fly" during a parameter optimization run, but it does detect and delete components that it deems unnecessary.) When AutoClass terminates, we extract the mixture model it thinks is best, and we use its maximum-likelihood parameters. (We also add the same uniform-background "slack distribution" used in our density trees to handle outliers, as discussed in Section 4.4.4; however, this appears to improve the accuracy of the mixture models only negligibly.)

We perform two different tests on AutoClass with each Bio dataset: one in which AutoClass is given two hours (per cross-validation fold) to find a good mixture, and another in which it is given roughly the same amount of time taken by our density-tree-based Bayesian network learning algorithm. We attempted to give AutoClass three hours on each Astro dataset; however, as currently implemented, the algorithm apparently pays no attention to the clock except when it reinitializes EM with a new starting point, and on this dataset the algorithm can take hours for a single run of EM if the number of centers is large. This resulted in the algorithm taking an average of over five hours per cross-validation fold rather than three. (Informal experiments in which only a randomly selected subsample of 10,000 datapoints were used for training were attempted, but this seemed to result in less accurate density estimators despite

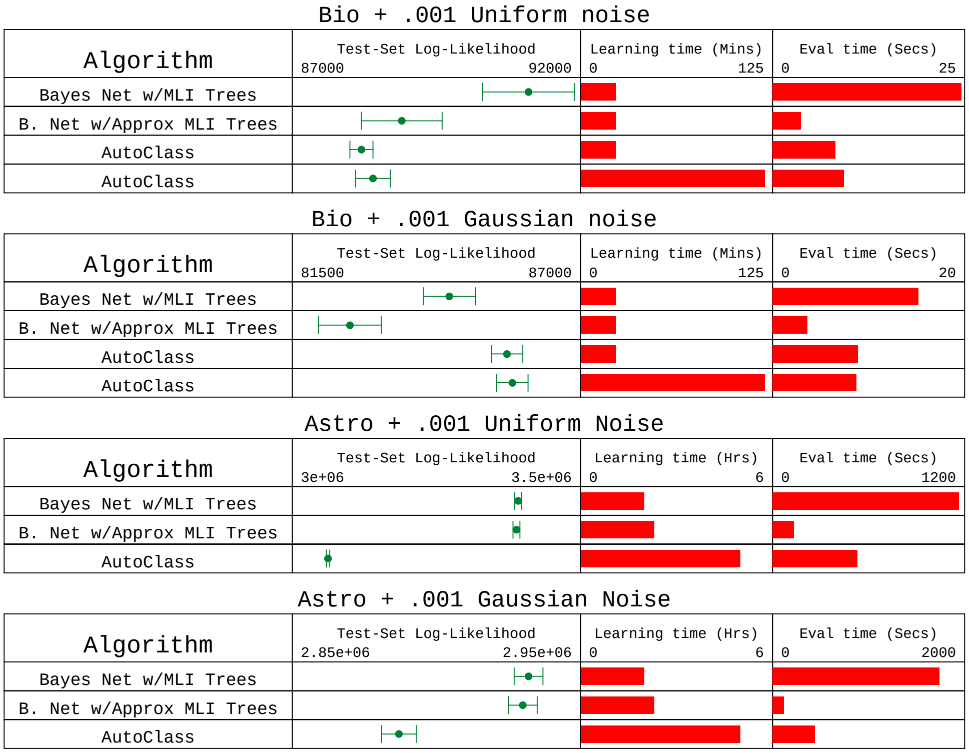


Figure 4.27: Automatically learned Bayesian networks w/density trees vs. global mixture models learned by AutoClass.

the greater number of EM trials that could be performed within the time limit with the smaller amount of data.)

We compare AutoClass to the same Bayesian network learning algorithm that was employed in Section 4.8.6: four iterations of our greedy-network algorithm are performed, with discretized data (8 values per variable) used for S_f, constant-density-leaf trees used for S_s, and multilinear-interpolation density trees for the final distribution. The results we list here are identical to those in Section 4.8.6, except here the results include the time required to learn the network structure.

The results are summarized in Figure 4.27.

AutoClass produced more accurate density estimators for the Bio dataset when the noise added to the dataset was Gaussian — the same type of distribution AutoClass uses for its mixture components. On the other hand, our density-tree-based Bayesian network algorithm produced more accurate density estimators on the Bio dataset was uniform — a distribution type more easily modeled by the multilinear leaves of the density tree. Thus, the results on the Bio dataset are somewhat inconclusive. However, our Bayesian network algorithm produced more accurate density estimation than Autoclass on the Astro dataset even when the noise added was Gaussian. Learning these Bayesian networks also took significantly less time; furthermore, evaluating the resulting networks was also faster than evaluating the mixture models when the network's density trees had been approximately conditionalized.

4.9 Conclusions, Related Work, and Possible Extensions

Throughout this chapter we have developed and evaluated a family of algorithms capable of quickly finding accurate factored probability density models over dozens of continuous and discrete variables from tens of thousands of datapoints. The potential applications for these algorithms are similar to the potential applications of Mix-nets described in Section 3.5. As with Mix-nets (Section 3.5.1), the conditional density tree-based algorithms can be applied to learning classifiers similar in nature to TAN classifiers [FGG97]. The results of some preliminary experiments along these lines are provided in Appendix A.5; however, further exploration is necessary to determine whether the resulting classifiers are useful and how they might be improved. The density tree-based models can also be used straightforwardly for anomaly detection (Section 3.5.2), although it is possible that the discontinuous nature of the probability densities modeled by conditional density trees makes it less useful than Mix-nets for that task. Inference may be also performed with density-tree-based Bayesian networks, either via sampling approaches such as likelihood weighting, or via message-passing algorithms employing dynamic discretization [KK97]. The density trees for which we have provided learning algorithms are very similar in nature to the representations used for discretization-based message-passing algorithms; thus, conditional density tree learning algorithms may be a natural choice to use when we are faced with a situation in which we wish to be able to perform message-passing-based inference but we do not know the distributions' parameters *a priori*. In order to guarantee convergence to the correct distribution, message-passing algorithms require the graphical models to be decomposable (i.e. chordal), and the network structure-search routine used here does not take this into account. However, efficient algorithms for performing searches over decomposable models have recently been developed [DGJ01]; modifying the network search algorithms used in this thesis to use these algorithms is one potentially interesting line of further research.

Because the conditional density tree algorithms here were designed for speed as well as accuracy, they present an appealing choice of representation to use for practical compression tasks. Naturally, when compressing continuous values, the compression must be lossy if it is to save a significant amount of space. Modifying our density tree learning algorithms to take a desired level of accuracy into consideration would be fairly straightforward. Furthermore, because the distributions represented by the trees decompose analytically into nonoverlapping regions — as opposed to the overlapping Gaussian mixture models used in Chapter 3 — no bits-back coding would be necessary.

In recent related research, a system called SPARTAN [BGR01] has been developed for lossily compressing datasets by using networks of CART-like decision and regression trees.[5] SPARTAN uses automatically learned Bayesian network structures as guides with which to create these networks. However, the network models learned by SPARTAN are not actually density estimators, and they have several important limitations. Each leaf of the trees SPARTAN employs only provides a point estimate of the variable being predicted. If this predicted value is insufficiently accurate for a particular datapoint, the actual value must be marked as an "outlier" and encoded via other means. Because only this point estimate is provided rather than a density over all possible values of the output variable, there is no mechanism with which to efficiently encode small corrections between the maximum-likelihood predicted values and the actual values. This in turn forces SPARTAN to restrict its prediction networks so that any given variable in the domain that is used to predict other variables cannot itself be predicted, and must therefore be encoded via other means — otherwise, prediction errors would accumulate. SPARTAN's CART-like trees could be modified so that the trees provide density estimates; however, as we have seen in Section 4.8.2, CART-like tree-based density estimation algorithms that do not allow splits on the variable being predicted can perform much worse than those that do. Having said that, SPARTAN's approach may be appropriate when decompression and compression speed is crucial, or when it is desirable to decompress certain variable values in a random access fashion without decompressing all the other variables.

[5] SPARTAN was developed after the material in Chapter 2 was published [DM99], although the authors appear unaware of previously existing research on Bayesian network-based compression. The material in this chapter of the thesis was developed independently of SPARTAN.

While allowing the density trees to split on the output variables would probably help compression performance, it is unclear whether other differences between the density tree algorithms examined here would have much practical impact on compression rates. For example, even if each variable value is 15% more likely on average when using density trees that use multilinear leaves rather than constant-density ones, this results in saving only a fraction of a bit per encoded value. In most such situations it would probably be better to simply use stratified conditional trees or conditionalized joint density trees with uniform-density leaves for maximum speed.

Because the density tree learning algorithms we use throughout this thesis treat the data in different subtrees independently, the resulting density estimates will generally have discontinuities at the tree's branch thresholds. It may be possible to improve the accuracy of the density estimators by attempting to enforce continuity whenever possible. For example, in the case where there is only one continuous variable, if we are given a density tree structure then it is easy to learn a set of linear interpolations in the leaves that provide a continuous density estimate. To do so, we could use the EM procedure described in Section 4.2.4 to fit all of the leaves' interpolations simultaneously; rather than having two independent hidden classes for every leaf, we would "tie together" the two classes to either side of any branch threshold. Unfortunately, there are problems with this type of approach. First, the simple divide-and-conquer nature of the tree structure-learning algorithm breaks down; the effect of performing a split in one part of the tree would now depend on the structure of other parts of the tree. Furthermore, this method does not generally apply to two or more dimensions unless the density tree has a gridlike structure. If the density tree structure is gridlike, then the multilinear interpolations within the different leaves can be constrained in a manner similar to the one-dimensional case discussed above to ensure continuity. However, if the density tree structure is not gridlike, then continuity cannot be enforced straightforwardly in this manner. For example, consider the simple two-dimensional density tree in Figure 4.28. Within leaf A, multilinear interpolation would interpolate between the values at corners 1, 2, 8, and 9; within leaf B, multilinear interpolation would use corners 2, 3, 4, and 5. As a result, a discontinuity will exist along the edge between corners 2 and 4 unless the value at corner 4 happens to be .75 times the value at corner 2 plus .25 times the value at corner 9; similar discontinuities exist along the edge (4, 6) and (6, 9). Naturally, this situation can be remedied by splitting leaf A into several new leaves, but these additional splits could create the need for further splits in other leaves adjacent to A, essentially cre-

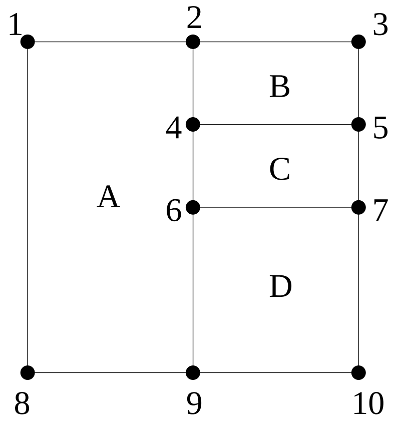


Figure 4.28: An illustration of a density tree structure for which ensuring continuity is difficult.

ating a gridlike structure. A complex algorithm exists for ensuring continuity in the two-dimensional case without creating arbitrarily many new splits [Gro89], but this does not scale to three dimensions or higher.

With this in mind, it may be worth investigating how well the interpolating density trees described in this chapter compare to grid-like density estimators. Such grid-like density estimators can be made continuous with little difficulty; this may help offset the negative effects of their fixed resolution. It may also be possible to combine multiple grids with different resolutions over different variables in a manner similar to CMACs [Alb81] or sparse grids [Zen91] to achieve higher predictive accuracy than with a single grid or tree over all the variables. However, evaluating these multiple grids would likely be significantly more computationally expensive than evaluating the density trees used here. Multivariate adaptive regression splines [Fri88] are also worth investigating.

The tree-based density estimators used in this chapter are more capable of handling high-arity discrete variables than the mixture tables used in Chapter 3, since the tree-learning algorithms will rarely create branches testing them unless these branches are justified by the data. Furthermore, high-arity discrete variables can help predict other variables even in joint density trees that never branch on them at all, since the leaves of the trees contain information about these variables' distributions. It may be possible to further increase the usefulness of such high-arity variables by using branches in which multiple variable values are mapped to the same child of the branch. Clustering techniques have been used in the past to find good choices for which values are mapped to which branch children (e.g. [BFOS84], [Cho91]) within the context of classification and regression trees; analogous techniques could be developed for interpolating density trees. However, employing such clustering techniques would probably significantly reduce the speed of our tree-learning algorithms.

While we have not yet carefully measured the memory consumption of the density trees employed throughout this thesis, informal examination of the amount used in our implementation suggests that they typically take up roughly as much memory as the data from which they are learned, to within an order of magnitude. For most applications, the amount of processor time required to learn the models is probably more restrictive than the amount of space taken up by the learned models, given the CPU speeds and memory capacities of current PCs. For compression, note that the number of bits required to represent the models compactly offline (e.g. on disk) is much less than the number used here in memory (where the model is optimized for efficient memory accesses, etc.). Furthermore, the tree-learning algorithms used in this chapter currently do not attempt to minimize space usage, but could be modified to do so; and even if the algorithms are not modified, they can be made to produce smaller trees simply by providing them with smaller training sets.

Chapter 5

Conclusions

5.1 Thesis contribution summary

In the first part of this thesis, I have developed Bayesian network-based algorithms that are capable of compressing discrete datasets with compression ratios much higher than achieved by state-of-the-art black-box compression programs, but that are still capable of megabyte-per-second decoding speeds (Chapter 2). In particular:

- I have shown that excellent compression can be achieved on real-world datasets by using arithmetic coding in conjunction with automatically learning Bayesian networks that are sparsely connected and employ no hidden variables. This allows decoding to be performed reasonably quickly (Sections 2.2.1 and 2.2.2).
- I have shown that even better compression can be achieved by automatically learning dynamic Bayesian networks that model dependencies between adjacent dataset items, possibly after the dataset has been sorted (Section 2.3).
- I have developed a type of modified Bayesian network to employ in conjunction with Huffman coding in order to address Huffman coding's limitations, and have developed algorithms for learning these networks (Section 2.3.1). This allows for significantly faster decoding than possible with arithmetic coding, with relatively little reduction in the compression rate.

In the second and third parts of the thesis, I have developed Bayesian network-based algorithms for learning joint distributions over discrete and continuous variables. In the second part (Chapter 3), I have shown how recently developed algorithms for quickly learning Gaussian mixture models over small sets of continuous variables [Moo99] can be practically used to model distributions over much larger sets of continuous and discrete variables via using automatically learned Bayesian networks. In the third part, which forms the bulk of the thesis, I have explored a wide variety of novel tree-based models for conditional density estimation, and shown how to use them in automatically learned Bayesian networks to model complex distributions over many continuous and discrete variables. In particular:

- I have described a wide variety of possible tree-based learning algorithms for representing joint distributions over small sets of variables (Sections 4.2 through 4.4).
- I have shown several novel ways of generalizing these models to learn and represent conditional distributions:
  - Stratified conditional density trees (Section 4.5.1), which are learned to directly model the conditional distribution. These trees can branch on the variable to be predicted once all branching on the input variables is finished, thus allowing the representation of complex conditional distributions.
  - Joint density trees that are used conditionally "on the fly" (Section 4.5.2). These are faster to learn than stratified conditional density trees, and are (somewhat surprisingly) frequently more accurate as well. Unfortunately, they are slow to evaluate.
  - Conditionalized joint density trees that are used either exactly (Section 4.5.3) or approximately (Section 4.5.4). These trees provide an appealing combination of fast learning, fast evaluation, and accuracy.
- I have provided a flexible class of heuristic Bayesian network structure-learning algorithms employing these conditional density trees (Section 4.6) to practically learn accurate distributions over dozens of continuous and discrete variables from many thousands of datapoints.
- I have presented a marginal distribution flattening method that can sometimes improve the performance of these tree-based conditional density estimators (Section 4.7).

- I have performed extensive experimental evaluations of all the above models (Section 4.8; Appendix A).

5.2 Possible avenues for further research

Possible extensions of the research performed in this thesis were already discussed at the ends of the appropriate chapters. In closing, we briefly recapitulate a few of the more important ones:

- Further comparisons of the compression techniques developed in Chapter 2 versus other techniques. In particular, it would be interesting to compare the compression rates achievable with the sparse Bayesian networks used here with that of the densely connected Bayesian networks used in Frey's work [Fre98], although such densely connected networks probably require significantly more computational overhead.

- Extensions of the compression techniques in Chapter 2 to adaptive coding — that is, allowing the parameters and structure of the networks to change as the data is processed in one pass.

- Application of the density trees developed in Chapter 4 to the compression of datasets containing continuous variables. A comparison of the compression rates and speed achievable with the density trees developed here versus that of the simpler trees used in SPARTAN [BGR01] would be particularly interesting.

- Application of the models developed in Chapter 3 and Chapter 4 to classification tasks. (Some preliminary results on applying density trees to classification are supplied in Appendix A.5, but further development and experimentation are warranted.)

Bibliography

[Ala96] S. Alag. Inference Using Message Propogation and Topology Transformation in Vector Gaussian Continuous Networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI96)*, 1996.

[Alb81] J. S. Albus. *Brains, Behaviour and Robotics.* BYTE Books, McGraw-Hill, 1981.

[Ber73] C. Berge. *Graphs and hypergraphs.* North-Holland, Amsterdam, 1973. Translated from French by E. Minieka.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Chapman & Hall, 1984.

[BFR98] P. S. Bradley, U. Fayyad, and C. A. Reina. Scaling EM (Expectation-Maximization) Clustering to Large Databases. Technical Report MSR-TR-98-35, Microsoft Research, Redmond, WA, November 1998.

[BGR01] S. Babu, M. Garofalakis, and R. Rastogi. SPARTAN: A Model-Based Semantic Compression System for Massive Data Tables. In *Proc. ACM SIGMOD*, 2001.

[Bri90] J. S. Bridle. Probabilistic Interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In F. F. Soulié and J. Hérault, editors, *Neurocomputing: Algorithms, Architectures and Applications*, pages 227–236. Springer-Verlag, 1990.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC-124, Digital Systems Research Center, May 1994.

[CH87] G. V. Cormac and R. N. Horspool. Data compression using dynamic Markov modelling. *The Computer Journal*, 30(6):541–550, December 1987.

[CH92] G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

[Chi96] D. Chickering. Learning Bayesian networks is NP-complete. In D. Fisher and H.-J. Lenz, editors, *Learning from Data: Artificial Intelligence and Statistics V*, pages 121–130. Springer-Verlag, 1996.

[Cho91] P. A. Chou. Optimal Partitioning for Classification and Regression Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4), April 1991.

[CKP85] Y. Choueka, S. T. Klein, and Y. Perl. Efficient variants of Huffman codes in high level languages. In *Proceedings of the Eighth International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 122–130. New York: ACM Press, June 1985.

[CL68] C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, IT-14:462–467, 1968.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* 1990.

[CMN$^+$95] J. Carpinelli, A. Moffat, R. Neal, W. Salamonsen, L. Stuiver, and I. Witten. *Word, Character, and Bit Based Compression Using Arithmetic Coding.* Available for download at ftp://munnari.oz.au/pub/arith_coder/, 1995.

[CS96] P. Cheeseman and J. Stutz. Bayesian classification (AutoClass): Theory and results. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurasamy, editors, *Advances in Knowledge Discovery and Data Mining.* MIT Press, 1996.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley, 1991.

[CW84] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.

[DGJ01] A. Deshpande, M. Garofalakis, and M. I. Jordan. Efficient stepwise selection in decomposable models. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI2001)*, 2001.

[DH73] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.

[DHNZ95] P. Dayan, G. E. Hinton, R. M. Neal, and R. S. Zemel. The Helmholtz machine. *Neural Computation*, 7:889–904, 1995.

[DK88] T. Dean and K. Kanazawa. Probabilistic temporal reasoning. In *AAAI-88 Proceedings*, pages 524–528, 1988.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, B 39:1–39, 1977.

[DM95] E. Driver and D. Morrell. Implementation of Continous Bayesian Networks Using Sums of Weighted Gaussians. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI95)*, 1995.

[DM99] S. Davies and A. Moore. Bayesian Networks for Lossless Dataset Compression. In *Conference on Knowledge Discovery in Databases (KDD-99)*, 1999.

[DM00] S. Davies and A. Moore. Mix-nets: Factored Mixtures of Gaussians in Bayesian Networks with Mixed Continuous and Discrete Variables. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI2000)*, 2000.

[FG96a] N. Friedman and M. Goldszmidt. Discretizing Continuous Attributes While Learning Bayesian Networks. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 157–165, 1996.

[FG96b] N. Friedman and M. Goldszmidt. Learning Bayesian Networks with Local Structure. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI96)*, 1996.

[FG97] N. Friedman and M. Goldszmidt. Sequential update of Bayesian network structure. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI97)*, 1997.

[FGG97] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.

[FGL98] N. Friedman, M. Godszmidt, and T. J. Lee. Bayesian Network Classification with Continuous Attributes: Getting the Best of Both Discretization and Parametric Fitting. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML)*, 1998.

[FHD96] B. J. Frey, G. E. Hinton, and P. Dayan. Does the wake-sleep algorithm produce good density estimators? In *Advances in Neural Information Processing Systems 8*. MIT Press, 1996.

[FHT$^+$02] E. Frank, M. Hall, L. Trigg, R. Kirkby, G. Schmidberger, M. Ware, X. Xu, R. Bouckaert, Y. Wang, S. Inglis, and I. H. Witten. *Weka 3 - Data Mining with Open Source Machine Learning Software in Java.* Available at http://www.cs.waikato.ac.nz/˜ml/weka/, 1998-2002.

[FI93] U. M. Fayyad and K. B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proc. of 13th Int. Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.

[FMR98] N. Friedman, K. Murphy, and S. Russell. Learning the Structure of Dynamic Probabilistic Networks. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI98)*, 1998.

[FN00] N. Friedman and I. Nachman. Gaussian Process Networks. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI2000)*, 2000.

[FNP99] N. Friedman, I. Nachman, and D. Peér. Learning Bayesian Network Structures from Massive Datasets: The Sparse Candidate Algorithm. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI99)*, pages 206–215, 1999.

[Fre98] B. J. Frey. *Graphical Models for Machine Learning and Digital Communication.* MIT Press, 1998.

[Fri88] J. H. Friedman. Multivariate Adaptive Regression Splines. Technical Report No. 102, Department for Statistics, Stanford University, 1988.

[GH94] D. Geiger and D. Heckerman. Learning Gaussian Networks. Technical Report MSR-TR-94-10, Microsoft Research, 1994.

[Gro89] E. Grosse. LOESS: Multivariate Smoothing by Moving Least Squares. In L. L. Schumaker C. K. Chul and J. D. Ward, editors, *Approximation Theory VI*. Academic Press, 1989.

[HDCM89] N. H. Heydon-Dumbleton, C. A. Collins, and H. T. MacGillivary. *MNRAS*, 268, 1989.

[HDFN95] G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The wake-sleep algorithm for unsupervised neural networks. *Science*, 268:1158–1161, 1995.

[HG95] D. Heckerman and D. Geiger. Learning Bayesian networks: a unification for discrete and Gaussian domains. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI95)*, 1995.

[HGC95] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: the combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

[HL90] D. S. Hirschberg and D. A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, April 1990.

[HM97a] David Heckerman and Christopher Meek. Embedded Bayesian network classifiers. Technical Report MSR-TR-97-06, Microsoft Research, Redmond, WA, March 1997.

[HM97b] David Heckerman and Christopher Meek. Models and selection criteria for regression and classification. In *Proceedings of Thirteenth Conference of Uncertainty in AI (UAI97)*, pages 223–228, Providence, RI, 1997. Morgan Kaufmann.

[HT95] R. Hofmann and V. Tresp. Discovering Structure in Continuous Variables Using Bayesian Networks. In D. S. Touretzsky, M. C. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*. MIT Press, 1995.

[Huf51] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, 1951.

[HZ94] G. E. Hinton and R. S. Zemel. Autoencoders, Minimum Description Length and Helmholtz Free Energy. In *Advances in Neural Information Processing Systems 6*. MIT Press, 1994.

[JGJS98] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul. An Introduction to Variational Methods for Graphical Models. In M. I. Jordan, editor, *Learning in Graphical Models*. Kluwer Academic Publishers, 1998.

[JL95] G. John and P. Langley. Estimating Continuous Distributions in Bayesian Classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI95)*, 1995.

[JP99] T. Jebara and A. Pentland. The Generalized CEM Algorithm. In *Advances in Neural Information Processing Systems 12*. MIT Press, 1999.

[KK97] A. Kozlov and D. Koller. Nonuniform dynamic discretization in hybrid networks. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI97)*, 1997.

[Koh96] R. Kohavi. Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, 1996.

[KS80] R. Kinderman and J. L. Snell. *Markov Random Fields and Their Applications.* American Mathematical Society, Providence USA, 1980.

[Lau96] S. Lauritzen. *Graphical Models.* Oxford University Press, 1996.

[LB94] W. Lam and F. Bacchus. Learning Bayesian belief networks: an approach based on the MDL principle. *Computational Intelligence*, 10:269–293, 1994.

[MC98a] S. Monti and G. F. Cooper. A Multivariate Discretization Method for Learning Bayesian Networks from Mixed Data. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI98)*, 1998.

[MC98b] S. Monti and G. F. Cooper. Learning Hybrid Bayesian Networks from Data. In M. I. Jordan, editor, *Learning in Graphical Models*. Kluwer Academic Publishers, 1998.

[MC99] S. Monti and G. F. Cooper. A Latent Variable Model for Multivariate Discretization. In *Proceedings of the Seventh International Workshop on AI & Statistics (Uncertainty 99)*, 1999.

[ML98] A. W. Moore and M. S. Lee. Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. *Journal of Artificial Intelligence Research*, 8, 1998.

[MM73] J. N. Morgan and R. C. Messenger. *THAID: a sequential search program for the analysis of nominal scale dependent variables.* Ann Arbor: Institute for Social Research, University of Michigan, 1973.

[MN83] P. McCullagh and J. A. Nelder. *Generalized Linear Models.* Chapman and Hall, 1983.

[MNW95] A. Moffat, R. Neal, and I. H. Witten. Arithmetic Coding Revisited. In *Proceedings of the IEEE Data Compression Conference*, March 1995.

[Moo99] A. W. Moore. Very Fast EM-based Mixture Model Clustering using Multiresolution kd-trees. In *Advances in Neural Processing Systems 12*. MIT Press, 1999.

[Moo00] A. W. Moore. The Anchors Hierarchy: Using the triangle inequality to survive high dimensional data. 2000.

[MSD97] A. W. Moore, J. Schneider, and K. Deng. Efficient Locally Weighted Polynomial Regression Predictions. In *Proceedings of the 1997 International Machine Learning Conference*. Morgan Kaufmann, 1997.

[MT97] A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, October 1997.

[NCC$^+$01] R. C. Nichol, S. Chong, A. J. Connolly, S. Davies, C. Genovese, A. M. Hopkins, C. J. Miller, A. W. Moore, D. Pelleg, G. T. Richards, J. Schneider, I. Szapudi, and L. Wasserman. Computational AstroStatistics: Fast and Efficient Tools for Analysing Huge Astronomical Data Sources. Invited talk at Statistical Challenges in Modern Astronomy III, July 2001.

[NH98] R. M. Neal and G. E. Hinton. A view of the EM algorithm that justifies incremental, sparse, and other variants. In M. I. Jordan, editor, *Learning in Graphical Models*. Kluwer Academic Publishers, 1998.

[Pas76] R. Pasco. Source Coding Algorithms for Fast Data Compression. Ph.D. Thesis, Stanford University, 1976.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan-Kaufmann, 1988.

[PTVF92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C (Second Edition)*. Cambridge University Press, 1992.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[Ris76] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, May 1976.

[Ros56] M. Rosenblatt. Remarks on Some Nonparametric Estimates of a Density Function. *Ann. Math. Statist.*, 27:832–837, 1956.

[Sah96] M. Sahami. Learning Limited Dependence Bayesian Classifiers. In *KDD-96: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 335–338. AAAI Press, 1996.

[Say96] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 1996.

[Sch78] G. Schwarz. Estimating the dimension of a model. *Annals of Statistics*, 6:461–464, 1978.

[Sco92] D. Scott. *Multivariate Density Estimation*. John Wiley & Sons, 1992.

[Sie88] A. Siemiński. Fast decoding of Huffman codes. *Information Processsing Letters*, 26(5):237–241, May 1988.

[SM00] P. Sand and A. W. Moore. Fast Structure Search for Gaussian Mixture Models. Submitted to Knowledge Discovery and Data Mining 2000, 2000.

[Wel84] T.A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, pages 8–19, June 1984.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc., 1999.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the Association for Computing Machinery*, 30:520–540, June 1987.

[Zen91] C. Zenger. Sparse grids. In W. Hackbusch, editor, *Parallel Algorithms for Partial Differential Equations, Proceedings of the Sixth GAMM-Seminar, Kiel, 1990*, pages 241–251. Vieweg, Braunschweig, 1991.

[Ziv78] J. Ziv. Coding theorems for individual sequences. *IEEE Transactions on Information Theory*, 24:389–394, 1978.

[ZL77] J. Ziv and A. Lempel. A Universal Algorithm for Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

Appendix A

Supplemental experimental results

A.1 Pruning, branch variable selection, and branch threshold selection

Throughout all other experiments in this thesis, we have used the greedy algorithm described in Section 4.4.1 for selecting branch variables, and the post-pruning method described in Section 4.4.3 for determining when to use leaves rather than branches. Furthermore, each branch on a continuous variable always used the midpoint of the branch variable's currently valid range as its split threshold. In this section we perform a series of experiments in which these aspects of the algorithms are varied:

- The "Joint Uniform" and "Joint MLI" ("MLI" for "multilinear interpolation") algorithms use the same pruning, branch variable selection, and branch threshold selection strategies as before.
- The "Joint Uniform w/Grid Select" and "Joint MLI w/Grid Select" algorithms are identical to "Joint Uniform" and "Joint MLI," respectively, except the variable on which to branch is determined by the "taking turns" strategy described in Section 4.4.1.
- The "w/Stopping" algorithms are identical to the correponding default algorithms except the pruning strategy has been changed from post-pruning to stopping (Section 4.4.3).

- The “w/Greedy Threshold” algorithms are identical to the corresponding default algorithms except the threshold selection algorithm has been changed from the midpoint method to the more expensive method described in Section 4.4.2.

The results are shown in Figures A.1 and A.2.

When uniform-density leaves are employed, the greedy variable selection algorithm actually performs worse than the simpler “taking turns” method on all the synthetic datasets and on the Bio dataset with high-magnitude noise. When multilinearly interpolated leaves are employed instead, however, greedy variable selection never performs worse than the “taking turns” method, and performs significantly better on the Bio and Astro datasets. This may be due to the fact that the greedy algorithm is able to essentially “look further ahead” due to the added representational power afforded by the nonuniform leaves it uses in the one-level density stumps it uses for testing. The fact that greedy variable selection does not generally help more may also be partially due to the relatively small number of variables modelled per tree in these experiments, and partially due the fact that the two variables in each of the synthetic datasets are essentially identical to each other on a global scale.

Comparing the results of the stopping and post-pruning algorithms reveals that when uniform-density leaves are used, the stopping algorithm generally results in worse accuracy than the post-pruning algorithm; however, the stopping algorithm often results in better accuracy than post-pruning when multilinearly interpolated leaves are employed. See Section 4.4.3 for a discussion of this phenomenon.

Comparing the “w/Greedy Threshold” algorithms with the algorithms employing the default midpoint split threshold reveals that the more complicated threshold algorithm does in fact significantly improve the accuracy of trees with uniform-density leaves in most cases. However, the accuracy of these trees is always still significantly worse than that of trees using multilinear leaves and the midpoint threshold method. Futhermore, employing the more complicated threshold-choosing algorithm significantly increases the computational cost of learning — so much so that some of the alternative learning algorithms using multilinear leaves are usually both faster and more accurate. The more complicated threshold-choosing algorithm usually *decreased* the accuracy of the trees using multilinear leaves. An analogous version of the more complicated threshold-choosing algorithm tuned for multilinear-leaf trees rather than uniform-leaf trees might in fact improve the accuracy of multilinear-leaf trees; however, such an algorithm would be even more prohibitively time-consuming.

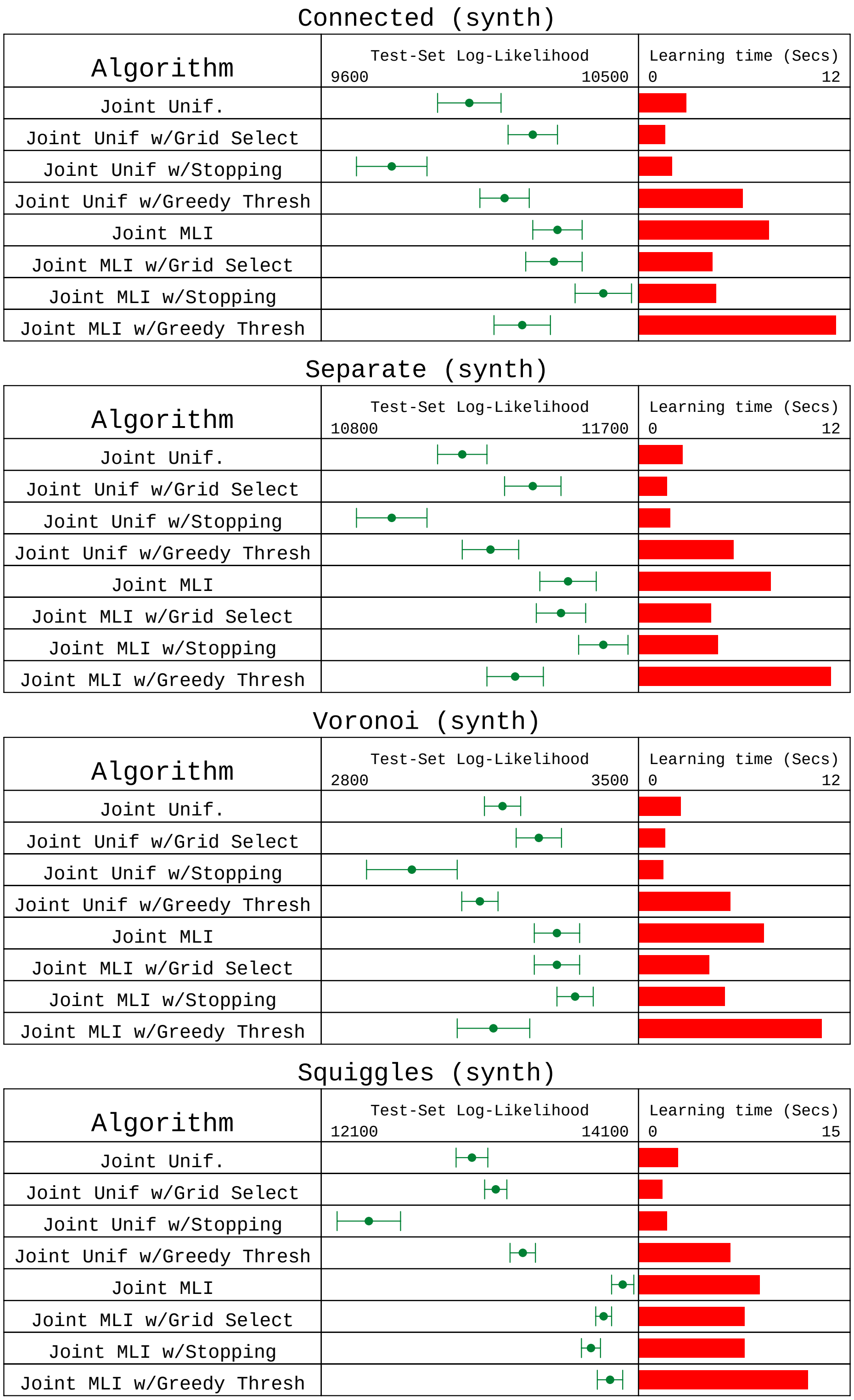
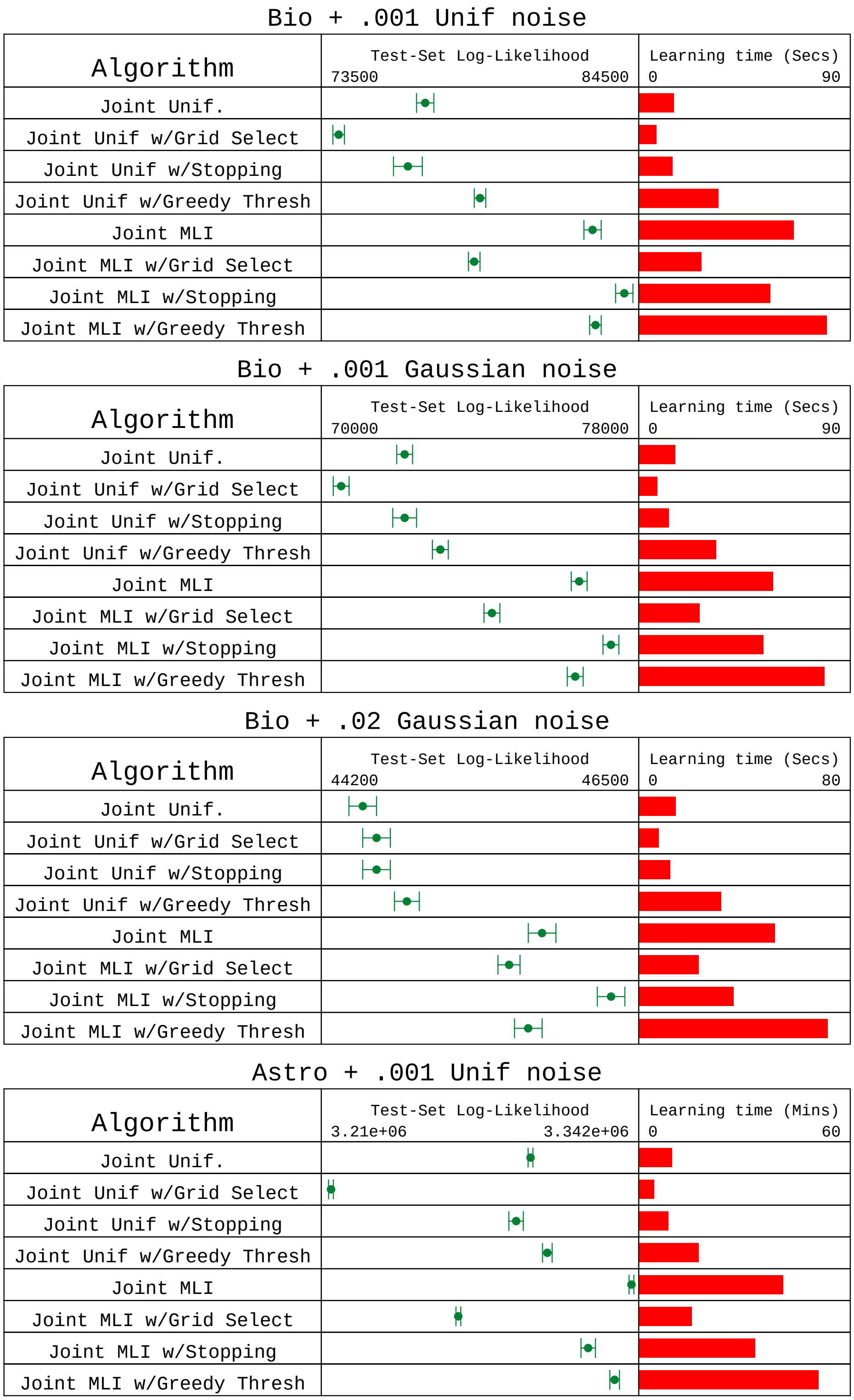


Figure A.1: Experiments on alternative pruning, branch variable selection, and branch threshold selection methods (synthetic datasets)



Bio + .001 Unif noise

Bio + .001 Gaussian noise

Bio + .02 Gaussian noise

Astro + .001 Unif noise

Figure A.2: Experiments on alternative pruning, branch variable selection, and branch threshold selection methods (scientific datasets)

A.2 "Switcheroo" experiments

In this section we perform a series of experiments in which one learning algorithm learns a density tree, and then another learning algorithm is constrained to use the previously learned tree's branching structure. These experiments serve as a sanity check ensuring that the differences in various algorithms' accuracies are not entirely due to subtle effects they have on the greedy density tree structure-learning algorithm. (For example, it might be conceivable that the main reason interpolated leaves perform better than uniform ones is that one-level decision stumps with interpolated leaves give more accurate "hints" to the greedy variable selection algorithm, not that they necessarily make better leaves to actually use in the final tree.)

Since the structure of the tree is fixed, we no longer have to hold out part of the training data for pruning or evaluating different branch variables, so we allow it to train the leaf distributions using *all* of the training data. Naturally, this by itself may cause the relearned tree to be more accurate, so we include experiments in which the second learning algorithm is identical to the first to control for this added accuracy. This type of leaf-relearning procedure was not used in any experiments outside this section, but it could have been used to slightly increase the accuracy of the resulting trees.

Joint vs. Stratified trees

In this series of experiments we verify the "soft branching" hypothesis by comparing the accuracy of identically structured stratified conditional density trees and joint density trees. While we're at it, we also examine the utility of refitting density trees' distributions with all the training data once their structures have been determined.

We compare five different density tree learning algorithms:

- "Stratified Cond MLI (Relearn)": stratified conditional density trees with leaves employing multilinear interpolation. After the tree's structure has been determined, all the leaves are refitted using all the training data.
- "Stratified Joint MLI (Relearn)": Joint density trees with leaves employing multilinear interpolation. Their structure is restricted so that all branching on the parent variables occurs before any branching on the child variable. The leaves are refitted after the tree structure is determined.

- “Stratified Cond MLI to Joint”: stratified conditional density trees with leaves employing multilinear interpolation are learned; then the trees are transformed into joint density trees that are then used conditionally as described in Section 4.5.2. In the process, all leaves are refit with all the training data.

- “Joint MLI (Relearn)”: joint density trees with multilinearly interpolated leaves are learned, their leaves are refitted, and then they are used conditionally.

- “Joint MLI (No Relearn)”: joint density trees with multilinearly interpolated leaves are learned and used conditionally, but their leaves are not relearned once the tree’s structure is fixed. (These results appear elsewhere in the thesis and are included here again for convenience.)

The network structure used for the Bio dataset experiments was identical to the structure used in Section 4.8.2. The results are shown in Figures A.3 and A.4.

On the datasets with the sharpest distributions (Squiggles, Bio .001 Unif, and Bio .001 Gaussian), “Two-level Cond MLI to Joint” significantly outperformed “Two-level Cond MLI (Relearn)”, despite the fact that the tree structures and leaf distributions used were identical. This clearly illustrates that “soft branching” — that is, learning distributions over the parent variables in the tree’s leaves, and then employing these to determine the likelihood with which each leaf generated the datapoint by using Bayes’s rule — can by itself somtimes lead to more accurate density estimates than those obtained more straightforwardly from the corresponding two-level conditional trees. It leads to slightly worse performance than that of the identically structured two-level conditional trees on the other problems. However, if the tree-learning algorithm is allowed to optimize the joint distribution rather than the conditional distribution — as the “Two-level Joint MLI (Relearn)” algorithm does — then the “soft branching” helps even more, and joint density trees perform better than two-level conditional density trees across the board. If the “Two-level” restriction on the joint density tree structure is removed (“Joint MLI (Relearn)”), the improvement becomes even greater.

Comparing “Joint MLI (Relearn)” with “Joint MLI (No Relearn)” reveals that refitting the tree leaves with the entire training dataset after the tree structure is fixed does result in better density estimation, but the improvement is usually relatively small compared to the other differences between learning algorithms.

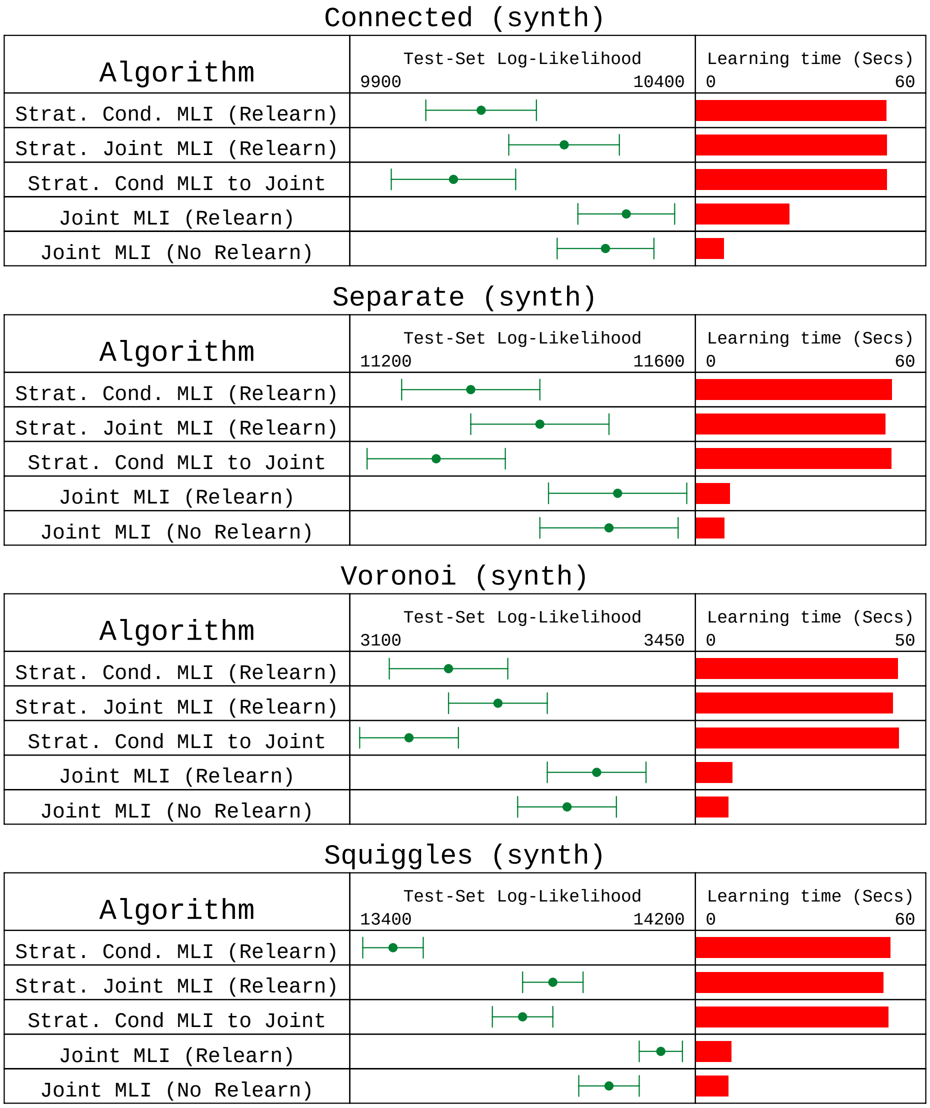


Figure A.3: Supplemental experiments on joint vs. stratifed trees (synthetic datasets)

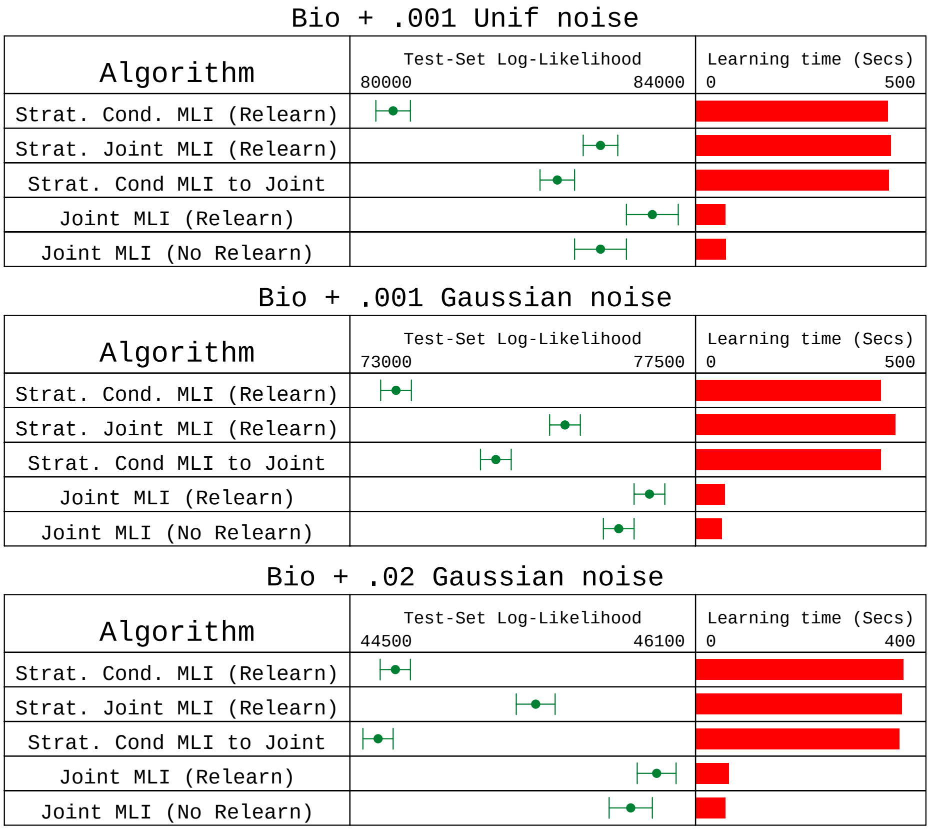


Figure A.4: Supplemental experiments on joint vs. stratifed trees (scientific datasets)

Constant vs. non-constant leaves

In this section we perform experiments in which density trees are learned with uniform leaves, but after the trees' structures are fixed, these leaves are replaced with multilinearly interpolated leaves ("Joint Unif to MLI") fitted with the entire training set. We compare these with refitted uniform-leaf density trees ("Joint Unif (Relearn)") and refitted multilinear-leaf density trees ("Joint MLI (Relearn)"). The results are shown in Figure A.5.

Replacing uniform-density leaves with multilinearly interpolated ones while holding the tree structure fixed significantly improved accuracy on many datasets (and never decreased accuracy). Therefore, the improvement in accuracy acquired by using multilinearly interpolated leaves cannot be due entirely to the differences in tree structure. It is also worth noting that using constant-density leaves during the tree structure-learning process and then replacing them with multilinear ones is much less computationally expensive than using multilinear leaves throughout the entire tree structure-learning process.

A.3 Effect of the greedy network-learning algorithm's MAXCHANGES parameter

In this section we illustrate the usefulness of setting the MAXCHANGES parameter higher than 1, thus allowing the algorithm to employ "out-of-date" estimates for which possible arc additions and deletions are the most promising. We compare the performance of the "Const→ML" algorithm used in Section 4.8.5, which has MAXCHANGES set to 10, with a version that has MAXCHANGES set to 1. The results are shown in Figure A.6.

While each iteration of the greedy algorithm took slightly less time with MAXCHANGES set to 1, it clearly improved the network structure significantly less per iteration. In fact, it's not clear from the learning curves whether the learner with MAXCHANGES set to 1 will ever converge to networks as accurate as those found by the learner with MAXCHANGES set to 10 — somewhat surprising, since one might expect the algorithm employing "out-of-date" estimates for the utility of various arc changes to get caught in worse local optima as a result. While this single experiment does not prove that using high values for MAXCHANGES will always

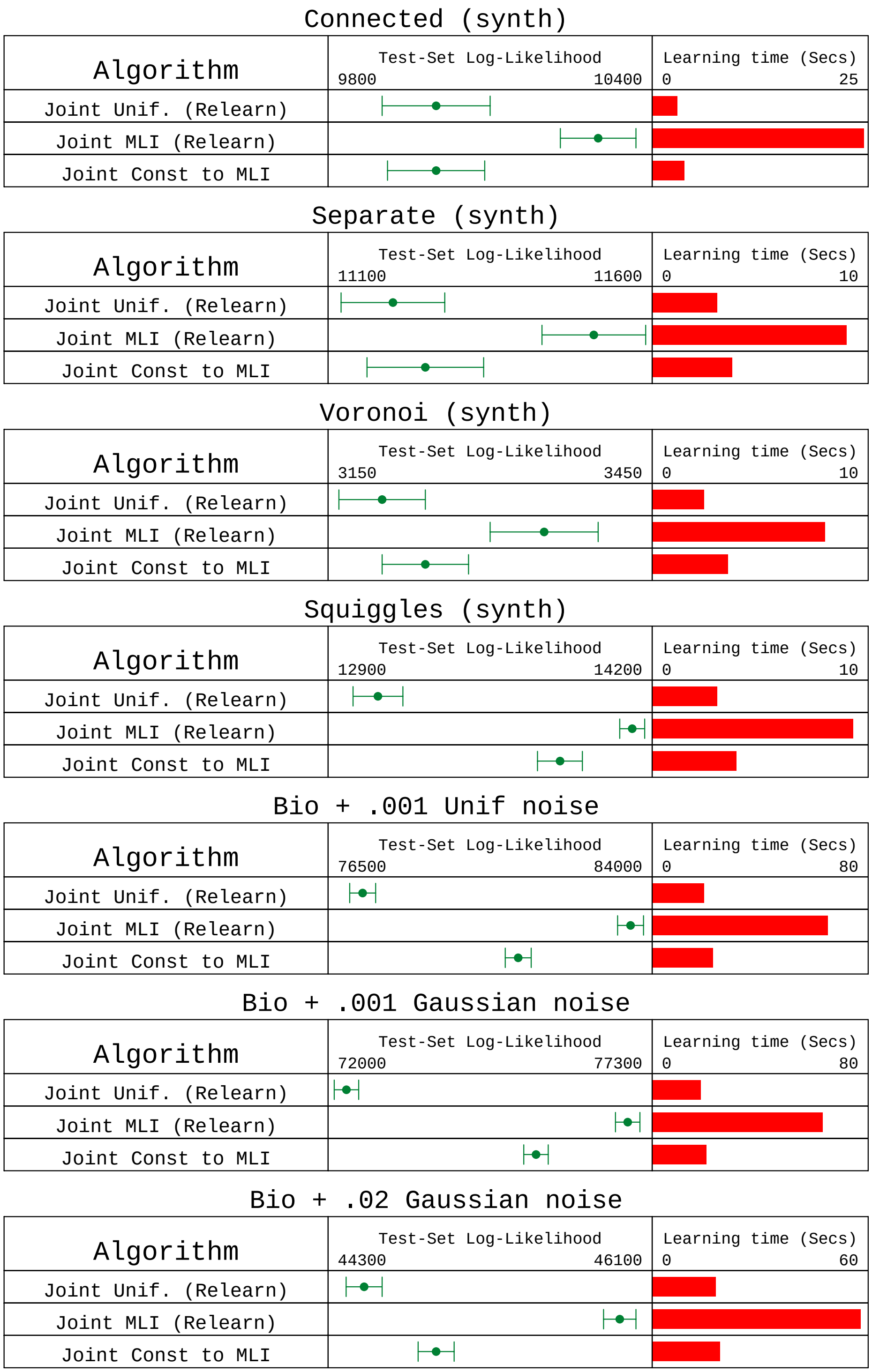


Figure A.5: Supplemental experiments on constant vs. multilinearly interpolated leaves

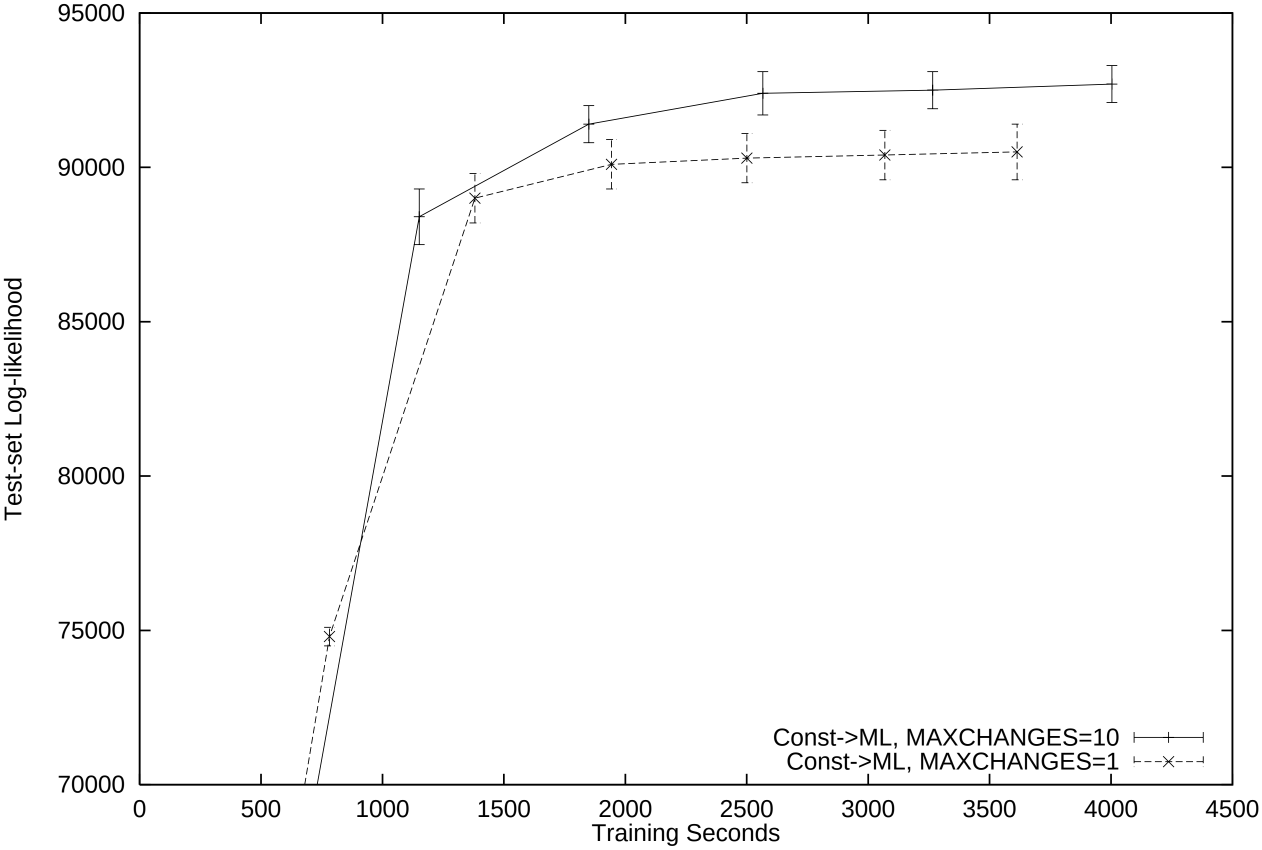


Figure A.6: Performance of the greedy structure-learning algorithm with MAXCHANGES set to 10 vs. with MAXCHANGES set to 1.

help, it does suggest that it is not generally a bad idea to try.

A.4 Diagnostic experiments on exponential-distribution density trees

In this section we perform experiments that support our hypothesis for why density trees with exponential-distribution leaves performed poorly on the Astro dataset (as shown in Section 4.8.6): namely, that the truncated exponential distribution has properties that cause it to interact poorly with the density tree learner's greedy variable selection algorithm when the data is strongly concentrated near the side of the current region's bounding box. We compare four density tree learning algorithms:

- "Joint Exp Greedy (Relearn)": Exponential-leaf density trees learned with the greedy variable selection algorithm described in Section 4.4.1; after the structure has been fixed, the leaf distributions are refitted with the entire training set.
- "Joint Exp Grid (Relearn)": the same as "Joint Exp Greedy (Relearn)" except the tree's branch variables are selected using the "taking turns" approach described in Section 4.4.1.
- "Joint Unif Grid (Relearn)": the same as "Joint Exp Greedy (Relearn)" except the tree uses constant-density leaves rather than exponential ones.
- "Joint Unif to Exp": the same as "Joint Unif Grid (Relearn)", except the constant-density leaves are replaced with exponential-density leaves after the tree's structure has been learned using the greedy variable selection mechanism in conjunction with constant-density leaves.

Results of these four algorithms on two different versions of the Astro dataset (namely, with uniform noise of magnitude .001 added, and with Gaussian noise with a standard deviation of .001 added) are shown in Figure A.7.

The results show that when exponential leaves are used in conjunction with the greedy structure-learning algorithm, performance is poor — significantly worse than using constant-density leaves rather than exponential ones. However, if the "grid" branch variable method is employed, or if a constant-leaf density tree's leaves are replaced with exponential leaves after the tree's structure has been determined, then performance improves. The results in Section A.1 indicate that greedy variable selection is significantly better than the "grid" branch variable selection mechanism when

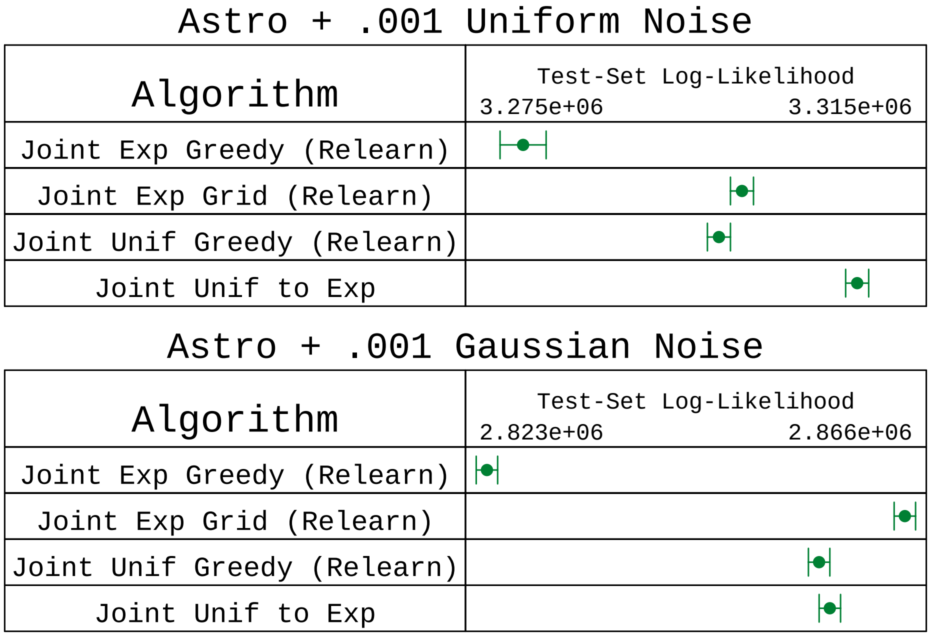


Figure A.7: Supplemental experiments on exponential-distribution density trees

the leaves are constant-density or multilinearly interpolated, so the problem is not generally attributable to the greedy variable selection method itself.

A.5 Preliminary experiments on using interpolating density trees for classification

The work in this thesis on density trees has focused primarily on estimating conditional distributions of continuous variables. In this section, we present the results of some preliminary experiments on using density tree-based algorithms for classification – that is, for predicting the value of a discrete variable.

The most direct approach to using density trees for classification is to simply learn a single density tree over the discrete output variable and some set of input variables. As discussed in Section 4.5.1, when the output variable is discrete and the type of tree used is a stratified conditional density tree, then the classifier is identical in form to the decision trees that have frequently been used in the past. However, none of the tree-learning algorithms described in this thesis have used any special-purpose methods to improve the performance of the trees on classification tasks, whereas classical decision-tree learning algorithms (e.g. ID3 [Qui86] and CART [BFOS84]) have branch threshold selection methods and pruning methods specifically designed for classification. Thus, one might expect these special-purpose decision-tree learning algorithms to generally perform better at classification than the density trees developed in this thesis.

First, we compare the classification performance of joint density trees using multilinear interpolation within the leaves versus the performance of J48, an implementation of the C4.5 [Qui93] decision-tree learning algorithm publically available in the Weka machine learning library [FHT$^+$02]. Since joint density trees can only be effectively learned when the number of input variables is relatively small, we use a version of the greedy Bayesian network-learning algorithm described in Figure 4.12 to perform feature selection. Namely, the MAXCHANGES parameter is set to 1; only arcs directly from the input variables to the output variable are considered; and the greedy algorithm is applied iteratively for five iterations, with the starting network for each iteration being the final network of the last. This effectively implements a best-first forward feature selection algorithm.

Figure A.8 shows the classification accuracy of the resulting trees on one of the discrete variables ("TNF") in the Bio dataset, with various forms and magnitudes of noise added to the continuous input variables. We show the mean classification accuracy in a ten-fold cross-validation, as well as its empirically estimated 95% confidence interval. The corresponding results are also shown for J48, both with and without a similar best-first forward feature selection algorithm employed. (Note, however, that because feature selection with J48 was rather slow, we let J48 "cheat" by using the entire dataset for feature selection, rather than have it perform feature selection once for each of the ten cross-validation splits.)

All the learners achieved very high accuracy on this problem when little or no noise was present; the variable appears to be a determininstic function (or nearly so) of one or two other variables in the domain. The joint density trees performed significantly better at predicting the target variable than J48 did in the two cases where noise was present in small amounts. However, they performed significantly worse in the cases where no noise was present or when high amounts of noise were present. Examination of the "Bio + .02 Gaussian noise" case revealed that performance of joint density trees was poor primarily because the feature selection algorithm chose poor feature sets in a few of the cross-validation splits; manually fixing the input features to a set of four

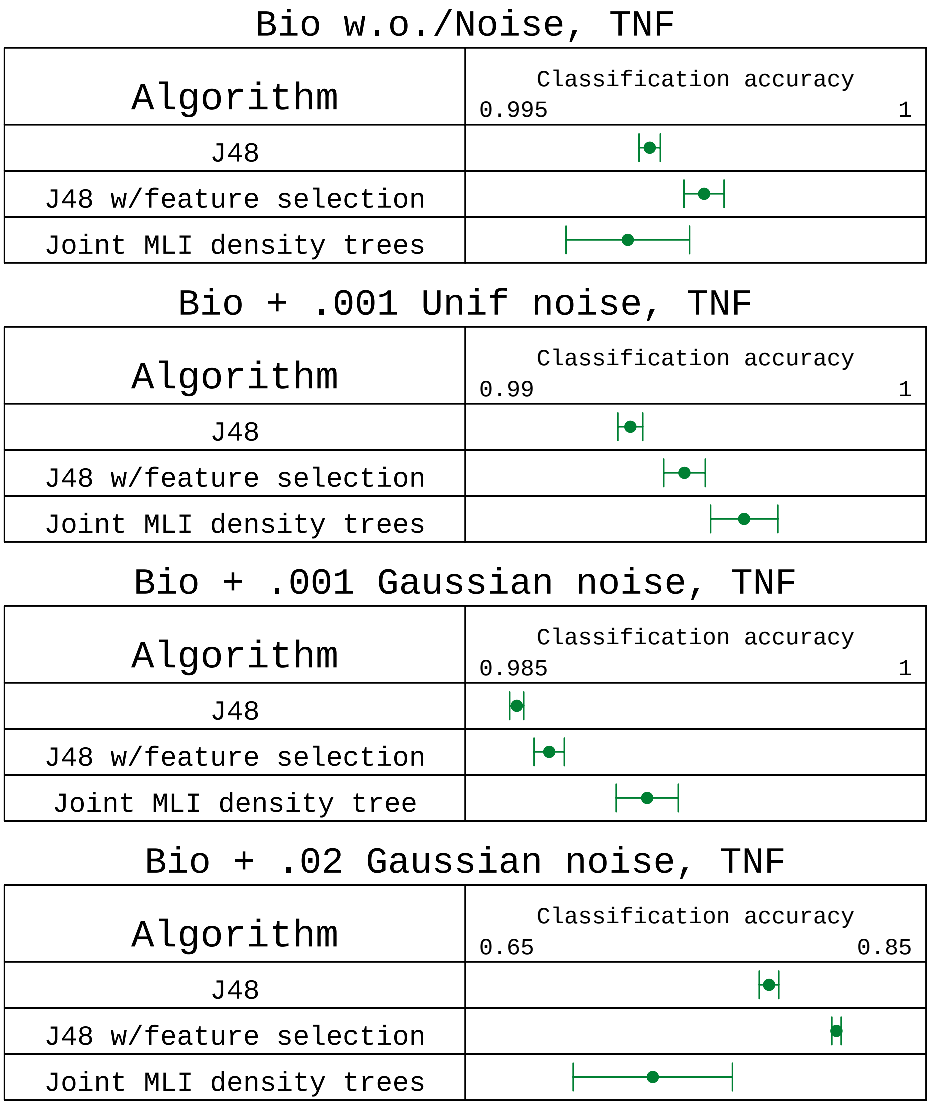


Figure A.8: Classification accuracy comparisons of J48 vs. joint multilinear density trees for the “TNF” variable in the Bio dataset, given different amounts and types of noise on the input variables.

"good" inputs allowed the joint density trees to perform better than J48 without feature selection, although still not as good as J48 with feature selection.

Figure A.9 shows the classification accuracy of joint multilinear density trees versus J48 on one of the discrete variables ("Type") in the Astro dataset. Feature selection with J48 was extremely slow — we aborted it after over 10 hours of CPU time — so we fixed the inputs to three features that had been selected by the Bayesian network-structure learning algorithm using joint multilinear density trees. Thus, the results for "J48, 3 input vars" in Figure A.9 should be considered only a lower bound on how well J48 would have performed with proper feature selection. (We also show the results for J48 with no feature selection; these results are significantly worse than with the manually selected features.) For consistency, we also fix the input feature set of the joint multilinear density trees to these three features.

Joint multilinear density trees had roughly the same accuracy as J48 in the "Astro + .001 Gaussian noise" case, but were significantly worse in the other two cases. Notably, the classification accuracy of joint multilinear density trees in the case where no noise had been added to the dataset was actually worse than the cases where small amounts of noise had been added. This suggests that the joint density trees may have been spending too much of their representational power modeling sharp peaks in the distributions of the input variables.

A second possible approach to using density trees for classification is to use them in Bayesian networks in which arcs go from the target variable to the input variables, rather than the other way around as in the previously discussed approach. When no other arcs are present, the Bayesian network effectively implements a Naive Bayes classifier. Further arcs between the input variables can be added to model important dependencies between them, as in TAN classifiers [FGG97]. We might expect this second approach to be more accurate than the first in cases where the target variable is better modelled as a noisy function of many input variables, rather than a near-deterministic function of a few inputs.

Figure A.10 shows the results of some preliminary experiments on using TAN-like networks with joint density trees for classification. The Bayesian network-learning algorithm employed is a modification of the greedy network-learning algorithm described in Figure 4.12 that evaluates candidate arc removals and additions based on their influence on the total conditional log-likelihood of the output variables given the input variables, as evaluated over a holdout set. A forward feature selection algo-

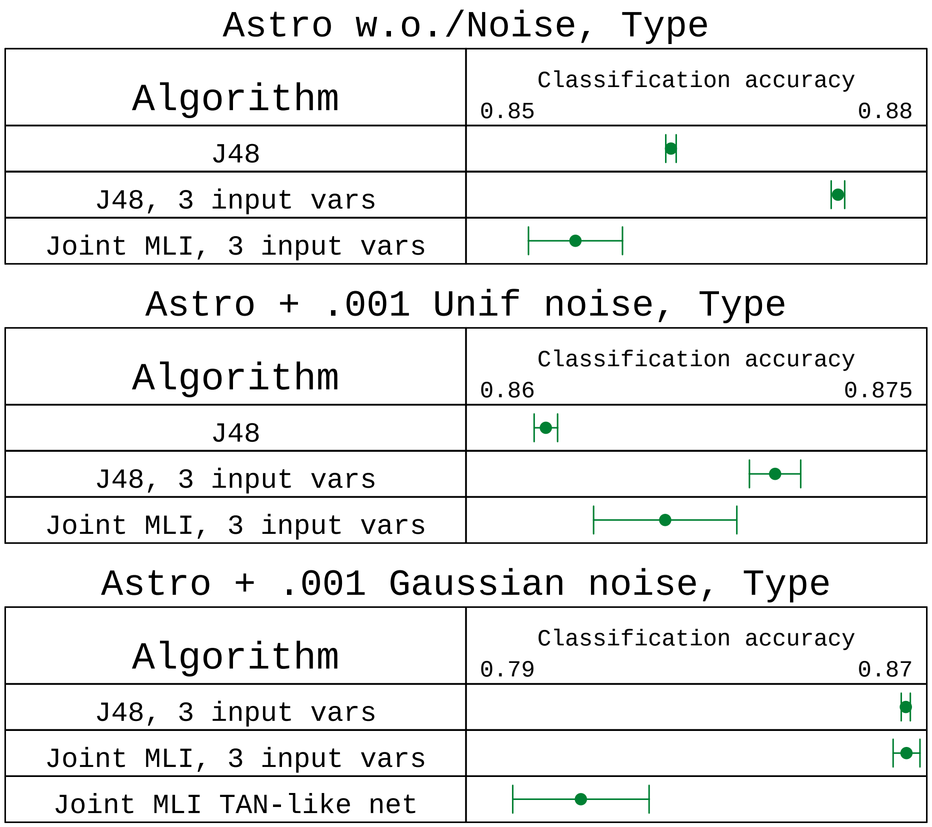


Figure A.9: Classification accuracy comparisons of J48 vs. joint multilinear density trees for the "Type" variable in the Astro dataset, given different amounts and types of noise on the input variables.

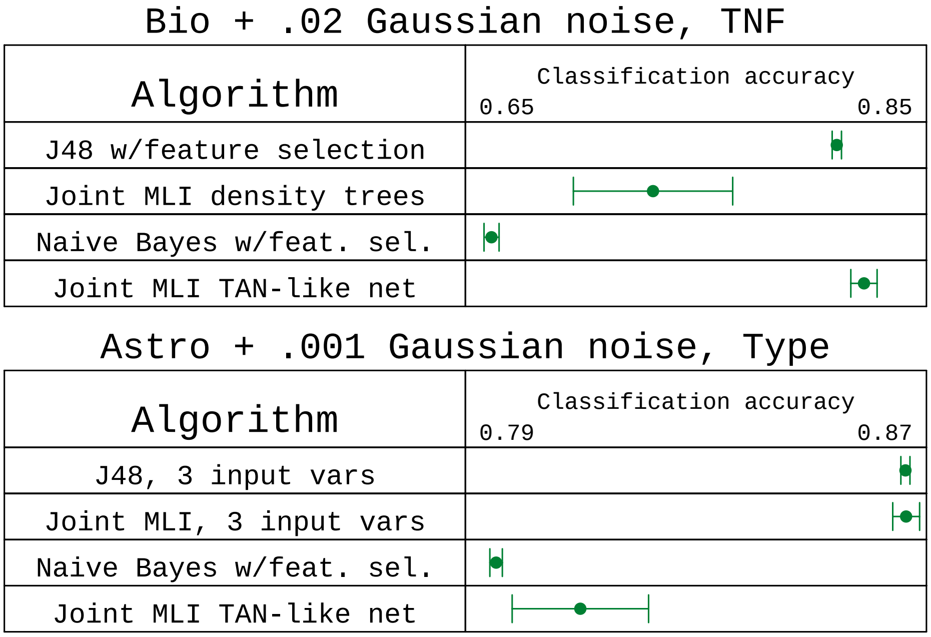


Figure A.10: Classification accuracy of TAN-like networks with joint multilinear density trees versus other classifiers.

rithm is used to initialize the network with a set of arcs from the target variable to a limited number of input variables; then, the modified greedy structure-learning algorithm is run for three iterations, with MAXPARENTS set to 3 and MAXCHANGES set to 1. In the case of the "Astro + .001 Gaussian Noise" dataset, the data was discretized during the network structure search; once the structure was learned, the network was reparameterized with joint density trees employing multilinear interpolation in the leaves. (Using such trees during the actual search would have been too computationally expensive on this dataset.)

In addition to the results for the TAN-like networks and the previous results for single density trees and decision trees, we provide results from Weka's implementation of a Naive Bayes classifier using best-first forward feature selection. This Naive Bayes implementation discretizes the continuous input variables according to a Minimum Description Length principle that takes the particular classification task into account [FI93], unlike the approaches explored in this thesis.

On the "Bio + .02 Gaussian Noise" task, the TAN-like network performed significantly better than the other classifiers; however, on the "Astro + .001 Gaussian Noise" task, it performed significantly worse than J48 and the single joint mulitlinear density tree. It is worth noting, however, that it performed significantly better than Naive Bayes in both cases. Naive Bayes performs better than decision trees do on many kinds of classification tasks (although clearly not on the tasks examined so far in this appendix); the TAN-like networks briefly explored here may be more useful on such problems. Further experimentation along such lines might be useful.