Learning with Mixtures of Trees

by

Marina Meilă-Predoviciu

M.S. Automatic Control and Computer Science (1985)
Universitatea Politehnica Bucureşti

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1999

© Massachusetts Institute of Technology 1999

Signature of Author .....
Department of Electrical Engineering and Computer Science
January 14, 1999

Certified by .....
Michael I. Jordan
Professor, Department of Brain Sciences
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Learning with Mixtures of Trees

by

Marina Meilă-Predoviciu

Revised version of a thesis submitted to the
Department of Electrical Engineering and Computer Science
on January 14, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

One of the challenges of density estimation as it is used in machine learning is that usually the data are multivariate and often the dimensionality is large. Operating with joint distributions over multidimensional domains raises specific problems that are not encountered in the univariate case. Graphical models are representations of joint densities that are specifically tailored to address these problems. They take advantage of the (conditional) independencies between subsets of variables in the domain which they represent by means of a graph. When the graph is sparse, graphical models provide an excellent support for human intuition and allow for efficient inference algorithms. However, learning the underlying dependence graph from data is generally NP-hard.

The purpose of this thesis is to propose and to study a class of models that admits tractable inference and learning algorithms yet is rich enough for practical applications. This class is the class of mixtures of trees models. Mixtures of trees inherit the excellent computational properties of tree distributions (themselves a subset of graphical models) but combine several trees in order to augment their modeling power, thereby going beyond the standard graphical model framework.

The thesis demonstrates the performance of the mixture of trees in density estimation and classification tasks. In the same time it deepens the understanding of the properties of the tree distribution as a multivariate density model. Among others, it shows that the tree classifier implements an implicit variable selection mechanism.

Learning mixtures of trees from data is a central subject of this thesis. The learning algorithm that is introduced here is based on the the EM and the Minimum Weight Spanning Tree algorithms and is quadratic in the dimension of the domain.

This algorithm can serve as a tool for discovering hidden variables in a special but important class of models where, conditioned on the hidden variable, the dependencies between the observed variables become sparse.

Finally, it is shown that in the case of sparse discrete data, the original learning algorithm can be transformed in an algorithm that is jointly subquadratic and that in simulations achieves speedups factors of up to a thousand.

Thesis Supervisor: Michael I. Jordan
Title: Professor, Department of Brain Sciences

Acknowledgments

The years at MIT have been a fantastic experience and it is a joy to remember those who concurred in making it so and to thank those whose influence contributed to this thesis.

Michael Jordan, my thesis advisor, fostered my interest in reinforcement learning, statistics, graphical models, differential geometry and ... who knows what next? He was the soul of the unforgettable "Jordan lab" and of its equally unforgettable group meetings. He supported me materially and morally during all these years while giving me complete freedom in pursuing my work. In spite of this freedom, my thesis owes enormously to his inspiring influence. I have deeply appreciated his relentless enthusiasm for learning and deepening one's understanding, the high standards he set, the genuinely positive attitude enriched by a sense for beauty that is his approach to science and life.

I thank Alan Willsky and Paul Viola, the two other members of my committee, for the interest and support they gave to this project and especially for asking just the right questions to make this thesis a better one. I very much hope that these discussions will be continue. Paul suggested a direction of work which resulted in the accelerated tree learning algorithms. The same algorithms benefited greatly from a series of very stimulating meetings with David Karger. Eric Grimson and Alvin Drake gave me good advice from my fist days on, Tomaso Poggio and his group "adopted" me during my last semester of studentship, Peter Dayan, to whom I should have talked more, found the time to listen to me and an insightful comment on every subject that I could have brought up.

Quaid Morris shared my enthusiasm for trees, coded the first mixtures of trees program, did the digits and random trees experiments. David Heckerman and the Microsoft DTAS group hosted me in Seattle for four intense months, forcing me to take a distance from this work that proved to be entirely to its benefit. David with Max Chickering provided code and ran part of the Alarm experiments.

Carl de Marcken was the most insightful critic of my ideas and my writing, answered my questions and thought me uncountably many things I needed to know, pondered with me over dozens of problems as if they were his own and was always the first to share with me the joy of a success.

I have had countless opportunities to appreciate the quality of the education I received at my Alma Mater, the Polytechnic University of Bucureşti; I take this one to thank Vasile Brînzănescu, Paul Flondor, Dan Iordache, Theodor Dănilă, Corneliu Popeea, Vlad Ionescu, Petre Stoica for everything they thought me.

I chose the thesis topic, but chance chose my officemates. I couldn't have been luckier: Greg Galperin, Tommi Jaakkola, Thomas Hofmann listened to my half baked ideas, answered patiently thousands of questions and let themselves be engaged in exciting discussions on every topic under the sun. They are now among my best friends and one of the reasons I find myself enriched by the years at MIT.

Finally, there is the circle of love that constantly and invisibly surrounds us with its strength. Gail, Butch, Paya and Natasha who welcomed me in their family. Felicia. Carl. My mother – teacher, friend and partner in adventures. My grandfather, Judge Ioan I. Predoviciu, to whose memory I dedicate this thesis.

This research was supported through the Center for Biological and Computational Learning at MIT funded in part by the National Science Foundation under contract No. ASC-92-17041.

Contents

1 Introduction **15**
- 1.1 Density estimation in multidimensional domains . . . 15
- 1.2 Introduction by example . . . 16
- 1.3 Graphical models of conditional independence . . . 17
  - 1.3.1 Examples of established belief network classes . . . 17
  - 1.3.2 Advantages of graphical models . . . 20
  - 1.3.3 Structure learning in belief networks . . . 21
  - 1.3.4 Inference and decomposable models . . . 22
- 1.4 Why, what and where? Goal, contributions and road map of the thesis . . . 23
  - 1.4.1 Contributions . . . 24
  - 1.4.2 A road map for the reader . . . 25

2 Trees and their properties **27**
- 2.1 Tree distributions . . . 27
- 2.2 Inference, sampling and marginalization in a tree distribution . . . 29
  - 2.2.1 Inference. . . . 29
  - 2.2.2 Marginalization . . . 30
  - 2.2.3 Sampling . . . 31
- 2.3 Learning trees in the Maximum Likelihood framework . . . 31
  - 2.3.1 Problem formulation . . . 31
  - 2.3.2 Fitting a tree to a distribution . . . 32
  - 2.3.3 Solving the ML learning problem . . . 35
- 2.4 Representation capabilities . . . 36
- 2.5 Appendix: The Junction Tree algorithm for trees . . . 36

3 Mixtures of trees **41**
- 3.1 Representation power of mixtures of trees . . . 43
- 3.2 Basic operations with mixtures of trees . . . 43
- 3.3 Learning mixtures of trees in the ML framework . . . 45
  - 3.3.1 The basic algorithm . . . 45
  - 3.3.2 Running time and storage requirements . . . 47
  - 3.3.3 Learning mixtures of trees with shared structure . . . 48
  - 3.3.4 Remarks on the learning algorithms . . . 49
- 3.4 Summary and related work . . . 50

4 Learning mixtures of trees in the Bayesian framework **53**
- 4.1 MAP estimation by the EM algorithm . . . 54
- 4.2 Decomposable priors for tree distributions . . . 55
  - 4.2.1 Decomposable priors over tree structures . . . 55
  - 4.2.2 Priors for tree parameters: the Dirichlet prior . . . 56

5 Accelerating the tree learning algorithm **61**
5.1 Introduction . . . 61
5.2 Assumptions . . . 62
5.3 Accelerated CL algorithms . . . 63
5.3.1 First idea: Comparing mutual informations between binary variables 63
5.3.2 Second idea: computing cooccurrences in a bipartite graph data representation . . . 64
5.3.3 Putting it all together: the **aCL-I** algorithm and its data structures 67
5.3.4 Time and storage requirements . . . 69
5.3.5 The **aCL-II** algorithm . . . 70
5.3.6 Time and memory requirements for **aCL-II** . . . 72
5.4 Generalization to discrete variables of arbitrary arity . . . 73
5.4.1 Computing cooccurrences . . . 73
5.4.2 Presorting mutual informations . . . 74
5.5 Using the aCL algorithms with EM . . . 76
5.6 Decomposable priors and the aCL algorithm . . . 76
5.7 Experiments . . . 77
5.8 Concluding remarks . . . 80
5.9 Appendix: Bounding the number of lists N_L . . . 80

6 An Approach to Hidden variable discovery **83**
6.1 Structure learning paradigms . . . 83
6.2 The problem of variable partitioning . . . 85
6.3 The tree H model . . . 88
6.4 Variable partitioning in the general case . . . 89
6.4.1 Outline of the procedure . . . 89
6.4.2 Defining structure as simple explanation . . . 89
6.5 Experiments . . . 91
6.5.1 Experimental procedure . . . 91
6.5.2 Experiments with tree H models . . . 91
6.5.3 General H models . . . 92
6.6 Approximating the description length of a model . . . 96
6.6.1 Encoding a multinomial distribution . . . 96
6.7 Model validation by independence testing . . . 97
6.7.1 An alternate independence test . . . 97
6.7.2 A threshold for mixtures . . . 99
6.7.3 Validating graphical models with hidden variables . . . 101
6.8 Discussion . . . 103

7 Experimental results **105**
7.1 Recovering the structure . . . 105
7.1.1 Random trees, large data set . . . 105
7.1.2 Random bars, small data set . . . 106
7.2 Density estimation experiments . . . 109
7.2.1 Digits and digit pairs images . . . 109
7.2.2 The ALARM network and data set . . . 110
7.3 Classification with mixtures of trees . . . 113
7.3.1 Using a mixture of trees as a classifier . . . 113

7.3.2 The AUSTRALIAN data set 113
7.3.3 The MUSHROOM data set 115
7.3.4 The SPLICE data set. Classification and structure discovery 115
7.3.5 The single tree classifier as an automatic feature selector 121

8 Conclusion **123**

List of Figures

1-1 The DNA splice junction domain. 17
1-2 An example of a Bayes net (a) and of a Markov net (b) over 5 variables. . 18
1-3 Structure of the thesis 25

3-1 A mixture of trees with $m = 3$ and $n = 5$. Note that although $b \perp_{T^k} c\,|, a$ for all $k = 1, 2, 3$ this does not imply in general $b \perp c\,|\,a$ for the mixture. . . 42
3-2 A Mixture of trees with shared structure represented as a graphical model. 42

5-1 The bipartite graph representation of a sparse data set. Each edge iv means "variable v is on in data point i". 65
5-2 The mean (full line), standard deviation and maximum (dotted line) of Kruskal algorithm steps n_K over 1000 runs plotted against $n \log n$. n ranges from 5 to 3000. The edge weights were sampled from a uniform distribution. 71
5-3 The **aCL-II** algorithm: the data structure that supplies the next weightiest candidate edge. Vertically to the left are the variables, sorted by decreasing N_u. For a given u, there are two lists: C_u, the list of variables $v \succ u$, sorted in decreasing order of I_{uv} and (the virtual list) $V_0(u)$ sorted by decreasing N_v. The maximum of the two first elements of these lists is that is inserted into an F-heap.The overal maximum of I_{uv} can then be extracted as the maximum of the F-heap. 71
5-4 Real running time for the accelerated (full line) and traditional (dotted line) **TreeLearn** algorithm versus number of vertices n for different values of the sparsity s. 78
5-5 Number of steps of the Kruskal algorithm n_K versus domain size n measured for the **aCL-II** algorithm for different values of s 79

6-1 The H-model (a) and a graphical model of the same distribution marginalized over h (b). 86
6-2 The tree H model used for training. 92
6-3 The models used for training: (a) S1, (b) S2. 92
6-4 Description lengths and accuracies for models with fixed $p = 2$ and variable $m = 2, 3, 4$ learned from data generated by the tree H model in figure 6-2. For $m = 3$ and 4 all structures are correct. (a) general, (b) detail of the upper left corner of (a). 93
6-5 Model structures and their scores and accuracies, as obtained by learning tree H models with $m = 3$ and $p = 3$ on 10 data sets generated from structure S1. Circles represent good structures and x's are the bad structures. A + marks each of the 5 lowest DL models. The x-axes measure the empirical description length in bits/example, the vertical axes measure the accuracy of the hidden variable retrieval, whose bottom line is at 0.33 94

6-6 Model structures and their scores and accuracies, as obtained by learning tree H models with $p = 3$ and variable $m = 2, 3, 4, 5$ on a data set generated from structure S1. Circles represent good structures and x's are the bad structures. The sizes of the symbols are proportional to m. The x-axis measure the empirical description length in bits/example, the vertical axis measure the accuracy of the hidden variable retrieval, whose bottom line is at 0.33. Notice the decrease in DL with increasing m. 95

7-1 Eight training examples for the bars learning task. 106

7-2 The true structure of the probabilistic generative model for the bars data. . 107

7-3 A mixture of trees approximate generative model for the bars problem. The interconnection between the variables in each "bar" are arbitrary. 107

7-4 Test set log-likelihood on the bars learning task for different values of the smoothing α and different m. Averages and standard deviations over 20 trials. 108

7-5 An example of a digit pair. 109

7-6 Average log-likelihoods (bits per digit) for the single digit (a) and double digit (b) datasets. MT is a mixture of spanning trees, MF is a mixture of factorial distributions, BR is the base rate model, HWS is a Helmholtz machine trained by the Wake-Sleep algorithm, HMF is a Helmholtz machine trained using the Mean Field approximation, FV is fully visible fully connected sigmoidal Bayes net. Notice the difference in scale between the two figures. 111

7-7 Classification performance of different mixture of trees models on the Australian Credit dataset. Results represent the percentage of correct classifications, averaged over 20 runs of the algorithm. 114

7-8 Comparison of classification performance of the mixture of trees and other models on the SPLICE data set. The models tested by DELVE are, from left to right: 1-nearest neighbor, CART, HME (hierarchical mixture of experts)-ensemble learning, HME-early stopping, HME-grown, K-nearest neighbors, Linear least squares, Linear least squares ensemble learning, ME (mixture of experts)-ensemble learning, ME-early stopping. TANB is the Tree Augmented Naive Bayes classifier of [24], NB is the Naive Bayes classifier, Tree represents a mixture of trees with $m = 1$, MT is a mixture of trees with $m = 3$. KBNN is the Knowledge based neural net, NN is a neural net. 116

7-9 Cumulative adjacency matrix of 20 trees fit to 2000 examples of the SPLICE data set with no smoothing. The size of the square at coordinates ij represents the number of trees (out of 20) that have an edge between variables i and j. No square means that this number is 0. Only the lower half of the matrix is shown. The class is variable 0. The group of squares at the bottom of the figure shows the variables that are connected directly to the class. Only these variable are relevant for classification. Not surprisingly, they are all located in the vicinity of the splice junction (which is between 30 and 31). The subdiagonal "chain" shows that the rest of the variables are connected to their immediate neighbors. Its lower-left end is edge 2–1 and its upper-right is edge 60-59. 118

7-10 The encoding of the IE and EI splice junctions as discovered by the tree learning algorithm compared to the ones given in J.D. Watson & al., "Molecular Biology of the Gene"[67]. Positions in the sequence are consistent with our variable numbering: thus the splice junction is situated between positions 30 and 31. Symbols in boldface indicate bases that are present with probability almost 1, other A,C,G,T symbols indicate bases or groups of bases that have high probability (>0.8), and a – indicates that the position can be occupied by any base with a non-negligible probability. 119
7-11 The cumulated adjacency matrix for 20 trees over the original set of variables (0-60) augmented with 60 "noisy" variables (61-120) that are independent of the original ones. The matrix shows that the tree structure over the original variables is preserved. 120

List of Tables

7.1 Results on the bars learning task. 109
7.2 Average log-likelihood (bits per digit) for the single digit (Digit) and double digit (Pairs) datasets. Boldface marks the best performance on each dataset. Results are averaged over 3 runs. 110
7.3 Density estimation results for the mixtures of trees and other models on the ALARM data set. Training set size $N_{train} = 10,000$. Average and standard deviation over 20 trials. 112
7.4 Density estimation results for the mixtures of trees and other models on a data set of size 1000 generated from the ALARM network. Average and standard deviation over 20 trials. 112
7.5 Performance comparison between the mixture of trees model and other classification methods on the AUSTRALIAN dataset. The results for mixtures of factorial distribution are those reported in [41]. All the other results are from [48]. 114
7.6 Performance of mixture of trees models on the MUSHROOM dataset. m=10 for all models. 115

Index of Algorithms

Absorb, 38
aCL-I, 68
aCL-I - outline, 68
aCL-II, 72

CollectEvidence, 39

DistributeEvidence, 39

H-learn – outline, 89
HMixTreeS, 88

ListByCooccurrence, 65

MixTree, 47
MixTree – outline, 47
MixTreeS, 49

PropagateEvidence, 40

TestIndependence, 102
TreeLearn, 35

Chapter 1

Introduction

Cetatea siderală in stricta-i descărnare
Imi dezveleşte-n Număr vertebra ei de fier.
Ion Barbu
–Pytagora
The astral city in naked rigor
Displays its iron spine - the Number.

1.1 Density estimation in multidimensional domains

Probability theory is a powerful and general formalism that has successfully been applied in a variety of scientific and technical fields. In the field of machine learning, and especially in what is known as *unsupervised learning*, the probabilistic approach has proven to be particularly fruitful.

The task of unsupervised learning is that, given a set of observations, or *data*, of producing a *model* or description of the data. It is often the case that the data is assumed to be generated by some [stationary] process and building a model represents building a description of that process. In any definition of learning is present an implicit assumption of redundancy: the assumption that the description of the data is more compact than the data themselves or that the model constructed from the present data can predict [properties of] future observations from the same source.

As opposed to *supervised learning*, where learning is performed in view of a specified task and the data presented to the learner are labeled consequently as "inputs" and "outputs" of the task, in unsupervised learning there are no "output" variables and the the envisioned usage of the model is not known at the time of learning. For example, after *clustering* a data set, one may be interested only in the number of clusters, or in the shapes of the clusters for analysis purposes, or one may want to classify future observations as belonging to one (or more) of the discovered clusters (as in document classification), or one may use the model for lossy data compression (as in vector quantization).

Density estimation is the most general form of unsupervised learning and provides a fully probabilistic approach to unsupervised learning. Expressing the domain knowledge as a probability distribution allows us to formulate the learning problem in a principled way, as a data compression problem (or, equivalently, as a maximum likelihood estimation problem). When prior knowledge exists, it is specified as a prior distribution over the class of models, and the task is one of Bayesian model selection or Bayesian model averaging.

Parametric density estimation with its probabilistic framework also enables us to separate what we consider as "essential" in the description of the data from what we consider

inessential or "random"[1].

One of the challenges of density estimation as it is used in machine learning is that usually the data are multivariate and often the dimensionality is large. Examples of domains with typically high data dimensionality are pattern recognition, image processing, text classification, diagnosis systems, computational biology and genetics. Dealing with joint distributions over multivariate domains raises specific problems that are not encountered in the univariate case. Distributions over domains with more than 3 dimensions are hard to visualize and to represent intuitively. If the variables are discrete, the size of the state-space grows exponentially with the number of dimensions. For continuous (and bounded) domains, the number of data points necessary to achieve a certain density of points per unit volume also increases exponentially in the number of dimensions. One other way of seeing this is that if the radius of the neighborhood around a data point is kept fixed while the number of dimensions, in the forthcoming denoted by n, is increasing the relative volume of that neighborhood is exponentially decreasing. This constitutes a problem for non-parametric models. While the possibilities of gathering data are usually limited by physical constraints, the increase in the number of variables leads, in the case of a parametric model class, to an increase in the number of parameters of the model and consequently to the phenomenon of *overfitting*. Moreover, the increased dimensionality of the parameter space may lead to an exponential increase in the computational demands for finding an optimal set of parameters. This ensemble of difficulties related to modeling multivariate data is known as *the curse of dimensionality*.

Graphical models are models of joint densities that, without attempting to eliminate the curse of dimensionality, limit its effects to the strictly necessary. They do so by taking advantage of the independences existing between (subsets of) variables in the domain. In the cases when the dependencies are sparse (in a way that will be formalized later on) and their pattern is known, graphical models allow for efficient inference algorithms. In these cases, as a side-result, the graphical representation is intuitive and easy to visualize and to manage by humans as well.

1.2 Introduction by example

Before discussing graphical models, let us illustrate the task of density estimation by an example. The *domain* represented in figure 1-1 is the DNA splice junction domain (briefly SPLICE) that will be encountered in the Experiments section. It consists of 61 discrete *variables*; 60 of them, called "site 1", ... "site 60" represent consecutive sites in a DNA sequence. They can each take 4 values, denoted by the symbols A, C, G, T representing the 4 bases that make up the nucleic acid. The variable called "junction" denoted the fact that, sometimes, the middle of the sequence represents a *splice junction*. A splice junction is the place where a section of non-coding DNA (called *intron*) meets a section of coding DNA (or *exon*). The "junction" variable takes 3 values: EI (exon-intron) when the first 30 variables belong to the exon and the next 30 are the beginning of the intron, IE (intron-exon) when the reverse happens and "none" whe the presented DNA section contains no junction[2]. An *observation* or *data point* is an observed instantiation for all the variables in the domain

[1]The last statement reveals the ill-definedness of "task-free" or purely unsupervised learning: what is essential for one task may be superfluous for another. Probability theory cannot overcome this difficulty but it can provide us with a better understanding of the assumptions underlying the models that we are constructing.

[2]The data are such as a junction appears in the middle position or not at all.

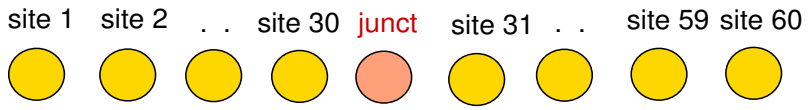


Figure 1-1: The DNA splice junction domain.

(in our case, a DNA sequence together with the corresponding value for the "junction" variable).

In density estimation, the assumption is that the observed data are generated by some underlying probabilistic process and the goal of learning is to reconstruct this process based on the observed data. This reconstruction, or or any approximation of thereof, is termed as a *model*. Therefore, in this thesis, a model will be always a joint probability distribution over the variables in the domain. This distribution incorporates our knowledge about the domain. For example, a model for the SPLICE domain, is expected to assign relatively high probability to the sequences that are biologically plausible (among them the sequences observed in the data set) and a probability close to 0 to implausible sequences. Moreover, a plausible sequence coupled with the correct value for the "junction" variable shoud have a high probability, whereas the same sequence coupled with any of the other two values for the "junction" variables should receive a zero probability.

The *state space*, i.e. the set of all possible configurations of the 61 variables, has a size of $3 \times 4^{60} \approx 10^{13}$. It is impossible to explicitly assign a probability to each configuration and therefore we have to construct more compact (from the storage point of view) and tractable (from the computation point of view) representations of probability distributions. To avoid the curse of dimensionality, we require that the models have a number of parameters that is small or slowly increasing with the dimension and that learning the models from data can also be done efficiently. In the next section we shall see that graphical probability models, have to first property but not fully satisfy the second requirement. Trees, a subclass of graphical models, enjoy both properties but sometimes need to be combined in a mixture to increase their modeling power.

In the present example, there is a natural ordering of the variables in a sequence. In other examples (Digits, Bars) the variables are arranged on a two-dimensional grid. But, in general, there is not necessary to have any spatial relationship between the variables in the domain. In the forthcoming, we will discuss arranging the variables in a graph. In this case, the graph may, but is not required to match the spatial arrangement of the variables.

1.3 Graphical models of conditional independence

1.3.1 Examples of established belief network classes

Here we introduce graphical models of conditional independence or in short *graphical models*. As [53] describes them, graphical models are "a computation-minded interpretation of probability theory, an interpretation that exposes the qualitative nature of this centuries-old formalism, its compatibility with human intuition and, most importantly, its amenability to network representation and to parallel and distributed computation.". Graphical models are also known as *belief networks* and in the forthcoming the two terms shall be used interchangeably.

We define *probabilistic conditional independence* as follows: If A, B, C are disjoint sets

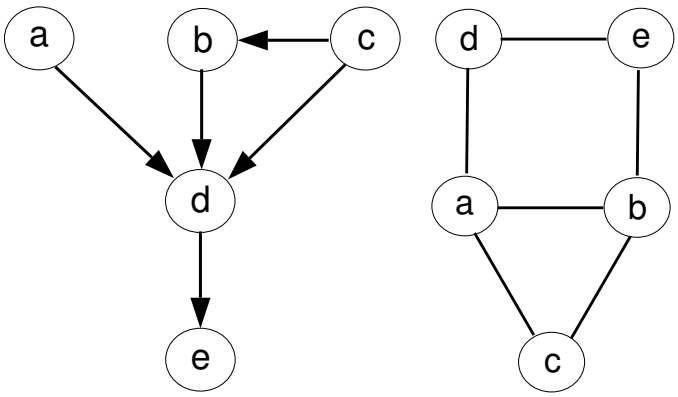


Figure 1-2: An example of a Bayes net (a) and of a Markov net (b) over 5 variables.

of variables, we say that A and B are *independent given* C, if the following relationship holds: $P_{AB|C} = P_{A|C}P_{B|C}$ for all configurations of A, B and C. In the graphical models language we say that C *separates* the sets A, B and we write

$$A \perp B \mid C \tag{1.1}$$

Equivalently, two variable sets are (conditionally) independent when knowing one does not affect our knowledge about the other. Graphical models, by taking independences into account, avoid processing irrelevant information where possible. This means that using a graphical model is at one's advantage in the measure that the variables in the domain can be grouped into (conditionally) independent subsets. For a domain where there are no independences a graphical model reduces to a n-way probability table or some other generic function of n variables.

A belief network encodes independences by means of a graph in the following way: each variable v is associated with a vertex of the graph. The graph's topology is used to represent dependencies. More rigorously, the absence of an edge indicates an independence relationship. This graph is called the *structure* of the belief network. The most common classes of belief networks are:

- Markov nets
- Bayes nets
- decomposable models
- chain graphs

A **Markov net** (better known as a Markov random field) is defined by a structure that is an undirected graph with arbitrary topology. Two variables that are connected by an edge are *neighbors*. The independences that a Markov net expresses are summarized by the *global Markov property*: a variable is independent of all the others given its neighbors. Figure 1-2(b) shows a Markov net and illustrates some of the independence relationships that it encodes. We define a *clique* of an undirected graph to be a maximal subset of variables that are all neighbors of each other. The probability distribution is a product of functions (called *clique potentials*) defined each on a clique of the graph. In the example of

figure 1-2(b) the cliques are $\{a,b,c\},\{a,d\},\{d,e\},\{b,e\}$; $c \perp d,e \,|\, a,b$; $d \perp b \,|\, a,e$; $e \perp a \,|\, b,d$ are among the independences represented by this net.

Thus, the Markov net is specified in two stages: first, a graph describes the structure of the model. The structure implicitly defines the cliques. Second, the probability distribution is expressed as a product of functions of the variables in each clique and some parameters. We say that the distribution factors according to the graph. For example, a distribution that factors according to the graph in figure 1-2 is represented by

$$P \;=\; \phi_{abc}\phi_{ad}\phi_{de}\phi_{be} \tag{1.2}$$

The number of variables in each clique is essential for the efficiency of the computations carried on by the model. The fewer variables in any clique, the more efficient the model. The totality of the parameters corresponding to the factor functions forms the parameter set associated with the given graph structure.

Bayes nets are belief nets whose structure is a directed graph that has no directed cycles; such a structure is called a *directed acyclic graph*, or shortly a DAG. We denote by $\vec{uv}$ an edge directed from u to v and in this case we call u the *parent* of v. A *descendent* of u is a variable w that can be reached by a directed path starting at u; hence, the children of u, their children, and so on, are all descendents of u. Figure 1-2(a) depicts an example of a DAG. The independences encoded by a Bayes net are summarized by the *directed Markov property*, that states that in a Bayes net, any variable is independent of its non-descendents given its parents. The probability distribution itself is given in a factorized form that corresponds to the independence relationships expressed by the graph. Each factor is a function that depends on one variable and all its parents (we call this set of variables a *family*). For example, in figure 1-2(a) the families are $\{a\},\{c\},\{b,c\},\{b,c,d\},\{d,e\}$. The list of independences encoded by this net includes $a \perp b$, $\{a,b,c\} \perp e|d$. Any distribution with this structure can be represented as

$$P \;=\; P_a P_c P_{b|c} P_{d|bc} P_{e|d} \tag{1.3}$$

In short, for both classes of belief nets, the model is specified by first specifying its structure. The structure determines the subsets of "closely connected" variables (cliques in one case, families in the other) on which we further define the factor functions (clique potentials or conditional probabilities) that represent the model's parametrization.

Any probability distribution P that can be factored according to a graph G (DAG or undirected) and thus possesses all the independences encoded in it is called *conformal* to G. The graph G is called an *I-map* of P. If G is an I-map of P then only the independences represented in G will be useful from the computational point of view even though a belief net with structure G will be able to represent P exactly. If the distribution P has no other dependencies then those represented by G, G is said to be a *perfect map* for P. For any DAG or undirected graph there is a probability distribution P for which G is a perfect graph. The converse is not true: there are distributions whose set of independences does not have a perfect map.

Decomposable models. Bayes nets and Markov nets represent distinct but intersecting classes of distributions. A probability distribution that can be mapped perfectly as both a Bayes net and a Markov net is called a *decomposable model*. In a decomposable model, the cliques of the undirected graph representation play a special role. They can be arranged in a tree (i.e acyclic graph) structure called *junction tree*. The vertices of the junction tree are the cliques. The intersection of two cliques that are neighbors in the tree

is always non-void and is called a *separator*. Any probability distribution that is factorized according to a decomposable model can be refactorized according to the junction tree into clique potentials ϕ_C and separator potentials ϕ_S in a way that is called *consistent* and that ensures the following important property:

The junction tree consistency property If a junction tree is *consistent* [38, 36] then for each clique $C \subset V$ the marginal probability of C is equal to the clique potential ϕ_C.

The factorization itself has the form

$$P = \frac{\prod_C \phi_C}{\prod_S \phi_S} \tag{1.4}$$

From now on, any reference to a junction tree should implicitly assume that the tree is consistent. This property endows decomposable models with a certain computational simplicity that is exploited by several belief network algorithms. We will discuss inference in junction trees in section 1.3.4.

Chain graphs [] are a more general category of graphical models. Their underlying graph comprises both directed and undirected edges. Bayes nets and Markov nets are both subclasses of chain graphs.

1.3.2 Advantages of graphical models

The advantages of a model that is factorized according to the independences between variables are of several categories:

- **Flexibility and modeling power.** The flexible dependence topology, complemented by the freedom in the choice of the factor functions, makes belief network a rich and powerful class among probabilistic models. In particular, belief networks encompass and provide a unifying view of several other model classes (including some that are used for supervised learning): hidden Markov models, Boltzmann machines, stochastic neural networks, Helmholtz machines, decision trees, mixture models, naive-Bayes models.

  More important perhaps than the flexibility of the topology is the flexibility in the usages that graphical models admit. Because a belief network represents a probability distribution, any query that can be expressed as a function of probabilities over subsets of variables is acceptable. This means that in particular, a belief network can be used as a classifier for any variable of its domain, can be converted to take any two subsets of variables as inputs and outputs respectively and to compute probabilities of the outputs given the inputs or can be used for diagnostic purposes by computing the most likely configuration of a set of variables given another set.

  This flexibility is theoretically a property of any probability density model, but not all density representations are endowed with the powerful inference machine that allows one to compute arbitrary conditional probabilities and thus to take advantage of it.

- **The models are easier to understand and to interpret.** A wide body of practical experience shows that graphical representations of dependencies are very appealing to the (non-technical) users of statistical models of data. Bayes nets, which allow for the interpretation of a directed edge $\vec{uv}$ as a causal effect of u on v, are particularly appreciated as a means of *knowledge elicitation* from human domain experts. Moreover, in a Bayes net the parameters represent conditional probabilities; it is found that

specifying or operating with conditional probabilities is much easier for a human than operating with other representations (as for example, specifying a joint probability) [53].

The outputs of the model, being probabilities, have a clear meaning. If we give these probabilities the interpretation of *degrees of belief*, as advocated by [53], then a belief network is a tool for *reasoning under uncertainty*.

- **Advantages in learning the parameters from data for a given structure.** More independences mean fewer free parameters compared to a *full* probability model (i.e. a model with no independences) over the same set of variables. Each parameter appears in a function of only a subset of the variables, thus depends on fewer variables. Under certain parametrizations that are possible for any Bayes net or Markov net structure, this allows for independent estimation of parameters in different factors. In general, for finite amounts of data, a smaller number of (independent) parameters implies an increased accuracy in the estimation of each parameter and thus a lower model variance.

- **Hidden variables and missing data.** A *hidden variable* is a variable whose value is never observed[3]. Other variables may on some occasions be observed, but not in others. When the latter happens, we say that the current observation of the domain has a missing value for that variable. If in an observation (or *data point*) no variable is missing, we say that the observation is *complete*. The graphical models framework allows variables to be specified as observed or unobserved for any data point, integrating thus supervised learning and naturally handling both missing data and hidden variables.

1.3.3 Structure learning in belief networks

Learning the structure of a graphical model from data, however, is not an easy task. [33] formulates the problem of structure learning in Bayes networks as a Bayesian model selection problem. They show that under reasonable assumptions the prior over the parameters of a Bayes network over a discrete variable domain has the form of a Dirichlet distribution. Moreover, given a set of observations, the posterior probability of a network structure can be computed in closed form. Similar results can be derived for continuous variable models with jointly Gaussian distributions [32]. However, finding the structure with the highest posterior probability is an intractable task. For general DAG structures, there are no known algorithms for finding the optimal structure that are asymptotically more efficient than exhaustive search.

Therefore, in the majority of the applications, the structure of the model is either assessed by a domain expert, or is learned by examining structures that are close to one elicited from prior knowledge. If neither of these is the case, then usually some simple structure is chosen.

[3]One can ask: why include in the model a variable that is never observed? One reason is that our physical model of the domain postulates the variable, although in the given conditions we cannot observe it directly (e.g. the state of a patients liver is only assessed indirectly, by certain blood-tests). Another reason is of computational nature: we may introduce a hidden variable because the resulting model explains the observed data well with fewer parameters than the models that do not include the hidden variable. This is often the case for hidden causes: for a second example from the medical domain, a disease is a hidden variable that allows one to describe in a simple way the interplay between a multitude of observation facts called symptoms. In chapter 6 we discuss this issue at length.

1.3.4 Inference and decomposable models

The task of using a belief network in conjunction with actual data is termed *inference*. In particular, inference means answering a generic query to the model that has the form: Q="what is the probability of variable v having value x_v given that the values of the variables in the subset $V' \subset V$ are known?". The variables in V' and their observed values are referred to as *categorical evidence* and denoted by $\mathcal{E}$. In a more general setting, one can define evidence to be a probability distribution over V' (called a *likelihood*) but this case will not be considered here. Thus, inference in the restricted sense can be formally defined as computing the probability $P(v = x_v|\mathcal{E})$ in the current model. This query is important for two reasons: One, the answers to a wide range of common queries can be formulated as a function of one or more queries of type Q or can be obtained by modified versions of the inference algorithm that solves the query Q. Two, Q serves as a benchmark query for the efficiency of the inference algorithms for a given class of belief networks.

Inference in Bayes nets. In [53] Pearl introduced an algorithm that performs exact inference in *singly connected* Bayes networks, called by him *polytrees*. A singly connected Bayes net is a network whose underlying undirected graph has no cycles. In such a net there will always be at most one (undirected) path between any two variables u and v. Pearl's algorithm, as it came to be named, assumes that each node can receive/send messages only from/to its *neighbors* (i.e. its parents and children) and that it performs computations based only on the information locally present at each node. Thus it is a *local* algorithm. Pearl proves that it is also asynchronous, exact and that it terminates finite time. The minimum running time is bounded above by the diameter of the (undirected) graph, which in turn is less or equal to the number of variables n.

The singly-connectedness of the graph is essential for both the finite termination and the correctness of the algorithm's output[4]. For general multiply connected Bayes nets, inference is provably NP-hard [8].

The standard way of performing inference in a Bayes net of general topology is to transform it into a decomposable model; this is always possible by a series of edge additions combined with removing the edges' directionality. Adding edges to a graphical model does not change the probability distribution that it represents but will "hide" (and thus make computationally unusable) some of its independences. Once the decomposable counterpart of the Bayes net is constructed, inference is performed via the standard inference algorithm for decomposable models, the *Junction Tree Algorithm* that will be described below.

Inference in junction trees. As defined before, a decomposable model (or junction tree) is a belief network whose cliques form a tree. Tree distributions, which will be introduced in the next section, are examples of decomposable models.

Inference in a graphical model has 3 stages. Here these stages are described for the case of the junction tree and they represent what is known as the *Junction Tree algorithm* [38, 36]:

> **Entering evidence.** This step combines a joint distribution over the variables (which can be thought of as a prior) with evidence (acquired from a different source of information) to produce a posterior distribution of the variables given the evidence.

[4]It is worth mentioning, however, that recently some impressively successful applications of Pearl's algorithm to Bayes networks with loops have been published. Namely the Turbo codes [3], Gallager [26] and Neal-MacKay [44] codes that are all based on belief propagation in multiply connected networks. Why Pearl's algorithm performs well in these cases is a topic of intense current research [68].

The expression of the posterior is in the same factorized form as the original distribution, but at this stage it does not satisfy all the consistency conditions implicit in the graphical model's (e.g. junction tree's) definition.

Propagating evidence. This is a stage of processing whose final result is the posterior expressed as a consistent and normalized junction tree. Often this phase is referred to as the Junction Tree algorithm proper. The Junction tree algorithm requires only local computations (involving only the variables within one clique). Similarly to Pearl's algorithm for polytrees, information is propagated along the edges of the tree, by means of the separators. The algorithm is exact and finite time; it requires a number of basic clique operations proportional to the number of cliques in the tree. If all the variables of the domain V take values in finite sets, the time required for the basic inference operations in a clique is proportional to the size of *total state space* of the clique (hence it is exponential in cardinality of the clique). The total time required for inference in a junction tree is $\mathcal{O}($ the sum of these state space sizes $)$.

Extracting the probabilities of the variables of interest by marginalization in the joint posterior distribution obtained at the end of the previous stage. In a consistent junction tree the marginal of any variable v can be computed by marginalization in any ϕ_C for which $v \in C$. If the clique sizes are small relative to n, this represents an important saving w.r.t. the time for computing the marginal. This is also the reason for the locality of the operations necessary in the previous step of the inference procedure.

Returning to Bayes nets, it follows that the time required for inference by the junction tree method is exponential in the size of the largest clique of the resulting junction tree. It is hence desirable to minimize this quantity (or alternatively the sum of the state space sizes) in the process of constructing the decomposable model. However, the former objective is provably NP-hard (being equivalent to solving a max-clique problem) [2]. It is expected, but not yet known, that the alternative objective is also an NP-hard problem. Moreover, it has been shown by [37] that any exact inference algorithm based on local computations is as least as hard as the junction tree algorithm and thus also NP-hard.

Inference in Markov nets. [30] showed that inference in Markov random fields with arbitrary topology is also intractable. They introduced a Markov chain Monte Carlo sampling technique for computing approximate values of marginal and conditional probabilities in such a network that is known as *simulated annealing*.

Similar approaches of approximate inference by Monte Carlo techniques have been devised and used for Bayes nets of small size also [31, 59].

Approximate inference in Bayes nets is a topic of current research. The existent approaches include: pruning the model and performing exact inference on the reduced model [40], cutting loops and bounding the incurred error [19], variational methods to bound the node probabilities in sigmoidal belief networks [35, 39].

1.4 Why, what and where? Goal, contributions and road map of the thesis

The previous sections have presented the challenge of density estimation for multidimensional domains and have introduced models as tools specifically designed for this purpose.

It has been shown that graphical models focus on expressing the dependencies between the variables of the domain, without constraining neither the topology of these dependencies, nor their functional form. The property of separating the dependency structure from its detailed functional form makes them an excellent support for human intuition, whitout compromising the modeling power of this model class.

The probabilistic semantics of a graphical model makes it possible to separate model learning from its usage: once a belief network is constructed from data, one can use it in any way that is consistent with the laws of probability.

We have seen also that inference in graphical models can be performed efficiently when the models are simple, but that in the general case it is an NP-hard problem. The same holds for learning a belief network structure from data: learning is hard in general, but we shall see that there are classes of structures for which both learning the structure and the parameters can be done efficiently.

It is the purpose of this thesis to propose and to describe a class of models that is rich enough to be useful in practical applications yet admits tractable inference and learning algorithms.

1.4.1 Contributions

The mixture of trees models represent this class. A tree can be defined as a Bayes net in which each node has at most one parent. But trees can represent only acyclic pairwise dependencies and thus have a limited modeling power. By combining tree distributions in a mixture, one can represent any distribution over discrete variables. The number of trees (or mixture components) is a means of controlling the model's complexity. Mixtures of trees can represent a different class of dependencies than graphical models. From the algorithmic perspective, this thesis shows that the properties of tree distributions, namely the ability to perform the basic operations of computing likelihoods, marginalization and sampling in linear time, directly extend to mixtures.

An efficient learning algorithm. This thesis introduces an efficient algorithm for estimating mixtures of trees from data. The algorithm builds upon a fundamental property of trees, the fact that unlike almost any other class of graphical models, a tree's structure and parameters can be learned efficiently. Embedding the tree learning algorithm in an Expectation-Maximization search procedure produces an algorithm that is quadratic in the domain dimension n and linear in the number of trees m and in the size of the data set N. The algorithm find Maximum Likelihood estimates but can serve for Bayesian estimation as well. To preserve the algorithm's efficiency in the latter case, one needs to use a restricted class of priors. The thesis characterizes this class as being the class of *decomposable* priors and shows that the restrictions that this class imposes are not stronger than the assumptions underlying the tree mixture of trees learning algorithm itself. It is also shown that many widely used priors belong to this class.

An accelerated learning algorithm for sparse binary data If one has in mind application over high-dimensional domains like document categorization and retrieval, preference data or image compression, a quadratic algorithm may not be sufficiently fast. Therefore, I introduce an algorithm that exploits a property of the data that is frequent in these domains - sparsity - to construct a family of algorithms that are jointly subquadratic. In controlled experiments on artificial data the new algorithm achieve speedup factors of up to 3000; the

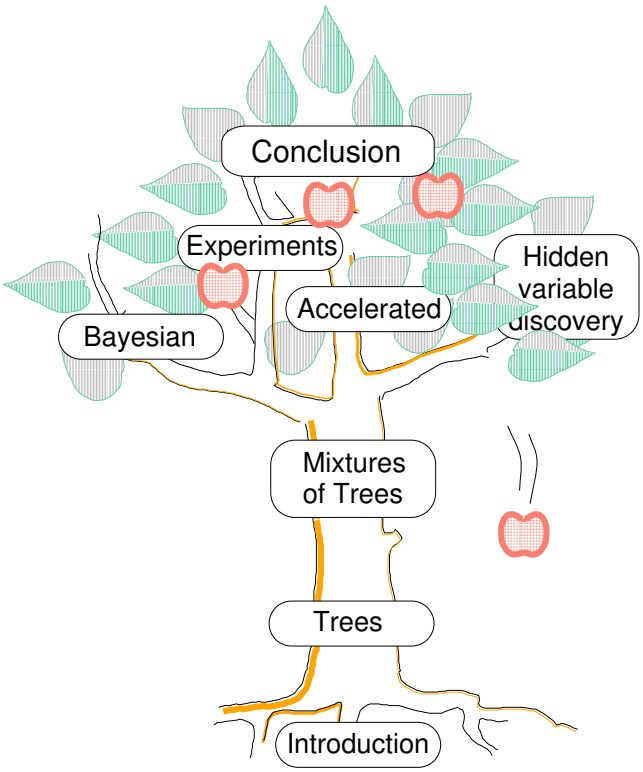


Figure 1-3: Structure of the thesis

performance is practically independent of n as long as the sparsity and the size of the data set remain constant.

A top-down approach to hidden variable discovery Learning the structure of a graphical model from data is a hard but important problem. The vast majority of algorithms in this field pursue a bottom up strategy in which the basic unit is the graph edge. There is no fundamental reason to make the graph edge play a special role; in fact, it is not the edges but the families or cliques that are the building parts of a graphical model. Therefore, in this thesis I pose the question: can structure search be performed in a top-down manner? I show that there is a hidden variable model that I call the H model for which this approach is natural and produces a partition of the variable set into clusters on which structure search can be performed independently. I show then that the mixture of trees learning algorithm is a tool for performing the partitioning operation. The result is a heuristic algorithm for discovering hidden variables in H models. Motivated by the need to validate the models obtained by the hidden variable discovery algorithm, I investigated the use of large deviation theory for testing probabilistic independence between discrete variables and obtained an alternate **distribution free independence test** based on Sanov's theorem. The optimality of the test is under study.

1.4.2 A road map for the reader

The structure of the thesis can be seen in the diagram 1-3. The introduction and chapter 2 lay the foundation by defining the fundamental concepts and by reviewing tree distributions

and their inference and learning algorithms.

Chapter 3 builds on this to define the mixture of trees with its variants, to introduce the algorithm for learning mixtures of trees from data in the Maximum Likelihood framework and to show that mixtures of trees can approximate arbitrarily closely any distribution over a discrete domain.

The following chapters will each develop this material into a different direction and thus can be read independently in any order. Chapter 4 discusses learning mixtures of trees in a Bayesian framework. The basic learning algorithm is extended to take into account a class of priors called decomposable priors. It is shown that this class is rich enough to contain important priors like the Dirichlet prior and Minimum Description Length type priors. In this process, the assumptions behind decomposable priors and the learning algorithm itself are made explicit and discussed.

Chapter 5 aims to improve the computational properties of the tree learning algorithm. It introduces a data representation and two new algorithms that exploit the properties of those domains to construct exact Maximum Likelihood trees in subquadratic time. The compatibility of the new algorithms with the EM framework and with the use of decomposable priors are also discussed.

The next chapter, 6, introduces the top-down method for learning structures with a hidden variable, methods for scoring the obtained models and a discussion of the independence test approach to validating them.

Finally there comes the chapter devoted to experimental assessments. Chapter 7 demonstrates the performance of mixtures of trees on various tasks. There are very good results in density estimation, even for data that are not generated from a mixture of trees. The last part of this chapter discusses classification with mixtures of trees. Although the model is a density estimator, its performance as a classifier in my experiments is excellent, even in competition with classifiers trained in supervised mode. I analize the behavior of the single tree classifier and demonstrate that it acts like an implicit feature selector.

The last chapter, 8 contains the concluding remarks.

Chapter 2

Trees and their properties

I think that I shall never see
A poem as lovely as a tree.
Joyce Kilmer
–Trees

In this section we introduce tree distributions as a subclass of decomposable models and we demonstrate some of the properties that make them attractive from a computational point of view. It will be shown that fundamental operations on distributions: inference, sampling and marginalizing carry over directly from their junction tree counterparts and are order n or less when applied to trees. Learning tree distributions will be formulated as Maximum Likelihood (ML) estimation problem. To solve it, an algorithm will be presented that finds the tree distribution T that best approximates a given target distribution P in the sense of the Kullback-Leibler divergence. The algorithm optimizes over both structure and parameters, in time and memory proportional to the size of the data set and quadratic in the number of variables n. Some final considerations on modeling power and ease of visualization will prepare the introduction of mixtures of trees in the next section.

2.1 Tree distributions

In this section we will introduce the tree model and the notation that will be used throughout the paper. Let V denote the set of variables of interest. As stated before, the cardinality of V is $|V| = n$. For each variable $v \in V$ let r_v denote its number of values, $\Omega(v)$ represent its domain and $x_v \in \Omega(v)$ a particular value of v. Similarly, for subset A of V, $\Omega(A) = \bigotimes_{v \in A} \Omega(v)$ is the domain of A and x_A an assignment to the variables in A. In particular, $\Omega(V)$ is the state-space of all the variables in V; to simplify notation x_V will be denoted by x. Sometimes we shall need the maximum of r_v over V; we shall denote this value by r_{MAX}.

According to the graphical model paradigm, each variable is viewed as a vertex of an (undirected) graph (V, E). An edge connecting variables u and v is denoted by (uv) and its significance will become clear in the following. The graph (V, E) is called a *tree* if it has no cycles[1]. Note that under this definition, a tree can have a number p between 1 and $|V|$ *connected components*. The number of edges $|E|$ and p are in the relationship

$$|E| + p \;=\; |V| \tag{2.1}$$

[1]This definition differs slightly from the definition of a tree in the graph theory literature. There, a tree is required to have no cycles *and* to be connected (meaning that between every two vertices there should exist a path); our definition of a tree allows for disconnected graphs to be trees and corresponds to what in graph theory is called a *forest*.

which means that adding an edge to a tree reduces the number of connected components by 1. Thus, a tree can have at most $|V| - 1 = n - 1$ edges.

Now we define a probability distribution T that is *conformal* with a tree. Let us denote by T_{uv} and T_v the marginals of T for u, $v \in V$ and $(uv) \in E$:

$$T_{uv}(x_u, x_v) = \sum_{x|u=x_u, v=x_v} T(x) \tag{2.2}$$

$$T_v(x_v) = \sum_{x|v=x_v} T(x). \tag{2.3}$$

They must satisfy the consistency condition

$$T_v(x_v) = \sum_{x_u} T(x_v, x_u) \quad \forall u, v, \ (uv) \in E \tag{2.4}$$

Let $\deg v$ be the *degree* of vertex v, i.e. the number of edges incident to $v \in V$. Then, the distribution T is conformal with the tree (V, E) if it can be factored as:

$$T(x) = \frac{\prod_{(u,v)\in E} T_{uv}(x_u, x_v)}{\prod_{v\in V} T_v(x_v)^{\deg v - 1}} \tag{2.5}$$

The distribution T itself will be called a tree when no confusion is possible. The graph (V, E) represents the *structure* of the distribution T. Noting that for all trees over the same domain V the edge set E alone uniquely defines the tree structure, in the following when no confusion is possible we identify E with the structure. If the tree is connected, i.e. it *spans* all the nodes in V, it is sometimes called a *spanning tree*.

Because a tree is a triangulated graph it is easy to see that a tree distribution is a decomposable model. In fact, the above representation (2.5) is identical to the junction tree representation of T. The cliques identify with the graph's edges (hence all cliques are size two) and the separators are all the nodes of degree larger than one. Thus the junction tree is identical to the tree itself with the clique and separator potentials being the marginals T_{uv} and T_v, $\deg v > 1$ respectively.

This shows a remarkable property of tree distributions: a distribution T that is conformal with the tree (V, E) is completely determined by its edge marginals $\{T_{uv}, \ (uv) \in E\}$.

Since every tree T is a decomposable model it can be represented in terms of conditional probabilities

$$T(x) = \prod_{v\in V} T_{v|\mathrm{pa}(v)}(x_v|x_{\mathrm{pa}(v)}) \tag{2.6}$$

We shall call the representations (2.5) and (2.6) the *undirected* and *directed* tree representations respectively. The form (2.6) is obtained from (2.5) by choosing an arbitrary root in each connected component and directing each edge away from the root. In other words, if for (uv) in E, u is closer to the root than v, then u becomes the parent of v and is denoted by $\mathrm{pa}(v)$. Note that in the *directed tree* thus obtained, each vertex has at most one parent. Consequently, the families of a tree distribution are either of size one (the root or roots) or of size two (the families of the other vertices). After having transformed the structure into a directed tree one computes the conditional probabilities corresponding to each directed edge by recursively substituting $\frac{T_{v\mathrm{pa}(v)}}{T_{\mathrm{pa}(v)}}$ by $T_{v|\mathrm{pa}(v)}$ starting from the root. $\mathrm{pa}(v)$ represents the parent of v in the thus directed tree or the empty set if v is the root

of a connected component. The directed tree representation has the advantage of having independent parameters. The total number of free parameters in either representation is

$$\sum_{(u,v)\in E} r_u r_v - \sum_{v\in V} (\deg v - 1) r_v - p \; = \\ = \sum_{(u,v)\in E} (r_u - 1)(r_v - 1) + \sum_{v\in V} r_v - n \tag{2.7}$$

The r.h.s. of (2.7) shows that each newly added edge (uv) increases the number of parameters by $(r_u - 1)(r_v - 1)$.

Now we shall characterize the set of independences represented by a tree. In a tree there is at most one path between every two vertices. If node w is on the path between u and v we say that w *separates* u and v. Correspondingly, if we set w as a root, the probability distribution T can be decomposed into the product of two factors, each one containing only one of the variables u, v. Hence, u and v are independent given w:

$$u \perp v \mid w$$

Therefore we conclude that in a tree two variables are separated by any set that intersects the path between them. Two subsets $A, B \subset V$ are independent given $C \subset V$ if C intersects every path between $u \in A$ and $v \in B$.

We can also verify that for a tree the undirected Markov property holds. The set of variables connected by edges to a variable v, namely its neighbors separates v from all the other variables in V.

2.2 Inference, sampling and marginalization in a tree distribution

This section discusses the basic operations: inference, marginalization and sampling for tree distributions. It demonstrates that the algorithms for performing these operations are direct adaptations of their counterparts for junction trees and are heavily relying on the generic Junction Tree algorithm. It is also shown that all basic operations are linear in the number of variables n. This fact is important because the analog operations on mixtures of trees use the algorithms for trees as building blocks. The details of the algorithms however, although presented here for the sake of completeness, can be skipped without prejudice for the understanding of the rest of the thesis.

2.2.1 Inference.

As shown already, tree distributions are decomposable models and the generic inference algorithm for trees is an instance of the inference algorithm for decomposable models called the Junction Tree algorithm. We shall define it as a procedure, **PropagateEvidence**$(T, \mathcal{E})$ that takes as inputs a tree having structure (V, E) presented in the undirected factored representation and some categorical evidence $\mathcal{E}$ on a set $A \subset V$. The procedure performs the first two steps of the inference process as defined in section 1.3.4. It enters the categorical evidence by multiplying the original distribution T with a $\{0, 1\}$-values function representing the categorical evidence. Then it calibrates the resulting tree by local propagation of information between the cliques (edges) of the tree. Finally, it outputs the tree T^* that is

factored conformally to the original T and represents the posterior distribution $T_{V|\mathcal{E}}$. The algorithm is described in detail in the last section of this chapter. It is also demonstrated that it takes a running time of the order $\mathcal{O}(r_{MAX}^2|E|)$.

The last step, extracting the probabilities of variables of interest from the above mentioned representation is done by marginalization and will be discussed in the next section. In particular, obtaining the posterior probability of any single variable involves marginalization over at most one other variable which takes $\mathcal{O}(r_{MAX})$ operations.

2.2.2 Marginalization

In a junction tree, computing the marginal distribution of any group of variables that are contained in the same clique can be performed by marginalization within the respective clique. For a tree, whose cliques are size two, the marginal of each variable v can be obtained either directly as T_v (if v is a separator and separator potentials are stored explicitly) or by marginalizing in the potential T_{uv} of the edge incident to v (if v is a leaf node). Thus, the marginal for any single variable can be obtained in $\mathcal{O}(r_{MAX})$ additions.

The pairwise marginals for all the variables that are neighbors in T are directly available as the clique potentials T_{uv}. The above enumeration exhausts all the cases where the marginals are directly available from the tree distribution. In the following I describe an algorithm that can efficiently compute the marginal distributions for arbitrary pairs of variables. The algorithm can be generalized to marginal distributions over arbitrary subsets of V.

First, it will be shown that the marginal T_{uv} depends only on the potentials of the edges on the path between u and v in E. Let $\text{path}(u,v) = (w^0 = u, w^1, w^2, \ldots, w^d = v)$ be the vertices on the path between u and v in the tree (V,E) and let $d \geq 1$ be its length. Then, the marginal T_{uv} can be expressed as

$$
\begin{aligned}
T_{uv} &= \sum_{x_{V\setminus\{u,v\}}} T \\
&= \sum_{x_{V\setminus\{u,v\}}} \left(T_u \prod_{i=1}^{d} T_{w^i|w^{i-1}} \prod_{w' \in V\setminus \text{path}(u,v)} T_{w'|\text{pa}(w')} \right) && (2.8) \\
&= \sum_{x_{V\setminus\{u,v\}}} T_u \prod_{i=1}^{d} T_{w^i|w^{i-1}} && (2.9) \\
&= \sum_{x_{w^i}, i=1,d-1} \frac{\prod_{i=1}^{d} T_{w^{i-1}w^i}}{\prod_{i=1}^{d-1} T_{w^i}} && (2.10)
\end{aligned}
$$

The first form (2.8) is obtained as the directed tree representation with a root at u. Summation over each $x_{w'}$ not on the path from u to v can be done recursively starting from the leaves of the tree and results in a factor of 1. Thus the form (2.9) is obtained. The third form, (2.10) is the rewriting of the previous equation in the undirected representation.

Each of the intermediate variables w^i appears in only two factors of (2.9); consequently, summation over the values of the intermediate variables can be done one variable at a time,

as indicated by the following telescoped sum:

$$T_{vu} = \sum_{x_{w^{d-1}}} T_{v|w^{d-1}} \dots \underbrace{T_{w^3|w^2} \sum_{x_{w^2}} \underbrace{(T_u \sum_{x_{w^1}} T_{w^1|u} T_{w^2|w^1})}_{T_{w^2 u}}}_{T_{w^3 u}} \tag{2.11}$$

The computation of one value of T_{uv} takes $\sum_{i=1}^{d-1} r_{w^i}$ additions and multiplications; to compute the whole marginal probability table we need to perform

$$r_u r_v \sum_{i=1}^{d-1} r_{w^i} = \mathcal{O}((d-1) r_{MAX}^3)$$

operations.

As the above equation shows, the intermediate sums involved in the computation of T_{uv} are themselves marginal distributions. This suggests that if the intermediate sums are stored and if the pairs $(uv) \notin E$ are enumerated in a judiciously chosen order, one can compute *all* the pairwise marginal tables by summing over 1 variable only for each of them; in other words, all the pairwise marginals can be computed with $\mathcal{O}((n-1)(n-2) r_{MAX}^3)$ operations.

Marginalization in the presence of evidence represents the third and last step of inference, as presented in the previous subsection. This problem can be approached in various ways, one of them being the polytree algorithm of [53]. The approach we present here follows directly from the procedures for entering evidence, calibration and marginalization introduced above. To find $T^*_{uv} = T_{uv|\mathcal{E}_A}$ for $u, v \in V$, $A \subset V$ one has to

1. enter and propagate evidence by $T^* \leftarrow$ **PropagateEvidence**$(T, \mathcal{E}_A)$
2. compute the marginal T^*_{uv}

2.2.3 Sampling

Sampling in a tree is best performed using the directed tree representation. The value of the root node(s) is sampled from its (their) marginal distribution. Then, the value of each of the other nodes is sampled from the conditional distribution given its parent $P(v|\mathrm{pa}(v))$, recursively, starting from the root(s). This simple algorithm is the specialization of the algorithm presented in [14] for sampling from a junction tree. In [12] an algorithm is presented for sampling without replacement in a junction tree that can be immediately specialized for trees.

Sampling in a tree in the presence of evidence is done, just like marginalization in the presence of evidence, in two steps. First, one incorporates the evidence by the **PropagateEvidence** algorithm; second, sampling by the procedure described above is performed on the resulting conditional distribution.

2.3 Learning trees in the Maximum Likelihood framework

2.3.1 Problem formulation

First, I shall formulate the learning problem as a Maximum Likelihood (ML) estimation task.

Assume a domain V and a set of observations from V called a *dataset* $\mathcal{D} = \{x^1, x^2, \ldots x^N\}$. We further assume that these data were generated by sampling independently from an unknown tree distribution T^0 over V. The learning problem consists in finding the generative model T^0. According to the Maximum Likelihood principle the estimate of T^0 from $\mathcal{D}$ is the model that maximizes the probability (or likelihood) of the observed data. Equivalently, one can search to optimize the logarithm of the likelihood, called *log-likelihood*, which leads us to formulate the *ML Learning Problem* for trees as follows:

Given a domain V and a set of complete observations $\mathcal{D}$, find a tree distribution T^* for which

$$T^* \;=\; \operatorname*{argmax}_T \sum_{x \in \mathcal{D}} \log T(x). \tag{2.12}$$

2.3.2 Fitting a tree to a distribution

The solution to the ML Learning Problem has been published in [7] in the broader context of finding the tree that best fits a given distribution P over the dataset $\mathcal{D}$. The goodness of fit is evaluated by the Kullback-Leibler (KL) divergence [43] between P and T :

$$KL(P \,||\, T) \;=\; \sum_{x \in \mathcal{D}} P(x) \log \frac{P(x)}{T(x)} \tag{2.13}$$

Since the Chow and Liu algorithm will constitute a building block for the algorithms that will be developed in my thesis, I shall present it and its derivation here. The impatient reader can skip to the next subsection.

Let us start by examining (2.13). It is known [11] that for any two distributions P and Q, $KL(P||Q) \geq 0$ and that equality is attained only for $Q \equiv P$. The KL divergence can be rewritten as

$$\begin{aligned} KL(P \,||\, Q) \;&=\; \sum_x P(x)[\log P(x) - \log Q(x)] \\ &=\; \sum_x P(x) \log P(x) - \sum_x P(x) \log Q(x) \end{aligned} \tag{2.14}$$

Notice that the first term above does not depend on Q. Hence, minimizing the KL divergence w.r.t. Q is equivalent to maximizing the second term of (2.14) (called the *cross-entropy* between P and Q) and we know that this is achieved for $Q = P$.

Now, let us return to our problem of fitting a tree to a fixed distribution P. Finding a tree distribution requires finding its structure (represented by the edge set E) and the corresponding parameters, i.e. the values of $T_{uv}(x_u, x_v)$ for all edges $(uv) \in E$ and for all values x_u, x_v.

Assume first that the structure E is fixed and expand the right-hand side of (2.13):

$$\begin{aligned} &KL(P \,||\, T) \;= \\ &\quad= \sum_{x \in \mathcal{D}} P(x)[\log P(x) - \log T(x)] \\ &\quad= -\sum_{x \in \mathcal{D}} P(x) \log \frac{1}{P(x)} - \sum_{x \in \mathcal{D}} P(x) \log[\prod_{v \in V} T_{v|\mathrm{pa}(v)}(x_v | x_{\mathrm{pa}(v)})] \\ &\quad= -H(P) - \sum_{v \in V} \sum_{x \in \mathcal{D}} P(x) \log T_{v|\mathrm{pa}(v)}(x_v | x_{\mathrm{pa}(v)}) \end{aligned} \tag{2.15}$$

$$
\begin{aligned}
&= -H(P) - \sum_{v \in V} \sum_{x_v, x_{\mathrm{pa}(v)}} P_{v,\mathrm{pa}(v)}(x_v, x_{\mathrm{pa}(v)}) \log T_{v|\mathrm{pa}(v)}(x_v | x_{\mathrm{pa}(v)}) \\
&= -H(P) - \sum_{v \in V} \sum_{x_{\mathrm{pa}(v)}} P_{\mathrm{pa}(v)}(x_{\mathrm{pa}(v)}) \sum_{x_v} P_{v|\mathrm{pa}(v)}(x_v | x_{\mathrm{pa}(v)}) \log T_{v|\mathrm{pa}(v)}(x_v | x_{\mathrm{pa}(v)})
\end{aligned}
$$

In the above, $H(P)$ denotes the entropy of the distribution P, a quantity that does not depend on T, and P_{uv}, P_v represent respectively the marginals of $\{u, v\}$, v under P. The inner sums in the last two lines are taken over the domains of v and $\mathrm{pa}(v)$ respectively. When v is a root node, $\mathrm{pa}(v)$ is the void set and its corresponding range has, by convention, one value with a probability of $P_{\mathrm{pa}(v)}(x_{\mathrm{pa}(v)}) = 1$. Moreover, note that the terms that depend on T are of the form

$$
-\sum_{x_v} P_{v|\mathrm{pa}(v)}(x_v | x_{\mathrm{pa}(v)}) \log T_{v|\mathrm{pa}(v)}(x_v | x_{\mathrm{pa}(v)})
$$

which differs only by a constant independent of T from the KL divergence

$$
KL(P_{v|\mathrm{pa}(v)} \,||\, T_{v|\mathrm{pa}(v)})
$$

We know that the latter is minimized by

$$
T_{v|\mathrm{pa}(v)}(. \,|\, x_{\mathrm{pa}(v)}) \;\equiv\; P_{v|\mathrm{pa}(v)}(. \,|\, x_{\mathrm{pa}(v)}) \;\; \forall v \in V. \tag{2.16}
$$

Hence, for a fixed structure E, the best tree parameters in the sense of the minimum KL divergence are obtained by copying the corresponding values from the conditional distributions $P_{v|\mathrm{pa}(v)}$. Let us make two remarks: first, the identity (2.16) can be achieved for all v and $x_{\mathrm{pa}(v)}$ because the distributions $T_{v|\mathrm{pa}(v)=x_{\mathrm{pa}(v)}}$ are each parameterized by its own set of parameters. Second, from the identity (2.16) it follows that

$$
T_{uv} \;\equiv\; P_{uv} \;\; \forall (u, v) \in E \tag{2.17}
$$

and subsequently, that the resulting distribution T is the same independently of the choice of the roots. For each structure E we denote by T^E the tree with edge set E and whose parameters satisfy equation (2.17). T^E achieves the optimum of (2.13) over all tree distributions conformal with (V, E).

Now, with the previous results in mind, we shall proceed to the minimization of $KL(P \,||\, T)$ over the tree structures. First, notice that this task is equivalent to maximizing the objective

$$
J(E) \;=\; \sum_{x \in \mathcal{D}} P(x) \log T^E(x). \tag{2.18}
$$

over all structures E.

Expanding the above formula and using (2.17) we obtain successively:

$$
\begin{aligned}
J(E) \;&= && (2.19) \\
&= \sum_{i=1}^{N} P(x^i) \log T^E(x^i) \\
&= \sum_{i=1}^{N} P(x^i) [\sum_{(u,v) \in E} \log T^E_{uv}(x^i_u x^i_v) - \sum_{v \in V} (\deg v - 1) \log T^E_v(x^i_v)] && (2.20)
\end{aligned}
$$

$$= \sum_{i=1}^{N} P(x^i)[\sum_{(u,v)\in E} \log P_{uv}(x_u^i x_v^i) - \sum_{v\in V} (\deg v - 1) \log P_v(x_v^i)] \tag{2.21}$$

$$= \sum_{(u,v)\in E} \sum_{i=1}^{N} P(x^i)[\log P_{uv}(x_u^i x_v^i) - \log P_u(x_u^i) - \log P_v(x_v^i)]$$
$$+ \sum_{v\in V} \sum_{i=1}^{N} P(x^i) \log P_v(x_v^i) \tag{2.22}$$

$$= \sum_{(u,v)\in E} \sum_{i=1}^{N} P(x^i) \log \frac{P_{uv}(x_u^i x_v^i)}{P_u(x_u^i) P_v(x_v^i)} + \sum_{v\in V} \sum_{i=1}^{N} P(x^i) \log P_v(x_v^i) \tag{2.23}$$

$$= \sum_{(u,v)\in E} \sum_{x_u x_v} P_{uv}(x_u, x_v) \log \frac{P_{uv}(x_u, x_v)}{P_u(x_u) P_v(x_v)} + \sum_{v\in V} \sum_{x_v} P_v(x_v) \log P_v(x_v) \tag{2.24}$$

$$= \sum_{(u,v)\in E} I_{uv} - \sum_{v\in V} H(P_v) \tag{2.25}$$

Equation (2.20) follows from the undirected tree representation (2.5) of T^E, (2.21) is obtained from (2.20) by taking into account (2.17); equation (2.23) follows from (2.22) by performing a summation over all $x \in \mathcal{D}$ that have the same x_u, x_v and using the definitions of P_{uv} and P_v; finally in equation (2.24) the terms I_{uv} under the first sum sign represent the *mutual information* between the variables u and v under the distribution P:

$$I_{uv} = \sum_{x_u x_v} P_{uv}(x_u, x_v) \log \frac{P_{uv}(x_u, x_v)}{P_u(x_u) P_v(x_v)} \tag{2.26}$$

The mutual information between two variables is a quantity that is always non-negative and equals 0 only when the variables are independent.

Remark two important facts about equation (2.25): first, the second sum does not depend on the structure E; second and more importantly, the dependence of $J(E)$ from E is additive w.r.t. to the elements of the set E. In other words, each edge in $(u, v) \in E$ contributes a certain positive amount to $J(E)$ and this amount I_{uv} is always the same, independently of the presence or absence of other edges and of the size of their contributions!

In this situation, maximization of J over all structures can be performed efficiently via a *Maximum Weight Spanning Tree* (MWST) algorithm [10] with weights $W_{uv} = I_{uv},\ u, v \in V$.

The MWST problem is formulated as follows: given a graph $(V, \overline{E})$ and a set of real numbers, called *weights* each corresponding to an edge of the graph, find a tree $(V, E),\ E \in \overline{E}$ for which the sum of the weights corresponding to its edges is maximized. This problem can be solved by a greedy algorithm that constructs the tree by adding one edge at a time, in decreasing order of the weights $\{W_{uv}\}$. There are several variants of the algorithm: the simplest one, called Kruskal's algorithm, runs in $\mathcal{O}(n^2 \log n)$ time. Note that if all the weights are strictly positive, a tree with the maximum number of edges and $p = 1$ connected components will result. If some of the weights W_{uv} are zero, it is possible to obtain trees with more than one connected component. More sophisticated MWST algorithms exist (see for example [10, 63, 25, 20, 58]) and they improve on Kruskal's algorithm on both running time and memory requirements. However, the running time of all published algorithms is at least proportional to the number of candidate edges ($|\overline{E}|$). In our case, this number is equal to $n(n-1)/2$ since all pairs of variables have to be considered. Hence, the best running

time achievable for a MWST algorithm will be $\mathcal{O}(n^2)$. Henceforth, we will assume that the MWST algorithm runs in $\mathcal{O}(n^2)$ time and will not further specify the implementation.

All of the above are summarized in the **TreeLearn** algorithm

Algorithm **TreeLearn**

Input	Probability distribution P over domain V
	Procedure MWST(weights) that fits a maximum weight spanning tree over V.
1.	Compute marginals $P_v,\ P_{uv}$ for $u,\ v \in V$.
2.	Compute mutual informations I_{uv} for $u,\ v \in V$.
3.	Call MWST($\{I_{uv}\}$) that outputs the edge set E of a tree distribution T.
4.	Set $T_{uv} \equiv P_{uv}$ for $(uv) \in E$.
Output	T

The algorithm takes as input a probability distribution P over a domain V and outputs a tree distribution T that minimizes the KL divergence $KL(P\,||\,T)$.

The running times for the algorithm's steps are as follows:

steps 1,2. For steps 1. and 2. (computing the marginals and the mutual informations for all pairs of variables) the running time is dependent on the representation of P. But, generally, it should be expected to be $\mathcal{O}(n^2)$, since there are $n(n-1)/2$ mutual information values to be computed.

step 3. The MWST algorithm takes $\mathcal{O}(n^2)$ operations (or $\mathcal{O}(n^2 \log n)$ in Kruskal's variant).

step 4. This step comprises only $\mathcal{O}(nr_{MAX}^2)$ assignments (remember that $|E| < n$).

Hence, the total running time of **TreeLearn** is $\mathcal{O}(n^2 + nr_{MAX}^2)$ or $\mathcal{O}(n^2)$ if we consider r_{MAX} to be a constant.

2.3.3 Solving the ML learning problem

The previous subsection has presented the algorithm **TreeLearn**(P) that finds the tree distribution T^* closest in KL divergence to a given distribution P,

$$T^* \;=\; \operatorname*{argmin}_T\, KL(P\,||\,T).$$

To solve the initial Maximum Likelihood estimation problem it is sufficient to call **Treelearn**$(\overline{P})$ where $\overline{P}$ is the uniform distribution over the data

$$\overline{P}(x) = \begin{cases} \frac{1}{N}, & x \in \mathcal{D} \\ 0 & \text{otherwise} \end{cases} \tag{2.27}$$

To see this, note that

$$KL(\overline{P}\,||\,T) \;=\; -\frac{1}{N}\sum_{i=1}^{N} \log T(x^i) - \log N. \tag{2.28}$$

Hence, minimizing the above expression is equivalent to maximizing the r.h.s. of equation (2.12).

For this case, the first step of the **TreeLearn** algorithm, computing the marginals, takes $\mathcal{O}(Nn^2r_{MAX}^2)$ time. This time dominates the times required for each of the following steps (unless $Nr_{MAX}^2 < \log n$, which is usually not the case) and thus, the running time of the ML tree estimation algorithm is $\mathcal{O}(Nn^2r_{MAX}^2)$.

2.4 Representation capabilities

If graphical representations are easy to grasp by means of human intuition, then the subclass of tree graphical models will be even more intuitive. For once, they are sparse graphs, having $n-1$ or fewer edges. More importantly and more precisely, between each two variables there is at most one path, or, in other words, the separation relationships between subsets of variables, which are not easy to read out in a general Bayes net topology, are obvious in a tree. Thus, in a tree, an edge corresponds to the simplest common sense notion of direct dependency and is the natural representation for it. However, the very simplicity that makes tree models intuitively appealing also limits their modeling power. In other words, the class of dependency structures representable by trees is a relatively small one. For instance, over a domain of dimension n there are n^{n-2} distinct (undirected) spanning trees [69], but a total of $2^{n(n-1)/2}$ undirected graphs[2]. Mixture of trees models, which will be presented next, are a way of circumventing this problem. This thesis will show that mixtures of trees can arbitrarily increase the representation power of tree distributions while sacrificing only little of their computational advantages and, to a certain extent, even preserving their appeal to human intuition.

2.5 Appendix: The Junction Tree algorithm for trees

This section will present the mechanisms for entering evidence in a tree density model and for maintaining consistent representations in the presence of evidence.

Since a tree distribution is also a junction tree, inference will be done via the Junction Tree algorithm using the undirected tree representation (2.5) described previously.

Entering evidence In the general sense, *evidence* on a subset A of variables is defined to be a (possibly unnormalized) probability distribution

$$\mathcal{E}_A \,:\, \Omega(A) \to [0,\infty). \tag{2.29}$$

A *finding* is a special type of evidence: $A = \{v\}$ for some $v \in V$ and $f_v \equiv \mathcal{E}_{\{v\}}$ takes values in the set $\{0,1\}$ only. A finding on v represents a statement that v cannot take certain values. Any finding f_v can be expressed as a sum of δ functions[3] with as many non-zero terms as the number of non-zero values of f_v.

$$f_v(x_v) \;=\; \sum_{\bar{x}_v \in \Omega v} f_v(\bar{x}_v)\delta_{\bar{x}_v, x_v} \tag{2.30}$$

If $\mathcal{E}_A$ consists of a collection of findings, one for each variable $v \in A$, such that:

$$\mathcal{E}_A \;=\; \prod_{v \in A} f_v \tag{2.31}$$

the evidence is said to be *categorical*. Here and in the rest of the thesis, only categorical evidence will be considered. For handling non-categorical evidence that is contained in one clique, the reader is referred to [38]. [53] discusses another special case of non-categorical evidence that is compatible with the polytree algorithm.

[2]To see that $\alpha = n^{n-2} < \beta = 2^{n(n-1)/2}$ note that α grows like n^n while β grows like $(2^{n/2})^n$ and that $n < 2^{n/2}$ for $n > 2$.

[3]Dirac's symbol $\delta_{\bar{x},x}$ is 1 if $x = \bar{x}$ and 0 otherwise

For any prior distribution P_V of the variables in V and for any categorical evidence $\mathcal{E}_A = \delta_{\bar{x}_A},\ A \subset V$,

$$P^*_V(x_{V\setminus A}, x_A) \ = \ P_V(x_{V\setminus A}, x_A)\mathcal{E}_A(x_A) \tag{2.32}$$

represents the probability of $x_{V\setminus A}, x_A$ after observing the evidence $\mathcal{E}_A$. This probability is of course 0 for $v \neq \bar{x}_v$. Moreover, the conditional probability of the remaining variables given the evidence is proportional to the above product

$$P^*_{V\setminus A|\mathcal{E}_A}(x_{V\setminus A}) \ \propto \ P_V(x).\mathcal{E}_A(x_A) \tag{2.33}$$

Using the convention $P^*(x_A|\mathcal{E}_A) \ = \ \mathcal{E}_A(x_A)$ we express the posterior conditional distribution of all the variables given the evidence $\mathcal{E}_A$ as

$$P^*_{V|\mathcal{E}_A}(x) \ \propto \ P_V(x).\mathcal{E}_A(x_A). \tag{2.34}$$

The above equations and discussion easily generalizes to the case where $\mathcal{E}_A$ is a product of sums of δ functions, i.e. the case of categorical evidence. Hence, for any categorical evidence $\mathcal{E}_A$ we define the operation of *entering evidence* by equation (2.34).

The normalization constant in equation (2.34) is

$$\sum_x P^*_V(x) \tag{2.35}$$

which represents the prior probability of the categorical evidence $P(\mathcal{E}_A)$ [38, 36]. Hence, the conditional probability of V given the (categorical) evidence $\mathcal{E}_A$ is given by:

$$P^*_{V|\mathcal{E}_A}(x) \ = \ \frac{P_V(x)\mathcal{E}_A(x_A)}{P(\mathcal{E}_A)} \tag{2.36}$$

For a tree distribution T, equation (2.34) rewrites as

$$T^*_{V|\mathcal{E}_A}(x) \ \propto \ T_V(x)\mathcal{E}_A(x_A) \tag{2.37}$$

and the prior of $\mathcal{E}_A$, given by (2.35) with P replaced by T, can be efficiently computed as it will be shown later.

Note also that because the posterior probability obtained by (2.34) is unnormalized, the evidence can be an unnormalized function as well.

Propagating evidence. When (categorical) evidence is entered in a tree T (or in a decomposable model), the resulting distribution $T^*_{V\setminus A|\mathcal{E}_A}$ is factored according to the graph of T, but its factors (i.e. the clique potentials) are neither consistent (as a junction tree) nor normalized. The following step, called tree *calibration* has the purpose to make T^* a normalized and consistent distribution again. The calibration represents the Junction Tree algorithm proper and is applicable in general to any decomposable model, but what will be presented here is its specialization for tree distributions. For the general case and the proofs of consistency, see [36] or [38].

It is important to keep in mind that calibration does not add any information to the model, but merely reorganizes the information already present in order to represent it in a form that is convenient for subsequent use (e.g. marginalizations).

The basic operation in evidence propagation is called *absorption*. We say that edge (vw) absorbs from neighboring edge (uv) when the following procedure is called:

Algorithm **Absorb**

Absorb(u, v, w)

Input	edge potentials T^*_{uv}, T_{vw}, $(uv), (vw) \in E$
1.	$T^*_v = \sum_{x_u} T^*_{uv}$
2.	$T_v = \sum_{x_u} T_{uv}$
3.	$T^*_{vw} \leftarrow T_{vw} \frac{T^*_v}{T_v}$
Output	T^*_{vw}

Note that absorption is an asymmetric operation: **Absorb**(u, v, w) and **Absorb**(w, v, u) produce different results. After an absorption, the potentials T^*_{vw} and T^*_{uv} are consistent. If T^*_{uv} is normalized, so will be T^*_{vw}. If the potentials are already consistent, then an absorb operation in either direction changes nothing. These facts can be easily proved:

$$\begin{aligned}
\sum_{x_w} T^*_{vw}(x_v, x_w) &= \sum_{x_w} T_{vw}(x_v, x_w) \frac{T^*_v(x_v)}{T_v(x_v)} \\
&= \frac{T^*_v(x_v)}{T_v(x_v)} \sum_{x_w} T_{vw}(x_v, x_w) \\
&= \frac{T^*_v(x_v)}{T_v(x_v)} T_v(x_v) \\
&= T^*_v(x_v) \\
&= \sum_{x_u} T^*_{uv}(x_u, x_v)
\end{aligned}$$

The above derivation proves consistency after absorption. Moreover, if the potentials are consistent before absorption, then the fraction $\frac{T^*_v(x_v)}{T_v(x_v)}$ equals 1 for all x_v and absorption in either direction leaves the potentials unchanged. Assume now that T^*_{uv} is normalized. Then

$$\begin{aligned}
\sum_{x_v, x_w} T^*_{vw}(x_v, x_w) &= \sum_{x_v} \sum_{x_w} T^*_{vw}(x_v, x_w) \\
&= \sum_{x_v} T^*_v(x_v) \\
&= \sum_{x_v} \sum_{x_u} T^*_{uv}(x_u, x_v) \\
&= 1
\end{aligned}$$

In the following it will be necessary to use absorption from multiple edges. This is defined as:

Algorithm **Absorb**

Absorb$(u_1, \ldots, u_m, v, w)$

Input	edge potentials $T^*_{u_i v}$, T_{vw}, $(u_i v), (vw) \in E$, $i = 1, \ldots m$
1.	for $i = 1, \ldots, m$ $T^i_v = \sum_{x_{u_i}} T^*_{u_i v}$
2.	$T_v = \sum_{x_u} T_{uv}$
3.	$T_{vw} \leftarrow T_{vw} \prod_{i=1}^m \frac{T^i_v}{T_v}$
Output	T_{vw}

We say that edge (vw) absorbs from adjacent edges $(u_1 v), \ldots (u_m v)$. Unlike the previous case, after absorption from multiple edges, the potentials involved in the operation are not

necessarily consistent. To make them consistent, it would be sufficient for each of the edges $(u_i v)$ to absorb from (vw). As we shall see shortly, this is the idea behind the Junction Tree propagation algorithm: the whole tree is made consistent by a series of absorptions from the periphery towards a "root edge" (or root clique) followed by a second series of absorptions in the opposite direction. The mechanism is implemented by two recursive procedures, **CollectEvidence** and **DistributeEvidence** which are described below.

When **CollectEvidence** from w is called for edge $(v, w) \in E$, then the edge calls **CollectEvidence** from v for all the other edges adjacent to v and then absorbs from them.

Algorithm **CollectEvidence**

CollectEvidence(T, v, w)
Input factored representation of tree T
nodes $v, w,\ (vw) \in E$
let $\mathrm{n}(v) = \{w, u_1, \ldots u_m\}$, $m \geq 0$
1. for $i = 1, \ldots, m$
CollectEvidence(T, u_i, v)
2. **Absorb**$(u_1, \ldots u_m, v, w)$
Output T

CollectEvidence(v, w) fulfills the task of recursively absorbing from the subtree rooted in node v of edge (vw). **DistributeEvidence**(w, v) has the reverse effect: when it is called from w for edge $(vw) \in E$, all the other edges adjacent to v absorb from (vw) then call **DistributeEvidence** from v.

Algorithm **DistributeEvidence**

DistributeEvidence(T, w, v)
Input factored representation of tree T
nodes $v, w,\ (vw) \in E$
let $\mathrm{n}(v) = \{w, u_1, \ldots u_m\}$, $m \geq 0$
1. for $i = 1, \ldots, m$
1.1. **Absorb**(w, v, u_i)
1.2. **DistributeEvidence**(T, v, u_i)
Output T

With the procedures above, we can introduce the Junction Tree algorithm, called here **PropagateEvidence**. The algorithm takes as input a tree distribution T over the space $\Omega(V)$ represented as a set of consistent, normalized marginal distributions T_{uv}, and some categorical evidence $\mathcal{E}_A$, $A \subset V$. It outputs the distribution $T^*_{V|\mathcal{E}_A}$ represented as a consistent, normalized tree distribution with the same structure as T.

Algorithm **PropagateEvidence**

PropagateEvidence$(T, \mathcal{E}_A)$

Input a tree distribution T in factored representation conformal to (V, E)
categorical evidence $\mathcal{E}_A$ on a subset $A \subset V$

1. enter evidence: $T^* \leftarrow T\mathcal{E}_A$
2. choose an edge $(vw) \in E$ as *root edge*
3. **CollectEvidence**(T^*, w, v)
   CollectEvidence(T^*, v, w)
4. normalize: $Z = \sum_{x_v x_w} T^*_{vw}$ $\qquad T^*_{vw} \leftarrow \frac{T^*_{vw}}{Z}$
5. **DistributeEvidence**(T^*, w, v)
   DistributeEvidence(T^*, v, w)

Output T^*

The proofs of correctness for this algorithm are given in [38, 36]. The normalization constant Z computed in step 4 of the algorithm represents the probability of the evidence $T(\mathcal{E}_A)$. To see this, notice that the potential T^*_{vw} of the root edge does not change after the normalization. Moreover, all the other tree edges absorb, directly or indirectly, from (vw). Therefore, at the end of the algorithm, all the edge potentials will be normalized. Since the tree is consistent, by simple function identification we conclude that indeed each $T^*_{v\prime w\prime}, (v\prime w\prime) \in E$ represents the

The propagation of evidence in a tree distribution takes order $\mathcal{O}(r^2_{MAX}|E|)$ time.

Chapter 3

Mixtures of trees

The previous section has shown that in the framework of graphical probability models, tree distributions enjoy many properties that make them attractive as modeling tools: they have a flexible topology, are intuitively appealing, sampling and computing likelihoods are linear time, simple efficient algorithms for marginalizing and conditioning ($\mathcal{O}(n^2)$ or less) exist. Fitting the best tree to a given distribution can also be done exactly and efficiently. Trees can capture simple pairwise interactions between variables but they can prove insufficient for more complex distributions. Therefore, this chapter introduces a more powerful model, the *mixture of trees*. As this thesis will show, mixtures of trees enjoy most of the computational advantages of trees and, in addition, they are universal approximators over the space of all distributions.

We define a mixture of trees to be a distribution of the form

$$Q(x) \;=\; \sum_{k=1}^{m} \lambda_k T^k(x) \tag{3.1}$$

with

$$\lambda_k \geq 0,\; k = 1, \ldots, m; \qquad \sum_{k=1}^{m} \lambda_k \;=\; 1. \tag{3.2}$$

The tree distributions T^k are the *mixture components* and λ_k are called *mixture coefficients*. From the graphical models perspective, a mixture of trees can be viewed as a containing an unobserved *choice variable* z, which takes value $k \in \{1, \ldots m\}$ with probability λ_k. Conditioned on the value of z the distribution of the visible variables V is a tree. The m trees may have different structures and different parameters.

Note that because of the variable structure of the component trees, a mixture of trees is neither a Bayesian network nor a Markov random field. Let us adopt the notation

$$A \;\perp_P\; B \mid C \tag{3.3}$$

for "A independent B given C under distribution P". If for some (all) $k \in \{1, \ldots m\}$ we have

$$A \;\perp_{T^k}\; B \mid C \;\text{ with } A, B, C \subset V$$

this will not imply that

$$A \;\perp_Q\; B \mid C.$$

On the other hand, a mixture of trees is capable of representing dependency structures that are conditioned on the value of one variable (the choice variable), something that a usual Bayesian network or Markov net cannot do. Situations where such a model is potentially useful abound in real life: Consider for example bitmaps of handwritten digits.

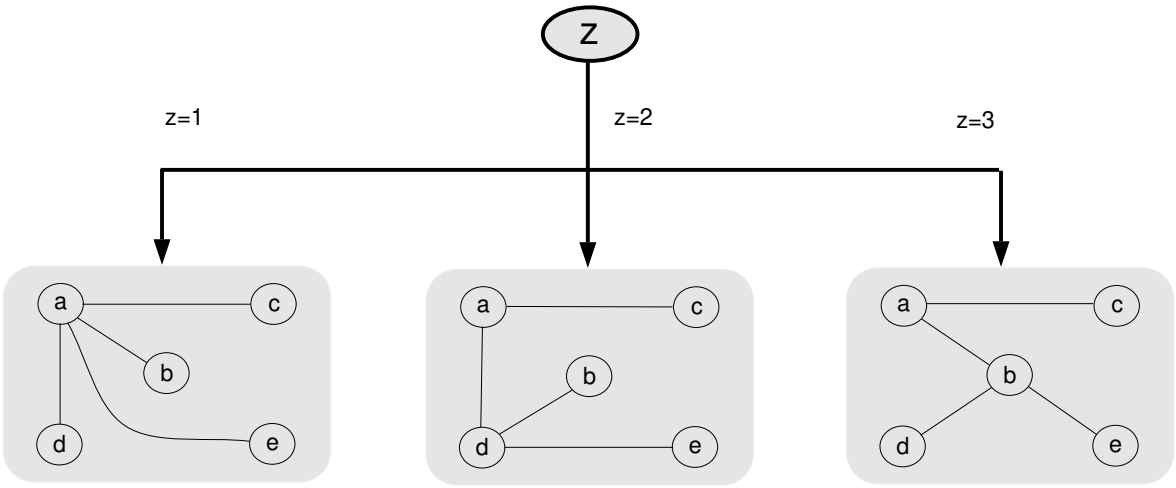


Figure 3-1: A mixture of trees with $m = 3$ and $n = 5$. Note that although $b \perp_{T^k} c \,|, a$ for all $k = 1, 2, 3$ this does not imply in general $b \perp c \,|\, a$ for the mixture.

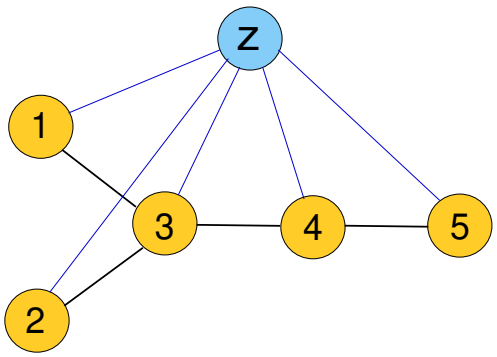


Figure 3-2: A Mixture of trees with shared structure represented as a graphical model.

They obviously contain many dependencies between pixels; however, the pattern of these dependencies will vary across digits. Imagine a medical database recording the body weight and other data for each patient. The body weight could be a function of age and height for a healthy person, but it would depend on other conditions if the patient suffered from a disease or were an athlete. If, in a situation like the ones mentioned above, conditioning on one variable produces a dependency structure characterized by sparse, acyclic pairwise dependencies, then a mixture of trees will be the model most able to uncover and exploit this kind of dependencies.

If we impose that all the trees in the mixture have the same structure we obtain a *mixture of trees with shared structure* (MTSS). Note that even in this case the resulting mixture does not preserve the independence relationships of the component trees. But a MTSS *can* be represented as a Bayes net (after choosing an orientation for the trees) or, more directly, as a *chain graph*. Chain graphs were introduced by []; they represent a superclass of both Bayes nets and Markov random fields. A chain graph contains both directed and undirected edges. The representation of the MTSS as a graphical model is given in figure 3-2.

A simple modification of both the mixture of trees and the mixture of trees with shared structure is to have the choice variable be observed as well. Such models will be called *mixtures with visible choice variable* in the forthcoming. Unless otherwise stated, it will be assumed that the choice variable is hidden. Therefore, sometimes it will be referred to as *hidden variable*.

3.1 Representation power of mixtures of trees

Here it will be shown that for discrete variable domains the class of mixtures of trees can represent any distribution.

Theorem Let $V = \{v_1, v_2, \ldots v_n\}$ a set of variables with finite ranges and P a probability distribution over $\Omega(V)$. Then P can be represented as a mixture of trees.

Proof. Let us denote by δ_{x^*} the distribution over $\Omega(V)$ defined by

$$\delta_{x^*}(x) = \begin{cases} 1, & x = x^* \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

δ_{x^*} can be identified with a tree T^{x^*} having n connected components (i.e. a factored distribution) and

$$T^{x^*}_{v_j} = \begin{cases} 1, & v_j = x^*_j \\ 0, & \text{otherwise} \end{cases} \quad \text{for } j = 1, \ldots n. \tag{3.5}$$

Moreover, any distribution P can be represented as a mixture of δ distributions:

$$P(x) \;=\; \sum_{x^*} P(x^*)\delta_{x^*}(x). \tag{3.6}$$

where the sum is taken over the range of V and $P(x^*)$ represent the mixing coefficients. This completes the proof. ■

3.2 Basic operations with mixtures of trees

Marginalization The marginal distribution of variables subset $A \subset V$ is

$$Q_A(x_A) \;=\; \sum_{x_{V \setminus A}} Q(x)$$

$$
\begin{aligned}
&= \sum_{x_{V\setminus A}} \sum_{k=1}^{m} \lambda_k T^k(x) \\
&= \sum_{k=1}^{m} \lambda_k \sum_{x_{V\setminus A}} T^k(x) \\
&= \sum_{k=1}^{m} \lambda_k T^k_A(x_A) \qquad (3.7)
\end{aligned}
$$

Hence, the marginal of Q is a mixture of the marginals of the component trees.

Inference (in the restricted sense) which is marginalization after conditioning on the evidence is performed in an analog way. Let $\mathcal{E}_B$ be the evidence and $B \subset V$ be the set of variables that the evidence is about. Then

$$
\begin{aligned}
Q_A(x_A|\mathcal{E}_B) &= \sum_{x_{V\setminus(A\cup B)}} Q(x|\mathcal{E}_B) \qquad (3.8) \\
&= \sum_{k=1}^{m} \lambda_k T^k_A(x_A|\mathcal{E}_B). \qquad (3.9)
\end{aligned}
$$

One can also infer the value of the hidden variable given some evidence $\mathcal{E}_B$ by applying Bayes' rule

$$
\begin{aligned}
Pr[z=k|\mathcal{E}_B] &= \frac{Pr[\mathcal{E}_B|z=k]Pr[z=k]}{\sum_{k'} Pr[\mathcal{E}_B|z=k']Pr[z=k']} \qquad (3.10) \\
&= \frac{\lambda_k T^k_B(\mathcal{E}_B)}{\sum_{k'} \lambda_{k'} T^{k'}_B(\mathcal{E}_B)} \qquad (3.11)
\end{aligned}
$$

In particular, when the evidence consists of observing all the visible variables $\mathcal{E} = \{v_1 = x_1, v_2 = x_2, \ldots v_n = x_n\}$, $B = V$ and $(x_1, x_2, \ldots x_n) = x$ the *posterior* probability distribution of z is

$$
Pr[z=k|\mathcal{E}] \equiv Pr[z=k|x] = \frac{\lambda_k T^k(x)}{\sum_{k'} \lambda_{k'} T^{k'}(x)} \qquad (3.12)
$$

Sampling Sampling in a mixture of trees, in the presence of evidence or not, is a straightforward extension of sampling from one tree discussed in chapter 2.2. The procedure for sampling from a mixture of trees uses **TreeSample(T, $\mathcal{E}$)** the procedure that samples from a tree distribution given evidence $\mathcal{E}$ (possibly void)

Algorithm **MixTreeSample**(Q, $\mathcal{E}$)

Input a mixture of trees Q
evidence $\mathcal{E}$

1. sample k the value of z from $(\lambda_1, \lambda_2, \ldots \lambda_n)$
2. sample x from T^k component of the mixture and returns a value x.

Output x

Conclusions As this subsection has shown, the basic operations on mixtures of trees, marginalization, conditioning and sampling, are direct extensions of the corresponding operations on tree distributions. Their complexity scales accordingly: Marginalization over a subset $V \setminus A$ in a mixture of trees takes m times the computation for marginalization over the same subset in a single tree. For instance, computing the marginal of a single

variable (that is not in a separator) takes $\mathcal{O}(mr_{MAX}^2)$. Inference in a mixture of trees takes m junction tree propagation operations, $\mathcal{O}(mnr_{MAX}^2)$. Inferring the value of the hidden variable takes mn multiplications. Sampling from a mixture is $\mathcal{O}(m + n.n_{\mathrm{s}})$, where n_{s} is the number of operations required to sample a value of a vertex in a clique, conditioned on the value of the other vertex. For the directed tree representation $n_{\mathrm{s}} \leq r_{MAX}$. The choice of the tree to sample from is an m-way choice requiring $\mathcal{O}(m)$ operations.

3.3 Learning mixtures of trees in the ML framework

3.3.1 The basic algorithm

This section will show how a mixture of trees can be fit to an observed dataset in the Maximum Likelihood paradigm via the EM algorithm [18].

The learning problem is similar to the one formulated for trees in chapter 2. We are given a set of observations $\mathcal{D} = \{x^1, x^2, \ldots, x^N\}$ and we are required to find the mixture of trees Q that satisfies

$$Q \;=\; \underset{Q'}{\operatorname{argmax}} \sum_{i=1}^{N} \log Q(x^i). \tag{3.13}$$

Here and in the rest of the thesis we will assume that there are no missing values for the variables in V. As for z we will first assume that it is hidden. We denote the (hidden) values of the choice variable by $\{z^i,\, i = 1, \ldots N\}$.

Learning the ML mixture of trees model will be done by means of the Expectation-Maximization (EM) algorithm. This is an iterative algorithm devised for fitting maximum likelihood parameters for models with hidden or missing variables. Each iteration consists of two steps, Expectation and Maximization, that will be described below.

The EM algorithm introduces a likelihood function called the *complete log-likelihood* which is the log-likelihood of both the observed and the unobserved data given the current model estimate $\mathcal{M} = \{m,\, T^k,\, \lambda^k,\, k = 1, \ldots m\}$

$$l_c(x^{1,\ldots N}, z^{1,\ldots N}|\mathcal{M}) \;=\; \sum_{i=1}^{N}\sum_{k=1}^{m} \delta_{k,z^i}(\log \lambda_k + \log T^k(x^i)) \tag{3.14}$$

The complete log-likelihood depends on the unknown values of the hidden variable and therefore it is not directly computable. The idea underlying the EM algorithm is to instead compute and optimize the *expected* value of l_c.

The Expectation (E) step consists of estimating the posterior probability of the hidden variable for each of the observations. In our case this means estimating the probability of each tree generating data point x^i

$$Pr[z^i = k|x^i, \mathcal{M}] \;=\; \gamma_k(i) \;=\; \frac{\lambda_k T^k(x^i)}{\sum_{k'} \lambda_{k'} T^{k'}(x^i)} \;=\; E[\delta_{k,z^i}] \tag{3.15}$$

One uses these posterior probabilities to compute the expectation of l_c, which is a linear function of the $\gamma_k(i)$ values.

$$E[l_c(x^{1,\ldots N}, z^{1,\ldots N}|\mathcal{M})] \;=\; \sum_{i=1}^{N}\sum_{k=1}^{m} \gamma_k(i)(\log \lambda_k + \log T^k(x^i)) \tag{3.16}$$

Let us introduce the following quantities:

$$\Gamma_k = \sum_{i=1}^{N} \gamma_k(x^i), \quad k = 1, \dots m \tag{3.17}$$

$$P^k(x^i) = \frac{\gamma_k(i)}{\Gamma_k} \tag{3.18}$$

The sums $\Gamma_k \in [0, N]$ can be interpreted as the total number of data points that are generated by component T^k. By normalizing the posteriors $\gamma_k(i)$ with Γ_k we obtain a probability distribution P^k over the data set. In the following it will be shown that P^k acts as a target distribution for T^k. For now, let us express the expected complete log-likelihood in terms of P^k and Γ_k.

$$E[l_c \,|\, x^{1,\dots N}, \mathcal{M}] = \sum_{k=1}^{m} \Gamma_k \log \lambda_k + \sum_{k=1}^{m} \Gamma_k \sum_{i=1}^{N} P^k(x^i) \log T^k(x^i) \tag{3.19}$$

The Maximization (M) step of the EM algorithm reestimates the parameters of the model so as to maximize $E[l_c \,|\, x^{1,\dots N}, \mathcal{M}]$. It can be proved [18] that the iteration over these two steps converges a local maximum of the log-likelihood of the visible data given the model.

By inspecting equation (3.19) one can see that the expression of $E[l_c]$ is a sum whose terms depend on disjoint subsets of the model's parameters. Hence, one can maximize separately each term of the sum w.r.t. to the part of the model that it depends on.

Thus, by maximizing the first term of (3.19) subject to the constraint

$$\sum_{k=1}^{m} \lambda_k = 1$$

one obtains the new values for the parameters λ

$$\lambda_k = \frac{\Gamma_k}{N} \text{ for } k = 1, \dots m \tag{3.20}$$

To obtain the new distributions T^k, we have to maximize for each k the expression that is the negative of the cross-entropy between P^k and T^k.

$$\sum_{i=1}^{N} P^k(x^i) \log T^k(x^i) \tag{3.21}$$

This problem is equivalent to the problem of fitting a tree to a given distribution and can be solved exactly as shown in section 2.3.2. Here I will give a brief reminder of the procedure. First, one has to compute the mutual information between each pair of variables in V under the target distribution P^k

$$I^k_{uv} = \sum_{x_u x_v} P^k_{uv}(x_u, x_v) \log \frac{P^k_{uv}(x_u, x_v)}{P^k_u(x_u) P^k_v(x_v)}, \quad u, v \in V, u \neq v. \tag{3.22}$$

Second, the optimal tree structure E_{T^k} is found by a Maximum Weight Spanning Tree algorithm using I^k_{uv} as the weight for edge $(u, v), \forall u, v \in V$. Once the tree is found, its marginals T^k_{uv} (or $T^k_{u|v}$), $(uv) \in E_{T^k}$ are exactly equal to the corresponding marginals P^k_{uv}

of the target distribution P^k. They are already computed as an intermediate step in the computation of the mutual informations I^k_{uv} (2.26). The algorithm is summarized here:

Algorithm **MixTree** – outline

MixTree$(\mathcal{D}, \mathcal{M}_0)$

Input: Dataset $\{x^1, \ldots x^N\}$
Initial model $\mathcal{M}_0 = \{m,\ T^k,\ \lambda^k,\ k = 1, \ldots m\}$
Procedure **TreeLearn**(P)
Iterate until convergence:
E step: compute $\gamma^i_k,\ P^k(x^i)$ for $k = 1, \ldots m,\ i = 1, \ldots N$ by (3.15), (3.17), (3.18)
M step: for $k = 1, \ldots m$
Ml. $\lambda_k \leftarrow \Gamma_k / N$
MT. T^k = **TreeLearn**(P^k)
Output model $\mathcal{M} = \{m,\ T^k,\ \lambda^k,\ k = 1, \ldots m\}$

Now we show the steps of the algorithm again in more detail. Next to each step is displayed its running time. Time and memory requirements will be discussed in the next subsection.

Algorithm **MixTree**

Input: Dataset $\{x^1, \ldots x^N\}$
Initial model $\mathcal{M} = \{m,\ T^k,\ \lambda^k,\ k = 1, \ldots m\}$
Procedure MWST(weights) that fits a maximum weight spanning tree over V
Iterate until convergence:

E step:	compute $\gamma^i_k,\ P^k(x^i)$ for $k = 1, \ldots m,\ i = 1, \ldots N$ by (3.15), (3.17), (3.18)	$\mathcal{O}(mnN)$
M step:	for $k = 1, \ldots m$	
M1.	$\lambda_k \leftarrow \Gamma_k / N$	$\mathcal{O}(m)$
M2.	compute marginals $P^k_v,\ P^k_{uv},\ u, v \in V$	$\mathcal{O}(mn^2N)$
M3.	compute mutual information $I^k_{uv}\ u, v \in V$	$\mathcal{O}(mnr^2_{MAX})$
M4.	call MWST($\{\ I^k_{uv}\ \}$) to generate E_{T^k}	$\mathcal{O}(mn^2)$
M5.	$T^k_{uv} \leftarrow P^k_{uv},\ ; T^k_v \leftarrow P^k_v$ for $(u, v) \in E_{T^k}$	$\mathcal{O}(mnr^2_{MAX})$

3.3.2 Running time and storage requirements

Running time. Computing the likelihood of a data point under a tree distribution (in the directed tree representation, where the parameters represent conditional distributions) takes $n - 1 = \mathcal{O}(n)$ multiplications. Hence, the E step should require $\mathcal{O}(mnN)$ floating point multiplications/divisions.

For the M step the situation is: m divisions for computing the λ_k values in step **M1**; $\mathcal{O}(mn^2N)$ for the marginals of P^k in step **M2**; another $\mathcal{O}(mn^2r^2_{MAX})$ for the mutual informations in step **M3**; $\mathcal{O}(mn^2)$ for running the MWST algorithm m times in step **M4**; finally $\mathcal{O}(mnr^2_{MAX})$ for computing the tree parameters in the directed representation in step **M5** of the algorithm. These values are presented at the right of the **MixTree** algorithm above. The total running time per EM iteration is thus

$$\mathcal{O}(mn^2N + mnr^2_{MAX})) \tag{3.23}$$

The dominating terms in this cost correspond to the computation of the marginals P^k_{uv} and of the $mn(n-1)/2$ mutual informations under the distributions P^k.

Storage requirements Storing the data takes mN integer numbers; additionally, to represent the model one needs $\mathcal{O}(mnr^2_{MAX})$ real parameters. This storage is necessary independently of the learning algorithm used.

The intermediate values that are needed by the EM algorithm are:

- the γ_k values that require mN storage locations. The P^k values can overwrite the γ_k values so that no additional storage is needed for them.
- the single variable and pairwise marginals of P^k that require $\frac{n(n+1)}{2} r^2_{MAX}$ storage. They can be overwritten for successive values of k.
- the mutual informations I^k_{uv} between pairs of variables that require $n(n-1)/2$ storage locations. They can also be overwritten for each k.
- the additional temporary storage required by the MWST algorithm. In the Kruskal algorithm, this amount is proportional to the number of candidate edges, namely $\mathcal{O}(n^2)$. Other algorithms, [10] require only linear space.

The total storage is

$$\mathcal{O}(mN + mnr^2_{MAX} + n^2 r^2_{MAX}) \tag{3.24}$$

Again, the dominant cost corresponds to the computation of the pairwise marginals of the distributions P^k.

3.3.3 Learning mixtures of trees with shared structure

It is possible to constrain the m trees to share the same structure, thus constructing a truly Bayesian network. The steps of the EM algorithm that learn mixtures of trees with shared structure, called **MixTreeS**, are given below. Most of the reasoning and calculations parallel the ones describing the fitting of a tree to a distribution in section 2.3.2.

The E step here is identical to the E step of the **MixTree** algorithm. So are the expression of the expected complete log-likelihood

$$E[l_c \,|\, x^{1,\ldots N}, \mathcal{M}] \;=\; \sum_{k=1}^{m} \Gamma_k \log \lambda_k + \sum_{k=1}^{m} \Gamma_k \sum_{i=1}^{N} P^k(x^i) \log T^k(x^i) \tag{3.25}$$

One can easily see that the reestimation of $\{\lambda_k,\, k = 1, \ldots m\}$ is decoupled from the estimation of the rest of the model and can be performed in the same way as before, i.e.

$$\lambda_k \;=\; \frac{\Gamma_k}{N} \;\text{ for } k = 1, \ldots m \tag{3.26}$$

The difference is in reestimating the k tree distributions, since now they are constrained to have the same structure. Hence, the maximization over the tree distributions cannot be decoupled into m separate tree estimations but must be performed simultaneously for all the trees. A reasoning similar to one in section 2.3.2 shows that for any given structure the optimal parameters of each tree edge T^k_{uv} are equal to the parameters of the corresponding marginal distribution P^k_{uv} (equation (2.17),reproduced here)

$$T^k_{uv} \;\equiv\; P^k_{uv} \;\; \forall (u,v) \in E \tag{3.27}$$

It remains only to find the optimal structure. The expression to be optimized is the second sum in the r.h.s. of equation (3.25):

$$\sum_{k=1}^{m} \Gamma_k \sum_{i=1}^{N} P^k(x^i) \log T^k(x^i) \tag{3.28}$$

By making the substitution (3.27) in the above expression and following a series of steps similar to (2.20–2.25) we obtain that (3.28) is equal to

$$\sum_{k=1}^{m} \Gamma_k [\sum_{(u,v)\in E} I^k_{uv} - \sum_{v\in V} H(P^k_v)] \tag{3.29}$$

$$= N \sum_{k=1}^{m} \lambda_k \sum_{(u,v)\in E} I^k_{uv} + \text{constant independent of structure} \tag{3.30}$$

$$= N \sum_{(u,v)\in E} I_{uv|z} + \text{constant} \tag{3.31}$$

Hence, optimizing the structure E can be again done by a MWST algorithm with the edge weights represented by $I_{uv|z}$ the *conditional mutual information* between the variables adjacent to it given z. This algorithm is essentially the same as the TANB learning algorithm of [24, 22]. The expressions in the right column represent the running time for each step of the algorithm.

Algorithm **MixTreeS**

Input: Dataset $\{x^1, \ldots x^N\}$

Initial model $\mathcal{M} = \{m,\ T^k,\ \lambda^k,\ k = 1, \ldots m\}$

Procedure MWST(weights) that fits a maximum weight spanning tree over V

Iterate until convergence:

E step:		compute $\gamma^i_k,\ P^k(x^i)$ for $k = 1, \ldots m,\ i = 1, \ldots N$ by (3.15), (3.17), (3.18)	$\mathcal{O}(mnN)$
M step:	**M1.**	for $k = 1, \ldots m$ compute marginals $P^k_v,\ P^k_{uv},\ u, v \in V$	$\mathcal{O}(mn^2N)$
	M2.	compute the conditional mutual information $I_{uv\|z}\ u, v \in V$	$\mathcal{O}(mnr^2_{MAX})$
	M3.	call MWST($\{\ I_{uv\|z}\ \}$) to generate E_T	$\mathcal{O}(n^2)$
	M4.	for $k = 1, \ldots m$ $\lambda_k \leftarrow \Gamma_k / N$	$\mathcal{O}(m)$
	M5.	for $k = 1, \ldots m$ $T^k_{uv} \leftarrow P^k_{uv},\ ; T^k_v \leftarrow P^k_v$ for $(u, v) \in E_T$	$\mathcal{O}(mnr^2_{MAX})$

3.3.4 Remarks on the learning algorithms

The above described procedures for learning mixtures of trees from data are based on one important assumption that should be made explicit now. It is the

Parameter independence assumption: *For any k, v and value of* $\mathrm{pa}(v)$ *the distribution $T^k_{v|\mathrm{pa}(v)}$ is a multinomial with $r_v - 1$ free parameters that are independent both of the tree structure and of any other parameters of the mixture.*[1]

[1]In [33] this assumption is stated as three distinct assumptions called: parameter modularity, global parameter independence and local parameter independence.

The parameter independence assumption is essential to the computational efficiency of the **MixTree** and **MixTreeS** algorithms. The independence of the parameters of $T^k_{v|\mathrm{pa}(v)}$ of any other parameters allows for the simple form of step **M5** of the EM algorithm. The independence from tree structures allows us to use the MWST algorithm to globally maximize over tree structures at each iteration. As a practical remark, note that although equation (3.27) calls for copying the marginals of the target distributions P^k, the algorithms should implement the directed tree representation and thus copy in its parameters the conditional probabilities $P^k_{v|\mathrm{pa}(v)}$.

Note also that, although the **TreeLearn** algorithm always attains a global solution, learning a mixture of trees by the EM algorithm will converge to a *local* maximum of the likelihood.

The tree structures obtained by the basic algorithm are connected. In the following chapters we will show reasons and ways to obtain disconnected tree structures.

Missing variables are handled elegantly by trees. Any number of nonadjacent missing variables can be marginalized out in $\mathcal{O}(r_{MAX})$ time and this bound grows exponentially with l, the size of the largest connected subset of missing variables.

Observed but unknown choice variable An interesting special case is the situation where the choice variable is in fact one of the observed variables (or a small subset thereof), but we don't know which one? To discover it, one can either: build several mixtures by conditioning on each one of the observables and then compare their posteriors, or: build one standard mixture model and then compare the mutual information between the structure variable and each of the others to identify the most likely candidate.

3.4 Summary and related work

This chapter has introduced the mixture of trees model with its variants and has shown that the basic operations on mixtures of trees are direct extensions of the same operations on tree distributions and thus inherit their excellent computational properties. It has also presented a tractable algorithm to learn the model from data in the Maximum Likelihood framework.

Trees have been noticed for their flexibility and computational efficiency as early as 1968 by [7] who also used them in a classification task, in what is called here a mixture of trees with visible choice variable (i.e. by fitting a different tree to each class). [53] builds on the Chow and Liu algorithm an exact algorithm for learning a polytree *when the true distribution is known to be a polytree*. Fitting the best polytree to an arbitrary distribution is NP-hard [33, 13]. [27] considered the same model, now with an explicit and observed choice variable as a special case of the *Bayesian multinet* [28]. The MTSS appeared in the work of [24] as the Tree Augmented Naive Bayes classifier and was further developed in [22, 23]. The latter work also considers continuous variables and suggests an ingenious heuristic for deciding whether the continuous variables should be discretized or whether their density should be approximated (e.g. by a mixture of Gaussians). The mixture of trees as it was presented here encompasses the aforementioned models as special cases; it was introduced in [46, 47]. Classification results with the mixture of trees (which will be presented in chapter 7 were first published in [47].

Mixture models, the second "parent" of mixtures of trees, have come an even longer way. Therefore I will only cite mixture models that are closely related to the present work. The mixture of factorial distribution in the form of the Naive-Bayes model (i.e with the choice

variable being a class variable in classification task), sometimes known as Auto-Class was given the name and an attentive study in [5] and was heavily used ever since for its excellent cost/performance ratio[2]. [41] successfully used a mixture of factorial distributions with a hidden variable for classification and this line was further followed by [49] who combined the two. The idea of learning tractable but simple belief networks and superimposing a mixture to account for the remaining dependencies was developed independently of this work by [64] into mixtures of Gaussian belief networks. Their work interleaves EM parameter search with Bayesian model search in a heuristic but general algorithm.

From here we shall go on to examine learning the same model in the Bayesian framework and show that the algorithms introduced here can be extended to that case as well. The quadratic time and storage complexity of the learning algorithm is satisfactory for medium scale problems but may become prohibitive for problems of very high dimensionality. We shall show how these requirements can be reduced under certain conditions of practical importance. We shall also examine how a mixture of trees can be used for classification. Finally we study the use of the mixture of trees learning algorithm as a heuristic for solving a challenging problem: hidden variable discovery.

[2]Section 7.3.5 shows that the Naive Bayes classifier has little sensitivity to irrelevant attributes, just like the single tree classifier, which partly explains its success.

Chapter 4

Learning mixtures of trees in the Bayesian framework

The ML paradigm assumes that the model is estimated using the data only and excluding other sources of knowledge about parameters or model structure. But, if prior knowledge exists and if it can be represented as a probability distribution over the space of mixtures of trees models, then one can use the Bayesian formulation of learning to combine the two sources of information.

In the Bayesian framework, the main object of interest is the posterior distribution over models (in our case mixtures of trees) given the observed data $Pr[Q|\mathcal{D}]$. By Bayes' formula, the posterior is proportional to

$$Pr[Q|\mathcal{D}] \;\propto\; Pr[Q,\,\mathcal{D}] \;=\; Pr[Q]\prod_{x\in\mathcal{D}} Q(x) \tag{4.1}$$

In the above $Pr[Q]$ represents the prior distribution over the class of mixture of trees models; the second factor is $Q(\mathcal{D})$, the likelihood of the data given the model Q.

The probability of an observation x is obtained by *model averaging*

$$Pr[x] \;=\; \int Q(x)Pr[Q|\mathcal{D}]\,dQ \tag{4.2}$$

However, except for a few special cases neither $Pr[x]$ nor the posterior $Pr[Q|\mathcal{D}]$ are representable in closed form. A common approach is to approximate the posterior distribution around its mode(s) for example by the Laplace approximation []. Another approach is to replace the integration in equation (4.2) by a finite sum over a set $\mathcal{Q}$ of models with high posterior probability.

$$Pr[x] \;=\; \sum_{Q\in\mathcal{Q}} Q(x)Pr[Q|\mathcal{D}] \tag{4.3}$$

This approximation is equivalent to setting $Pr[Q|\mathcal{D}]$ to 0 for all the mixtures not in $\mathcal{Q}$. Consequently, the normalization constant in the above formula is computed over $\mathcal{Q}$ only.

Finally, if we are to choose one model only to summarize the posterior distribution $Pr[Q|\mathcal{D}]$ then a natural choice is the mean of the distribution. As it will be shown the mean can sometimes be expressed as the MAP estimate under a certain parameterization.

Finding the modes of the posterior distribution $Pr[Q|\mathcal{D}]$ is a necessary step in all the three approaches; therefore, although *Maximum A-Posteriori (MAP)* estimation of Q is not seen as a purpose per se, this chapter will be concerned with maximizing the posterior distribution $Pr[Q|\mathcal{D}] \propto Pr[Q,\mathcal{D}]$.

4.1 MAP estimation by the EM algorithm

According to equation (4.1) in the previous section, maximizing the posterior log-likelihood of a model is equivalent to maximizing

$$\log Q(\mathcal{D}) \; + \; \log Pr[Q] \tag{4.4}$$

which differs from $\log Pr[Q|\mathcal{D}]$ only by an additive constant. The EM algorithm previously used to find Maximum Likelihood estimates can be adapted to maximize the above expression (which represents the expression of the log-likelihood plus the term $\log Pr[Q]$) [50]. The quantity to be iteratively maximized by EM is now

$$E[\log Pr[Q|x^{1,\ldots N}, z^{1,\ldots N}]] \; = \; \log Pr[Q] \; + \; E[l_c(x^{1,\ldots N}, z^{1,\ldots N}|Q)] \tag{4.5}$$

It is easy to see that the added term does not have any influence on the E step of the EM algorithm, which will proceed exactly as before. However, in the M step, we must be able to successfully maximize the r.h.s. of (4.5). We have seen that in the previous chapter exact maximization was enabled by the fact that we obtained a separate set of equations for λ and for each of the mixture components T^k. Therefore, we will look for priors over the mixtures of trees models that are amenable to this decomposition.

For a given m a prior over the space of mixtures of trees with m mixture components comprises a prior over the distribution $\lambda_{1,\ldots m}$ of the hidden variable and priors over each of the m trees' structures and parameters. We require that the prior is expressed in the product form

$$Pr[Q] \; = \; Pr[\lambda_{1,\ldots m}] \prod_{k=1}^{m} \underbrace{Pr[E_k] Pr[\text{parameters} T^k | E_k]}_{Pr[T^k]}. \tag{4.6}$$

Moreover, remember that the key to the efficient maximization in equation (3.21) was the fact that its r.h.s could be further decomposed into a sum of independent terms, each of them corresponding to an edge of T^k. Therefore, a similar property is desirable for the prior $Pr[T^k]$. A prior probability for a tree has two components: the prior for the tree structure (represented by the edge set E_k) and the prior for the tree parameters, given the structure. We shall require that both components decompose in a way that matches the factoring of the likelihood in equation (3.21):

$$Pr[T^k] \; = \; Pr[E_k] \prod_{(u,v)\in E_k} Pr[T^k_{uv}] \tag{4.7}$$

$$\propto \; \prod_{(u,v)\in E_k} \beta^k_{uv} Pr[T^k_{uv}] \tag{4.8}$$

A prior over the class of mixtures of trees having all the above properties is called a *decomposable prior*.

In the context of decomposable priors, at the M step of the EM algorithm the new model Q^{new} is obtained by solving the following set of decoupled maximization problems:

- for the λ parameters

$$\lambda^{new}_{1,\ldots m} \; = \; \underset{\Sigma_k \lambda_k = 1}{\text{argmax}} \sum_{k=1}^{m} \Gamma_k \log \lambda_k \; + \; \log P(\lambda_{1,\ldots m}). \tag{4.9}$$

- for each tree T^k equation (3.21) in the previous chapter is replaced by

$$T^{k\,new} \;=\; \underset{T^k}{\operatorname{argmax}} \sum_{i=1}^{N} P^k(x^i) \log T^k(x^i) \;+\; \log Pr[T^k] \tag{4.10}$$

Requiring that the prior be decomposable is equivalent to making strong independence assumptions: for example, it means that the prior probability of each tree in the mixture is independent of all the others as well as of the probability distribution of the mixture variable. In the following sections, the implications of some of these independence assumptions will be made more explicit. It will also be shown that, although the independence assumptions made are strong ones, they are not too restrictive. The class of decomposable priors is rich enough to contain members that are interesting and of practical importance.

4.2 Decomposable priors for tree distributions

4.2.1 Decomposable priors over tree structures

The general form of a decomposable prior for the tree structure E is one where each edge contributes a constant factor independent of the presence or absence of other edges in E.

$$Pr[E] \;\propto\; \prod_{(u,v)\in E} \exp(-\beta_{uv}) \tag{4.11}$$

With this prior, the expression to be maximized in the M step of the EM algorithm (former equation (3.19))becomes

$$E[l_c \,|\, x^{1,\ldots N}, \mathcal{M}] \;=\; \sum_{k=1}^{m} \Gamma_k \log \lambda_k + \sum_{k=1}^{m} \Gamma_k \left(\sum_{i=1}^{N} P^k(x^i) \log T^k(x^i) - \sum_{(u,v)\in E_k} \frac{\beta_{uv}^k}{\Gamma_k} \right) \tag{4.12}$$

Consequently, each edge weight in tree T^k is penalized by its corresponding β divided by the total number of points that tree k is responsible for:

$$W_{uv}^k \;=\; I_{uv}^k - \frac{\beta_{uv}^k}{\Gamma_k} \tag{4.13}$$

A negative β_{uv} increases the probability of uv being present in the final solution. On the contrary, if β_{uv} is positive, it acts like a penalty on the presence of edge uv in the tree. If β_{uv} is sufficiently large the weight W_{uv} becomes negative and the edge is not added to the tree. Hence, by introducing edge penalties, one can obtain trees T^k having fewer than $n-1$ edges and thus being disconnected. Notice that the strength of the prior decreases with Γ_k the total number of points assigned to mixture component k. Thus, for equal priors for all trees T^k, trees accounting for fewer data points will be penalized stronger and therefore will be likely to have fewer edges.

If one chooses the edge penalties to be proportional to the increase in the number of parameters caused by the addition of edge uv to the tree,

$$\beta_{uv} \;=\; \beta(r_u - 1)(r_v - 1) \tag{4.14}$$

then a *Minimum Description Length* (MDL) [55] type of prior is implemented. Note that the effective penalty for each parameter is inversely proportional to the number of data points Γ_k the tree that the parameter belongs to is responsible for.

In [33], in the context of learning Bayesian networks, the following prior is suggested:

$$Pr[E] \ = \propto \kappa^{\Delta(E,E^*)} \tag{4.15}$$

where $\Delta(\cdot)$ is a distance metric and E^* is the prior network structure. Thus, this prior penalizes deviations from the prior network. For trees, the distance metric becomes the symmetric difference:

$$\Delta(E, E^*) \ = \ |(E \setminus E^*) \bigcup (E^* \setminus E)| \tag{4.16}$$

This prior is also factorable, entailing

$$\beta_{uv} \ = \ \begin{cases} -\ln \kappa & (uv) \in E^* \\ \ln \kappa & (uv) \notin E^* \end{cases} \tag{4.17}$$

What happens in the case of mixtures of trees with shared structure? Recall that in this case each edge uv had a weight proportional to the mutual information between u and v conditioned on z: $I_{uv|z}$ (3.31). The effect of the decomposable prior is to penalize this weight by $-\frac{\beta_{uv}}{N}$.

4.2.2 Priors for tree parameters: the Dirichlet prior

We will introduce now an important subclass of decomposable priors for parameters called *Dirichlet* priors. The Dirichlet prior is the conjugate prior of the *multinomial* distribution. Since the distribution of the mixture variable is a multinomial as well, the facts shown in this section will cover this case too.

Let z be a discrete random variable taking r values and let $\theta_j = P_z(j), \ j = 1, \ldots r$. Then, the probability distribution of an i.i.d. sample of size N from P_z is given by

$$P(z^{1,\ldots N}) \ = \ \prod_{j=1}^{r} \theta_j^{N_j} \tag{4.18}$$

where N_j, $j = 1, \ldots r$ represent the number of times the value j is observed in and are called the *sufficient statistics* of the data. The sample itself is said to obey a *multinomial* distribution.

The Dirichlet distribution is defined over the domain of $\theta_{1,\ldots r}$ and depends on r real parameters $N'_{1,\ldots r} > 0$ by

$$D(\theta_{1,\ldots r}; N'_{1,\ldots r}) \ = \ \frac{\Gamma(\sum_j N'_j)}{\prod_j \Gamma(N'_j)} \prod_j \theta_j^{N'_j - 1} \tag{4.19}$$

In the above $\Gamma()$ represents the Gamma function defined by

$$\Gamma(p) \ = \ \int_0^\infty t^{p-1} e^t dt. \tag{4.20}$$

For any nonnegative integer n

$$\Gamma(n+1) \ = \ n! \tag{4.21}$$

The importance of the Dirichlet distribution in connection with a multinomially distributed variable resides in the following fact: If the parameters θ of the multinomial distribution have as prior a Dirichlet distribution with parameters $N'_j, \ j = 1, \ldots r$, then, after

observing a sample with sufficient statistics $N_j,\ j = 1, \ldots r$, the posterior distribution of θ is a Dirichlet distribution with parameters $N'_j + N_j,\ j = 1, \ldots r$. This justifies denoting the distribution's parameters by N'. One popular alternative parameterization for the Dirichlet distribution is given by:

$$N' = \sum_{j=1}^{r} N'_j \tag{4.22}$$

$$P'_j = \frac{N'_j}{N'} \quad j = 1, \ldots r \tag{4.23}$$

$$\tag{4.24}$$

Note that in this parameterization the means of the parameters θ_j are equal to P'_j. We say that the Dirichlet distribution is a conjugate prior for the class of multinomial distributions. The property of having conjugate priors is characteristic for the exponential family of distributions.

The Dirichlet prior in natural coordinates

The multinomial distribution was represented before as defined by the parameters $\theta_j,\ j = 1, \ldots r$. But there are infinitely many ways to parametrize the same distribution. For each set of parameters $y_{1,\ldots r_y}$ the corresponding representation of the Dirichlet prior results from the well known change of variable formula

$$D(y_{1,\ldots r_y};\ N'_{1,\ldots r-1}) \ = \ D(\theta_{1,\ldots r}(y_{1,\ldots r_y}); N'_{1,\ldots r}).|\frac{\partial \theta}{\partial y}| \tag{4.25}$$

with $|\frac{\partial y}{\partial \theta}|$ representing the absolute value of the determinant of the Jacobian of $\theta(y)$. Note that because of the presence of this factor, the maximum of $D(.; N'_{1,\ldots r})$ has both a different value and a different position in each parametrization. This dependence of the parametrization is one fundamental drawback of MAP estimation which justifies the Bayesian and approximate Bayesian approaches mentioned above.

By contrast, the mean of $f(y)D(y; N')$ of any measurable function f over any measurable set is independent of the parametrization. In particular, the mean of the Dirichlet distribution is independent of the parametrization and equal to

$$E[\theta_j] \ = \ \frac{N'_j}{\sum_{j'=1}^{r} N'_{j'}} \quad j = 1, \ldots r \tag{4.26}$$

Of special interest is the so called *natural* parametrization of the multinomial, defined by $r - 1$ unconstrained parameters ϕ:

$$\phi_i \ = \ \log \frac{\theta_i}{\theta_r} \quad i = 1, \ldots r - 1. \tag{4.27}$$

The parameters ϕ take values in $(-\infty, \infty)$ when $\theta_{1,\ldots r} > 0$. The reverse transformation from ϕ coordinates to θ coordinates is defined by:

$$\theta_r = \frac{1}{1 + \sum_{i=1}^{r-1} e^{\phi_i}} \tag{4.28}$$

$$\theta_j = \frac{e^{\phi_j}}{1 + \sum_{i=1}^{r-1} e^{\phi_i}}, \quad j = 1, \ldots r - 1 \tag{4.29}$$

$$\tag{4.30}$$

In the natural parametrization, the Dirichlet distribution is expressed as

$$D(\phi_{1,\dots r-1};\, N'_{1,\dots r}) \;=\; \frac{\Gamma(\sum_{j=1}^{r} N'_j)}{\prod_{j=1}^{r} \Gamma_(N'_j)} \prod_{i=1}^{r-1} \left(\frac{e^{\phi_i}}{1+\sum_{i=1}^{r-1} e^{\phi_i}} \right)^{N'_i} \frac{1}{(1+\sum_{i=1}^{r-1} e^{\phi_i})^{N'_r}} \tag{4.31}$$

A remarkable property of the natural parametrization is that its mode coincides in position with the mean. To see this, it suffices to equate to 0 the partial derivatives of the Dirichlet distribution w.r.t the ϕ parameters. After some calculations, one obtains

$$\frac{e^{\phi_i}}{1+\sum_{i'=1}^{r-1} e^{\phi_{i'}}} \;=\; \frac{N'_i}{\sum_{j=1}^{r} N'_j} \quad i = 1, \dots r-1 \tag{4.32}$$

or equivalently

$$\theta_j \;=\; \frac{N'_i}{N'} \quad j = 1, \dots r \tag{4.33}$$

Dirichlet priors for trees and mixtures

[33] show that the assumptions of *likelihood equivalence* which says that data should not help discriminate between structures which represent the same probability distribution, *parameter modularity* which says that the parameters corresponding to an edge of the tree should have the same prior every time the edge is present in the tree and *parameter independence* which says that in any directed tree parametrization the parameters of each edge are independent of anything else. These combined with some weak technical assumptions[1] imply that the parameter prior is Dirichlet.

In the case of trees, likelihood equivalence is automatically assured by definition (all tree parametrizations, directed or not, represent the same distribution). Parameter modularity and parameter independence are implicitly assumed as a basis for the tree fitting procedure that we have used. Moreover, requiring a decomposable prior over tree structures preserves parameter modularity. Hence, requiring decomposable priors not only fits naturally in the framework already adopted, but introduces almost no additional constraints. [33] also show that the likelihood equivalence constrains the Dirichlet priors for all the parameter sets to share a common equivalent sample size N'. All these results hold for trees and mixtures as described by the present framework. Moreover, it is easy to see that when learning tree distributions the Dirichlet parameters collapse into the fictitious sufficient statistics represented by the pairwise "counts" N'_{uvij} = "# times $u = i$ and $v = j$". Therefore, for the case of trees, the prior over all parameters of all possible edges of tree T^k can be described by the set of fictitious marginal counts

$$\{N'^k_{uvij},\; u, v \in V,\; i = 1, \dots r_u,\; j = 1, \dots r_v\}$$

Alternatively, one can normalize the counts and express the Dirichlet prior over all trees as a table of fictitious marginal probabilities P'_{uv} for each pair u, v of variables plus an *equivalent sample size* N' that gives the strength of the prior.

However, it is important to note that the fictitious marginal counts or alternatively the pairwise marginals P'_{uv} cannot be set arbitrarily[2]. The values P'_{uv} have to satisfy

$$P'_{uv}(x_u x_v) \;=\; \sum_{x \in \Omega(V): u = x_u, v = x_v} P'_V(x) \;\; \forall u, v, x_u, x_v \tag{4.34}$$

[1] These technical assumptions assumptions amount to the positivity of the joint prior.

[2] Take for example a set of three binary variables a, b, c. Assume that $P_{ab}(01) = P_{ab}(10) = 0$ and $P_{bc}(01) = P_{bc}(10) = 0$. This implies that $a = b = c$ and thus constrains $P_{ac}(01) = P_{ac}(10) = 0$.

for some joint distribution P' over $\Omega(V)$.

The above represent a system of linear equality and inequality constraints. Verifying them directly is theoretically straightforward but computationally intractable on account on the exponential number of columns of the system's matrix (a number of the order of $|\Omega(V)|$). Still one can notice that the *uninformative* prior given by

$$P'_{uv}(x_u x_v) \;=\; \frac{1}{r_u r_v} \tag{4.35}$$

is valid since it represents the set of pairwise marginals of the uniform distribution over $\Omega(V)$[3].

If the Dirichlet prior is represented in the natural parameters and the empirical distribution is P, with sample size N, then, from the fact that the Dirichlet prior is a conjugate prior, it follows that finding the MAP tree is equivalent to finding the ML tree for

$$\tilde{P} \;=\; \frac{1}{N+N'}(N'P' + NP). \tag{4.36}$$

Consequently, the parameters of optimal tree will be

$$T_{u|v} \;=\; \frac{\tilde{P}_{uv}}{\tilde{P}_v} \tag{4.37}$$

and, according to the previous section and equation (4.33) they will also represent the mean of the posterior distribution. Moreover, using the parameter independence assumption, we can conclude that the optimal tree distribution itself is the mean of the posterior distribution given the structure.

For mixtures, maximizing the posterior translates into replacing P by P^k and N by Γ_k in equation (4.36) above. Namely, each step of the EM algorithm fits the optimal tree to the distribution

$$\tilde{P}^k \;=\; \frac{1}{\Gamma_k + N'}(N'P' + \Gamma_k P^k). \tag{4.38}$$

If we want to distinguish between trees, then different Dirichlet priors can be used for each tree. By the previous arguments, at each M step the resulting trees T^k and λ_k values represent the mean of their respective posterior distribution. However, this does not imply that the mixture $Q = \sum \lambda_k T^k$ is also the mean of its own posterior. The presence of the hidden variable cancels the independence assumption that allows us to conclude this for the single tree.

[3]This prior is called the BDeu prior in [33]. We note in passing that the uninformative prior denoted there as the K2 metric is not a valid prior from the point of view of equations (4.34).

Chapter 5

Accelerating the tree learning algorithm

"So do something! Fast!" said the Fish.
Dr. Seuss
–The Cat in the Hat

5.1 Introduction

The Chow and Liu (CL) algorithm presented in chapter 2 as the **TreeLearn** algorithm as well as the **MixTree** algorithm that builds on it are quadratic in the dimension of the domain n. This is due to the fact that to minimize the the KL divergence

$$KL(P \,||\, T) \;=\; \sum_x P(x) \log \frac{P(x)}{T(x)} \tag{5.1}$$

the **TreeLearn** algorithm needs the mutual information I_{uv} under P between each pair of variables u, v in the domain. When P is an empirical distribution obtained from and i.i.d. set of data, the computation of all the mutual information values requires time and memory quadratic in the number of variables n and linear in the size of the dataset N. It has been shown (section 3.3.2) that this is the most computationally expensive step in fitting a tree to data. The time and memory requirements of this step are acceptable for certain problems but they may become prohibitive when the dimensionality n of the domain becomes large.

An example of such a domain is information retrieval. In information retrieval, the data points are documents from a data base and the the variables are words from a vocabulary. A common representation for a document is as a binary vector whose dimension is equal to the vocabulary size. The vector component corresponding to a word v is 1 if v appears in the document and 0 otherwise. The number N of documents in the data base is of the order of 10^3 – 10^4. Vocabulary sizes too can be in the thousands or tens of thousands. This means that fitting a tree to the data necessitates $n^2 \;\sim\; 10^6 - 10^9$ mutual information computations and $n^2 N \;\sim\; 10^9 - 10^{12}$ counting operations! However, the problem has a particularity: each document contains only a relatively small number of words (of the order of 10^2) and therefore most of the component values of its corresponding binary vector are null. We call this property of the data *sparsity*. Can we use sparsity to improve on the time (and memory) requirements of the **TreeLearn** algorithm?

The rest of this chapter shows that the answer to this question is yes. It also shows how this improvement can be carried over to learning mixtures of trees or TANB models. We call the algorithms obtained *accelerated Chow and Liu* algorithms (**aCL**). The table below offers a preview of the results that will be achieved. The symbol $s \leq n$ represents a measure of the data sparsity, and N_K is the number of steps of the Kruskal algorithm.

	TreeLearn	aCL
E step	$\mathcal{O}(mnN)$	$\mathcal{O}(msN)$
M step	$\mathcal{O}(mn^2N)$	$\tilde{\mathcal{O}}[m(s^2N + sn + n_k)]$

It is obvious that in general the proportionality with the size of the data set N cannot be improved on, since one needs to examine each data point at least once. Hence the focus will be on improving on the dependence on n. Here the situation is as follows: there are several MWST algorithms, some more efficient than others, but all the algorithms that I know of run in time at least proportional to the number of candidate edges n_e. For the tree learning problem $n_e = n(n-1)/2$ which results in algorithms that are at least quadratic in the number of variables n. But we also have additional information: weights are not completely arbitrary; each edge (uv) has a weight equal to the mutual information I_{uv} between the variables u and v. Can we use this fact to do better? Remark that the only way the MWST algorithm uses the edge weights is in comparisons. The idea the present work is based on is to compare mutual informations between pairs of variables **without actually computing them** whenever this is possible. This way, by partially sorting the edges before running the MWST algorithm significant savings in running time can be achieved. A second but no less important idea is to exploit the sparsity of the data in computing the pairwise marginals P_{uv}. Combining the two will result in algorithms for fitting a tree to a distribution that (under certain assumptions) are jointly subquadratic in N and n w.r.t. both running time and memory. Let us start by stating the assumption underlying them.

5.2 Assumptions

Binary variables. All variables in V take values in the set $\{0, 1\}$. When a variables takes value 1 we say that it is "on", otherwise we say it is "off". Without loss of generality we can further assume that a variable is off more times than it is on in the given dataset. The binary variables assumption will be eliminated later on (section 5.4).

Integer counts. The target distribution P is derived from a set of observations of size N. Hence,

$$P_v(1) \;=\; \frac{N_v}{N} = 1 - P_v(0) \tag{5.2}$$

where N_v represents the number of times variable v is on in the dataset. According to the first assumption,

$$0 \;<\; P_v(1) \leq \frac{1}{2} \quad \text{or} \quad 0 \;<\; N_v \leq \frac{1}{2}N \tag{5.3}$$

We can also exclude as non-informative all the variables that are always on (or always off) thereby ensuring the strict positivity of $P_v(1)$ and N_v.

Let us denote by N_{uv} the number of times variables u and v are simultaneously on. We call each of these events a *cooccurrence* of u and v. The marginal P_{uv} of u and v is given by

$$N.P_{uv}(1,1) \;=\; N_{uv} \tag{5.4}$$
$$N.P_{uv}(1,0) \;=\; N_u - N_{uv} \tag{5.5}$$
$$N.P_{uv}(0,1) \;=\; N_v - N_{uv} \tag{5.6}$$
$$N.P_{uv}(0,0) \;=\; N - N_v - N_u + N_{uv} \tag{5.7}$$

All the information about P that is necessary for fitting the tree is summarized in the counts N, N_v and N_{uv}, $u, v = 1, \ldots, n$ that are assumed to be non-negative integers. From

now on we will consider P to be represented by these counts. Later on (section 5.3.5) this assumption will be considerably relaxed.

Sparse data. Let us denote by $0 \leq |x| \leq n$ the number of variables that are on in observation x. Further, define s, the *sparsity* of the data by

$$s = \max_{i=1,N} |x^i| \tag{5.8}$$

If, for example, the data are documents and the variables represent words from a vocabulary, then s represents the maximum number of distinct words in a document. The time and memory requirements of the accelerated CL algorithm that we are going to introduce depend on the sparsity s. The lower the sparsity, the more efficient the algorithm. From now on, s will be assumed to be a constant and

$$s << n,\, N. \tag{5.9}$$

As we shall see, of the four assumptions introduced here, the sparsity assumption is the only assumption that we cannot dispense with in our program of accelerating the tree learning algorithm.

Data/dimension ratio bounded The ratio of the number of data points N vs. the dimension of the domain n is bounded and bounded away from 0.

$$0 \;<\; R_{min} \;\leq \frac{N}{n} \;\leq\; R_{max} \tag{5.10}$$

This is a technical assumption that will be useful later. It is a plausible assumption for large n and N.

5.3 Accelerated CL algorithms

5.3.1 First idea: Comparing mutual informations between binary variables

The mutual information between two variables $u, v \in V$ can be expressed with the above notations as:

$$\begin{aligned} I_{uv} &= H_u + H_v - H_{uv} \\ &= \frac{1}{N}\{-N_u \log N_u - (N - N_u)\log(N - N_u) + N \log N \\ &\quad - N_v \log N_v - (N - N_v)\log(N - N_v) + N \log N \\ &\quad + N_{uv} \log N_{uv} + (N_u - N_{uv})\log(N_u - N_{uv}) + (N_v - N_{uv})\log(N_v - N_{uv}) \\ &\quad + (N - N_u - N_v + N_{uv})\log(N - N_u - N_v + N_{uv}) - \log N\} \end{aligned} \tag{5.11}$$

Knowing that N is fixed for the dataset, it follows that I_{uv} is a function of 3 variables: N_v, N_u and N_{uv}. Let us fix N_{uv} and N_u and analyze the variation of the mutual information w.r.t. N_v:

$$\begin{aligned} \frac{\partial I_{uv}}{\partial N_v} &= \log N_v - \log(N - N_v) - \log(N_v - N_{uv}) + \log(N - N_u - N_v + N_{uv}) \\ &= \log \frac{N_v(N - N_u - N_v + N_{uv})}{(N_v - N_{uv})(N - N_v)} \end{aligned} \tag{5.12}$$

Equating the above derivative to 0 one obtains the extremum:

$$N_v^* \;=\; \frac{N_{uv}N}{N_u} \tag{5.13}$$

It is easy to verify that N_v^* is a minimum and that it corresponds to mutual independence between u and v. Note that N_{uv}^* is not necessarily an integer.

But more importantly, this calculation has the following practical implication: assume that u is fixed and thereby N_u is also fixed. Assume also that the variables in V are sorted by decreasing N_v and that we are interested only in the variables following u in this ordering. We denote this latter fact by $v \succ u$. To express it in an intuitive way, assume that edge uv is directed from the lower ranking variable toward the higher ranking one. In this case the mutual informations between u and the variables v following u correspond to the weights of the edges outgoing from u. Partition this set of variables (or edges) into sets $V_c, c = 0, 1, \ldots$ according to the number $c = N_{uv}$ of cooccurrences with u. Further partition the sets $V_c,\ c > 0$ into

$$V_c^+ \;=\; \{v \in V | v \succ u \text{ and } N_{uv} = c \text{ and } N_v > N_v^*\} \tag{5.14}$$

$$V_c^- \;=\; \{v \in V | v \succ u \text{ and } N_{uv} = c \text{ and } N_v \leq N_v^*\} \tag{5.15}$$

For $c = 0\ N_v^* = 0$ so that $V_0 \equiv V_0^+$. For each of these subsets I_{uv} varies monotonically (increasing or decreasing) with N_v. It means that if V_c^- and V_c^+ are sorted by increasing respectively decreasing N_v then the variables within each set are sorted in the decreasing order of their mutual information with u. This way, we can achieve a partial sorting of the mutual informations I_{uv} for a given u without computing any of the mutual informations involved. To obtain $v = \underset{v \succ u}{\operatorname{argmax}}\, I_{uv}$ we only need to calculate and compare the first elements of the lists $V_c^{\pm},\ c = 0, 1, \ldots$. This procedure can save substantial amounts of computation provided that the total number of values of $c = N_{uv}$ is small for each u.

If the data are sparse, then most pairs of variables do not cooccur. Consequently, the list $V_c^{\pm}(u)$ for $c > 0$ have, together, much fewer elements than the $V_0(u)$ lists. One can pool the former together, for each u, without significant cost in computation time. Of the two accelerated algorithms that are introduced here, the first costructs and uses all the $V_c^{\pm}$ lists, whereas the second one uses only V_0.

5.3.2 Second idea: computing cooccurrences in a bipartite graph data representation

The bipartite graph data representation Let $\mathcal{D} = \{x^1, \ldots x^N\}$ be a set of observations over n binary variables whose probability of being on is less than 1/2. It is efficient to represent each observation in $\mathcal{D}$ as a list of the variables that are on in the respective observation. Thus, data point $x^i, i = 1, \ldots N$ will be represented by the list $xlist^i = \text{list}\{v \in V | x_v^i = 1\}$. The name of this representation comes from its depiction in figure 5-1 as a bipartite graph $(V, \mathcal{D}, \mathcal{E})$ where each edge $(vi) \in \mathcal{E}$ corresponds to variable v being on in observation i. The space required by this representation is no more than

$$sN \;<<\; nN,$$

and much smaller than the space required by the binary vector representation of the same data.

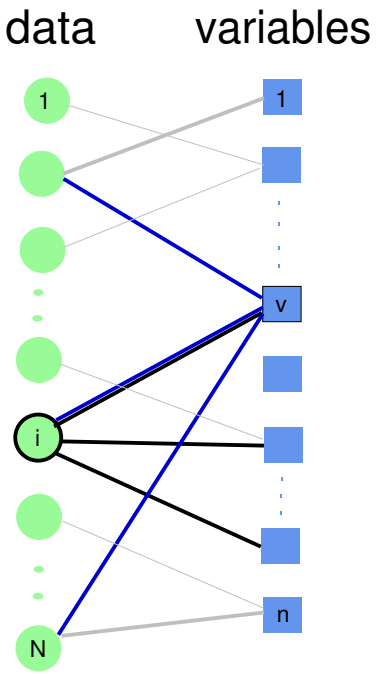


Figure 5-1: The bipartite graph representation of a sparse data set. Each edge iv means "variable v is on in data point i".

Computing cooccurrences in the bipartite graph representation First, let us note that the total number N_C of cooccurrences in the dataset $\mathcal{D}$ is

$$N_C = \sum_{v \succ u} N_{uv} \leq \frac{1}{2} s^2 N \tag{5.16}$$

where s is the previously defined sparsity of the data. Indeed, each data point x contains at most s variables that are on, therefore contributing at most $s(s-1)/2$ cooccurrences to the sum in (5.16). Hence, the above result.

Therefore, as it will be shown shortly, computing all the cooccurrence counts takes the same amount of time, up to a logarithmic factor. The following algorithm not only computes all the cooccurrence numbers N_{uv} for $u, v \in V$ but also constructs the lists $V_c^{\pm}$, $c > 0$. Because the data are sparse, we expect the lists $V_0(u)$ to be on average much larger that the other $V_c^{\pm}(u)$ lists. Therefore, instead of representing $V_0(u)$ explicitly, we construct its complement $\overline{V_0}(u)$ representing the sorted union of all the lists $V_c^{\pm}(u)$, $c > 0$ or, in other words, the list of all the variables that cooccur with u at least once. We assume that the variables are already sorted by decreasing N_v, with ties broken arbitrarily.

This algorithm will serve as a building block for the accelerated Chow and Liu (**aCL**) algorithms that the next subsections will introduce. One of them, **aCL-I**, uses all the $V_c^{\pm}$ lists as shown below. The second one, **aCL-II** simplifies this process by creating only the $\overline{V}_0(u)$ lists and pooling the contents of the rest of the lists (i.e. the variables that cooccur with u) in one list C_u sorted by the mutual information I_{uv}.

For each variable u we shall initialize a temporary storage of cooccurrences denoted by $cheap_u$ organized as a Fibonacci heap (or F-heap) [20]. Then, the lists $V_c^{\pm}(u)$, $c > 0$, $u \in V$ can be computed as follows:

Algorithm **ListByCooccurrence**

Input list of variables sorted by decreasing N_u
dataset $\mathcal{D} = \{xlist^i,\ i = 1, \ldots N\}$

1.for $u = 1, \ldots n$
  initialize $cheap_u$
2. for $i = 1, \ldots N$
  for $u \in xlist^i$
    for $v \in xlist^i, v \succ u$
      insert v into $cheap_u$
3.for $u = 1, \ldots n$
  construct lists $V_c^{\pm}(u), c > 0$
  construct list $\overline{V_0}(u)$

Output lists $V_c(u),\ \overline{V_0}(u)\ u = 1, \ldots n,\ c > 0$

To construct the set of lists $V_c(u), c > 0$ we proceed as follows:
  ConstructLists(u)

compute N_v^*
$c = 0$
$v = u$
while $cheap_u$ not empty
  $vnew$ = extract max $cheap_u$
  if $vnew \succ v$ // new set of cooccurrences
    Insert $v,\ c$ *in corresponding lists*
    $v = vnew$
    $c = 1$
  else // one more cooccurence for the same v
    c++
Insert $v,\ c$ *in corresponding lists* //insert last set of cooccurrences

To insert $v,\ c$ in corresponding lists we do:

if $v \succ u$ // not first step
  if $N_v < N_v^*$
    insert v in $V_c^-(u)$ at the beginning
  else
    insert v in $V_c^+(u)$ at the end
  insert v in $\overline{V_0}(u)$ at the end

The algorithm works as follows: $cheap_u$ contains one entry for each cooccurrence of u and v. Because we extract the elements in sorted order, all cooccurrences of v with u will come in a sequence. We store v and keep increasing the count c until a new value for v comes out of the F-heap. Then it is time to store the previous variable in its corresponding $V_c(u)$. Since the different v's come in decreasing order, we know that in V_c^+ we have to

insert at the end, whereas in V_c^- we have to insert at the beginning to obtain the desired sorting of the lists. $\overline{V_0}(u)$ is the list of all variables that cooccur with u, sorted by decreasing N_v. Since it is assumed that most variables do not cooccur with u, it is more efficient to store $\overline{V_0}(u)$ than its complement $V_0(u)$.

Insertion in each V_c assumes implicitly that we test if the list was created and create it if necessary. To handle insertion in $V_c^\pm$ efficiently, we store the current maximum value of c for each u in the variable $c_{max}(u)$. We maintain a vector of pointers to the V_c lists with dimension $2c_{max}(u)$. If a list is empty its corresponding pointer is a null pointer. This way, each insertion in a $V_c^\pm$ list takes constant time. This time includes checking that the list is initialized. Each insertion in one of the $V_c^\pm$ lists is followed by the insertion of the same variable at the end of the list $\overline{V_0}(u)$, which is also a constant time operation. The amount of additional storage incurred by this method will be discussed later on.

Running time. As shown above, an insertion at the extremity of a list takes constant time. Extracting the maximum of an F-heap takes logarithmic time in the size of the heap. Thus, extracting all the elements of a heap $cheap_u$ of size L_u takes $\sum_{l=1}^{L_u} \log l = \log L_u! \leq L_u \log L_u$.

Bounding the time to empty the heaps $cheap_u$. Extracting all elements of all heaps $cheap_u$ takes therefore less than $\tau = \sum_u L_u \log L_u$. Knowing already that $\sum_u L_u = N_C \leq 1/2 s^2 N$ it is easy to prove that the maximum of τ is attained for all L_u equal. After performing the calculations we obtain $\tau \sim \mathcal{O}(s^2 N \log \frac{N}{n})$. The total number of list insertions is at most proportional to τ. It remains to compute the time needed to create $cheap_u$. But we know that insertion in a F-heap takes constant time and there are N_C cooccurrences to insert. Therefore the whole algorithm runs in $\mathcal{O}(s^2 N \log \frac{N}{n})$ time.

Memory requirements. The memory requirements for the temporary heaps $cheap_u$ are equal to N_C. The space required by the final lists $V_c^\pm(u)$ is no more than proportional to the total number of cooccurrences, namely $\mathcal{O}(s^2 N)$. It remains to bound the space taken up by the vectors of pointers to the lists. This space is no larger than

$$\sigma_1 \;=\; \sum_{u \in V} 2c_{max}(u).$$

The total number of cooccurrences N_C is

$$\sum_{u \in V} \sum_{c=1}^{\infty} c[\text{length}(V_c^+(u)) + \text{length}(V_c^-(u))] \;=\; N_C \;\leq\; s^2 N/2 \tag{5.17}$$

Under this constraint, σ_1 is maximized when $\text{length}(V_c^\pm(u)) = 0$ for all but one of the lists with $c = c_{max}(u)$ for which it is equal to 1. Then the l.h.s. of (5.17) becomes

$$\sum_{u \in V} c_{max}(u).$$

It follows that $\sigma_1 \leq s^2 N$. Thus the total space required for this algorithm is $\mathcal{O}(s^2 N)$.

5.3.3 Putting it all together: the aCL-I algorithm and its data structures

So far, we have an efficient method of partially sorting the mutual informations of all the candidate edges. What we aim for is to create a mechanism that will output the edges uv in the decreasing order of their mutual information. We shall set up this mechanism in the form of an F-heap called *vheap* that contains an element for each $u \in V$, represented

by the edge with the highest mutual information among the edges outgoing from u. The maximum over this set will obviously be the maximum over the mutual informations of all the possible edges not yet eliminated. The record in *vheap* is of the form (c, u, v, I_{uv}), with $v \succ u$ and I_{uv} being the key used for sorting. Once the maximum is extracted, the used edge has to be replaced by the next weightiest edge in u's lists.

With this in place the Kruskal algorithm can by used to construct the desired spanning tree. The outline of the algorithms is

Algorithm **aCL-I** - outline

$n_e = 0$ // the number of tree edges
$E = \Phi$
while $n_e < n - 1$
  (uv) = extract max(*vheap*)
  if (uv) does not create a cycle with edges already in E
    add (uv) to E
    n_e++
  get a new (uv') and insert it in *vheap*
Output E

It remains to show how to efficiently construct a suitable *vheap*. This is not hard, once we have the lists $V_c^{\pm}(u)$, $\overline{V_0}(u)$ and *vlist* the sorted list of all variables. It suffices to take the first element of each list, let it be called $v_c^{\pm}$, $c > 0$, compute the respective mutual information I_{uv} and insert the triple $(c, v_c^{\pm}, I_{uv})$ in a F-heap $iheap_u$. How to extract the first element of the implicitly represented $V_0(u)$ will be discussed later. Extracting the maximum of $iheap(u)$ will provide us with the desired maximum over all edges originating in u. The quadruple (c, u, v, I_{uv}) is inserted in *vheap* and c is also used to replace the eliminated edge with one from the same list.

It remains to show how to handle the variables that do not cooccur with u. For this we maintain a pointer p_u into *vlist*. Initially p_u points to the successor of u in *vlist*. We compare p with the first element of $\overline{V_0}(u)$ and if they are equal we increment p by one and delete the head of $\overline{V_0}(u)$ recursively, until we find a v in *vlist* that does not cooccur with u. For this v we compute the mutual information and insert the triple $(0, v, I_{uv})$ in *ilist*. To get the next variable not cooccurring with u we increment p and repeat recursively the above comparison procedure with the head of $\overline{V_0}(u)$. Now we can summarize the algorithm as

Algorithm **aCL-I**

Input variable set V of size n
    dataset $\mathcal{D} = \{xlist^i,\ i = 1, \ldots N\}$
1. compute N_v for $v \in V$
  create *vlist*, list of variables in V sorted by decreasing N_v
2. **ListByCooccurrence** // partial sort the mutual informations
3. create *vheap*
  for $u \in vlist$
    create $iheap_u$
    (c, v, I_{uv}) = extract max with replacement from $iheap_u$

insert (c, u, v, I_{uv}) in *vheap*

4. E = **KruskalMST**(*vheap*) storing the $c = N_{uv}$ values for the edges added to E
5. for $(uv) \in E$
   compute the probability table T_{uv} using N_u, N_v, N_{uv} and N.

Output T

5.3.4 Time and storage requirements

Running time In the first step we have to compute the variables' frequencies N_v. This can be done by scanning trough the data points and by increasing the corresponding N_v each time v is found in $xlist^i$ for $i = 1, \ldots N$. This procedure will take at most sN operations. Adding in the time to initialize all N_v ($\mathcal{O}(n)$) and the time to sort the N_v values ($\mathcal{O}(n \log n)$) gives an upper bound on the running time for step 1 of the **aCL** algorithm of

$$\mathcal{O}(n \log n + sN)$$

The running time of the second step is already estimated to

$$\mathcal{O}(s^2 N \log \frac{N}{n})$$

Step 3 takes one mutual information computation for the head of each $V_c^{\pm}(u)$ list, plus $n \log C_{max}$ operations to extract the maximum from each $iheap_u$ and insert it into *vheap* (the reader is reminded that insertion in an F-heap is constant time). By C_{max} we denote the maximum over u of the number of non-empty $V_c^{\pm}(u)$ lists associated to a variable $u \in V$. The total number of lists is denoted by N_L. With these notations the total time for this step is

$$\mathcal{O}(N_L + n \log C_{max})$$

Step 4 is the Kruskal algorithm. For each edge extracted from *vheap* another one has to be inserted. Extraction takes $\mathcal{O}(\log n)$ because the size of *vheap* is always $\leq n$; a new insertion involves an extraction from an *iheap*, done in $\mathcal{O}(\log C_{max})$ and one computation of a mutual information, taking constant time. Checking that the current edge (uv) does not create a cycle and adding it to E can be done in constant time. Thus, if we denote by n_K the total number of edges examined by the Kruskal algorithm this step takes a number of operations of the order

$$\mathcal{O}(n_K \log nC_{max})$$

The last step computes $n - 1$ probability tables, each of them taking constant time. Thus, its running time is

$$\mathcal{O}(n)$$

Adding up these five terms we obtain the upper bound for the running time of the aCL algorithm as

$$\mathcal{O}(n \log n + s^2 N \log \frac{N}{n} + N_L + n \log C_{max} + n_K \log(nC_{max})) \tag{5.18}$$

Let us now further refine this result, by bounding N_L and C_{max}. Appendix 5.9 computes an upper bound on the number of lists N_L. If our third assumption holds (N/n bounded

away from 0 and ∞) then this bound reduces to $\mathcal{O}(sn)$. For C_{max} we shall use the obvious upper bound

$$C_{max} \leq N_C.$$

Notice that since C_{max} influences the running time only by its logarithm, any bound that is polynomial in s, N and n will do. Now the expression of the running time for the aCL algorithm becomes

$$\mathcal{O}(n \log n + s^2 N \log \frac{N}{n} + sn + n \log s^2 N + n_K \log(s^2 nN)) \tag{5.19}$$

which further simplifies to

$$\mathcal{O}(n \log n + s^2 N \log \frac{N}{n} + sn + n_K \log(s^2 nN)) = \tilde{\mathcal{O}}(n + s^2 N + sn + n_K) \tag{5.20}$$

This bound, ignoring the logarithmic factors, is a polynomial of degree 1 in the three variables n, N and n_K. However, we know that n_K the total number of edges inspected by Kruskal's algorithm has the range

$$n - 1 \leq n_K \leq \frac{n(n-1)}{2}. \tag{5.21}$$

Hence, in the worst case the above algorithm is quadratic in n. However, there are reasons to believe that in practice the dependence of n_K on n is subquadratic. Random graph theory suggests that if the distribution of the weight values is the same for all edges , then Kruskal's algorithm should take a number of steps proportional to $n \log n$ [69]. This result is sustained by experiments we have conducted: we ran Kruskal algorithm on sets of random weights over domains of dimension up to $n = 3000$. For each n, 1000 runs were performed. Figure 5-2 shows the average and maximum n_K plotted versus $n \log n$. The curves display a close to linear dependence.

Memory requirements To store data and results we need: $\mathcal{O}(sN)$ for the dataset in the bipartite graph representation, $\mathcal{O}(n)$ to store the variables and another $\mathcal{O}(n)$ to store the resulting tree structure and parametrization.

The additional storage required by the algorithm includes $\mathcal{O}(n)$ for *vlist*, then $\mathcal{O}(s^2 N)$ for all the lists created in step 2 of the algorithm. In step 3, *vheap* is created taking up $\mathcal{O}(n)$ memory. The space occupied by all the *ilist*s is proportional to the number of their elements, namely N_L. As we have already seen, this number is $\mathcal{O}(sn)$.

The last two steps do not use auxiliary storage so that the total space used by the algorithm is

$$\mathcal{O}(s^2 N + n + N_L) \tag{5.22}$$

or, using the bound in appendix 5.9

$$\mathcal{O}(s^2 N + sn) \tag{5.23}$$

5.3.5 The aCL-II algorithm

When the data are sparse enough, one can simplify the **aCL-I** algorithm. The $V_c^{\pm}(u)$ lists for $c > 0$ can be replaced by a single list C_u of all the variables cooccurring with u. The difference between C_u and the list $\overline{V}_0(u)$ introduced previously is that the records in C_u contain as an additional field the mutual information I_{uv} and are sorted thereby. The list

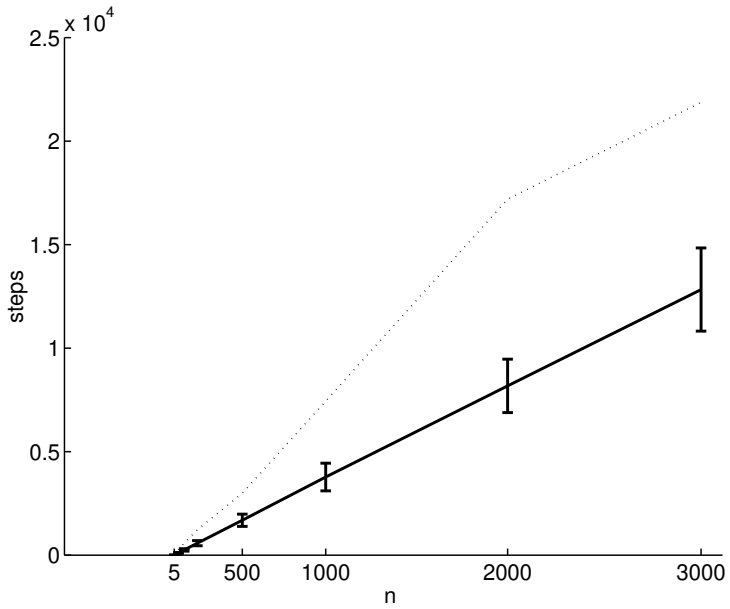


Figure 5-2: The mean (full line), standard deviation and maximum (dotted line) of Kruskal algorithm steps n_K over 1000 runs plotted against $n \log n$. n ranges from 5 to 3000. The edge weights were sampled from a uniform distribution.

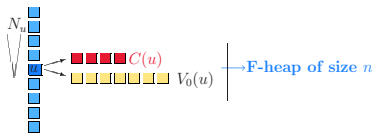


Figure 5-3: The **aCL-II** algorithm: the data structure that supplies the next weightiest candidate edge. Vertically to the left are the variables, sorted by decreasing N_u. For a given u, there are two lists: C_u, the list of variables $v \succ u$, sorted in decreasing order of I_{uv} and (the virtual list) $V_0(u)$ sorted by decreasing N_v. The maximum of the two first elements of these lists is that is inserted into an F-heap.The overal maximum of I_{uv} can then be extracted as the maximum of the F-heap.

$\overline{V}_0(u)$ preserves its function of supplying (together with *vlist*) the variables not cooccurring with u in decreasing order of their mutual information with u.

This implies computing all $I_{uv},\ v \in C_u,\ u \in V$ ahead of time, but they are no more than $N_C < 1/2s^2N$. The maximum weight edge outgoing from u is obtained by comparing the two mutual informations corresponding to the current heads of lists C_u and $V_0(u)$. The data structure used is shown schematically in figure 5-3 and the algorithm is described below.

Algorithm **aCL-II**

Input variable set V of size n
dataset $\mathcal{D} = \{xlist^i,\ i = 1, \ldots N\}$
1. compute N_v for $v \in V$
create *vlist*, list of variables in V sorted by decreasing N_v
2. construct lists $\overline{V}_0(u),\ u \in V$
compute mutual informations $I_{uv},\ v \in \overline{V}_0(u),\ u \in V$.
create and sort $C_u,\ u \in V$
3. create *vheap*
for $u \in vlist$

$$v = \underset{\text{head}C_u, \text{head}V_0(u)}{\text{argmax}} I_{uv}$$

insert (c, u, v, I_{uv}) in *vheap*

4. $E =$ **KruskalMST**(*vheap*) storing the $c = N_{uv}$ values for the edges added to E
5. for $(uv) \in E$
compute the probability table T_{uv} using N_u, N_v, N_{uv} and N.
Output T

5.3.6 Time and memory requirements for aCL-II

Steps 1, 4 and 5 of the above algorithm copy the corresponding steps of **aCL-I**. Hence we analyze only steps 2 and 3.

Constructing $\overline{V}_0$ has been discussed previously and found to take $\mathcal{O}(s^2N \log \frac{N}{n})$ time and $\mathcal{O}(s^2N)$ memory. Then we need to compute no more than $\mathcal{O}(s^2N)$ mutual informations and to sort the resulting lists. The worst case for the latter is when all the lists are equal. Then, the total time is bounded by

$$n(\frac{s^2N}{n} \log \frac{s^2N}{n}) \ = \ s^2N \log \frac{s^2N}{n}$$

a result encountered before.

Constructing *vheap* takes $\mathcal{O}(n)$ units of time and memory. Each extraction from it is $\mathcal{O}(\log n)$. All the extractions from the virtually represented $V_0(u)$ take no more than $n_K + N_C$ time steps since there are at most N_C elements that have to be skipped. n_K is again the number of steps taken by Kruskal's algorithm. Therefore, the running time of the **aCL-II** algorithm is

$$\mathcal{O}(n \log n + sn + s^2N \log(s^2nN) + n_K \log n) \tag{5.24}$$

The additional memory requirement are bounded by

$$\mathcal{O}(n + s^2N). \tag{5.25}$$

Comparing (5.24) above to (5.20) it appears that it is not possible to determine which algorithm is asymptotically faster from the bounds alone. The algorithms have large parts that are identical and the factor $\log s^2 nN$ is a loose overestimate in both formulas; thus, it may be that the running times of the two algorithms are close in reality. But more important is the fact that by the **aCL-II** algorithm we have managed to relax the one of the assumptions made in the previous section: the latter algorithm does not rely on integer counts. This difference will become essential in the context of the EM algorithm.

5.4 Generalization to discrete variables of arbitrary arity

This section will show how the **TreeLearn** algorithm can be accelerated in discrete domains where the variables can take more than two values. The algorithm that we are going to develop is a simple extension of the **aCL-II** algorithm. Therefore, we shall present only the basic ideas of the extension and the modifications to the **aCL-II** algorithm that they imply.

The **aCL** algorithms introduced previously were exploiting the data sparsity. If they are to be generalized, it is first necessary to extend the notion of sparsity itself. Thus, in the forthcoming we shall assume that for each variable exists a special value that appears with higher frequency than all the other values. This value will be denoted by 0, without loss of generality. For example, in a medical domain, the value 0 for a variable would represent the "normal" value, whereas the abnormal values of each variable would be designated by non-zero values. Similarly, in a diagnostic system, 0 will indicate a normal or correct value, whereas the non-zero values would be assigned to the different failure modes associated with the respective variable. An occurence for variable v will be the event $v \neq 0$ and a cooccurrence of u and v means that u and v are both non-zero in the same data point. Therefore, we define $|x|$ as the number of non-zero values in observation x

$$|x| \;=\; n - \sum_{v \in V} \delta_{x_v} \tag{5.26}$$

The sparsity s will be the maximum of $|x|$ over the data set, as before.

From the above it can be anticipated that the high frequency of the 0 values will help accelerate the tree learning algorithm. As before, we shall represent only the occurrences explicitly, creating thereby a compact and efficient data structure. Moreover, we shall demonstrate a way to presort mutual informations for non-cooccurring variables similar to that used by the **aCL-II** algorithm.

5.4.1 Computing cooccurrences

Following the previously introduced idea of not representing explicitly 0 values, each data point x will be replaced by the list $xlist$ of the variables that occur in it. However, since there can be more than one non-zero value, the list has also to store this value along with the variable index. Thus

$$xlist \;=\; \text{list}\{(v, x_v), v \in V, x_v \neq 0\}.$$

Similarly, a cooccurrence will be represented by the quadruple $(u, x_u, v, x_v),\ x_u, x_v \neq 0$. Counting and storing cooccurrences can be done in the same time as before and with a proportionally larger amount on memory, required by the additional need to store the (non-zero) variable values.

Instead of one cooccurrence count N_{uv} we shall now have a two-way contingency table N_{uv}^{ij}. Each N_{uv}^{ij} represents the number of data points where $u = i, v = j$, $i, j \neq 0$. This contingency table, together with the marginal counts N_v^j (defined as the number of data points where $v = j,\ j \neq 0$) and with N completely determine the joint distribution of u and v (and consequently the mutual information I_{uv}). Constructing the cooccurrence contingency tables multiplies the storage requirements of this step of the algorithm by $\mathcal{O}(r_{MAX}^2)$ but does not change the running time.

5.4.2 Presorting mutual informations

As in subsection 5.3.1, our goal is to presort the mutual informations I_{uv} for all $v \succ u$ that do not cooccur with u. We shall show that this can be done exactly as before. The derivations below will be clearer if they are made in terms of probabilities; therefore, we shall use the notations:

$$P_v(i) = \frac{N_v^i}{N}, \quad i \neq 0 \tag{5.27}$$

$$P_{v0} \equiv P_v(0) = 1 - \sum_{i \neq 0} P_v(i) \tag{5.28}$$

The above quantities represent the (empirical) probabilities of v taking value $i \neq 0$ and 0 respectively. Entropies will be denoted by H.

A "chain rule" expression for the entropy of a discrete variable. The entropy H_v of any multivalued discrete variable v can be decomposed in the following way:

$$\begin{aligned}
H_v &= -P_{v0} \log P_{v0} - \sum_{i \neq 0} P_v(i) \log P_v(i) \\
&= -P_{v0} \log P_{v0} - (1 - P_{v0}) \sum_{i \neq 0} \frac{P_v(i)}{(1 - P_{v0})} \left[\log \frac{P_v(i)}{(1 - P_{v0})} + \log(1 - P_{v0}) \right] \\
&= \underbrace{-P_{v0} \log P_{v0} - (1 - P_{v0}) \log(1 - P_{v0})}_{H_{v0}} - (1 - P_{v0}) \underbrace{\sum_{i \neq 0} \frac{P_v(i)}{(1 - P_{v0})} \log \frac{P_v(i)}{(1 - P_{v0})}}_{-H_{\overline{v}}} \\
&= H_{v0} + (1 - P_{v0}) H_{\overline{v}}
\end{aligned} \tag{5.29}$$

This decomposition represents a sampling model where first we choose whether v will be zero or not, and then, if the outcome is "non-zero" we choose one of the remaining values by sampling from the distribution $P_{v|v\neq 0}(i) = \frac{P_v(i)}{1-P_{v0}}$. H_{v0} is the uncertainty associated with the first choice, whereas $H_{\overline{v}} \equiv H_{v|v\neq 0}$ is the entropy of the outcome of the second one. The advantage of this decomposition for our purpose is that it separates the 0 outcome from the others and "encapsulates" the uncertainty of the latters in the number $H_{\overline{v}}$.

The mutual information of two non-cooccurring variables We shall use the above fact to find an expression of the mutual information I_{uv} of two non cooccurring variables u, v in terms of P_{u0}, P_{v0} and $H_{\overline{u}}$ only.

$$I_{uv} = H_u - H_{u|v} \tag{5.30}$$

The second term, the conditional entropy of u given v is

$$H_{u|v} \;=\; P_{v0}H_{u|v=0} + \sum_{j\neq 0} P_v(j)\,\underbrace{H_{u|v=j}}_{0} \tag{5.31}$$

The last term in the above equation is 0 because, for any non-zero value of v, the condition $N_{uv} = 0$ implies that u has to be 0. Let us now develop $H_{u|v=0}$ using the decomposition in equation (5.29).

$$H_{u|v=0} \;=\; H_{u0|v=0} + (1 - P_{u=0|v=0})H_{u|u\neq 0,v=0} \tag{5.32}$$

Because u and v are never non-zero in the same time, all non-zero values of u are paired with zero values of v. Hence, knowing that $v = 0$ brings no additional information once we know that $u \neq 0$. In probabilistic terms: $Pr[u = i|u \neq 0, v = 0] = Pr[u = i|u \neq 0]$ and

$$H_{u|u\neq 0,v=0} \;=\; H_{\overline{u}} \tag{5.33}$$

The term $H_{u=0|v=0}$ is the entropy of a binary variable whose probability is $Pr[u = 0|v = 0]$. This probability equals

$$\begin{aligned} Pr[u = 0|v = 0] \;&=\; 1 - \sum_{i\neq 0} P_{u|v=0}(i) \\ &=\; 1 - \sum_{i\neq 0} \frac{P_u(i)}{P_{v0}} \\ &=\; 1 - \frac{1 - P_{u0}}{1 - P_{v0}} \end{aligned} \tag{5.34}$$

Note that in order to obtain a non-negative probability in the above equation one needs

$$1 - P_{u0} \;\leq\; P_{v0}$$

a condition that is always satisfied if u and v do not cooccur. Replacing the previous three equations in the formula of the mutual information, we get

$$I_{uv} \;=\; P_{u0}\log P_{u0} - P_{v0}\log P_{v0} + (P_{u0} + P_{v0} - 1)\log(P_{u0} + P_{v0} - 1) \tag{5.35}$$

an expression that, remarkably, depends only on P_{u0} and P_{v0}. Taking its partial derivative with respect to P_{v0} yields

$$\frac{\partial I_{uv}}{\partial P_{v0}} \;=\; \log\frac{P_{v0} + P_{u0} - 1}{P_{v0}} \;<\; 0 \tag{5.36}$$

a value that is always negative, independently of P_{v0}. This shows the mutual information increases monotonically with the "occurrence frequency" of v given by $1 - P_{v0}$. Note also that the above expression for the derivative is a rewrite of the result obtained for binary variables in (5.12) in the case $N_{uv} = 0$.

We have shown that the **aCL-II** algorithm can be extended to variables taking more than two values by making only one (minor) modification: the replacement of the scalar counts N_v and N_{uv} by the vectors N_v^j, $j \neq 0$ and, respectively, the contingency tables N_{uv}^{ij}, $i, j \neq 0$.

5.5 Using the aCL algorithms with EM

So far it has been shown how to accelerate the CL algorithm under the assumption that the target probability distribution P is defined in terms of integer counts $N,\ N_v,\ N_{uv},\ u, v \in V$. This is true when fitting one tree distribution to an observed data set or in the case of classification with TANB models where the data points are partitioned according to the observed class variable. But an important application of the CL algorithm are mixtures of trees, and in the case of learning mixtures by the EM algorithm the counts defining P for each of the component trees are not integer.

Recall that each E step of the EM algorithm computes the posterior probability of each mixture component k of having generated data point x^i. This is $\gamma_k(i)$ defined in equation (3.15). The values γ have the effect of "weighting" the points in the dataset $\mathcal{D}$ with values in $[0, 1]$, different for each of the k trees in the mixture. The counts N_v^k and N_{uv}^k corresponding to tree k are defined in terms of the γ values as

$$N^k \equiv \Gamma_k = \sum_i \gamma_k(i) \tag{5.37}$$

$$N_v^k = \sum_{i:x_v^i=1} \gamma_k(i) \tag{5.38}$$

$$N_{uv}^k = \sum_{i:x_v^i=1 \wedge x_u^i=1} \gamma_k(i). \tag{5.39}$$

These counts are in general not integer numbers. Therefore, for learning mixtures of trees the **aCL-II** algorithm is recommended.

Now we shall examine steps 1 and 2 of the **aCL-II** algorithm and show how to modify them in order to handle weighted data.

First, remark that step 1 will have to sort the variables m times, producing m different *vlists*, one for each of the components. Computing the N_v^k values is done similarly to the previous section; the only modification is that for each occurrence of v in a data point one adds $\gamma_k(i)$ to N_v^k instead of incrementing a counter. Remark that no operations are done for pairs of variables that do not cooccur in the original data set, preserving thereby the algorithm's guarantee of efficiency.

For step 2, a similar approach is taken. At the time of inserting in $cheap_u^k$ one must store not only v but also $\gamma_k(i)$. When the heap is emptied, the current "count" c sums all the $\gamma_k(i)$ values corresponding to the given v.

Note also that one can use the fact that the data are sparse to accelerate the E step as well. One can precompute the "most frequent" likelihood $T_0^k \triangleq T^k(0, \ldots, 0)$ for each k. Then, if v is 1 for point x^i one multiplies T_0^k by the ratio $\frac{T^k_{v|\mathrm{pa}(v)}(1|\mathrm{pa}(v))}{T^k_{v|\mathrm{pa}(v)}(0|\mathrm{pa}(v))}$ also precomputed. This way the E step will run in $\mathcal{O}(msN)$ time instead of the previously computed $\mathcal{O}(mnN)$.

5.6 Decomposable priors and the aCL algorithm

All of the above assumes that the tree or mixture is to be fit to the data in the maximum likelihood framework. This section will study the possibility of using priors in conjunction with the aCL algorithm. The classes of priors that we shall be concerned with are the priors

discussed in chapter 4. We shall first examine priors on the tree's structure having the form

$$P(E) \propto \exp\left(-\sum_{uv\in E}\beta_{uv}\right)$$

As shown in section 4.2.1, this prior translates into a penalty on the weight of edge (uv) as seen by a MWST algorithm

$$W_{uv} \leftarrow I_{uv} - \frac{\beta_{uv}}{N}$$

It is easily seen that for general β_{uv} values such a modification cannot be handled by the **aCL** algorithms. Indeed, this would affect the ordering of the edges outgoing from u in a way that is inpredictable from the counts $N_v,\ N_{uv}$. However, if β_{uv} is constant for all pairs $u, v \in V$, then the ordering of the edges is not affected. All we need to do to use an **aCL** algorithm with a constant penalty β is to compare the I_{uv} of each edge, at the moment it is extracted from *vheap* by Kruskal's algorithm, with the quantity $\frac{\beta}{N}$. The algorithm stops as soon as one edge is found whose mutual information is smaller than the penalty $\frac{\beta}{N}$ and proceeds as before otherwise. Of course, in the context of the EM algorithm, N is replaced by $N^k \equiv \Gamma_k$ and β can be different for each component of the mixture. Remark that if all the variables are binary (or have the same number of values) an MDL type edge penalty translates into a constant β_{uv}.

Regarding Dirichlet priors on the tree's parameters, it has already been shown that they can be represented as a set of fictitious counts $N'_{uv},\ u, v \in V$ and that maximizing the posterior probability of the tree is equivalent to minimizing the KL divergence

$$KL(\tilde{P}\,||\,T)$$

with $\tilde{P}$ a mixture between the empirical distribution P and the fictitious distribution P' defined by N'_{uv}. This challenges two of the assumptions that the accelerated algorithms are based upon. First, the counts N_v, N_{uv} cease to be integers. This affects only the **aCL-I** algorithm.

Second, both algorithms rely on the fact that most of the N_{uv} values are 0. If the counts N'_{uv} violate this assumption then the **aCL** algorithms become inefficient. In particular, the **aCL-II** algorithm degrades to a standard **TreeLearn** algorithm. Having many or all $N'_{uv} > 0$ is not a rare case. In particular, it is a characteristic of the non-informative priors that aim at smoothing the model parameters. This means that smoothing priors and aCL algorithms will in general not be compatible.

Somehow suprisingly, the uniform prior (4.35) constitutes an exception: for this prior, in the case of binary variables, all the fictitious cooccurrence counts are equal to $N'/4$. Using this fact, one can prove that for very small and for large values of N' (> 8) the order of the mutual informations in $V_0(u)$ is preserved and respectively reversed. This fact allows us to run the **aCL-II** algorithm efficiently after only slight modification.

5.7 Experiments

The following experiments compare the (hypothesized) gain in speed of the accelerated **TreeLearn** algorithm w.r.t the traditional version presented in chapter 2 under controlled conditions on artificial data.

The binary domain had a dimensionality n varying from 50 to 1000. Each data point had a fixed number s of variables being on. The sparsity s took the values 5, 10, 15

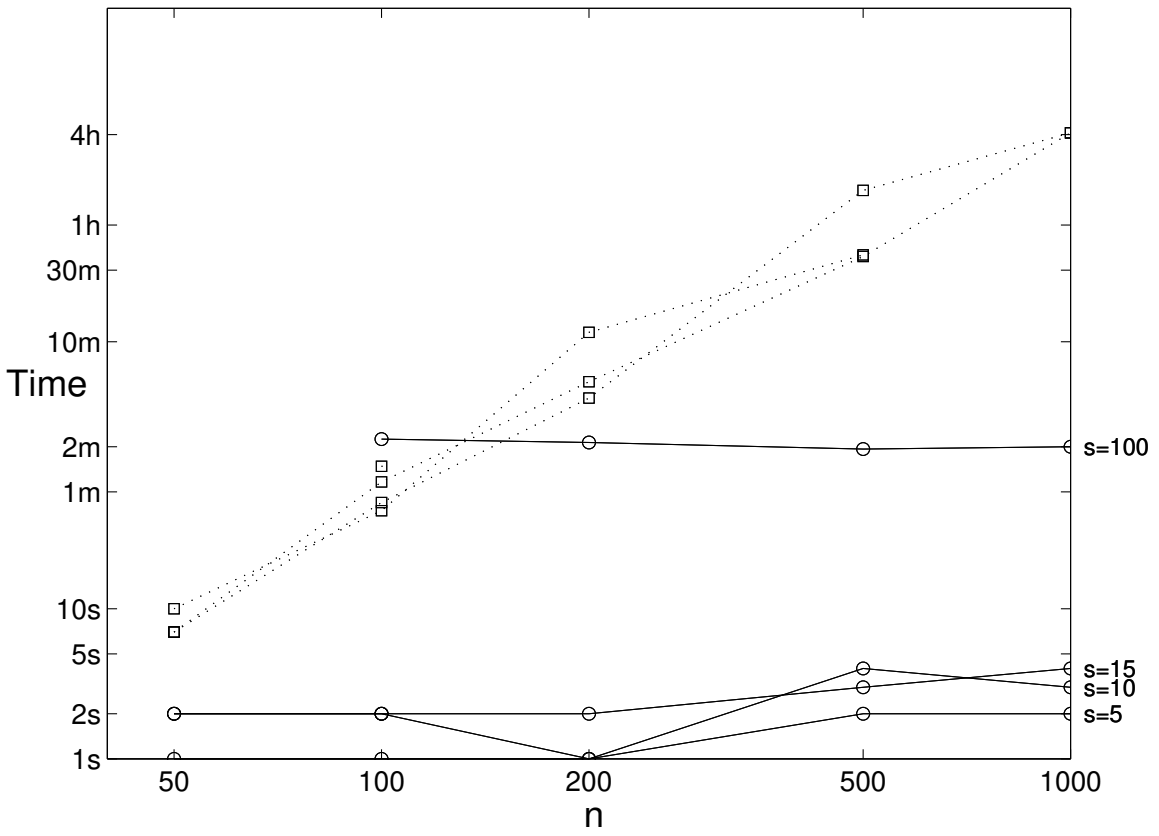


Figure 5-4: Real running time for the accelerated (full line) and traditional (dotted line) **TreeLearn** algorithm versus number of vertices n for different values of the sparsity s.

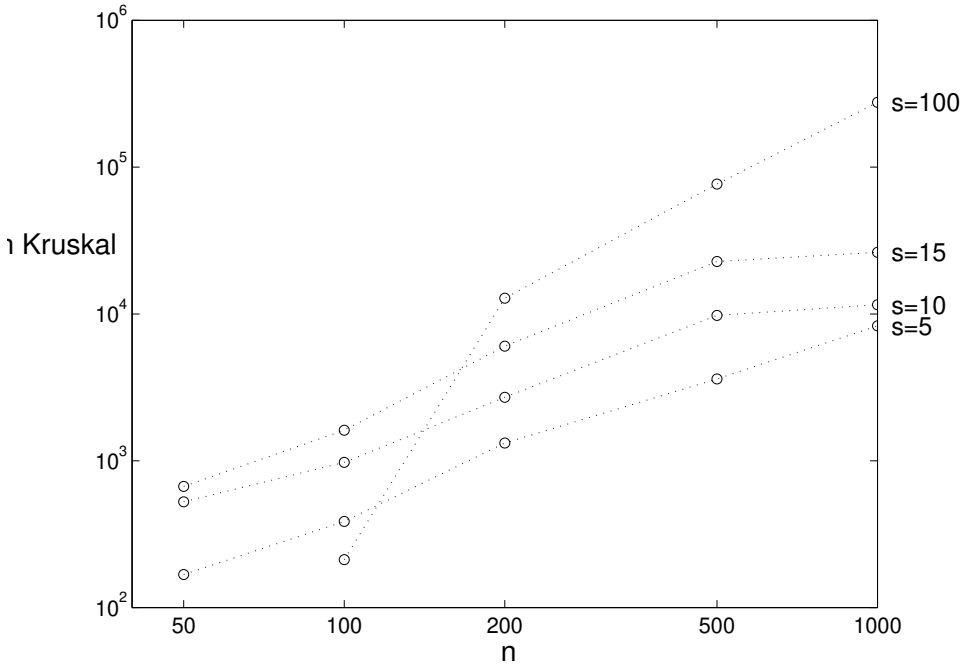


Figure 5-5: Number of steps of the Kruskal algorithm n_K versus domain size n measured for the **aCL-II** algorithm for different values of s

and 100. The small values were chosen to gauge the advantage of the accelerated algorithm under extremely favorable conditions. The larger value will help us see how the performance degrades under more realistic circumstances. Each data point (representing a list of variables being on) was generated as follows: the first variable was picked randomly from the range $1, \ldots n$; the subsequent points were sampled from a random walk with a random step size between -4 and 4. For each pair n, s a set of 10,000 points was generated.

Each data set was used by both **TreeLearn** and **aCL-II** to fit one tree distribution and the running times were recorded and plotted in figure 5-4. The improvements over the traditional version for sparse data are spectacular: learning a tree over 1000 variables from 10,000 data points takes 4 hours by the traditional algorithm and only 4 seconds by the accelerated version when the data are sparse ($s = 15$). For $s = 100$ the **aCL-II** algorithm takes 2 minutes to complete, improving on the traditional algorithm by a factor of "only" 123.

What is also noticeable is that the running time of the accelerated algorithm seems to be almost independent of the dimension of the domain. On the other side, the number of steps n_K (figure 5-5) grows with n. This observation implies that the bulk of the computation lies with the steps preceding the Kruskal algorithm proper. Namely, that it is in computing cooccurrences and organizing the data that most of the time is spent. This observation would deserve further investigation for large real-world applications.

Figure 5-4 also confirms that the running time of the **TreeLearn** algorithm grows quadratically with n and is independent of s.

5.8 Concluding remarks

This chapter has presented a way of taking advantages of sparsity in the data to accelerate the tree learning algorithm. The ideas introduced have been developed into the two algorithms presented without exhausting the number of possible variants.

The methods achieve their performance by taking advantage of characteristics of the data (sparsity) and of the problem (the weights represent mutual informations) that are external to the Maximum Weight Spanning Tree algorithm proper. Here the algorithms have been evaluated considering that the data sparsity s is a constant. If this is not the case, however, the actual complexity of the algorithms can be computed easily from the bounds provided here, since the dependence on s is made explicit in each of them.

Moreover, it has been shown empirically that a very significant part of the algorithms' running time is spent in computing cooccurrences. This prompts future work on applying trees and mixtures of trees to high-dimensional tasks to focus on methods for structuring the data and for computing or approximating marginal distributions and mutual informations in specific domains.

The **aCL-II** algorithm relies heavily on the fact that most cooccurrences between variables are zero. Given this fact, it can smoothly handle real values for the non-zero cooccurrence counts and even some classes of priors. The **aCL-I** algorithm is more restrictive in the sense that it requires integer counts and from the present perspective, it is at clear disadvantage w.r.t **aCL-II** both in simplicity and in versatility.

5.9 Appendix: Bounding the number of lists N_L

Let us denote the total number of nonempty lists $V_c^{\pm}(u), c > 0, u \in V$ by N_L. It is obvious N_L is maximized when all list lengths are 1 for $c \leq C$, 0 for $c > C+1$ and 0 or 1 for $c = C$, where C is a constant to be determined. In this case, the number of lists can be expressed successively as

$$\begin{aligned} N_L &= n\sum_{c=1}^{C} 2c + \mathcal{O}(n)(C+1) \\ &= 2n\frac{C(C+1)}{2} + \mathcal{O}(n)(C+1) \\ &= n(C+C^2) + \mathcal{O}(n)(C+1) \end{aligned}$$

Therefore

$$n(C+C^2) \;\leq\; N_C \;\leq\; s^2N/2 \tag{5.40}$$

Solving for C we obtain

$$C \;\leq\; -\frac{1}{2} + \sqrt{\frac{s^2N}{2n} + 1} \tag{5.41}$$

which amounts to

$$\begin{aligned} N_L &= nC + \mathcal{O}(n) \\ &= -\frac{n}{2} + sn\sqrt{\frac{N}{n}}\sqrt{\frac{1}{2} + \frac{n}{s^2N}} + \mathcal{O}(n) \\ &= \mathcal{O}\left(sn\sqrt{\frac{N}{n}}\sqrt{\frac{1}{2} + \frac{n}{s^2N}}\right) \end{aligned} \tag{5.42}$$

Under the assumption that $\frac{N}{n}$ is bounded above and below the equation simplifies to

$$N_L \;=\; \mathcal{O}(sn) \tag{5.43}$$

Chapter 6

An Approach to Hidden variable discovery

Atâtea clăile de fire stângi!
Găsi-vor oare gest inchis să le rezume,
Să nege, dreaptă, linia ce frângi–
Ochi în virgin triunghi tăiat spre lume?
Ion Barbu
–Grup
So many heaps of tangled straw!
Will they find clean gesture to comprise them,
To straighten them into the golden beam
Shone by the divine triangle eye?

This chapter presents a way of using mixtures of trees and the MixTreeS learning algorithm for discovering structure in a particular class of graphical models with hidden variables. In the process of structure discovery model selection and model validation are important components. With this goal in mind, I introduce and a novel method for validating independencies in graphical models with hidden variables whose scope is much broader than the class of models investigated here.

6.1 Structure learning paradigms

Structure discovery has been one of the most challenging and most fascinating problems in graphical models ever since the beginning of the field. This is to no surprise given that the task of identifying dependencies and independencies in observations is at the core of scientific discovery as an intellectual process. Work in this field can be loosely categorized into three areas of research.

The first category of structure learning methods (and the oldest also) is based on conditional independence tests. [66] introduce an algorithm that constructs a Bayesian network that is consistent with a given list of independence statements. A similar result exists for decomposable models: [16] described a method to learn decomposable models or decomposable approximations to Markov network models using an oracle for conditional independence queries. The algorithm is exponential in the size of the largest separator. The two works are impressive in their success in inverting the relationship between a graphical model structure and its list of independencies. On the other hand, the strength of the results reflects the strength of the underlying assumptions. In other words, the methods assume as input an information that is at least as hard to obtain as the graph itself. Testing the conditional

independence $A \perp B \mid C$ is exponential in $|A|.|B|.|C|$. The first method assumes that we know beforehand all independencies in a domain V (and implicitly all the pairs A, B that are *not* independent). But who is to give us such a list of independencies? Moreover, assuming that we were to undertake the intractable task of performing all the required independence tests, we would be faced with uncertain and perhaps contradictory information that none of the above methods are prepared to handle.

[53] presents an algorithm for finding a polytree structure from mutual information measurements when the underlying distribution is known to be a polytree. The same chapter presents a method for learning *star decomposable graphs*[1] the case of binary variables and ideal correlation measurements. These methods have the same drawback of requiring ideal information that cannot be obtained from data.

[6] introduces an algorithm for learning a minimal I-map of a distribution from data. The algorithm is based on evaluation of pairwise mutual informations between variables, and independence tests that require computational effort proportional to the size of the largest separator, but it is shown to work efficiently on sparse graphs. The implicit assumption that complete independence information can be obtained from pairwise independence tests - an assumption that will be examined later in this chapter. It is also assumed that there is sufficient data to ensure a correct answer to all the independence tests, an assumption that brings this paper in the same group with the ones requiring ideal information.

The second category of methods explicitly takes into account the existence of a finite sample of data points from which the structure has to be learned. The most influential is the Bayesian approach pioneered by [9] and developed by Heckerman, Geiger and Chickering in the now classical paper [33]. They state the problem of Bayesian learning of Bayesian belief networks from data, make explicit the assumptions underlying their approach and show how to calculate, for each network structure E the *score* $Pr[\mathcal{D}, E]$. The score is then used for model comparison. They also prove that optimizing the Bayesian score over networks structures is NP-hard when each node is allowed to have more than 1 parent. Therefore, both [9] and [33] use a local search: The search algorithms typically generate proposed additions or deletions of an edge from the graph, and these proposed moves are evaluated by computing (an approximation to) the marginal likelihood of the updated graph (exact calculations can be performed in some cases for graphs without hidden nodes). Local search is the most common approach to structure learning thus far. Methods based on non-Bayesian scores have been proposed by [45]. The cited works assume that data are complete and that there are no hidden variables. [60] develop a method for scoring equivalence classes of DAGs with hidden variables in the special case when the dependence relationships are jointly Gaussian, augmented by a heuristic search algorithm.

The third category of methods includes methods that also consider learning from data, but are explicitly aimed at finding dependencies. Some of the methods do not attempt to construct a complete graphical model, but to find as many dependencies or independencies as possible. [62] and [17] directly find dependencies in discrete databases by counting coincidences in subsamples. [34] constructs minimal Markov nets in a heuristic way. The first two methods are aiming at discovering structure in high-dimensional domains, where applying more sophisticated algorithms is computationally too expensive. This is a little explored area of research, but it is the area where the approach presented here most likely belongs.

[1] A star decomposable graph is a tree over the visible domain augmented with some hidden variables where each hidden variable has a degree at least 3.

The method to be developed in the forthcoming is fundamentally a method of learning from data. It belongs to the third category of the above taxonomy by the fact that it is mainly concerned with finding (a certain class of) dependency structures rather than a fully determined graphical model. But it relates to the second category by the fact that it generates multiple models and compares them by a score that will be further defined.

6.2 The problem of variable partitioning

The approach to structure learning presented in this thesis differs from all the methods mentioned above. Rather than adopting a local search heuristic based on single edge addition or deletion like the methods in the second category, or than looking for dependencies like the methods in the third category, it attempts to find *separators*.

A separator[2] is a variable or group of variables S that, when observed, separates $V \setminus S$ into two independent subsets A_1, A_2. If a separator exists and it is found, then learning the structure of the domain V reduces to learning the structures of $A_1 \bigcup S$ and $A_2 \bigcup S$. Since the number of possible structures grows superexponentially with n, by partitioning the domain the search space is reduced in a superexponential manner. Moreover, if more separators can be found in A_1, A_2, then one can proceed recursively, effectively implementing a divide-and-conquer structure learning algorithm. We call this approach *variable partitioning*.

If the separator S is included in V it is called a *visible* separatore. A visible separator of small size (practically 1 or 2 variables) can be found in a brute-force manner, by trying all possibilities. The number of such trials is $\mathcal{O}(n^{|S|}$ and each trial implies testing the conditional independence of two or more large sets, an exponential task by itself. As it will be shown later in this chapter (section 6.7.3), under certain assumptions the aforementioned independence tests can be reduced to tests on pairs of variables. This makes each trial quadratic in n.

This work focuses on a different situation: the case when the separator is a hidden variable. For this case the brute force approach is not sufficient and a different method needs to be devised. Much of the machinery developed for this case is applicable to visible separators; such that, in some sense, a visible separator is a simple case of hidden variable separator.

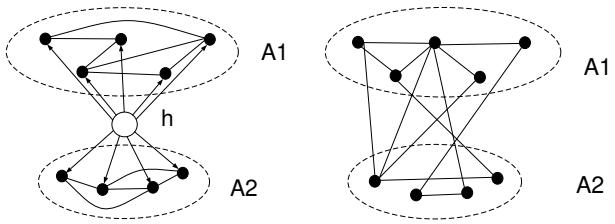
The hidden variable separator, or shortly hidden variable h, is assumed to partition V into several clusters of variables that are mutually independent conditioned on h. To maintain computational tractability, we assume a restrictive architecture for each cluster, in particular we assume that each cluster is a tree. We propose an algorithm whose goal it is to find a good set of trees and to find a good choice of tree structure within each cluster. Later we examine what happens when the true distribution of the data violates this restrictive assumption.

Let V denote as usual the set of variables of interest. Remember also that x_v is a particular value of V, x_A an assignment to the variables in the subset A of V and x is a shorthand for x_V. The variables in V will be referred to as being *visible*. We introduce an additional variable, a *hidden* (unobservable) variable denoted by h. This variable takes values ranging from 1 to m. The joint space $V \cup \{h\}$ is denoted by V^h. The following assumption about the joint distribution P over V^h is essential:

Assumption 1. The hidden variable h partitions V into p mutually disjoint subsets

[2]The term originates in the junction tree literature.

Figure 6-1: The H-model (a) and a graphical model of the same distribution marginalized over h (b).



$A_1, \ldots, A_p$, $\bigcup_j A_j = V$ that are conditionally independent given h. We write

$$A_j \perp A_{j'} \mid h \quad j, j' = 1, \ldots m \tag{6.1}$$

Consider the graphical model G corresponding to P (figure 6-1,a). Assumption 1 implies the graph topology $G = (V^h, E \cup E')$[3]. The edge set is the union of E', which contains a *directed* edge from the hidden variable h to each of the visible variables in V, and of E, which contains only undirected edges between pairs of visible variables. Moreover, the induced graph over V only has p connected components. Thus, in the graphical model language, conditional independence given h translates into h *separating* the respective variable clusters in G. We can view this model as a mixture of factorial models over the p hyper-variables $A_{1,\ldots p}$. Therefore, we call a model that satisfies Assumption 1 a *hyper mixture of factorials*, or in short an *H model*. H models are chain graphs.

Note that A_j being mutually independent given h does not imply that A_j are independent after marginalizing over the values of h. Graphically, if A_j are mutually separated (by h) in G, they are in general *not* separated in the graph $G^M = (V, E^M)$ corresponding to the marginal distribution of V

$$Q(x) \;=\; \sum_{k=1}^{m} P(x, h = k) \tag{6.2}$$

Let us compute the number of parameters of the models represented by P and Q under the simplifying assumption that $|v| = 2$ for all v, $p = 2$ and that P, $P_{A_j|h}$, $j = 1, 2$ are represented by probability tables. We have

$$\#pars(P) \;=\; m(2^{|A_1|} + 2^{|A_2|} - 2) \;\ll\; 2^n - 1 \;=\; \#pars(Q) \;\text{ for } m \ll 2^n/(2^{|A_1|} + 2^{|A_2|}) \tag{6.3}$$

Hence, if the sizes of A_1 and A_2 are close, introducing a hidden variable can result in exponential savings in terms of the number of parameters.

We formulate the *variable partitioning task* as follows: given a domain $V^h = \{h\} \bigcup_j A_j$ such that $A_j \perp A_{j'} | h \;\forall j, j'$ and a set of observations $\mathcal{D} = \{x^1, \ldots, x^N\}$ from $V = \bigcup_j A_j$, find the sets A_j, the number of values of h, the distribution P_h and the distribution $\gamma^i = P(h|x^i)$ over the values of the hidden variable for each of the observed instances x^i in $\mathcal{D}$.

Of course, the final goal is to construct a model P of the domain; to do this, once variable partitioning is completed, one has two choices: either to use the model P that

[3]More precisely, a graph that is consistent with Assumption 1 is equivalent to a structure of type G

our algorithm learns, or to construct by some other method (including a recursive call to our variable partitioning algorithm) a probabilistic model for each of the domains $A_j \cup h$. The sizes of the domains being smaller than n, this task should be considerably easier than constructing a model over V directly.

However, one should be aware of the important difficulties that this quest has to face. Searching over the space of all possible partitions of V (of size $\sim \frac{p^n}{p!}$) is a forbidding task. It has been shown that learning the structure of belief networks from data even in the absence of hidden variables is an intractable problem [33]. Second, the H model is a mixture model and we also know that learning mixtures in certain instances is NP-hard [4]. The above reasons show that all one can do is to provide heuristic procedures for coping with this large-scale but important search problem.

There are theoretical difficulties as well: as it has been shown, graphical models with hidden variables do not belong to the curved exponential family and thus the BIC/MDL criterion is not a consistent estimate of the marginal likelihood of the model structure, as it is for exponential family models [29]. Computing the Bayesian posterior marginal probabilities of the model structure represents an equally intractable computation.

It is also generally known that models with hidden variables are subject to non-identifiability problems. A simple example will illustrate this: Suppose that the observed distribution over the two binary variables a, b is uniform and that we know that this distribution is the marginal of a mixture distribution

$$Q_{ab} \;=\; \sum_{k=1,2} P_h(k) P_{ab|h=k}$$

The above is an underdetermined system of equations in the unknowns P_h, $P_{ab|h}$. Two of the possible solutions are

$$P_h^{(1)}(k) \;=\; 0.5 \;\; k = 1,2; \quad P_{ab|0}^{(1)} \;=\; \left\{ \begin{array}{ll} 0 & a = b \\ 0.5 & a \neq b \end{array} \right. \quad P_{ab|1}^{(1)} \;=\; \left\{ \begin{array}{ll} 0.5 & a = b \\ 0 & a \neq b \end{array} \right.$$

and

$$P_h^{(2)}(k) \;=\; 0.5 \;\; k = 1,2; \quad P_{ab|0}^{(2)} \;=\; \left\{ \begin{array}{ll} 0 & a = 0 \\ 0.5 & a = 1 \end{array} \right. \quad P_{ab|1}^{(2)} \;=\; \left\{ \begin{array}{ll} 0.5 & a = 0 \\ 0 & a = 1 \end{array} \right.$$

Both of the above distributions have a uniform marginal over a, b and so does any convex combination of $P^{(1)}, P^{(2)}$. Hence there is an infinity of mixture models that have the same marginal over the observed variables.

Non-identifiability can cause severe convergence problems in learning the model. For our task identifiability is even more important, since in its absence we are likely to find several hidden variable solutions (potentially an infinity!) that explain the data equally well. Identifiability of hidden variable models is an area of current research [29, 57, 61]. The existent results are only preliminaries to a general theory. However, in the case of models with binary variables only, [61] gives the following sufficient condition that is highly relevant to the present problem.

Theorem [61] An H-model over a domain V^h consisting of binary variables only is identifiable if either (a) or (b) hold:

(a) $p \geq 3$

(b) $p = 2$, $V = A_1 \bigcup A_2$ and $P_{A_1|h}, P_{A_2|h}$ are not fully connected graphical models.

6.3 The tree H model

In the following we present an algorithm for variable partitioning derived under additional assumptions about the model structure. We shall also assume for now that m and p are known.

We define a *tree H model* over $V^h = \{h\} \bigcup \left(\bigcup_{j=1}^{p} A_j \right)$ as an H model for which $P_{A_j|h=k} \equiv T^k_{A_j}$ are trees for all j, k. The structure of $T^k_{A_j}$ is fixed for all k, but its parameters may vary with k. We denote the structure of each connected component[4] j by E_j. Their union is the edge set $E = \bigcup_j E_j$. Because there are p distinct connected components we have

$$|E| \;=\; n - p \tag{6.4}$$

The conditional distribution

$$P_{V|h}(x|k) \;\equiv\; T^k(x) \;=\; \prod_j T^k_{A_j}(x_{A_j})$$

can be viewed either as the product of p independent trees over the A_js or as one tree T^k over V having p connected components. This remark enables us: (1) to avoid explicitly specifying p from now on by using the notation T^k and (2) to view the H as a special case of mixture of trees with shared structure. This allows us to use the efficient algorithms devised for mixtures of trees for fitting tree H models.

A tree H model is obviously a mixture of trees with shared structure; therefore, learning it can be done by a variant of the MixTreeS learning algorithm. The derivation of the algorithm closely parallels the one given in section 3.3.3, so only the final description thereof is be included here.

Algorithm **HMixTreeS**

Input: Dataset $\mathcal{D} = \{x^1, \ldots x^N\}$
Procedure Kruskal(p, weights) that fits a maximum weight tree with $n - p$ edges over V
p a minimum number of connected components
Initial model $\mathcal{M}_0$
Initialize EM with $\mathcal{M} = \{E, \lambda_k, T^k,\ k = 1, \ldots m\}$
Iterate until convergence
E step: compute $\gamma^i_k,\ P^k(x^i)$ for $k = 1, \ldots m,\ i = 1, \ldots N$
M step:
M1. $\lambda_k \leftarrow \Gamma_k / N,\ k = 1, \ldots m$
M2. compute marginals $P^k_v,\ P^k_{uv},\ u, v \in V,\ k = 1, \ldots m$
M3. compute mutual information $I^k_{uv} u, v \in V,\ k = 1, \ldots m$
M4. call Kruskal$(p, \{\, I_{uv|h}\})$ to generate E
M5. $T^k_{uv} \leftarrow P^k_{uv},\ ; T^k_v \leftarrow P^k_v$ for $(u, v) \in E$
Output $\mathcal{M}$

As one can see, the only change w.r.t the standard MixTreeS algorithm is in step **M4** where we need to construct the maximum weight tree with the prescribed maximum number of edges. Kruskal's algorithm fits this purpose easily, since it adds the edges in the order of

[4]The reader should be aware of the distinction between *mixture components* indexed by the values of h and *connected components* which are subtrees of each tree mixture component and are indexed by $j = 1, \ldots p$.

their decreasing weight. It can be proved[5] that stopping Kruskal's algorithm after it adds $n - p$ edges results in the maximum weight spanning tree with (at most) that number of edges.

6.4 Variable partitioning in the general case

6.4.1 Outline of the procedure

We have presented a tractable algorithm for learning a special case of H models, the tree H models. Our next step is to apply the same algorithm to data that are not known to be generated by a tree H model. Hence, we need to proceed with caution and to use model selection and validation criteria to evaluate the output of the algorithm.

The procedure I used can be outlined as a two stage process: The first stage generates models with different values for m and p and different structures and parameters by repeatedly calling the **HMixTreeS** algorithm. The second stage validates the models produced in the first stage by testing that the independencies they imply are not contradicted by the data and computes a score for each (valid) model. The different models can be then compared based on this score. The method is summarized as follows:

Algorithm **H-learn** – outline

Input: Dataset $\mathcal{D} = \{x^1, \dots x^N\}$
Confidence level δ
Parameters $m_{MAX}, p_{MAX}, N^{trials}$
Procedure **HMixTreeS**
for $m = 2, 3, \dots, m_{MAX}$
for $p = 2, 3, \dots p_{MAX}$
for $t = 1, \dots, N^{trials}$
Initialize randomly $\mathcal{M}_0$
$\mathcal{M}_{m,p,t}$=**HMixTreeS**$(\mathcal{D}, \mathcal{M}_0, p)$
Validate $\mathcal{M}_{m,p,t}$ with confidence δ
if $\mathcal{M}_{m,p,t}$ valid
Compute $score(\mathcal{M}_{m,p,t})$
Output $\mathcal{M} = \underset{\mathcal{M}_{m,p,t}\text{valid}}{\text{argmin}}\ score(\mathcal{M}_{m,p,t})$ or the l models with best score

The rest of the chapter studies and develops this approach. For two of the subtasks involved - evaluating the description length of a model and testing independence - detailed descriptions are given in sections 6.6 and 6.7 respectively.

6.4.2 Defining structure as simple explanation

When we set our goal to be learning about a real domain that has inherently unobserved variables, having little or no prior knowledge about it, how can we know when we have discovered the "right" interactions between variables ? We shall *define* a good structure in this case as a structure that is both simple and likely in view of the data, simplicity being measured by the description length of the model. The reader may have realized already

[5] By a proof similar to the proof of Theorem 8.2.18. in [69].

that we what we have stated is a version of the well known Minimum Description Length (MDL) principle [55].

$$\mathcal{M}^* \;=\; \operatorname*{argmin}_{\mathcal{M}} \overbrace{[DL(\mathcal{M}) \;+\; DL(\mathcal{D}|\mathcal{M})]}^{DL(\mathcal{D},\mathcal{M})} \tag{6.5}$$

The last term in (6.5) is $-\log Pr[\mathcal{D}|\mathcal{M}]$ the negative base 2 log-likelihood. The first term can be decomposed as

$$DL(\mathcal{M}) \;=\; \text{bits}(m) + \text{bits}(\text{structure}) + \sum_{k=1}^{m} \text{bits}(\text{parameters} \,|\, \text{structure}) \tag{6.6}$$

Note that any tree structure requires the same number of bits[6], independently of p (1 for each variable, specifying its parent or ϕ). The first term is also constant over all models[7]. Hence, computing the description length amounts to computing the description length of the parameters for the distributions T^k. However, this is not a straightforward task.

We can approach it in two ways. The first is to use the approximation to the model description length derived by Rissanen [55] and also known as the Bayes Information Criterion (BIC). It assumes $\frac{1}{2} n_{pars} \log N$ as the description length of the parameters. According to BIC, each edge $(uv) \in E_T$ contributes

$$\Delta_{uv}^{BIC} \;=\; \frac{m}{2}(r_u - 1)(r_v - 1) \log N \tag{6.7}$$

bits in addition to the description of a "tree" with no edges that we take as bottom line. The minimization of the DL over tree structures can be automatically incorporated into the EM algorithm, by assigning the weigths

$$W_{uv} = N I_{uv|h} - \Delta_{uv}^{BIC}. \tag{6.8}$$

The method has the advantage of indirectly controling the number of connected components p. A drawback of equation (6.8) is that it penalizes parameters in each component equally. To see why this is a problem, imagine that there is one component that is responsible for no data at all; the parameters of this component will have no influence on the data likelihood but the model will still be penalized for them. So, it would be sensible to replace m in the above formula with something representing an "effective number of parameters".

A heuristic remedy that I tried for this problem is to take into account the fact that parameters are estimated from different amounts of data for each k. Hence, I considered that each edge $(uv) \in E_T$ contributes to the total penalty by

$$\Delta_{uv}^{BICh} \;=\; \frac{1}{2}(r_u - 1)(r_v - 1) \sum_{k=1}^{m} \max(\log \Gamma_k, 0) \tag{6.9}$$

which gives the $BICh$ criterion.

A second approach is to directly approximate the DL of a model. To achieve this, we represent the probability values on a number of bits b (smaller than the machine precision) obtaining an approximate model $\mathcal{M}_b$. For each b in a chosen range, $DL(\mathcal{M}_b)$ is easily

[6] Unless we have a nonuniform prior over tree structures.

[7] If m is small,as it is implicitly assumed throughout the paper and if there is no prior on m. Otherwise, the first term will vary with m.

computed and so is $Pr[\mathcal{D}|\mathcal{M}]$. Then we approximate the true DL by $\operatorname{argmin}_b DL(\mathcal{D}, \mathcal{M}_b)$. A full description of this procedure will be included in section 6.6. This method being a purely evaluatory method, it can be used even for models learned using the penalty from equations (6.8) or (6.9) above.

Number of connected components p As a consequence of the weight modification (6.8), the weights can now be negative. Kruskal's algorithm will stop adding edges when it first encounters a weight ≤ 0, so this implicitly controls p. Moreover, the penalty will be likely to increase with m thus favoring sparser structures. Alternatively, one can make p vary over a previously fixed range (e.g. between 2 and n) and compare all the resulting models by their description lengths.

Selecting m. The description length DL provides us with a consistent criterion for comparing between model structures with different numbers of trees, thus enabling us to select m.

Local optima We deal with this problem in the usual way: by restarting the EM algorithm several times, each time from different random points in the model space. Consequently, for each new run of the EM algorithm we may end up with a different model structure (and parameters) but this is a help inasmuch as it is a problem, since the role of the EM algorithm is to present us with good, but also diverse candidate structures.

6.5 Experiments

6.5.1 Experimental procedure

The experiments described in the following aim to assess the ability of the **HMixTreeS** algorithm to discover the correct structure and the hidden variable in two cases: when the data are generated by a mixture of trees and when they are not. In both cases, we will also evaluate the description length and the alternative scores discussed above as model selection criteria for this task.

The experiments therefore were run on artificial data generated in two steps. First, an H-model structure is created (figures 6-2 and 6-3 show the structures used here). The domain has 12 visible variables and one hidden variable, all taking 3 values. Second, for each structure, 10 sets of random parameters were sampled; each resulting model was used to generate a data set of 3,000 examples, of which 2,000 were used for training and 1,000 were used for testing. The values of the hidden variable were recorded but served only for testing purposes. The first structure (figure 6-2) is a tree H model, the other two (figure 6-3) are general H models. In the latter case, because pilot experiments have shown that discovering the variable partitioning is relatively easy when the true partitioning is visible in the marginal distribution of the data, all the data sets were checked for this property and accepted only if the true structure could not be recovered by fitting a single tree.

6.5.2 Experiments with tree H models

This group of experiments demonstrates that the **HMixTreeS** algorithm is capable of recovering structure when the generative model is a tree H model. The model structure is shown in figure 6-2. Two random parametrizations for this structure were generated and from each them a training set ($N = 3,000$) and a test set ($N_{test} = 1000$).

For each training set, models with $m = 2, 3, 4$ were learned using the MDL edge penalty to choose the number of connected components p. For comparison the case $m = 3,\ p =$

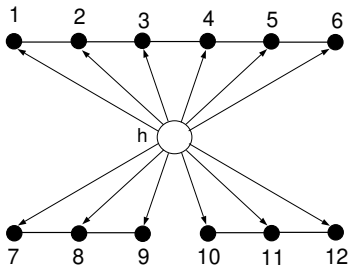


Figure 6-2: The tree H model used for training.

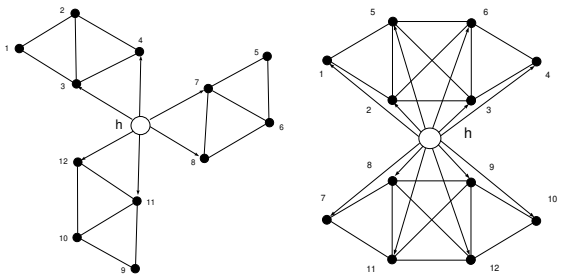


Figure 6-3: The models used for training: (a) S1, (b) S2.

4, $\beta = 0$ was tried too. For each set of parameters, the algorithm is run 20 times with different random initializations.

The results for one of the data sets are displayed in figure 6-4. From the figure we can see that the correct structure $p = 3$ is found in all 20 cases corresponding to $m = 3$ and MDL edge penalty. The high accuracy means that the values of the hidden variable are correctly estimated. It can also be seen that for $m = 2$ and 4 and for $p = 4$, the description length is higher than for the true model $m = 3, p = 3$. Hence, the number of values of the hidden variable is correctly selected too. The results for the second data set are similar, except that there the correct structure is found in only 19 out of 20 trials.

Another example of successful structure learning and hidden variable discovery is the Bars learning task that will be presented in section 7.1.2.

6.5.3 General H models

The following experiments were run on 10 data sets generated from structure S1. For every parameter setting, the learning algorithm was run 20 times from different initial points. The initial points were chosen randomly without including any knowledge about the desired structure. Unless otherwise stated, the empirical description length DL was used as model selection criterion. We diverge a little from the algorithm outlined previously by leaving model validation last, after examining the properties of all the resulting models.

The first series of experiments were run on models with $m = 3$ and fixed $p = 3$ (the true values). As figure 6-5 shows, in 8 out of 10 trials the correct structure prevailed both in terms of the number of times it appeared and in terms of its description length. For these 8 cases, according to the DL criterion, the best model had the correct structure and in many

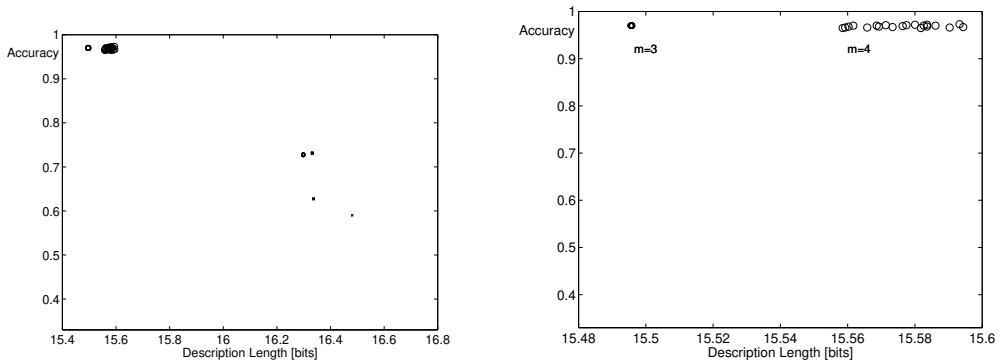


Figure 6-4: Description lengths and accuracies for models with fixed $p = 2$ and variable $m = 2, 3, 4$ learned from data generated by the tree H model in figure 6-2. For $m = 3$ and 4 all structures are correct. (a) general, (b) detail of the upper left corner of (a).

cases all five best models were correct. The accuracy of the hidden variable retrieval is also relatively high (the baseline for accuracy is 0.33), but not as high. For the remaining two data sets (c and j in figure 6-5) the correct structure is present but it overtaken in score by other structures (however, at least one of the best 5 models has the correct structure).

I have run a series of similar experiments with $m = 3$ but applying the MDL edge penalty to set p. In this situation, the learning algorithm consistently overestimated p by 1, but otherwise found a structure compatible with the correct one. Therefore, I searched the space of p values in a manner similar to the search for m. For this purpose, models with $m = 3, 4$ and $p = 2, 4$ were trained. The results were not encouraging: For $p = 2$ both $m = 3$ and $m = 4$ produced a significant number of models with shorter description length. This suggests the presence of residual dependences between components that are taken up by the additional link (especially in the case $m = 3$), something to be investigated in the model validation phase. But, as a conclusion so far, estimating p appears to be a hard task.

The next set of experiments studies the effect of varying m with p fixed at the true value 3. H models with $m = 2, 4, 5, 6$ were trained on all the data sets. Figure 6-6 summarizes the results for set d. One sees that the models for $m = 2$ cluster separately in a region of higher description lengths. For $m = 4, 5, 6$ three things happen: the proportion of good structures is increased (for all data sets); the description length of the models decreases until $m = 5$ and remains about the same for $m = 6$ (all but 1 case, where it settles at $m = 4$); the hidden variable accuracy remains stationary (all cases). Moreover, in all cases, the minimum DL model has the best structure (but $m > 3$) and except for data set c, all 5 best models are correct. The explanation for this behavior is that, since the true mixture components are not tree distributions, adding more components to the mixture of trees helps model the true distribution more closely: some of the original three mixture components are allocated more than one tree. The explanation is sustained by examining the confusion matrices: for each of the m trees the data it represents comes mostly from one component of the original mixture.

Here is a confusion matrix for data set d, $m = 5$. The columns are the trees, the rows are the original components, and the matrix element i, j represents the proportion of data

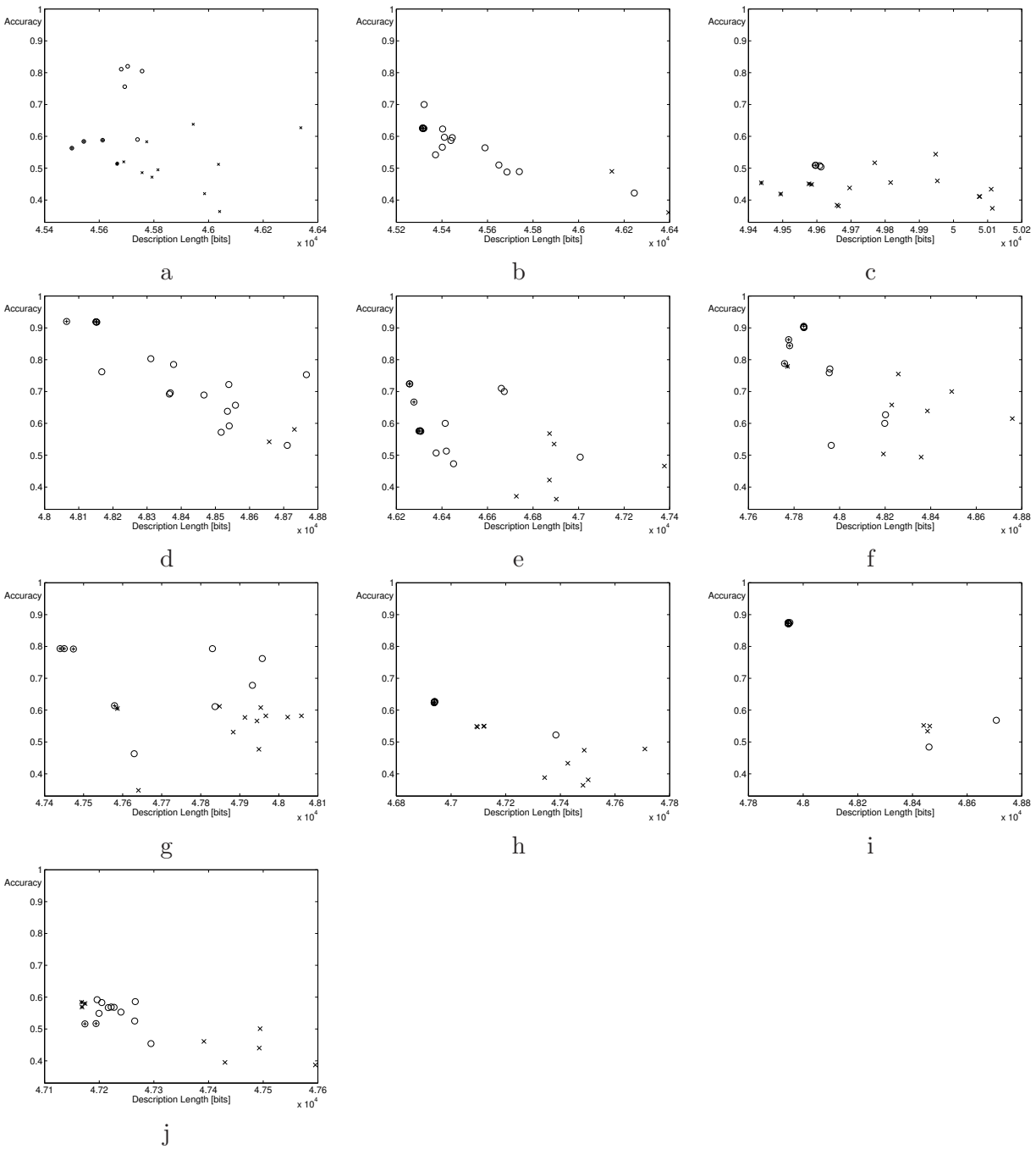


Figure 6-5: Model structures and their scores and accuracies, as obtained by learning tree H models with $m = 3$ and $p = 3$ on 10 data sets generated from structure S1. Circles represent good structures and x's are the bad structures. A + marks each of the 5 lowest DL models. The x-axes measure the empirical description length in bits/example, the vertical axes measure the accuracy of the hidden variable retrieval, whose bottom line is at 0.33

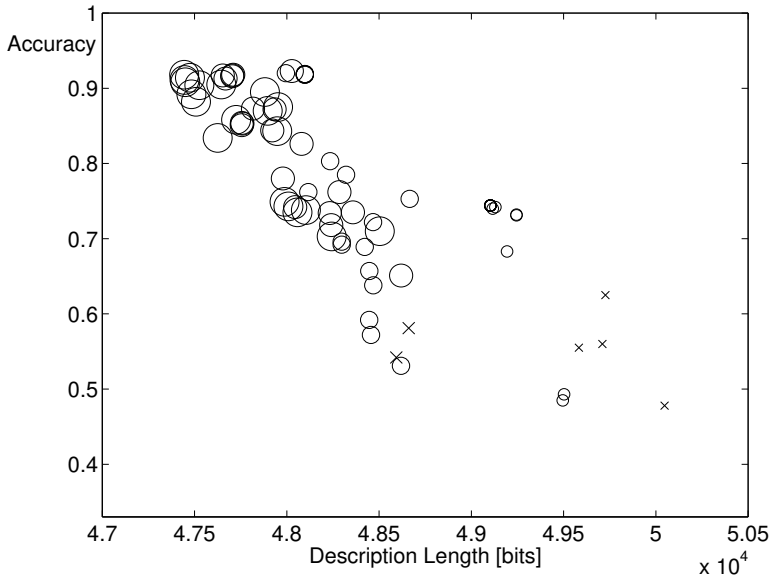


Figure 6-6: Model structures and their scores and accuracies, as obtained by learning tree H models with $p = 3$ and variable $m = 2, 3, 4, 5$ on a data set generated from structure S1. Circles represent good structures and x's are the bad structures. The sizes of the symbols are proportional to m. The x-axis measure the empirical description length in bits/example, the vertical axis measure the accuracy of the hidden variable retrieval, whose bottom line is at 0.33. Notice the decrease in DL with increasing m.

from component i modeled by tree j.

```
      0     1     2     3     4
0: 0.504 0.391 0.063 0.018 0.025
1: 0.010 0.018 0.659 0.012 0.301
2: 0.050 0.023 0.045 0.837 0.045

Cluster Accuracy = 0.914
```

The experiments show that choosing m solely by minimizing the description length can lead to an overestimation of the number of components. Based on the gap in the DL dimension between the models with $m = 2$ and the rest of the models (all data sets except c) one could study the possibility of using it as a criterion for selecting m.

Comparison between the different model section criteria The above results were plotted again (not shown), replacing in turn the BIC, the modified BIC and the test set log-likelihood as the horizontal axis, in order to see if there are differences in selecting the correct structure. We found that the test log-likelihood fails to select the best model in 3 cases when DL does, however we think that this is not sufficient evidence that the log-likelihood is less good than the description length. The other differences seemed also to favor the empirical description length, but in an even less significant way.

The results obtained for the second structure are qualitatively similar.

6.6 Approximating the description length of a model

Here is described the method used to approximate the DL of a mixture of trees model. It is a general enough method to be used with any probability model, graphical or not. The idea is to encode the model $\mathcal{M}$ as best we can on a fixed number of bits B and then to compute the description length of the data under this approximated model called $\mathcal{M}_B$. If B is smaller than the length of the original representation of $\mathcal{M}$ and if $\mathcal{M}$ is a local maximum of the likelihood, then the term $DL(\mathcal{D}\,|\,\mathcal{M}_B)$ will be larger then the original $DL(\mathcal{D}\,|\,\mathcal{M})$; however, the model DL will be equal to B and thus smaller than the original $DL(\mathcal{M})$. By varying B over a reasonable range and minimizing the total description length we obtain

$$DL^{emp} \;=\; \operatorname*{argmin}_{B} DL(\mathcal{D}, \mathcal{M}_B), \tag{6.10}$$

an upper bound to the local minimum of the desired description length.

The better the encoding for each B (in the sense of preserving the likelihood of each observed data point), the closer is the above minimization to the true $DL(\mathcal{D}, \mathcal{M})$. In practice, we realize a tradeoff between encoding quality and computational cost. We want cheap encoding schemes that preserve the represented distribution reasonably well.

To evaluate the DL of a mixture of trees, we use the above idea only to approximate the DL of the tree parameters within the mixture's DL. We assume that the structure description takes the same number of bits for any tree (n pointers, one – possibly null – for the parent of each node) and thus can be ignored from the DL. The DL of the λ parameters is approximated by

$$b_\lambda \;=\; \frac{m-1}{2} \log N.$$

The parameters of the tree distributions are encoded with the same number of bits b. This number is a sum of two precisions, b_m and b_M, that will be defined below. The number of parameters is given by equation (2.7). Thus, the total number of bits (excluding the constant for structure) is

$$DL(Q) \;=\; b_\lambda + m(b_m + b_M)[\sum_{(u,v)\in E} (r_u - 1)(r_v - 1) + \sum_{v\in V} r_v - n] + \sum_{(u,v)\in E} r_u \log r_v \tag{6.11}$$

Now we show how to encode the parameters of the trees. In our directed tree based implementation, the parameters represent the values of the conditional probabilities $T_{v|\mathrm{pa}(v)}(x_v|x_{\mathrm{pa}(v)})$ (where, of course, sometimes $\mathrm{pa}(v)$ is the empty set). For each fixed value of $\mathrm{pa}(v)$, this distribution is a multinomial. Therefore, in the following we describe a way of encoding multinomial distribution.

6.6.1 Encoding a multinomial distribution

Encoding a set of real numbers on a finite number of bits always involves some sort of quantization. So, to encode the parameters of a distribution one needs to first quantize them. The quantization of the multinomial distribution parameters is done in the natural parametrization introduced in section 4.2.2.

Let the distribution values be $\theta_j,\ j = 1 \ldots r$ and let j_{MAX} be the index of the largest θ. The natural parameters are given by (4.27), reproduced here in slightly modified form

$$\phi_i \;=\; -\frac{\log \theta_i}{\log \theta_r} \quad i = 1, \ldots r,\ i \neq j_{MAX}. \tag{6.12}$$

Adding a minus sign in front of the logarithm function ensures that all the ϕ parameters are positive. Since the θ parameters are allowed to be 0, their range is in $[0, \infty]$.

We are going to represent the ϕ values on a finite number of bits. This is done in two stages: First, the large ϕ values are "truncated" to the constant 2^{b_M} if they exceed it. This corresponds to bounding the smallest θ values away from 0. Then, the resulting values are represented as b_m bit precision binary numbers, or equivalently, they are multiplied by 2^{b_m} and stored as integers. Note that while b_M controls the representation precision for small probabilities, b_m control the precision of large probabilities. Both precisions are relative to the largest value of θ. The full encoding of $\theta_{1\ldots r}$ is represented by r, the encoding precision b_m, the position of the maximum j_{MAX} and the values

$$\phi_i^{enc} \;=\; 2^{b_m} \min(\phi_i, 2^{b_M}), \quad i = 1, \ldots r,\; i \neq j_{MAX} \tag{6.13}$$

To decode the values of θ from the above representation one has to apply formula (4.28) to $\phi_i^{enc} 2^{b_m}$ and replace the maximum θ in position j_{MAX}. Note that the value b_M is not necessary for decoding.

If the value of r is given by a separate table (remember that we are encoding several distributions over the same domain) and that b_m is also encoded separately, the total number of bits to encode $\theta_{1,\ldots r}$ is

$$\log\lceil r \rceil + (b_m + b_M)(r-1) \;\approx\; \log r + (b_m + b_M)(r-1) \tag{6.14}$$

By adding up the right-hand sides of (6.14) values for all $T_{v|\mathrm{pa}(v)}$ one obtains the last two terms of (6.11). To see this remark that: (1) each free parameter is encoded on $b_M + b_m$ bits and (2) $r_u \log r_v = r_v \log r_u$ for all positive numbers r_u, r_v. To make the search tractable, the values of b_m and b_M are common to all distributions in Q and thus b_m is recorded only once.

6.7 Model validation by independence testing

6.7.1 An alternate independence test

Assume that we have two independent discrete variables u and v taking r_u, r_v values respectively, and an i.i.d. sample of size N drawn from their joint distribution. Since the variables are independent, their mutual information is obviously zero, but its estimate from the sample $\hat{I}_{uv}$ does not generally exactly equal zero. In this section we want to bound the probability that the *sample mutual information* $\hat{I}_{uv}$ is larger than a threshold $M > 0$.

Of course, $\hat{I}_{uv}$ will never exceed $\max(\log r_u, \log r_v)$ so it will be assumed that M is below this value. We shall use large deviation theory, and in particular Sanov's theorem [11] to obtain this bound.

Let us denote by $\hat{P}_{uv}$, $\hat{P}_u$, $\hat{P}_v$ respectively the sample joint distribution of u, v and its marginals. We shall consider the following two sets of probability distributions over the domain $\Omega(uv)$

$$E \;=\; \{P \mid I_{uv}^P \geq M\} \tag{6.15}$$

$$F \;=\; \{Q \mid I_{uv}^Q = 0\} \tag{6.16}$$

The set F represents all the factorized distributions over $\Omega(uv)$. With this notation our problem can be formulated as: what is the probability of the sample distribution $\hat{P}$ to belong to E given that the true distribution is in F?

Sanov's theorem gives us the following result: For any distribution Q and any closed set of probability distributions E, the probability that the sample distribution $\hat{P}$ of an i.i.d. sample of size N drawn from Q is in E is denoted by $Q^N(E)$ and satisfies

$$Q^N(E) \;\leq\; (N+1)^r 2^{-N.KL(P^*_Q \,||\, Q)} \tag{6.17}$$

where r is the size of Q's domain and

$$P^*_Q \;=\; \operatorname*{argmin}_{P \in E} KL(P \,||\, Q) \tag{6.18}$$

is the distribution in E that is closest to Q in KL-divergence[8]. We also know [70] that for any distribution P over $\Omega(uv)$

$$\min_{Q \in F} KL(P \,||\, Q) \;=\; KL(P \,||\, P_u P_v) \;=\; I^P_{uv}. \tag{6.19}$$

Hence

$$\min_{P \in E, Q \in F} KL(P \,||\, Q) \;=\; \min_{P \in E} I^P_{uv} \;=\; M \tag{6.20}$$

Using this result in Sanov's theorem above we conclude that for any $Q \in F$

$$Q^N(E) \;\leq\; (N+1)^{r_u r_v} 2^{-NM} \tag{6.21}$$

In other words, for any two independent variables u, v, the probability that their sample mutual information $\hat{I}_{uv}$ exceeds a threshold $M > 0$ is bounded by

$$(N+1)^{r_u r_v} 2^{-NM} \;=\; \delta.$$

Conversely, given a probability δ we can derive the corresponding M:

$$M \;=\; \frac{1}{N}[\log \frac{1}{\delta} \;+\; r_u r_v \log(N+1)] \tag{6.22}$$

This bound is distribution free: it requires no assumption about the distributions of u and v. As expected, the value of M for fixed δ decreases with the sample size N. Contrast this with the χ^2 test, that *does* make assumptions about the underlying distribution but whose p-values (the analogues of M) are independent of the sample size.

Because E is the closure of its interior, Sanov's theorem and a reasoning similar to the above one enables us to establish a lower bound for $Q^N(E)$

$$Q^N(E) \;\geq\; \frac{1}{(N+1)^{r_u r_v}} 2^{-NM} \tag{6.23}$$

It is interesting to compare the test devised here, call it *Sanov* test with the well-known χ^2 test for independence. It is easy to show that the χ^2 statistic is approximating a KL divergence. For this, let us compute the quantity $KL(\hat{P}_u \hat{P}_v \,||\, \hat{P}_{uv})$ as a function of the counts N_{ij} =#times $u = i$, $v = j$, N_i =#times $u = i$, N_j =#times $v = j$ and

$$n_{ij} \;=\; \frac{N_i N_j}{N}, \;\; i = 1, \ldots r_u, \; j = 1, \ldots r_v.$$

[8] A somewhat tighter bound for the r.h.s of equation (6.17) is $[r + N^{r-1}]2^{-N.KL(P^*_Q \,||\, Q)}$.

$$
\begin{aligned}
KL(\hat{P}_u\hat{P}_v \,||\, \hat{P}_{uv}) &= \sum_{ij} \frac{n_{ij}}{N} \log \frac{n_{ij}}{N_{ij}} && (6.24)\\
&= -\frac{1}{N \ln 2} \sum_{ij} n_{ij} \ln \frac{N_{ij}}{n_{ij}} && (6.25)\\
&= -\frac{1}{N \ln 2} \sum_{ij} N_{ij} \ln(1 + \frac{N_{ij} - n_{ij}}{n_{ij}}) && (6.26)\\
&\approx -\frac{1}{N \ln 2} \sum_{ij} [n_{ij} \frac{N_{ij} - n_{ij}}{n_{ij}} - \frac{1}{2} n_{ij} \frac{(N_{ij} - n_{ij})^2}{n_{ij}^2} + \ldots] && (6.27)\\
&= \frac{1}{N \ln 2} \frac{1}{2} \sum_{ij} \frac{(N_{ij} - n_{ij})^2}{n_{ij}} && (6.28)\\
&= \frac{1}{2 \ln 2} \chi^2 && (6.29)
\end{aligned}
$$

Hence, whereas the Sanov test is based on $KL(\hat{P}_{uv} \,||\, \hat{P}_u\hat{P}_v)$, the χ^2 independence test is an approximation of the reverse divergence $KL(\hat{P}_u\hat{P}_v \,||\, \hat{P}_{uv})$. Of course, the motivation for the χ^2 test is not in this divergence but in Central Limit Theorem arguments.

6.7.2 A threshold for mixtures

Now we undertake to estimate an approximate confidence interval for the sample mutual information when the variables u and v are independent given a third variable z. In this case the true joint distribution of u, v is

$$
Q \;=\; \sum_{k=1}^{m} \lambda_k Q_u^k Q_v^k \tag{6.30}
$$

where, as usual, the values λ_k represent the probabilities of variable z being in state $k \in \{1, \ldots m\}$. Again, we assume an i.i.d. sample of size N; given this sample, we group the data points by the values of z in m groups of size N_k respectively. Therefore, we have

$$
\sum_{k=1}^{m} N_k \;=\; N.
$$

We denote by $\hat{I}_{uv}^k$ the sample mutual information between u and v in group k. Let us further denote by $\rho(N, M)$ the probability that the sample mutual information from a size N sample does not exceed $M > 0$.

$$
\rho(N, M) \;=\; 1 - Q^N(E) \tag{6.31}
$$

Of course, ρ is also a function of r_u, r_v and the true distribution Q. To simplify the notation we drop these variables. This is acceptable because r_u and r_v are fixed and because the dependence on Q will be masked by the use of bounds that hold for any Q.

Our goal is now to fix a confidence level δ and to find the thresholds $M_k,\ k = 1, \ldots m$ depending on δ such that $\hat{I}_{uv}^k < M_k$ for all k with probability δ. We start by expressing the probability of the event $\mathcal{E}$="$\hat{I}_{uv}^k < M_k$ for all k" as a function of the values M_k and after we arrive at a convenient expression we will equate that with δ.

Let us first introduce some notation: let Z be the i.i.d. sample of size N representing the values of z, and $\underline{N} = (N_1, \ldots N_m)$ an m-tuple of N_k values as defined above. By

$$\binom{N}{\underline{N}} \;=\; \frac{N!}{N_1!N_2!\ldots N_m!} \tag{6.32}$$

we denote "N choose $N_1, \ldots N_m$", the multinomial coefficient indexed by $\underline{N} = (N_1, \ldots N_m)$. Then,

$$\begin{aligned} Pr[\mathcal{E}] &= \sum_Z Pr[Z] \prod_k Pr[\hat{I}^k_{uv} < M_k \mid Z] \\ &= \sum_Z \prod_k \lambda_k^{N_k(Z)} \rho(N_k(Z), M_k) \\ &= \sum_{\underline{N}} \binom{N}{\underline{N}} \prod_k \lambda_k^{N_k} \prod_k \rho(N_k, M_k) \end{aligned}$$

If the factors $\rho(N, M)$ vary slowly with N (further on we show when this happens), then the above sum is dominated by the term corresponding to the largest $\binom{N}{\underline{N}} \prod_k \lambda_k^{N_k}$, a value attained for

$$N^*_k \;\approx\; \lambda_k N, \;\; k = 1, \ldots m. \tag{6.33}$$

Moreover, using Stirling's approximation $N! \approx \left(\frac{N}{e}\right)^N$ we have

$$\begin{aligned} \binom{N}{\underline{N^*}} \prod_k \lambda_k^{N^*_k} &= \frac{N!}{\prod_k (\lambda_k N)!} \prod_k \lambda_k^{\lambda_k N} \\ &= \frac{N!}{N^N} \prod_k \frac{(\lambda_k N)^{\lambda_k N}}{(\lambda_k N)!} \\ &\approx e^{-N + \sum_k N\lambda_k} \\ &= 1 \end{aligned}$$

This allows us to write

$$Pr[\mathcal{E}] \;\approx\; \prod_{k=1}^{m} \rho(N\lambda_k, M_k) \tag{6.34}$$

If we now choose M_k according to (6.22) i.e.

$$M_k \;=\; \frac{1}{\lambda_k N}[\log \frac{1}{\tilde{\delta}} \;+\; r_u r_v \log(\lambda_k N + 1)] \tag{6.35}$$

we have

$$1 - \tilde{\delta} \;\leq\; \rho(\lambda_k N, M_k) \;\leq\; 1 - \frac{\tilde{\delta}}{(\lambda_k N + 1)^{r_u r_v}} \tag{6.36}$$

and

$$Pr[\mathcal{E}] \;\approx\; (1 - \tilde{\delta})^m = 1 - \delta \tag{6.37}$$

For a given δ the corresponding $\tilde{\delta}$ is

$$\tilde{\delta} \;=\; 1 - (1 - \delta)^{\frac{1}{m}} \;\approx\; \frac{\delta}{m} \tag{6.38}$$

To conclude, we have to prove our assumption that the functions $\rho(N_k, M_k)$ vary slowly with the first argument. More precisely, we are interested in upper bounding them. But

$$\frac{\rho(N_k, M_k)}{\rho(N_k^*, M_k)} \;\leq\; \frac{1}{\rho(N_k^*, M_k)} \;\leq\; \frac{1}{1-\tilde{\delta}} \tag{6.39}$$

For small values of δ the above quantity is close to unity validating our previous derivation.

6.7.3 Validating graphical models with hidden variables

In this section we will apply the previous test to the models of interest to us: graphical models with a hidden variable that separates the observed ones into two conditionally independent subsets. Thus, we will assume that we have a set of variables V, a dataset $\mathcal{D} = \{x^1, \dots x^N\}$ of observations from V and the model

$$Q(x) \;=\; \sum_{k=1}^{m} \lambda_k T_A^k(x_A) T_B^k(x_B) \tag{6.40}$$

where A, B is a partition of V. Thus Q assumes that there is a hidden variable z taking values $k = 1, \dots m$ with probabilities λ_k and that $A \perp B \mid z$. We want to test whether the data $\mathcal{D}$ supports the independence implied by Q. In the following we will be referring to the Sanov test developed in section 6.7.1, but arguments similar to those presented here hold for the χ^2 independence test.

There are two issues that need to be addressed: First, in this case the variable z is not observed. Therefore, we use Monte Carlo sampling from the posterior $\gamma_k(i)$ to obtain sequences $Z = (z^1, \dots z^N)$. For each of them we compute $\hat{I}_{uv}^k$, $k = 1, \dots m$ and compare them to the respective thresholds.

Second, establishing independence between two sets of variables implies a summation over $\Omega(A) \times \Omega(B) = \Omega(V)$. This configuration space is usually much larger than the size N of the available sample. With $r_u r_v = |\Omega(V)|$ the threshold M becomes giant (6.22) and the comparison meaningless. And, if N becomes comparable to $\Omega(V)$ then the calculations become intractable. Therefore, we will test the independence of A and B by performing independence tests on pairs of variables $u \in A$, $v \in B$. In general it is **not true** that

$$u \perp v \mid z \;\; \forall u \in A,\, v \in B \quad \Longrightarrow \quad A \perp B \mid z. \tag{6.41}$$

But (6.41) is true for a broad class of distributions. In particular, if the *true* distribution of the data is a chain graph (6.41) holds. Now we have to test the independence of $|A||B|$ pairs of variables, something that is computationally intensive, but tractable. However, we will be performing simultaneous tests and we have to take this into account.

We use therefore a Bonferroni inequality [42] that states that if each hypothesis $\mathcal{H}_{uv}$ is true with confidence $1 - \epsilon_{uv}$, then their conjunction is true with confidence at least

$$1 - \sum_{uv} \epsilon_{uv}.$$

In the present case, $\mathcal{H}_{uv}$ represent the hypotheses "$u \perp v \,|\, z$"; by $\mathcal{E}_{uv}$ we denote the events "$\hat{I}_{uv}^k(Z, \mathcal{D}) < M_k \, \forall k$". There is no special reason to treat any of them differently; hence

$$\epsilon_{uv} \;=\; \frac{\epsilon}{|A||B|} \;=\; \tilde{\epsilon} \quad \forall u \in A,\; v \in B \tag{6.42}$$

The test procedure can be summarized as follows:

Algorithm **TestIndependence**

Input	model Q, data set $\mathcal{D}$, confidence ϵ
1.	$\tilde{\epsilon} = \epsilon/(\lvert A\rvert\lvert B\rvert)$
	fix δ, compute $\tilde{\delta}$ by (6.38)
2.	compute $\underline{\gamma} = (\gamma_k(i),\ k = 1, \dots m,\ i = 1, \dots N)$ the posterior probability that x^i was generated by component k of the mixture
3.	for $u \in A$, $v \in B$
	3.1 compute the thresholds M_k by (6.22)
	3.2 repeat
	3.2.1 sample Z from $\underline{\gamma}$
	3.2.2. for $k = 1, \dots m$ compute $\hat{I}^k_{uv}$ and compare to M_k until $Pr[\mathcal{E}_{uv}]$ <> $\tilde{\delta}$ with confidence $1 - \tilde{\epsilon}$; ">" means reject $\mathcal{H}_{uv}$, "<" means accept it
Output	accept independence hypothesis if all the tests in step 3. accept it, otherwise reject

The crux of the method is step 3 where we need to estimate the low probabilities $Pr[\mathcal{E}_{uv}]$ with *very* high confidence $\tilde{\epsilon}$ for each pair u, v. We have of course the freedom to choose a $\tilde{\delta}$ closer to 0.5 in order to require fewer iterations of 3.1–3.3 to compare $Pr[\mathcal{E}_{uv}]$ with $\tilde{\delta}$ at the required confidence level. But a larger δ means a lower threshold M_k and this in turn means loosening the approximation in Sanov's theorem[9] and thus increasing the probability of type I error (false acceptance).

The choice of ϵ reflects our tradeoff between type I and type II errors. For a small ϵ the probability of rejecting a model (i.e. rejecting the independence hypothesis) that is correct is low, but we are likely to accept independence when this is not true. This latter probability is inflated in our case by further dividing ϵ between the $|A||B|$ tests in (6.42). Therefore it is better to accept a somewhat larger false rejection risk by choosing a larger ϵ lest the test becomes too permissive.

Let us examine if the assumptions our tests are based upon are true in the above procedure. The $|A||B|$ individual independence tests are dependent on each other because they share the observed values $(x^1, \dots x^N)$, but the Bonferroni inequality accounts for that. The dataset $\mathcal{D}$ is not the dataset the model was trained on.

It is in using an independence test based on $Pr[\mathcal{E}_{uv}]$ that we depart from the assumptions. Because the data $\mathcal{D}$ are already observed, the quantity that is actually estimated by repeating 3.2.1 – 3.2.2 is not $Pr[\mathcal{E}_{uv}]$ but $Pr[\mathcal{E}_{uv}|\mathcal{D}]$. It can be expressed as

$$Pr[\mathcal{E}_{uv}|\mathcal{D}] = \sum_Z Pr[Z|\mathcal{D}] \prod_k Pr[I^k_{uv} < M_k|Z, \mathcal{D}] \tag{6.43}$$

$$= \sum_Z \prod_i \gamma_{z^i}(i) \prod_k \delta_{I^k_{uv}(Z,\mathcal{D}) < M_k} \tag{6.44}$$

$$= \sum_{Z: I^k_{uv}(Z,\mathcal{D}) < M_k\, \forall k} \prod_i \gamma_{z^i}(i) \tag{6.45}$$

Hence, a correct test would predict or bound the above probability. But this involves the intractable summation over $\{I^k_{uv}(Z, \mathcal{D}) < M_k\, \forall k\}$.

Finally, a general fact about independence tests and their utility. The independence test can only reject independence and provide an estimate of the risk in doing it. It can

[9] This can be seen by examining the proof of Sanov's theorem.

never accept independence and estimating the risk of error associated with this decision is usually difficult.

6.8 Discussion

This chapter has been an exploration in to the domain of learning the structure of graphical models with hidden variables. We proposed to use the mixture of trees learning algorithm in order to uncover the presence of a hidden variable that partitions the visible variables set into mutually independent subsets. Such a model is useful because (if the number of values of the hidden variable is small) it offers the possibility to drastically reduce the complexity of the resulting density. Moreover, if such a hidden variable and the corresponding separation are discovered, the model's structure can be refined by subsequent structure learning separately in each of the variable clusters. It is, to my knowledge, the first top down approach to the problem of structure learning. However, as we stressed, the task that we attempt to solve is known to be a difficult one. Searching the space of possible structures is proven to be NP-hard. We have transformed it into a continuous domain optimization problem via the EM algorithm and are relying on multiple initializations to obtain good candidate structures. Empirically we showed that this is not unreasonable: the EM algorithm is indeed capable of uncovering the hidden structure and to converge to it even when this structure is not detectable in the marginal distribution of the visible variables.

The problem of model selection for domains with hidden variable is a topic of current research. We introduced a method to empirically evaluate the description length of a distribution. The empirical description length and a number of other criteria (BIC, holdout set likelihood and modified BIC) were compared on the task of selecting the best H model structure. It was found that the examined criteria are relatively similar in accuracy, with slight advantage for the empirical description length. None of the criteria proved completely reliable in selecting the good structure. One reason for this is that the task is such that the scoring function, no matter which one, is in impossibility to distinguish between the failure to discover the independency structure (the information of interest) on one side, and inaccuracy in the representation of the conditional distributions once the structure is found (information irrelevant to our task). In the context of comparing models by score, we found by experiment that determining the correct number of connected components p is not an easy task. In particular, for the data used in the experiments, the MDL edge penalty is systematically too strong, biasing towards models with false independencies.

An alternate approach is to test the obtained models for the independencies they assume. Testing independence in models with hidden variables is a yet unsolved problem. Here we propose a heuristic based on Monte Carlo sampling, together with a novel approach to independence testing based on large deviation theory. Independence tests allow to eliminate the structures in which the hypothesised variable clusters are not independent given the hidden variables, but they cannot help if the model has fewer connected components than the correct one (however, in some cases this can be considered a minor structure error). Moreover, the risk of discarding a structure that is correct but has residual dependencies because the trees are too simple to represent the distributions in each variable cluster and thereby induce inaccuracy in the values of the hidden variable.

By contrast, the case when the working assumption – that the data was generated by a tree H model – was correct has proved to be a much easier one. The rate of structure recovery by the EM algorithm, when the structure parameters m and p are correct is very

high, often 100%. Likewise, the accuracy of the values of the hidden variable is very good. For model selection with fixed m and p, as well as for search across values of m, p, the model selection criteria, be they empirical DL or others, are found to be reliable.

Chapter 7

Experimental results

This section describes the experiments that were run in order to assess the capabilities and usefulness of the mixture of trees model. The first experiments examine the ability of the **MixTree** algorithm to recover the original distribution when the data are generated by a mixture of trees. For these models, only artificial data are used. The next group of experiments studies the performance of the mixture of trees as a density estimator; the data used in these experiments are *not* generated by mixtures of trees. Finally, we perform classification experiments. In these experiments, we study both the mixture of trees and a classifier consisting of a single tree trained as a density estimator. Comparisons are made with classifiers trained in both supervised and unsupervised mode. The section ends with a discussion of the single tree classifier and its feature selection properties.

In all the experiments below, unless otherwise stated, the training algorithm is initialized at random, independently of the data or of the knowledge about the desired solution. Log-likelihoods are expressed in bits/example[1] and therefore are sometimes called *compression rates*. The lower the value of the compression rate, the better the fit to the data.

In the experiments that involve small data sets, a smoothing procedure has sometimes been used. The technique I applied is called *smoothing with the marginal* and is described in [24, 51]. One computes the pairwise marginal distributions for the whole data set P_{uv}^{total} and replaces the marginals P_{uv}^{k} by

$$\tilde{P}_{uv}^{k} \;=\; (1-\alpha)P_{uv}^{k} + \alpha P_{uv}^{total} \tag{7.1}$$

Intuitively, the effect of this operation is to give a small probability weight to unseen instances and to make the m trees more similar to each other, thereby reducing the effective model complexity. For the method to be effective in practice, α is a function of Γ_k and P_{uv}^{k}. In particular, for the experiments in this thesis, a global smoothing parameter α is apportioned between mixture components and, within each mixture component, beween tree edges. Whereas α in the above equation is always between 0 and 1, the global smoothing parameter may be larger than 1. In some cases (e.g. when fitting a single tree), I used smoothing with a uniform distribution instead of smoothing with the marginal.

7.1 Recovering the structure

7.1.1 Random trees, large data set

For the first experiment, we generated a mixture of 5 trees over 30 variables with $r = 4$ for all vertices. The distribution of the choice variable λ as well as each tree's structure

[1] 1 bit is the negative base 2 logarithm of a likelihood of 0.5.

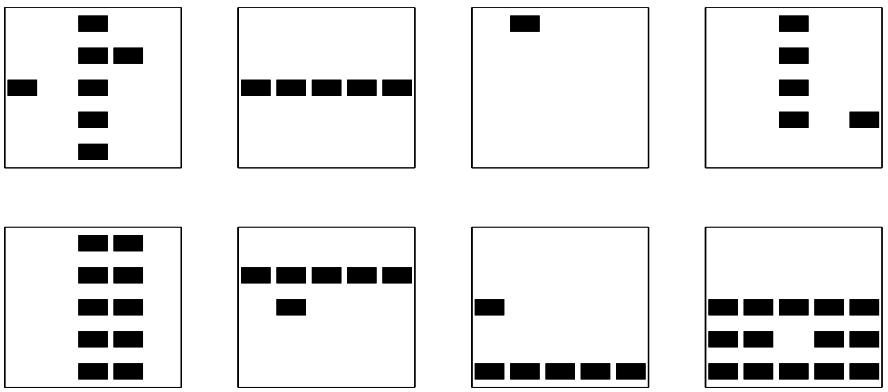


Figure 7-1: Eight training examples for the bars learning task.

and parameters were sampled at random. The mixture was used to generate 30,000 data points that were used as training set for a **MixTree** algorithm. The initial model had $m = 5$ components but otherwise was random. We compared the structure of the learned model with the generative model and computed the likelihoods of both the learned and the original model on a test dataset consisting of 1000 points.

The results on retrieving the original trees were excellent: out of 10 trials, the algorithm failed to retrieve correctly only 1 tree in 1 trial. This result can be accounted for by sampling noise; the tree that wasn't recovered had a λ of only 0.02. Instead of recovering the missing tree, the algorithm fit two identical trees to the generating tree with the highest λ. The difference between the log likelihood of the samples of the generating model and the approximating model was 0.41 bits per example. On all the correctly recovered trees, the approximating mixture had a higher log likelihood for the sample set than the generating distribution. This shows that not only the structure but the whole distribution was recovered correctly.

7.1.2 Random bars, small data set

The "bars" problem is a benchmark structure learning problem in neural network unsupervised learning algorithms. It has many variants, of which I describe the one used in the present experiments.

The domain V is the $l \times l$ square of binary variables depicted in figure 7-1. The data are generated the following manner: first, one flips a fair coin to decide whether to generate horizontal or vertical bars; this represents the hidden variable in our model. Then, each of the l bars is turned on independently (black in figure 7-1) with probability p_b. Finally, noise is added by flipping each bit of the image independently with probability p_n.

A learner is shown data generated by this process; the task of the learner is to "discover" the data generating mechanism.

What would it mean to discover the data generating mechanism with a mixture of trees? A mixture of trees model that approximates the true structure (figure 7-2) for low levels of noise is shown in figure 7-3. Note that any tree over the variables forming a bar is an equally good approximation. Thus, we will consider that structure has been discovered when the model learns a mixture with $m = 2$, each T^k having l connected components, one for each bar. Additionally, we shall test the "classification accuracy" of the learned model

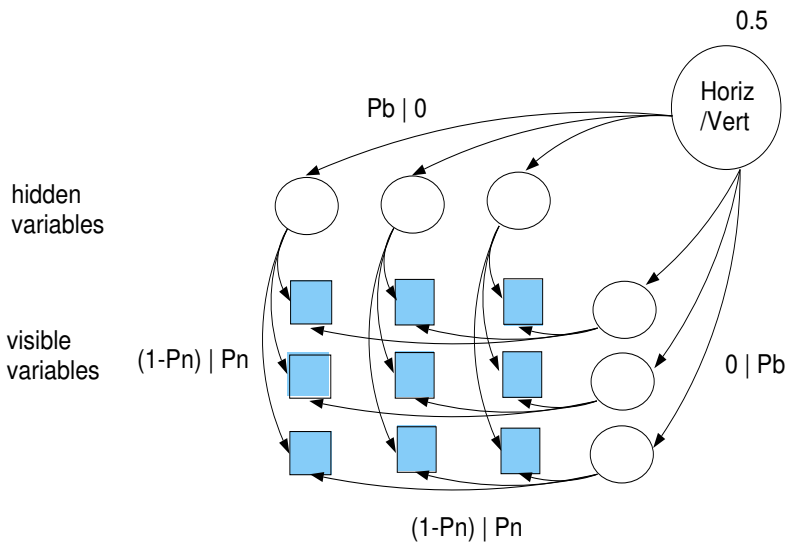


Figure 7-2: The true structure of the probabilistic generative model for the bars data.

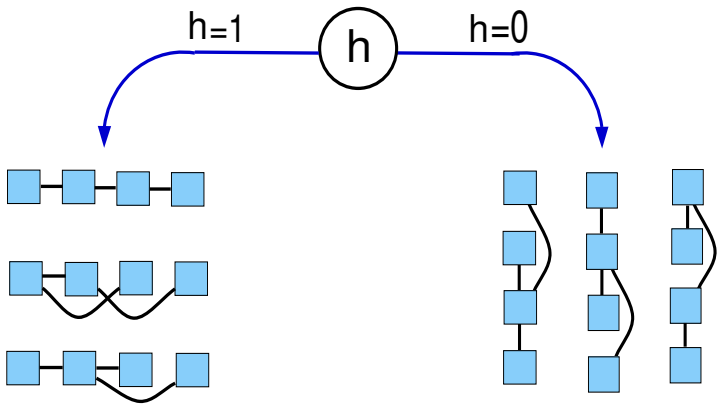


Figure 7-3: A mixture of trees approximate generative model for the bars problem. The interconnection between the variables in each "bar" are arbitrary.

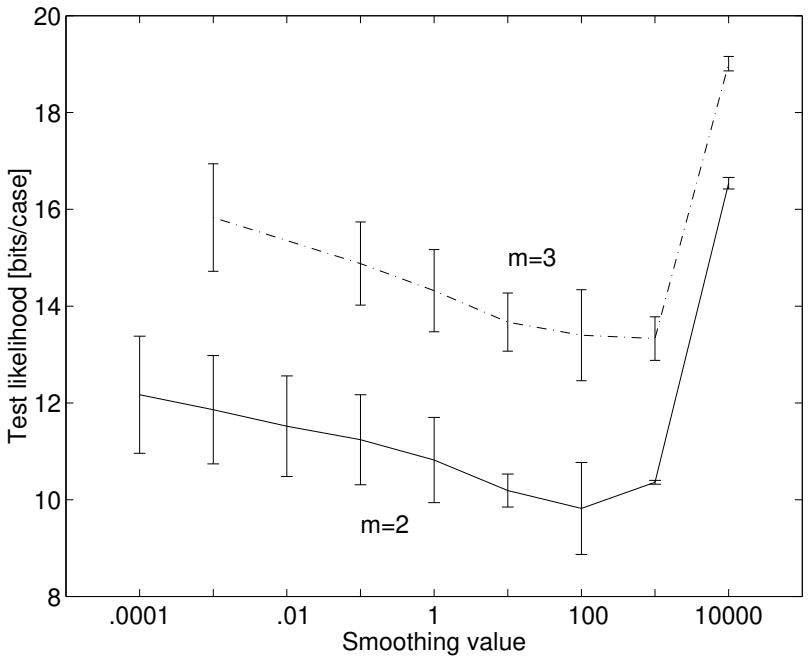


Figure 7-4: Test set log-likelihood on the bars learning task for different values of the smoothing α and different m. Averages and standard deviations over 20 trials.

by comparing the true value of the hidden variable (i.e. "horizontal" or "vertical") with the value estimated by the model for each data point in a test set.

We do not compare all of our results directly against neural network results, mainly because most of the published results are qualitative, focusing on whether the model is able to learn the structure of the data. We closely replicate the experiment described in [15] in a way that makes the present experiment slightly more difficult than theirs. The training set size is $N_{train} = 400$ (in [15] it is 500), the data set contains ambiguous examples (examples where no bars are present, which consequently are ambiguous from the point of view of classification) and the noise level is small but not zero (it is zero in [15]). Because the other authors studying this problem assume incomplete knowledge of structure (in this case, the number of hidden units of the neural net), we do something similar: we train models with $m = 2, 3, \ldots$ and choose the final model by the likelihood on a holdout set. Typical values for l in the literature are $l = 4, 5$ (the problem becomes harder with increasing l); we choose $l = 5$ following [15]. The probabilities p_b and p_n are 0.2 and 0.02 respectively. The test set size is $N_{test} = 200$. To obtain trees with several connected components we use a small edge penalty $\beta = 5$. Because the data set is small smoothing with different values for α is used. For each value of m we run the **MixTree** learning algorithm 20 times on the same training set with different random initial points and average the results.

Selection of m The average test set log-likelihoods (in bits) for $m = 2, 3$ are given in figure 7-4, together with their standard deviations. Clearnly, $m = 2$ is the best model.

Structure recovery. For $m = 2$ we examined the resulting structures: in 19 out of 20 trials, structure recovery was perfect (in the sense explained above); in the remaining trial the learned model missed the correct structure altogether, converging to a different local minimum apparently closer to its initial point. For comparison, [15] examined two training methods and the structure was recovered in 27 and respectively 69 cases out of 100. This

Table 7.1: Results on the bars learning task.

Test set	ambiguous	unambiguous
l_{test} [bits/datapt]	9.82 ± 0.95	13.67 ± 0.60
Class accuracy	0.852 ± 0.076	0.951 ± 0.006

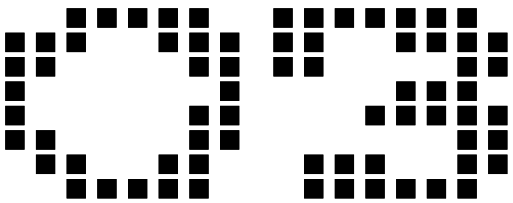


Figure 7-5: An example of a digit pair.

result held for the whole range of the smoothing parameter α.

Classification. The classification performance is shown in table 7.1. The result reported is obtained for a value of α chosen based on the test set log-likelihood. Note that on data generated by the previously described mechanism, no model can achieve perfect classification performance. This is due to the "ambiguous" examples, like the one shown in figure 7-1 (upper row, third from left), where no bars are on (or, less likely, where all bars are on). The probability of an ambiguous example for the current value of p_b is

$$p_{ambig} \;=\; p_b^l + (1-p_b)^l \;=\; 0.25 \tag{7.2}$$

Since ambiguous examples appear in either class with equal probability, the error rate caused by them is $0.5 p_{ambig} = 0.125$. Comparing this theoretical upper bound with the value in the corresponding column of table 7.1 shows that the model has indeed a very good classification performance, even trained on ambiguous examples.

To further support this conclusion, a second test set of size 200 was generated, this time including only non-ambiguous examples. The classification performance, shown in the corresponding section of table 7.1 rose to 0.95. The table also shows the likelihood of the (test) data given the learned model. For the first, "ambiguous" test set, this is 9.82, 1.67 bits away from the true model entropy of 8.15 bits/data point. For the "non-ambiguous" test set, the compression rate is significantly worse, no surprise since the distribution of the test set is now different from the distribution the model was trained on.

7.2 Density estimation experiments

7.2.1 Digits and digit pairs images

We tested the mixture of trees as a density estimator by running it on a subset of binary vector representations of handwritten digits and measuring the average log-likelihood. The datasets consist of normalized and quantized 8x8 binary images of handwritten digits made available by the US Postal Service Office for Advanced Technology. One data set (that we call "digits") contained images of single digits in 64 dimensions, the second (called "pairs" hitherto) contained 128 dimensional vectors representing randomly paired digit images. Figure 7-5 displays an example of a digit pair. The training, validation and test

Table 7.2: Average log-likelihood (bits per digit) for the single digit (Digit) and double digit (Pairs) datasets. Boldface marks the best performance on each dataset. Results are averaged over 3 runs.

m	Digits	Pairs
16	**34.72**	79.25
32	34.48	**78.99**
64	34.84	79.70
128	34.88	81.26

set contained 6000, 2000, and 5000 exemplars respectively. The data sets, the training conditions and the algorithms we compared with are described in [21]. Each model is trained on the same training set until the likelihood of the validation set stops increasing. We tried mixtures of 16, 32, 64 and 128 trees, fitted by the basic algorithm. For each of the digits and pairs datasets we chose the mixture model with the highest log-likelihood on the validation set and using it we calculated the average log-likelihood over the test set (in bits per example). The averages (over 3 runs) are shown in table 7.2. Notice the small difference in test likelihood between models that is due to early stopping in the training algorithm.

In figure 7-6 we compare our result (for $m = 32$) with the results published by [21].The other algorithms plotted in the figure are the mixture of factorial distributions (MF), the completely factored model (which assumes that every variable is independent of all the others) called "Base rate" (BR), the Helmholtz Machine trained by the wake-sleep algorithm [21] (HWS), the same Helmholtz Machine where a mean field approximation was used for training (HMF) and a fully visible and fully connected sigmoid belief network (FV). Table 7.2 displays the performances of all the mixture of trees models that we tested.

The results are very good: the mixture of trees is the absolute winner for compressing the simple digits and comes in second as a model for pairs of digits. A comparison of particular interest is the comparison in performance between the mixture of trees and the mixture of factored distribution. In spite of the structural similarities, the mixture of trees performs significantly better than the mixture of factorial distribution indicating that there exists some structure that is exploited by the mixture of spanning trees but can't be captured by a mixture of independent variable models. Comparing the values of the average likelihood in the mixture of trees model for digits and pairs we see that the second is more than twice the first. This suggests that our model (just like the mixture of factored distributions) is able to perform good compression of the digit data but is unable to discover the independence in the double digit set.

7.2.2 The ALARM network and data set

The second set of density estimation experiments features the ALARM network [33, 6]. This Bayes net model is a benchmark model for structure learning experiments. It is a medical diagnostic alarm message system for patient monitoring and it was constructed from expert knowledge. The domain has $n = 37$ discrete variables taking between 2 and 4 values, connected by 46 directed arcs. Since the ALARM net was not constructed from a complete data set I use in the experiments data generated from the model itself. This is artificial data, but it is an interesting experiment for two reasons: (1) the data generating

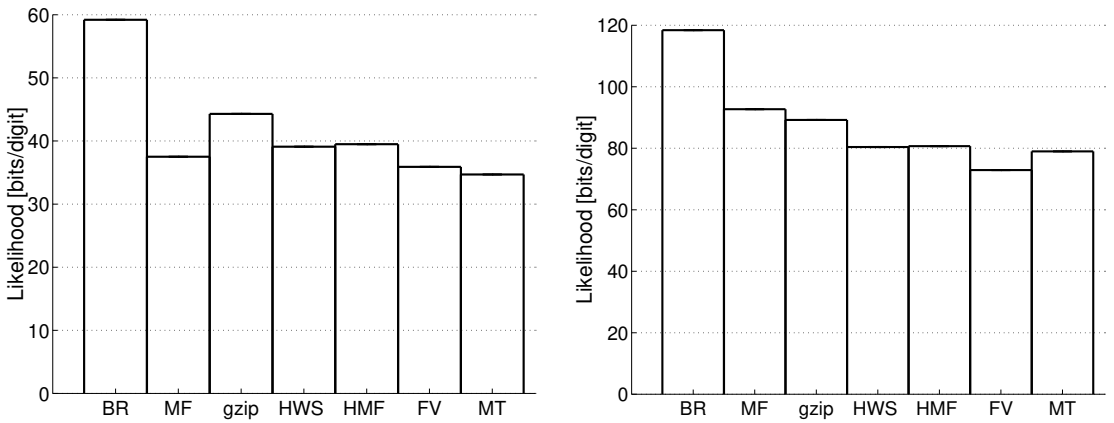


Figure 7-6: Average log-likelihoods (bits per digit) for the single digit (a) and double digit (b) datasets. MT is a mixture of spanning trees, MF is a mixture of factorial distributions, BR is the base rate model, HWS is a Helmholtz machine trained by the Wake-Sleep algorithm, HMF is a Helmholtz machine trained using the Mean Field approximation, FV is fully visible fully connected sigmoidal Bayes net. Notice the difference in scale between the two figures.

distribution is *not* a tree or a mixture of trees (17 nodes have more than one parent) but the topology of the DAG is sparse suggesting that the dependence can be approximated by a mixture of trees with a small number of components m; (2) the generative model captures the features of a real domain of practical importance.

We generated a training set having $N_{train} = 10,000$ data points and a separate test set of $N_{test} = 2,000$ data points. On these sets we train and compare the following models: mixtures of trees, mixtures of factorial distributions, the true model (without training), gzip (without training). For mixtures of trees and factorial distributions, the comparison is made in the following way: we separate a validation set of $N_{valid} = 1,000$ data points from the training set; then, models with different values of m are trained on the remaining data and evaluated (after the training process converges) on the validation set. For each value of m we run 20 trials. The averaged value of the log-likelihood on the validation set is used to choose the optimal m. Then, the models with that m are evaluated on the test set and the likelihood achieved on it is compared against other classes of models.

The results are presented in table 7.3. The result reported for gzip was obtained by writing the data in a file in binary format, with all additional information removed and presenting it to the gzip program.

One can see that, even with a 9,000 data set, the mixture of trees does not approximate the true distribution perfectly. However, note the difference in likelihood values for the true model between the training and the test set, suggesting that a data set in the thousands is still not a large data set for this domain. Among the other models used to approximate the true distribution, the mixture of trees is a clear winner. If being ahead of gzip and the base rate model was expected, it was not so with the mixture of factorials. However, the superiority of the mixture of trees w.r.t this model is also clear.

To examine the sensitivity of the algorithms to the size of the data set also ran the experiment with a training set of size 1,000. The results are presented in table 7.4.

Again, the mixture of trees is the closest to the true model, beatingthe mixture of

Table 7.3: Density estimation results for the mixtures of trees and other models on the ALARM data set. Training set size $N_{train} = 10,000$. Average and standard deviation over 20 trials.

Model	Train likelihood [bits/data point]	Test likelihood [bits/data point]
ALARM net	13.148	13.264
Mixture of trees $m = 18$	13.51 $\pm$0.04	14.55 $\pm$ 0.06
Mixture of factorials $m = 28$	17.11 $\pm$ 0.12	17.64 $\pm$ 0.09
Base rate	30.99	31.17
gzip	40.345	41.260

Table 7.4: Density estimation results for the mixtures of trees and other models on a data set of size 1000 generated from the ALARM network. Average and standard deviation over 20 trials.

Model	Train likelihood [bits/data point]	Test likelihood [bits/data point]
ALARM net	13.167	13.264
Mixture of trees $m = 2$, $\alpha = 50$	14.56 $\pm$0.16	15.51 $\pm$ 0.11
Mixture of factorials $m = 12$, $\alpha = 100$	18.20 $\pm$ 0.37	19.99 $\pm$ 0.49
Base rate	31.23 $\pm$ 0.21	31.18 $\pm$ 0.01
gzip	45.960	46.072

factorials by an even larger margin than for the large data set. Notice that the degradation in performance for the mixture of trees is relatively mild (about 1 bit), whereas the model complexity is reduced drastically (from 18 to only 2 mixture components, and this in the context of additional parameter smoothing!). This indicates the important role played by the tree structures in fitting the data and motivates the advantage of the mixture of trees over the mixture of factorials for this data set.

7.3 Classification with mixtures of trees

7.3.1 Using a mixture of trees as a classifier

Classification is a common and important task for which probabilistic models are often used. Very often density models are trained on data with the sole purpose of classification. This procedure is not asymptotically optimal, but there is empirical evidence that it can offer good performance in practice. Therefore, this section will be devoted to experimentally assessing the performance of the mixture of trees model in classification tasks.

A density estimator can be turned into a classifier in two ways, both of them being essentially *likelihood ratio* methods. Denote the class variable by c and the set of input variables by V. In the first method, adopted in our classification experiments under the name of *mixture of trees classifier*, one (mixture of trees) model Q is trained on the domain $\{c\} \bigcup V$, treating the class variable like any other variable and pooling all the training data together. In the testing phase, a new instance $x \in \Omega(V)$ is classified by picking the most likely value of the class variable given the other variables' settings.

$$c(x) \;=\; \operatorname*{argmax}_{x_c} Q(x_c, x) \tag{7.3}$$

The second method calls for partitioning the training set according to the values of the class variable and for training a density estimator on each partition. This is equivalent with training a mixture of trees with visible choice variable, the choice variable being the class c. This method was first used by [7]; if the trees are forced to have the same structure we obtain the Tree Augmented Naive Bayes (TANB) classifier of [24]. To classify a new instance x one turns to Bayes formula

$$c(x) \;=\; \operatorname*{argmax}_{k} P[c = k] T^k(x) \tag{7.4}$$

7.3.2 The AUSTRALIAN data set

We investigated the performance of mixtures of trees on three classification tasks from the UCI repository [1]. In the first experiment, the data set was the AUSTRALIAN database [1]. It has 690 examples each consisting of 14 attributes and a binary class variable. Six of the attributes (numbers 2, 3, 7, 10, 13 and 14) were real-valued and they were discretized into 4 (for attribute 10) respectively 5 (for all other attributes) roughly equally sized bins. In our experiments, in order to compare with [41] the test and training set sizes were 70 and 620 respectively (a ratio of roughly 1/9). For each value of m that was tested we ran our algorithm for a fixed number of epochs on the training set and then recorded the performance on the test set. This was repeated 20 times for each m, each time with a random start and with a random split between the test and the training set. In a first series of runs all training parameters were fixed except for m. In the second series of runs, we let β,

Table 7.5: Performance comparison between the mixture of trees model and other classification methods on the AUSTRALIAN dataset. The results for mixtures of factorial distribution are those reported in [41]. All the other results are from [48].

Method	**% correct**	**Method**	**% correct**
Mixture of trees $m = 20$, $\beta = 4$	**87.8**	Backprop	84.6
Mixture of factorial distributions (D-SIDE in [41])	87.2	C4.5	84.6
Cal5	86.9	SMART	84.2
ITrule	86.3	Bayes Trees	82.9
Logistic discrimination	85.9	K-nearest neighbor	81.9
Linear discrimination	85.9	AC2	81.9
DIPOL92	85.9	NewID	81.9
Radial basis functions	85.5	LVQ	80.3
CART	85.5	ALLOC80	79.9
CASTLE	85.2	CN2	79.6
Naive Bayes	84.9	Quadratic discrimination	79.3
IndCART	84.8	Flexible Bayes	78.3

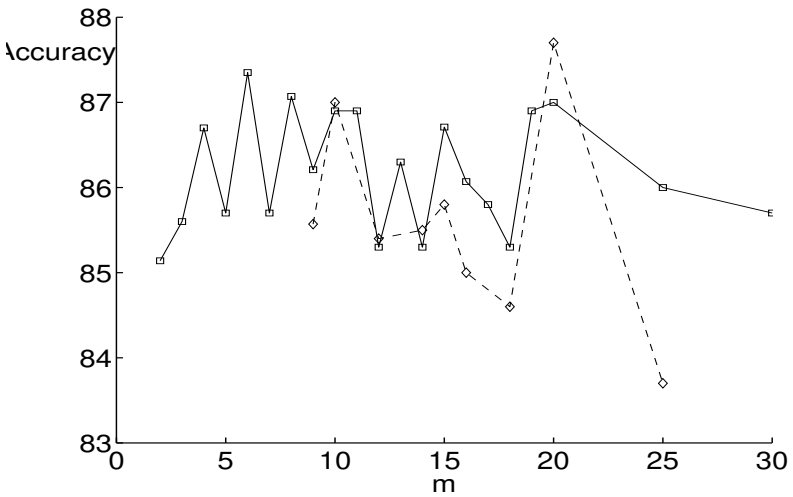


Figure 7-7: Classification performance of different mixture of trees models on the Australian Credit dataset. Results represent the percentage of correct classifications, averaged over 20 runs of the algorithm.

Table 7.6: Performance of mixture of trees models on the MUSHROOM dataset. m=10 for all models.

Algorithm	**Correctly classified**	**Soft class** $\sum \gamma_{\text{true}}/N$	**Test likelih. (bits/datapoint)**	**Train likelih. (bits/data point)**
No smoothing	.998	.997	27.63	27.09
Smooth w/ marginal $\alpha_M = 0.3,\ \alpha_P = 20$	1	.9999	27.03	26.97
Smooth w/ uniform $\alpha_M = 0.3,\ \alpha_P = 200$	1	.9997	27.39	27.02

the edge pruning parameter, change in such a way as to keep $m\beta$ approximatively constant. This results in roughly the same fraction of pruned edges independently of the number of components. The results, which are slightly better than in the previous experiment, are presented in figure 7-7. What is common to both series of experiments is the relatively large set of values of m for which the performance stays at the top of its range. We hypothesize that this is caused by the multiple ways the models are smoothed: edge pruning, smoothing with the marginals and early stopping.

The best performance of the mixtures of trees in the second case is compared to other published results for the same dataset in table 7.5. For comparison, correct classification rates obtained and cited in [41] on training/test sets of the same size are: 87.2% for mixtures of factorial distributions and 86.9% for the next best model (a decision tree called Cal5). The full description of the other methods can be found in [41, 48]. It can be seen that on this dataset the mixture of trees achieves the best performance, followed by the mixture of factorial distributions from [41].

7.3.3 The MUSHROOM data set

The second data set used was the MUSHROOM database [56]. This data set has 8124 instances of 23 discrete attributes (including the class variable, which is treated like any other attribute for the purpose of model learning). The training set comprised 6000 randomly chosen examples, and the test set was formed by the remaining 2124. The smoothing methods used were a) a penalty α_P on the entropy of the mixture variable and b) smoothing with the marginal according to (7.1) or with a uniform distribution. The smoothing coefficient α_M was divided between the mixture components proportionally to $1/\Gamma_k$. For this dataset, smoothing was effective both in reducing overfitting and in improving classification performance. The results are shown in table 7.6. The soft classification colums expresses an integrated measure of the confidence of the classifier. Note that besides the classification being correct, the classifier also has achieved high confidence.

7.3.4 The SPLICE data set. Classification and structure discovery

The last task was the classification of DNA SPLICE-junctions. The domain consists of 60 variables, representing a sequence of DNA bases, and an additional class variable. The

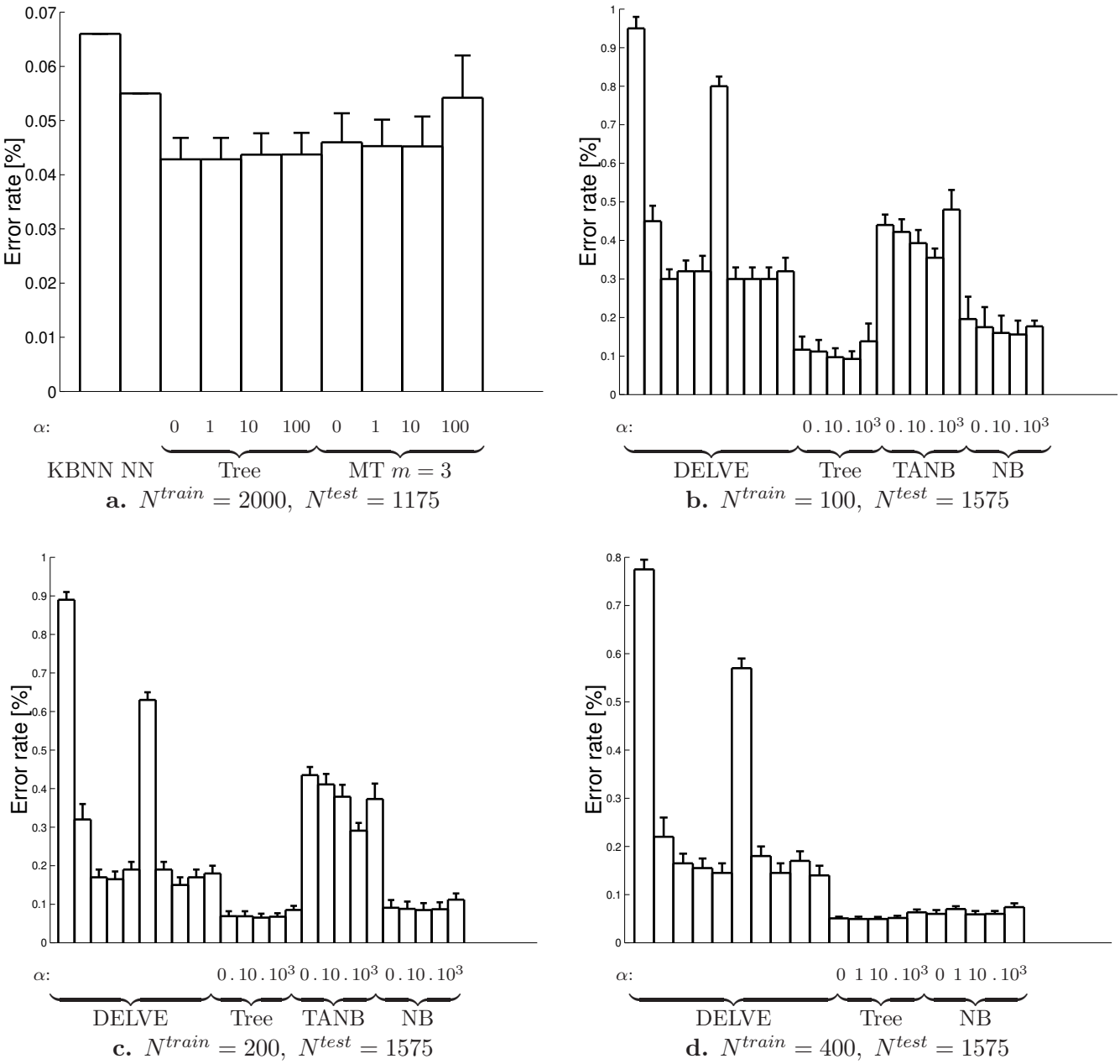


Figure 7-8: Comparison of classification performance of the mixture of trees and other models on the SPLICE data set. The models tested by DELVE are, from left to right: 1-nearest neighbor, CART, HME (hierarchical mixture of experts)-ensemble learning, HME-early stopping, HME-grown, K-nearest neighbors, Linear least squares, Linear least squares ensemble learning, ME (mixture of experts)-ensemble learning, ME-early stopping. TANB is the Tree Augmented Naive Bayes classifier of [24], NB is the Naive Bayes classifier, Tree represents a mixture of trees with $m = 1$, MT is a mixture of trees with $m = 3$. KBNN is the Knowledge based neural net, NN is a neural net.

task is to determine if the middle of the sequence is a splice junction and what is its type[2]. Hence, the class variable can take 3 values (EI, IE or no junction) and the other variables take 4 values corresponding to the 4 possible DNA bases coded here by the symbols (C, A, G, T). The data set consists of 31175 labeled examples. We compared the Mixture of trees model with two categories of classifiers and thus we performed two series of experiments. A third experiment involving the SPLICE data set will be described later.

For the first series, we compared our model's performance against the reported results of [65] and [52] who used multilayer neural networks and knowledge-based neural networks for the same task. The sizes of the training set and of the test set are reproduced from the above cited papers; they are 2000 and 1175 examples respectively. We constructed trees ($m = 1$) and mixtures of $m = 3$ trees with different smoothing values α. Fitting the single tree can be done in 1 step. To fit the mixture, we separated N_{valid}=300 examples out of the training set and learned the model using the EM algorithm on the remaining 1700. The training was stopped when the likelihood of the validation set stopped decreasing. This can be regarded as an a additional smoothing for the $m = 3$ model. The results, averaged over 20 trials, are presented in figure 7-8,a. It can be seen that the tree and the mixture of trees model perform very similarly, with the single tree showing an insignificantly better classification accuracy. Since a single tree has about three times fewer parameters than a mixture with $m = 3$ we strongly prefer the former model for this task. Notice also that in this situation smoothing does not improve performance; this is not unexpected since the data set is relatively large. With the exception of the "oversmoothed" mixture of trees model ($\alpha = 100$) all the trees/mixture of trees models significantly outperform the other models tested on this problem. Note that whereas the mixture of trees contains no prior knowledge about the domain, the other two models do: the neural network model is trained in supervised mode, optimizing for class accuracy, and the KBNN includes detailed domain knowledge before the training begins.

The second set of experiments pursued a comparison with benchmark experiments on the SPLICE data set that are part of the DELVE repository [54]. The DELVE benchmark uses subsets of the SPLICE database with 100, 200 and 400 examples for training. Testing is done on 1500 examples in all cases. The algorithms tested by DELVE and their performances are shown in figure 7-8,b,c, and d. We fitted single trees ($m = 1$) with different degrees of smoothing. We also learned naive Bayes (NB) and Tree Augmented Naive Bayes (TANB) models [24]. A NB or a TANB model can be fit in one step, just like a tree. According to the DELVE protocol, we ran our algorithms 20 times with different random initializations on the same training and testing sets.

The results are presented in figures 7-8b,c and d. Most striking in these plots is the dramatic difference between the methods in DELVE and the classification obtained by a simple tree: in all cases the error rates of the tree models are *less than half* of the performance of the best model tested in DELVE. Moreover, the average error of a single tree trained on 400 examples is 5.5%, which is only 1.2% away from the average error of trees trained on the 2000 examples dataset. The explanation for this remarkable accuracy preservation with the decrease of the number of examples is discussed later in this section. The Naive Bayes model exhibits a behavior that is very similar to the tree model and only slightly less accurate. However, augmenting the Naive Bayes model to a TANN significantly hurts

[2]The DNA is composed of sections that are useful in coding proteins, called *exons*, and of inserted sections of non-coding material called *introns*. Splice junctions are junctions between an exon and an intron and they are of two types: exon-intron (EI) represents the end of an exon and the beginning of an intron whereas intron-exon (IE) is the places where the intron ends and the next exon, or coding section, begins.

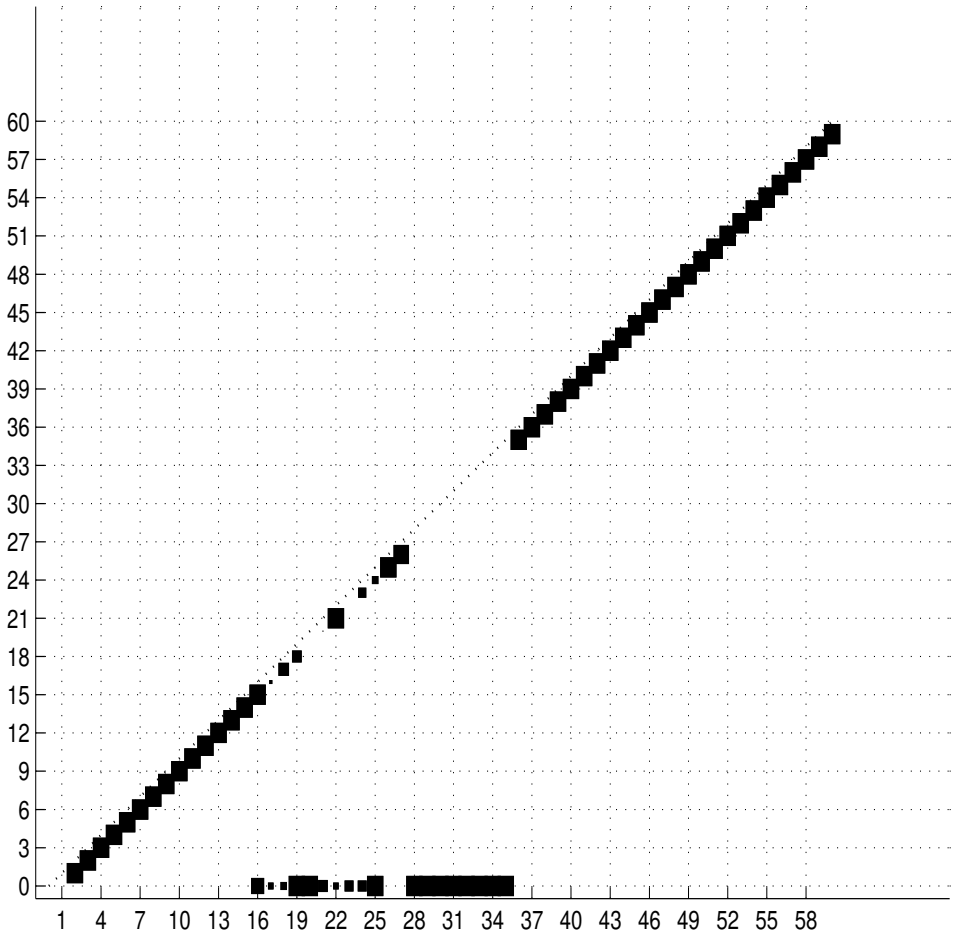


Figure 7-9: Cumulative adjacency matrix of 20 trees fit to 2000 examples of the SPLICE data set with no smoothing. The size of the square at coordinates ij represents the number of trees (out of 20) that have an edge between variables i and j. No square means that this number is 0. Only the lower half of the matrix is shown. The class is variable 0. The group of squares at the bottom of the figure shows the variables that are connected directly to the class. Only these variable are relevant for classification. Not surprisingly, they are all located in the vicinity of the splice junction (which is between 30 and 31). The subdiagonal "chain" shows that the rest of the variables are connected to their immediate neighbors. Its lower-left end is edge 2–1 and its upper-right is edge 60-59.

EI junction

	Exon			Intron					
	28	29	30	31	32	33	34	35	36
Tree	CA	**A**	**G**	**G**	**T**	AG	A	G	
True	**CA**	**A**	**G**	**G**	**T**	AG	A	G	T

IE junction

	Intron									Exon
	15	16	...	25	26	27	28	29	30	31
Tree	–	CT	CT	CT	–	–	CT	**A**	**G**	**G**
True	CT	CT	CT	CT	–	–	CT	**A**	**G**	**G**

Figure 7-10: The encoding of the IE and EI splice junctions as discovered by the tree learning algorithm compared to the ones given in J.D. Watson & al., "Molecular Biology of the Gene"[67]. Positions in the sequence are consistent with our variable numbering: thus the splice junction is situated between positions 30 and 31. Symbols in boldface indicate bases that are present with probability almost 1, other A,C,G,T symbols indicate bases or groups of bases that have high probability (>0.8), and a – indicates that the position can be occupied by any base with a non-negligible probability.

the classification performance. The plots also allow us to observe the effect of the degree of smoothing when it varies from none to very large. In contrast to the previous experiment on SPLICE data, here smoothing has a beneficial effect on the classification accuracy for values under a certain threshold; for larger values the accuracy is strongly degraded by smoothing. These accuracy profiles can be observed in tree models as well as in the TANN and Naive Bayes models and they are similar to the bias-variance tradeoff curves commonly encountered in machine learning applications. Also not surprisingly, the effect diminishes with the increasing size of the data set (and is almost invisible for a training set size of 400).

Discovering structure. Figure 7-9 presents a summary of the tree structures learned from the N=2000 example set in the form of a cumulated adjacency matrix. The adjacency matrices of the 20 graph structures obtained in the experiment have been summed[3]. The size of the black square at coordinates i, j in the figure is proportional to the value of the i, j-th element of the cumulated adjacency matrix. No square means that the respective element is 0. Since the adjacency matrix is symmetric, only half the matrix is shown. From figure 7-9 we can see first that the tree structure is very stable over the 20 trials. Variable 0 represents the class variable; the hypothetic splice junction is situated between variables 30 and 31. The figure shows that the splice junction (variable 0) depends only on DNA sites that are in its vicinity. The sites that are remote from the splice junction are dependent on their immediate neighbors. Moreover, examining the tree parameters, for the edges adjacent to the class variable, we observe that these variables build certain patterns when the splice junction is present, but are random and almost uniformly distributed in the absence of a splice junction. The patterns extracted from the learned trees are shown in figure 7-10. The same figure displays the "true" encodings of the IE and EI junctions

[3]The reader is reminded that the adjacency matrix of a graph has a 1 in position ij if the graph has an edge connecting vertices i and j and a 0 otherwise.

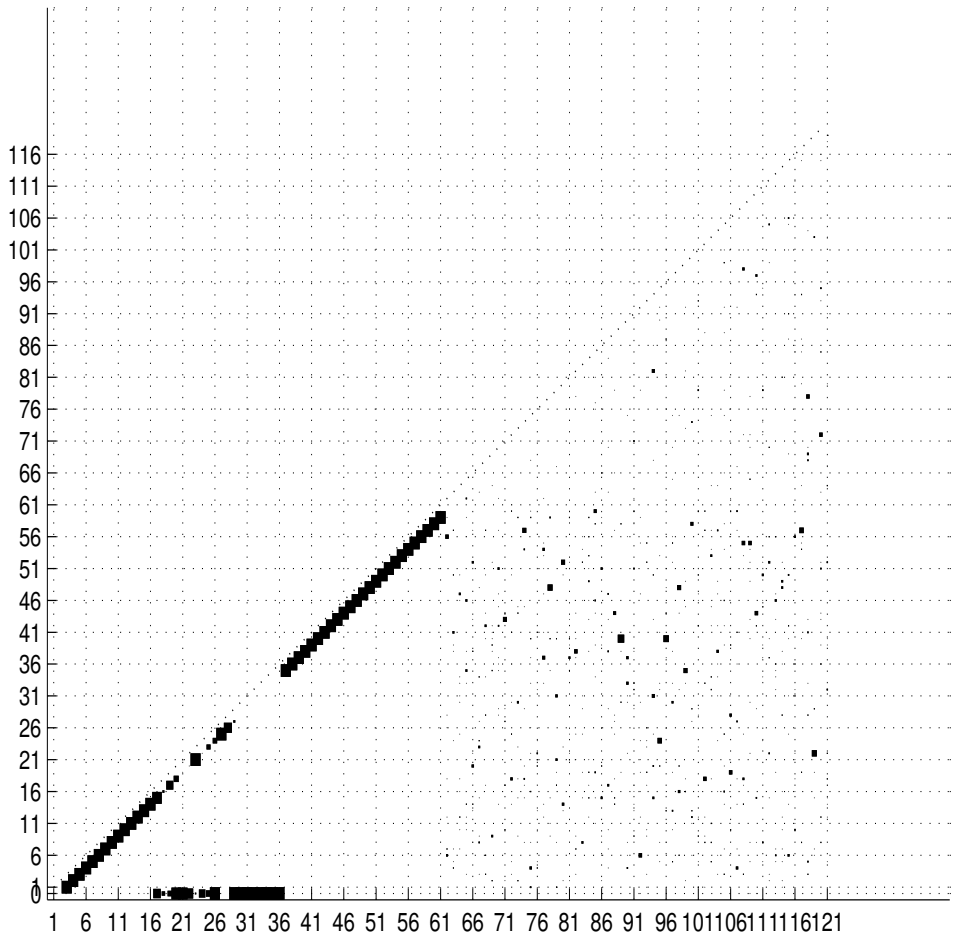


Figure 7-11: The cumulated adjacency matrix for 20 trees over the original set of variables (0-60) augmented with 60 "noisy" variables (61-120) that are independent of the original ones. The matrix shows that the tree structure over the original variables is preserved.

as given in [67]. The match between the two encodings is almost perfect. Thus, we can conclude that for this domain, the tree model not only provides a good classifier but also discovers the true underlying model of the physical reality that generates the data. Remark also that the algorithm arrives at this result *without any prior knowledge*: it does not know which variable is the class and it doesn't even know that 60 variables out of 61 are in a sequence.

7.3.5 The single tree classifier as an automatic feature selector

.

Let us examine the single tree classifier that was used for the SPLICE data set more closely. According to the separation properties of the tree distribution, the probability of the class variables depends only on its neighbors, that is, on the variables to which the class variable is connected by tree edges. Hence, a tree acts as an implicit variable selector for classification: only the variables adjacent to the queried variable (this set of variables is called the *Markov blanket* [53]) will be relevant for determining its probability distribution. This property also explains the observed preservation of the accuracy of the tree classifier when the size of the training set decreases: out of the 60 variables, only 18 are relevant to the class; moreover, the dependence is parametrized as 18 independent pairwise probability tables $T_{class,v}$. Such parameters can be accurately fit from relatively few examples. Hence, as long as the training set contains enough data to establish the correct dependency structure, the classification accuracy will degrade slowly with the decrease in the size of the data set.

Now we can explain the superiority of the tree classifier over the models in DELVE in the previous example: out of the 60 variables, only 17 are relevant to the class. The tree finds them correctly. a classifier that is not able to perform a reasonable feature selection will be hindered by the remaining irrelevant variables, especially if the training set is small.

For a given Markov blanket, the tree classifier classifies in the same way as a naive Bayes model with the Markov blanket variables as inputs. Remark also that the naive Bayes model itself has a built-in feature selector: if one of the input variables v is not relevant to the class, the distributions $P_{v|c}$ will be roughly the same for all values of c. Consequently, in the posterior $P_{c|v}$ that serves for classification, the factors corresponding to v will simplify and thus v will have (almost) no influence on the classification. This explains why the naive Bayes model also performs well on the SPLICE classification task. Notice however that the variable selection mechanisms implemented by the tree classifier and the naive Bayes classifier are not the same.

Sensitivity to irrelevant features. To verify that indeed the single tree classifier acts like a feature selector, we performed the following experiment, using again the SPLICE data. We augmented the variable set with another 60 variables, each taking 4 values with randomly and independently assigned probabilities. The rest of the experimental conditions (training set, test set and number of random restarts) were identical to the first SPLICE experiment. We fitted a set of models with $m = 1$, a small $beta = 0.1$ and no smoothing and compared its structure and performance to the corresponding model set in the first experiment series. The structure of the new models, in the form of a cumulative adjacency matrix, is shown in figure 7-11. We see that the structure over the original 61 variables is unchanged and stable; the 60 noise variables connect in a random uniform patterns to the original variables and among each other. As expected after examining the structure, the classification performance of the new trees is not affected by the newly introduced variables: in fact the average accuracy of the trees over 121 variables is 95.8%, 0.1% higher than the

accuracy of the original trees[4].

[4]The standard deviation of the accuracy is 3.5% making this difference insignificant.

Chapter 8

Conclusion

The subject of this thesis has been the tree as a tool for statistical multivariate modeling. A tree, like any graphical model, has the ability to express the dependencies between variables (by means of the graph) separately from the detailed form of these dependencies (contained in the parameters), a property that makes it an excellent support for human intuition and allows the design of general inference and learning algorithms.

Trees are simple models: this is especially evident when one examines the algorithm that fits a tree to a given distribution. All the information about the target distribution that a tree can capture is contained in a small number (at most $n-1$) pairwise marginals. Simplicity leads to computational efficiency. Efficient inference is a direct consequence of the fact that trees are decomposable models with small clique size, but the existence of an efficient algorithm for learning the tree structure is a property that makes trees quasi-unique in the graphical models family. As this thesis shows, the limitation in the trees' modeling power can be overcome by combining them into mixtures. The number of trees in the mixture is a smoothing parameter that allows one to control the complexity of the resulting model.

From the point of view of their ability to graphically represent independencies, mixtures are not strictly speaking graphical models because the independence relationships between the visible variables cannot be determined from the graph topology alone. In other words, the dependency structure is not separated from the parametrization as it is for other classes of belief networks. But from a computational point of view, mixtures of trees are tributary to the belief network perspective and methods. They inherit the computational properties of trees, both in terms of inference and of learning from data, to such an extent that in fact all mixtures of trees algorithms in this thesis are combinations and modifications of the corresponding algorithms for trees.

The learning algorithm for mixtures of trees introduced here is a version of the well known EM algorithm. As such, it converges to a local optimum only. This property is a characteristic of mixtures in general, rather than of the present model. It is also an illustration of the fact that in the search for the best model structure there is no free lunch: just as learning optimal graphical model structure requires brute-force search, learning mixtures suffers from the local optima problem.

Therefore, it is sensible to ask: when should one class of models be chosen instead of the other? A rule suggested by the above discussion on structure search is: Use graphical models for those problems where there is significant knowledge about the dependency structure of the data, so that the structure search is minimal or not necessary; when there is little or no prior knowledge about the structure of the dependencies take mixtures of trees as the preferred candidate. And if the structure is unknown, but simple, like in the splice junction example, a single tree may suffice for the task. The reader should keep in mind however

that the final answer will depend on the properties of the actual data.

The thesis has also examined the use of the mixture of trees learning algorithm as a heuristic method for hidden variable discovery. The research is motivated in part by the cases in science and everyday life when identifying a hidden variable enables one, by providing a computationally simple model, to change one's way of thinking (like diseases, hidden variables that are at the core of medical knowledge). A second motivation was seeing this problem as an instance of a more general divide and conquer approach to the problem of structure learning. Because structure search requires model selection and model validation, this thesis has explored these topics as well. It has devised and tested as a model selection criterion an empirical measure of a model's description length. On the model validation side it has introduced a new distribution free independence test.

Making the learning scale better with the dimensionality of the domain by taking advantage of certain properties of the data is the last, but not the least of the research topics that this work is concerned with. The property it was built onto is data sparsity, loosely defined as low entropy in the marginal distributions of the individual variables. The algorithms introduced reduced running time by up to 3 orders of magnitude, mainly by intelligently reorganizing the data in a manner that exploits sparsity. This underlines the importance of combining the graphical models perspective with other data mining techniques in enabling us to conquer the high dimensional domains that exist out there.

After devoting a whole thesis to trees, one may want to ask: are they unique in the properties and advantages demonstrated here? Are there other classes of graphical models that can be learned efficiently from data? There is one other such class. It is the class of Bayes nets for which the variables' topological ordering is fixed and the number of parents of each node is bounded by l. For this class the optimal model structure for a given target distribution can be found in $\mathcal{O}(n^{l+1})$ time by a greedy algorithm. These models and trees are instances of *matroids* [69]. The matroid is the unique algebraic structure for which the "Maximum Weight" problem, corresponding to the Maximum Weight Spanning Tree problem, is solved optimally by a greedy algorithm. Therefore, any class of graphical models that were also matroids would have a structure learning algorithm that is better than brute-force. Finding such classes is an open problem.

By introducing and studying the mixture of trees, this thesis has extended the scope of graphical modeling for unsupervised learning. At the same time, it has aimed to demonstrate that the potential of this exciting field of research is far from being exhausted.

Bibliography

[1] U. C. Irvine Machine Learning Repository. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/.

[2] Ann Becker and Dan Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *UAI 96 Proceedings*, 1996.

[3] C. Berrou and A. Glavieux. Near-optimum error correcting coding and decoding Turbo codes. *IEEE Transactions on Communications*, 44:1261–1271, 1996.

[4] P. Brucker. On the complexity of clustering algorithms. In R. Henn, B. Corte, and W. Oletti, editors, *Optimierung und Operations Research*, Lecture Notes in Economics and Mathematical Systems, pages 44–55. Springer Verlag, 1978.

[5] Peter Cheeseman and John Stutz. *Bayesian classification (AutoClass): Theory and results*, pages 153–180. AAAI Press, 1995.

[6] Jie Cheng, David A. Bell, and Weiru Liu. Learning belief networks from data: an information theory based approach. In *Proceedings of the Sixth ACM International Conference on Information and Knowledge Management*, 1997.

[7] C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, IT-14(3):462–467, May 1968.

[8] Gregory F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

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

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[11] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley, 1991.

[12] Robert Cowell. Sampling without replacement in junction trees. Statistical Research Paper 15, City University, London, 1997.

[13] Sanjoy Dasgupta. Learning polytrees. 1998.

[14] A. P. Dawid. Applications of a general propagation algorithm for probabilistic expert systems. *Statistics and Computing*, 2:25–36, 1992.

[15] Peter Dayan and Richard S. Zemel. Competition and multiple cause models. *Neural Computation*, 7(3), 1995.

[16] Luis M. de Campos and Juan F. Huete. Algorithms for learning decomposable models and chordal graphs. In Dan Geiger and Prakash Pundalik Shenoi, editors, *Proceedings of the 13th Conference on Uncertainty in AI*, pages 46–53. Morgan Kaufmann, 1997.

[17] Carl G. de Marcken. Discovering dependencies by repeated sampling. 1998.

[18] 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*, 39:1–38, 1977.

[19] Denise L. Draper and Steve Hanks. Localized partial evaluation of belief networks. In *Proceedings of the 10th Conference on Uncertainty in AI*. Morgan Kaufmann Publishers, 1994.

[20] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Jounal of the Association for Computing Machinery*, 34(3):596–615, July 1987.

[21] Brendan J. Frey, Geoffrey E. Hinton, and Peter Dayan. Does the wake-sleep algorithm produce good density estimators? In D. Touretsky, M. Mozer, and M. Hasselmo, editors, *Neural Information Processing Systems*, number 8, pages 661–667. MIT Press, 1996.

[22] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.

[23] Nir Friedman, Moises Goldszmidt, and Tom Lee. Bayesian network classification with continous attributes: Getting the best of both discretization and parametric fitting. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[24] Nir Friedman and Moses Goldszmidt. Building classifiers using Bayesian networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 96)*, pages 1277–1284, Menlo Park, CA, 1996. AAAI Press.

[25] H. N. Gabow, Z. Galil, T. Spencer, and Robert Endre Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[26] Robert G. Gallager. *Low-density parity-check codes*. MIT Press, 1963.

[27] Dan Geiger. An entropy-based learning algorithm of bayesian conditional trees. In *Proceedings of the 8th Conference on Uncertainty in AI*, pages 92–97. Morgan Kaufmann Publishers, 1992.

[28] Dan Geiger. Knowledge representation and inference in similarity networks and bayesian multinets. *"Artificial Intelligence"*, 82:45–74, 1996.

[29] Dan Geiger and Christopher Meek. Graphical models and exponential families. In *Proceedings of the 14th Conference on Uncertainty in AI*, pages 156–165. Morgan Kaufmann Publishers, 1998.

[30] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, 1984.

[31] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. *Markov chain Monte Carlo in practice*. Chapman & Hall, London, 1996.

[32] David Heckerman. A tutorial on learning Bayesian networks. Technical Report MSR–TR–95–06, Microsoft Research, 1995.

[33] David Heckerman, Dan Geiger, and David M. Chickering. Learning Bayesian networks: the combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[34] Reimar Hofmann and Volker Tresp. Nonlinear Markov networks for continuous variables. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Neural Information Processing Systems*, number 10, pages 521–529. MIT Press, 1998.

[35] Tommi S. Jaakkola. *Variational methods for inference and estimation in graphical models*. PhD thesis, Massachusetts Institute of Technology, 1997.

[36] Finn V. Jensen. *An introduction to Bayesian networks*. Springer, 1996.

[37] Finn V. Jensen and Frank Jensen. Optimal junction trees. In *Proceedings of the 10th Conference on Uncertainty in AI*. Morgan Kaufmann Publishers, 1994.

[38] Finn V. Jensen, Steffen L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 4:269–282, 1990.

[39] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. *Learning and inference in graphical models*, chapter "An introduction to variational methods for graphical models", pages 75–104. 1998.

[40] Uffe Kjærulff. Approximation of Bayesian networks through edge removals. Technical Report Report, Aarlborg University, Department of Mathematics and Computer Science, 94–2007, 1993.

[41] Petri Kontkanen, Petri Myllymaki, and Henry Tirri. Constructing bayesian finite mixture models by the EM algorithm. Technical Report C-1996-9, Univeristy of Helsinky, Department of Computer Science, 1996.

[42] S. Kotz and N. L. Johnson (Eds.). *Encyclopedia of Statistical Sciences*. Wiley, 1982–1989.

[43] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Stat.*, 22:79–86, March 1951.

[44] David J. C. MacKay and Radford M. Neal. Near Shannon limit performance of low density parity check codes. *Electronics Letters*, 33:457–458, 1997.

[45] David Madigan and Adrian Rafferty. Model selection and accounting for model uncertainty in graphical models using occam's window. *Journal of the American Statistical Association*, 89, 1994.

[46] Marina Meilă and Michael I. Jordan. Estimating dependency structure as a hidden variable. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Neural Information Processing Systems*, number 10, pages 584–590. MIT Press, 1998.

[47] Marina Meilă, Michael I. Jordan, and Quaid D. Morris. Estimating dependency structure as a hidden variable. Technical Report AIM–1648,CBCL–165, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1998. (revised version of AIM–1611).

[48] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood Publishers, 1994.

[49] Stefano Monti and Gregory F. Cooper. A Bayesian network classfier that combines a finite mixture model and a naive Bayes model. Technical Report ISSP-98-01, University of Pittsburgh, March 1998.

[50] Radford M. Neal. Connectionist learning of belief networks. *Artificial Intelligence*, 56:71–113, 1992.

[51] Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech and Language*, 8:1–38, 1994.

[52] Michiel O. Noordewier, Geoffrey G. Towell, and Jude W. Shawlik. Training Knowledge-Based Neural Networks to recognize genes in DNA sequences. In Richard P. Lippmann, John E. Moody, and David S. Touretztky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 530–538. Morgan Kaufmann Publishers, 1991.

[53] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman Publishers, San Mateo, CA, 1988.

[54] Carl E. Rasmussen, Radford M. Neal, Geoffrey E. Hinton, Drew van Camp, Michael Revow, Zoubin Ghahramani, R. Kustra, and Robert Tibshrani. *The DELVE Manual*. http://www.cs.utoronto.ca/ delve, 1996.

[55] Jorma Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific Publishing Company, New Jersey, 1989.

[56] Jeff Schlimmer. Mushroom database. U.C. Irvine Machine Learning Repository.

[57] Raffaella Settimi and Jim Q. Smith. On the geometry of Bayesian graphical models with hidden variables. In *Proceedings of the 14th Conference on Uncertainty in AI*, pages 472–479. Morgan Kaufmann Publishers, 1998.

[58] R. Sibson. SLINK: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16, 1973.

[59] D. J. Spiegelhalter, A. Thomas, N. G. Best, and W. R. Gilks. *BUGS: Bayesian inference Using Gibbs Sampling, Version 3.0*. Medical Research Council Biostatistics Unit, Cambridge, 1994.

[60] Peter Spirtes and Thomas Richardson. A polynomial time algorithm for determining DAG equivalence in the presence of latent variables and selection bias. In *Proceedings of the AISTATS-97 workshop*, pages 489–500, 1997.

[61] Elena Stanghellini and Barbara Vantaggi. On identification of graphical log-linear models with one unobserved variable. Technical Report 99/1, Universitá di Perugia, 1999.

[62] Evan W. Steeg. *Automated motif discovery in protein structure prediction.* PhD thesis, University of Toronto, 1997.

[63] Robert Endre Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, 1983.

[64] Bo Thiesson, Christopher Meek, D. Maxwell Chickering, and David Heckerman. Learning mixtures of Bayes networks. Technical Report MSR–POR–97–30, Microsoft Research, 1997.

[65] Geoffrey Towell and Jude W. Shawlik. Interpretation of artificial neural networks: Mapping Knowledge Based Neural Networks into rules. In John E. Moody, Steve J. Hanson, and Richard P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4, pages 9–17. Morgan Kaufmann Publishers, 1992.

[66] Thomas Verma and Judea Pearl. An algorithm for deciding if a set of observed independencies has a causal explanation. In Didier Dubois, Michael P. Wellman, Bruce D'Ambrosio, and Philippe Smets, editors, *Proceedings of the 8th Conference on Uncertainty in AI*, pages 323–330. Morgan Kaufmann, 1992.

[67] James D. Watson, Nancy H. Hopkins, Jeffrey W. Roberts, Joan Argetsinger Steitz, and Alan M. Weiner. *Molecular Biology of the Gene*, volume I. The Benjamin/Cummings Publishing Company, 4 edition, 1987.

[68] Yair Weiss. Belief propagation and revision in networks with loops. Technical Report AIM–1616,CBCL–155, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1997.

[69] Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, 1996.

[70] Joe Whittaker. *Graphical models in applied multivariate statistics.* John Wiley & Sons, 1990.