LEARNING TRACTABLE GRAPHICAL MODELS

by

AMIRMOHAMMAD ROOSHENAS

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2017

DISSERTATION APPROVAL PAGE

Student: Amirmohammad Rooshenas

Title: Learning Tractable Graphical Models

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Daniel Lowd	Chair
Dejing Dou	Core Member
Christopher Wilson	Core Member
John Conery	Core Member
Yashar Ahmadian	Institutional Representative

and

Scott L. Pratt	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded March 2017

DISSERTATION ABSTRACT

Amirmohammad Rooshenas

Doctor of Philosophy

Department of Computer and Information Science

March 2017

Title: Learning Tractable Graphical Models

Probabilistic graphical models have been successfully applied to a wide variety of fields such as computer vision, natural language processing, robotics, and many more. However, for large scale problems represented using unrestricted probabilistic graphical models, exact inference is often intractable, which means that the model cannot compute the correct value of a joint probability query in a reasonable time. In general, approximate inference has been used to address this intractability, in which the exact joint probability is approximated. An increasingly popular alternative is tractable models. These models are constrained such that exact inference is efficient. To offer efficient exact inference, tractable models either benefit from graph-theoretic properties, such as bounded treewidth, or structural properties such as local structures, determinism, or symmetry. An appealing group of probabilistic models that capture local structures and determinism includes arithmetic circuits (ACs) and sum-product networks (SPNs), in which marginal and conditional queries can be answered efficiently. In this dissertation, we describe ID-SPN, a state-of-the-art SPN learner as well as novel methods for learning tractable graphical models in a discriminative setting, in particular through introducing

Generalized ACs, which combines ACs and neural networks. Using extensive experiments, we show that the proposed methods often achieves better performance comparing to selected baselines.

CURRICULUM VITAE

NAME OF AUTHOR: Amirmohammad Rooshenas

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA

Sharif University of Technology, Tehran, Iran

Shahid Beheshti University, Tehran, Iran

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2017, University of Oregon

Master of Science, Computer and Information Science, 2016, University of Oregon
Master of Information Technology, Computer Networks, 2010, Sharif University of Technology
Bachelor of Science, Software Engineering, 2007, Shahid Beheshti University

AREAS OF SPECIAL INTEREST:

Probabilistic Models, Machine Learning

PROFESSIONAL EXPERIENCE:

Graduate Research & Teaching Assistant, Department of Computer and Information Science, University of Oregon, 2011 to present

Research Assistant, Department of Computer and Information Science, Sharif University of Technology, 2008 to 2010

GRANTS, AWARDS AND HONORS:

Gurdeep Pall Scholarship, Department of Computer Science, University of Oregon, 2014.

Clarence and Lucille Dunbar Scholarship, College of Art and Sciences, University of Oregon, 2014.

Member of Upsilon Pi Epsilon, the International Computer Science Honors Society.

PUBLICATIONS:

Rooshenas, A. and Lowd, D. (2016). Discriminative structure learning of arithmetic circuits. In Proceedings of the Nineteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2016), Cadiz, Spain

Lowd, D. and Rooshenas, A. (2015). The Libra toolkit for probabilistic models. Journal of Machine Learning Research, 16:24592463

Rooshenas, A. and Lowd, D. (2014). Learning sum-product networks with direct and indirect variable interactions. In Proceedings of the 31st International Conference on Machine Learning, pages 710718

Rooshenas, A.and Lowd, D. (2013). Learning tractable graphical models using mixture of arithmetic circuits. In AAAI (Late-Breaking Developments).

Lowd, D. and Rooshenas, A. (2013). Learning Markov networks with arithmetic circuits. In Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2013), Scottsdale, AZ

ACKNOWLEDGEMENTS

I would like to thank Daniel Lowd for being a great adviser and for his extensive support during my Ph.D. education.

I also want to thank my committee members Dejing Dou, Christopher Wilson, John Conery, and Yashar Ahmadian for their supports.

I graciously acknowledge that this work was funded in part was by grants from National Science Foundation and a Google faculty research award.

Finally, I want to thank my beloved wife Sara for her emotional support, without which this work was not possible at all.

To my wife Sara,

to my mother Farahnaz,

and

to the memory of my father Mehdi

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

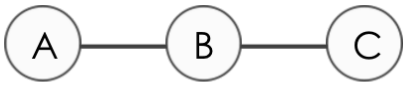
Probabilistic models are important in many areas such as medicine, biology, robotics, etc. The goal of probabilistic models is to probabilistically reason about phenomena or events. For example, given a probabilistic model of medical records of several patients, we are interested in inferring the risk of a particular patient getting cancer. Suppose each record indicates whether a patient has anemia, fatigue, dizziness, fever, pain, and leukemia. Then, we want to reason about the probability of a patient having leukemia if we only have partial information that he or she has anemia.

For this kind of reasoning, we may define a probabilistic model considering one variable for each of the 6 pieces of information we have about patients, which gives us a space of 64 different configurations.

A joint probability distributions over these variables assigns a probability to each of these configurations. More formally, a joint probability distribution $P(\mathcal{X})$ over a set of variables $\mathcal{X}$ is a function from $\mathcal{X}$ to $[0, 1]$ such that $\sum_{\mathbf{x} \in X} P(\mathbf{x}) = 1$, where X is the set of all possible configurations. Therefore, given a probabilistic distribution for medical records, we are interested in computing P(Leukemia = true), which requires iterating over the space of all variables in order to compute the corresponding value. This computation becomes intractable as the space of possible configurations grows exponentially in the number of variables.

This kind of probabilistic reasoning is called inference, which in general is finding the probability of an assignment to variables. If the assignment is partial, the reasoning is called marginal inference since it runs inference on a

a)



b)

A	B	Φ(A,B)
0	0	1
0	1	120
1	0	430
1	1	1

B	C	Φ(B,C)
0	0	100
0	1	1
1	0	1
1	1	210

FIGURE 1.1. a) Example of a Markov network over three random variables A, B, and C. b) The potentials that describing edges $A - B$ and $B - C$.

marginal probability distribution over a smaller set of variables. We can also answer conditional probabilities using probabilities of full assignments and partial assignments. The other important category of inference is finding the most probable explanation (MPE) of variables, also known as maximum a posteriori (MAP) inference. MPE or MAP inference finds the most likely state of variables given a probability distribution.

For an exponentially large probability space, inference is intractable. Probabilistic graphical models such Bayesian or Markov networks encode a probability space as factorizations to reduce the complexity of inference and representation although inference remains intractable in general. These factorizations are based on independencies and conditional independencies among variables.

A Markov network (MN) represents an undirected graph, in which every variable is represented as a node, and the edges indicate direct interactions between variables. For example, Figure 1.1.a illustrates a Markov network over three random variables A, B, and C. The given Markov network encodes the following interaction: A is conditionally independent of C given B, for which we have $P(A, C|B) = P(A|B)P(C|B)$. A probability distribution factorizes over a Markov

network if it can be written as a normalized products of factors:

$$P(\mathcal{X}) = \frac{1}{Z} \prod_c \phi_c(\mathcal{X}_c) \qquad \text{(Equation 1.1)}$$

where each ϕ_c is a non-negative, real-valued function called a potential function, $\mathcal{X}_c \subset \mathcal{X}$, and Z is a normalization constant or partition function: $Z = \sum_{\mathbf{x}} \prod_c \phi_c(\mathbf{x}_c)$. For the Markov network of Figure 1.1.a, the probability distribution $P(A, B, C)$ factorizes using the potentials $\phi_1(A, B)$ and $\phi(B, C)$ for edges $A - B$ and $B - C$, respectively, shown in Figure 1.1.b.

We can also represent a Markov network with positive potentials as

$$P(\mathcal{X}) = \frac{1}{Z} \exp(-E(\mathcal{X})), \qquad \text{(Equation 1.2)}$$

where $E(\mathcal{X})$ is a free energy function. In this notation, MAP inference is equal to the minimization of the free energy function:

$$\mathbf{x} = \arg\min_x E(\mathcal{X}). \qquad \text{(Equation 1.3)}$$

This formulation is very useful since we can use optimization techniques to compute the MAP state, or an approximation, for a particular energy function.

In addition to optimization approaches, there exists a plenty of other algorithmic solutions for exact and approximate inference in graphical models (Sontag et al., 2011; Wainwright and Jordan, 2008; Globerson and Jaakkola, 2007; Heskes et al., 2002; Murphy et al., 1999; Chavira and Darwiche, 2008).

Exact inference algorithms, such as variable elimination and the junction tree algorithm can be used to compute arbitrary marginal and conditional probabilities

in a probabilistic graphical model. However, the complexity of such methods is exponential in the treewidth of the model. Many relatively simple structures, such as Markov networks with pairwise interactions, may have very large treewidth, rendering exact inference intractable in most cases. For example, a NxN grid structure over n^2 variables has a treewidth of n, which makes inference over grid structures becomes intractable even for small values of n.

In recent years, there has been growing interest in learning tractable probability distributions, which can perform exact inference efficiently. The most widely explored approach is to learn a graphical model with bounded treewidth, often called a thin junction tree. For a treewidth of one (Markov trees), the maximum likelihood tree-structured model can be learned in polynomial time using the Chow-Liu algorithm (Chow and Liu, 1968). A number of methods have been proposed to learn thin junction trees with larger treewidths (Bach and Jordan, 2001; Chechetka and Guestrin, 2008; Elidan and Gould, 2008; Shahaf et al., 2009). In general, finding a maximum likelihood bounded treewidth structure is NP-hard (Korhonen and Parviainen, 2013), so most algorithms only find a local optimum or require very long running time for graphs with treewidth greater than three.

While having bounded treewidth is a sufficient condition for tractable inference, it is not always necessary. When additional forms of local structure exist, such as context-specific independence (Boutilier et al., 1996) and determinism (Chavira and Darwiche, 2008), even a model with a very large treewidth may still admit efficient inference. Mixture models (Meila and Jordan, 2000; Lowd and Domingos, 2005), sum-product networks (Poon and Domingos, 2011), arithmetic circuits (Darwiche, 2003), cutset networks (Rahman

et al., 2014a), feature trees (Gogate et al., 2010), and sentential decision diagrams (Darwiche, 2011) are examples of representations that can exploit these local structures and offer tractable inference.

A mixture of tractable distributions is an interesting class of tractable models. The simplest example is a naive Bayes mixture model, in which the observed variables are independent given the latent class variable: $P(\mathcal{X}) = \sum_c P(C = c) \prod_i P(X_i|C = c)$. This is equivalent to a tree-structured graphical model with one additional variable, C, which is never observed in the training data. Mixture models can be learned with the expectation maximization algorithm, which iteratively assigns instances to clusters using the current model and then updates the model parameters using this assignment. Naive Bayes mixture models are often as effective as learning a Bayesian network without hidden variables (Lowd and Domingos, 2005).

A sum-product network (SPN) (Poon and Domingos, 2011) is a deep probabilistic model for representing a tractable probability distribution. SPNs are attractive because exact inference is linear in the size of their networks. They have also achieved impressive results on several computer vision problems. An SPN consists of a rooted, directed, acyclic graph representing a probability distribution over a set of random variables. Each leaf in the SPN graph is a tractable distribution over a single random variable. Each interior node is either a sum node, which computes a weighted sum of its children in the graph, or a product node, which computes the product of its children.

An arithmetic circuit (AC) (Darwiche, 2003) is an inference representation that is as expressive as SPNs, and can represent many other types of tractable probability distributions, including thin junction trees and latent tree models.

Like an SPN, an AC is a rooted, directed, acyclic graph in which interior nodes are sums and products. The representational differences are that ACs use indicator nodes and parameter nodes as leaves, while SPNs use univariate distributions as leaves and attach all parameters to the outgoing edges of sum nodes. ACs have interesting mathematical properties that make them suitable for learning tractable probabilistic methods. For example, we can efficiently differentiate the function represented by an AC with respect to its parameters with two passes over the circuit, which is advantageous for gradient-based optimization algorithms.

1.1. Contributions

In this section, we briefly enumerate the main contributions of this dissertation:

- We introduce ID-SPN, a state-of-the-art SPN learner. The main advantage of ID-SPN over the prior algorithms for learning SPNs is considering both direct and indirect interactions of random variables through tractable Markov networks and mixture models, respectively. We experimentally show that ID-SPN is better than the baselines.

- We introduce conditional ACs, which are more compact representations than ordinary ACs for conditional distributions. We also introduce DACLearn as the first discriminative structure learning of tractable conditional distributions. DACLearn searches over high-order features over output and input variables while maintaining a compact conditional AC for tractable exact inference. DACLearn learns more accurate conditional models in comparison to generative and discriminative baselines. We also show that

discriminative structure learning results in more accurate conditional models rather than discriminative parameter learning.

– We introduce generalized ACs (GACs) to represent tractable conditional distributions. GACs generalize the operation of ACs to include non-linear operations, and can also represent high-dimensional discrete or continuous input variables. We also introduce GACLearn as a method for learning the GAC representations. GACLearn learns more accurate conditional models comparing to DACLearn and other generative and discriminative baselines. We also show that GACLearn achieves the state-of-the-art results on the problem of multilable classification.

1.2. Dissertation outline

Chapter II describes ACs and SPNs as the main background of this dissertation. It also includes a brief introduction to other tractable representations for probabilistic models. Chapter III describes the ID-SPN algorithm for learning SPNs and discusses the representational equivalence of SPNs and ACs. Chapter IV and Chapter V focus on learning tractable conditional distributions. Chapter IV introduces conditional ACs, which are more compact for representing conditional distributions rather than ACs. In Chapter IV, we also show that how we can learn the structure of conditional ACs from data. Chapter V, extends conditional ACs into generalized ACs (GACs), which are more powerful representations for conditional distributions. We also introduce GACLearn as a method for learning the structure of GACs. Finally, Chapter VI concludes this dissertation and addresses some potential future directions.

CHAPTER II

BACKGROUND

It is well understood that learning tractable high-treewidth models is possible if the models leverage local structures such as context-specific independence (CSI) (Boutilier et al., 1996), determinism (Chavira and Darwiche, 2008) or other structural properties such as associativity (Taskar et al., 2004) and exchangeability (Niepert and Domingos, 2014). Exploiting CSIs and determinism leads to the introduction of alternative model representations such as arithmetic circuits (Darwiche, 2003), sum-product networks (Poon and Domingos, 2011), and probabilistic decision diagrams (Kisa et al., 2014). To achieve tractable inference through associativity or exchangeability, we do not need different representations other than Markov networks, and benefiting from these properties, we can solve the inference problem using a closed form solution or a tractable optimization formulation. In this chapter, we explore arithmetic circuits and sum-product networks as well as the alternative representations and structural properties in more detail.

2.1. Arithmetic circuits

An arithmetic circuit (AC) (Darwiche, 2003) is a tractable probabilistic model over a set of discrete random variables, $P(\mathcal{X})$. An AC consists of a rooted, directed, acyclic graph in which interior nodes are sums and products. Each leaf is either a non-negative model parameter or an indicator variable that is set to one if a particular variable can take on a particular value.

For example, consider a simple Markov network over two binary variables with features $f_1 = y_1 \wedge y_2$ and $f_2 = y_2$:

$$P(Y_1, Y_2) = \frac{1}{Z} \exp(w_1 f_1 + w_2 f_2).$$

Figure 4.1 represents this probability distribution as an AC, where $\theta_1 = e^{w_1}$ and $\theta_2 = e^{w_2}$ are parameters, and $\lambda_{y_1} = 1_{(y_1=1)}$ and $\lambda_{y_2} = 1_{(y_2=1)}$ are indicator variables.

In an AC, to compute the unnormalized probability of a complete configuration $\tilde{P}(\mathcal{X} = \mathbf{x})$, we first set the indicator variable leaves to one or zero depending on whether they are consistent or inconsistent with the values in $\mathbf{x}$. Then we evaluate each interior node from the bottom up, computing its value as a function of its children. The value of the root node is the unnormalized probability of the configuration. However, the real strength of ACs is their ability to efficiently marginalize over an exponential number of variable states. To compute the probability of a partial configuration, set all indicator variables for the marginalized variables to one and proceed as with a complete configuration. The normalization constant Z can similarly be computed by setting all indicator variables to one. Conditional probabilities can be computed as probability ratios. For example, for the AC in Figure 4.1, we can compute the unnormalized probability $\tilde{P}(y_1)$ by setting $\lambda_{\neg y1}$ to zero and all others to one, and then evaluating the root. To obtain the normalization constant, we set all indicator variables to one and again evaluate the root.

However, we can compute the normalization constant using aforementioned bottom-up evaluation only if the circuit is a valid representation for a probability distribution, which can be expressed using the following terms Darwiche (2003):

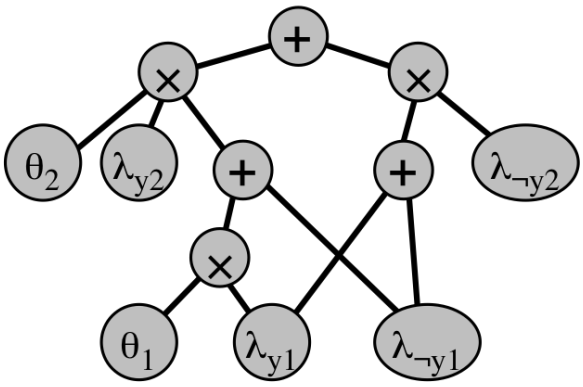


FIGURE 2.1. Simple arithmetic circuit that encodes a Markov network with two variables y_1 and y_2 and two features $f_1 = y_1 \wedge y_2$ and $f_2 = y_2$.

- An AC is *decomposable* if the children of a product node have no common descendant variable.

- An AC is *deterministic* if the children of a sum node are mutually exclusive, meaning that at most one is non-zero for any complete configuration.

- An AC is *smooth* if the children of a sum node have identical descendant variables.

An AC is a valid representation for a probability distribution if it is decomposable and smooth. For decomposable and smooth ACs, marginal and conditional inference is linear in the number of edges, so for compact ACs inference is tractable. If a circuit is deterministic, exact MAP inference is also linear in the number of edges.

2.1.1. Learning ACs

Two AC learning methods have been proposed. Lowd and Domingos (2008) adapt a greedy Bayesian network structure learning algorithm by maintaining an equivalent AC representation and penalizing structures by the number of edges in

the AC. This biases the search towards models where exact inference is tractable without placing any a priori constraints on network structure. ACMN (Lowd and Rooshenas, 2013) extends this idea to learning Markov networks with conjunctive features, and find that the additional flexibility of the undirected representation leads to slightly better likelihoods at the cost of somewhat slower learning times.

ACMN performs a greedy search through structure space, similar to the methods of Della Pietra et al. (1997) and McCallum (2003). The initial structure is the set of all single-variable features. The search operations are to take an existing feature in the model, f, and combine it with another variable, V, creating two new features: $f \wedge v$ and $f \wedge \neg v$. This operation is called "split".

Splits are scored according to their effect on the log-likelihood of the MN and the size of the corresponding AC:

$$\text{score}(s) = \Delta_{ll}(s) - \gamma \Delta_e(s)$$

Here, Δ_{ll} is a measure of how much the split will increase the log-likelihood. Measuring the exact effect would require jointly optimizing all model parameters along with the parameters for the two new features. Therefore, log-likelihood gain is measured by modifying only the weights of the two new features, keeping all others fixed. This gives a lower bound on the actual log-likelihood gain. This gain is computed by solving a simple two-dimensional convex optimization problem, which depends only on the empirical counts of the new features in the data and their expected counts in the model, requiring performing inference just once to compute these expectations. A similar technique was used by Della Pietra et al. (1997) and McCallum (2003) for efficiently computing feature gains.

$\Delta_e(s)$ denotes the number of edges that would be added to the AC if this split were included. Computing this has similar time complexity to actually performing the split. γ determines the relative weightings of the two terms. The combined score function is equivalent to maximizing likelihood with an exponential prior on the number of edges in the AC.

2.2. Mixture models

When probability distributions are multi-modal, we can suppose that the distribution can be expressed as a weighted sum (mixture) of some uni-modal distributions. Therefore, if each mixture component represents a uni-modal distribution $P^i(\mathcal{X})$, the distribution represented by the mixture models become as:

$$P(\mathcal{X}) = \sum_i w_i P^i(\mathcal{X}),$$
$$w_i \geq 0, \sum_i w_i = 1. \qquad \text{(Equation 2.1)}$$

However, in mixture models, $P^i(\mathcal{X})$ can be described using graphical models with different structures over the same set of variables $\mathcal{X}$. Therefore, there may not exist any graphical model (Markov network or Bayesian network) that can represent the same distribution, which means that mixture models are more powerful than graphical models. Moreover, if we describe every mixture component, $P^i(\mathcal{X})$, with a tractable model, then the whole mixture model is tractable.

Many researchers introduce different tractable mixture models, and empirically show their representation power. Mixture of trees (Meila and Jordan,

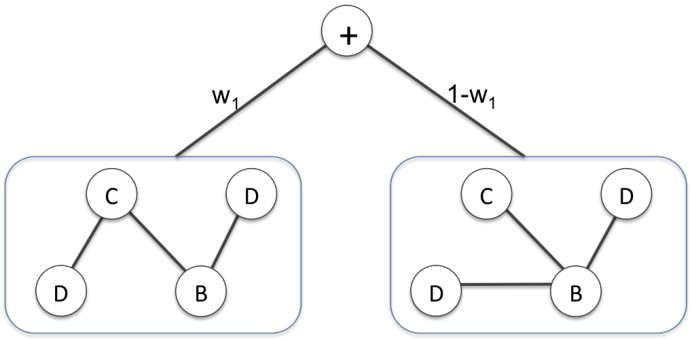


FIGURE 2.2. A simple illustration of mixture of trees with two mixture components, and four observable variables.

2000), mixture of arithmetic circuits (Rooshenas and Lowd, 2013), and mixture of cutsets (Rahman et al., 2014b) are examples of tractable mixture models.

Here, we describe the learning algorithm for mixture of trees (MT) (Meila and Jordan, 2000) since other algorithms follow more or less similar approaches for learning mixture models, however, with different representation for mixture components. Initially, MT randomly assigns each sample to a mixture component, and then models the probability distribution of each component using Chow-Liu algorithm (Chow and Liu, 1968). To learn the mixture parameters, MT uses expectation maximization (EM), which is an iterative algorithm for learning parameters based on maximum likelihood when some of the parameters are not observed. In each iteration of the EM, MT distributes samples among the mixture components by assigning each sample to the component that is most likely to generate the sample, and updates the Chow-Liu trees for each component. Figure 2.2 demonstrates a MT with two components. In the figure, the latent variable representation embeds the mixture parameters as well. Mixtures of trees are fast and accurate in comparison to the state-of-the-art tractable graphical models (Rooshenas and Lowd, 2014).

2.3. Sum-product networks

A sum-product network (SPN) (Poon and Domingos, 2011) is a deep probabilistic model for representing a tractable probability distribution. SPNs are attractive because they can represent many other types of tractable probability distributions, including thin junction trees, latent tree models, and mixtures of tractable distributions. They have also achieved impressive results on several computer vision problems (Poon and Domingos, 2011; Gens and Domingos, 2012; Amer and Todorovic, 2012) as well as problems in speech and language modeling (Peharz et al., 2014; Cheng et al., 2014).

SPNs are very similar to ACs. Like ACs, SPNs are rooted, directed, acyclic graphs representing probability distributions over a set of random variables. In SPNs, each interior node is either a sum node, which computes a weighted sum of its children in the graph, or a product node, which computes the product of its children. The scope of a node is defined as the set of variables appearing in the univariate distributions of its descendants. In order to be valid, the children of every sum node must have identical scopes, and the children of every product node must have disjoint scopes (Gens and Domingos, 2013)[1]. Intuitively, sum nodes represent mixture and product nodes represent independencies. SPNs can also be described recursively as follows: every SPN is either a tractable univariate distribution, a weighted sum of SPNs with identical scopes, or a product of SPNs with disjoint scopes.

The representational differences between ACs and SPNs are that ACs use indicator nodes and parameter nodes as leaves, while SPNs use tractable univariate

[1]Poon and Domingos (2011) also allow for non-decomposable product nodes, but we adopt the more restrictive definition of Gens and Domingos (2013) for simplicity.

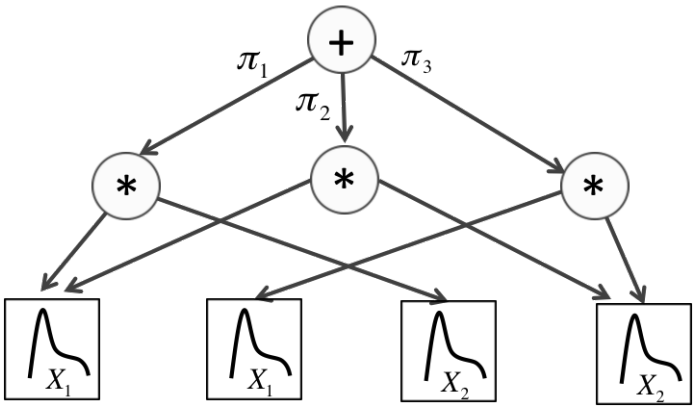


FIGURE 2.3. An SPN representation of a naive Bayes mixture model over two random variables.

distributions as leaves and attach all parameters to the outgoing edges of sum nodes.

As we show in Chapter III, for discrete domains, every decomposable and smooth AC can be represented as an equivalent SPN with fewer or equal nodes and edges, and every SPN can be represented as an AC with at most a linear increase in the number of edges.

As an example of SPNs, consider a naive Bayes mixture model (sometimes called a mixture of Bernoullis): $P(\mathcal{X}) = \sum_i \pi_i \prod_j P^i(X_j)$. This can easily be represented as an SPN, where the root node computes the weighted sum and its children are products of the univariate distributions. Figure 2.3 depicts an SPN that represents a naive Bayes mixture over two variables. SPNs can also represent thin junction trees by introducing sum and product nodes for the different states of the clique and separator sets; see Poon and Domingos (2011) for a simple example. Moreover, SPNs can represent mixtures of thin junction trees, mixture of trees, and latent tree models. In addition, SPNs can also represent context-specific independence and other types of finer-grained structure.

To compute the probability of a complete configuration, we have to compute the value of each node starting at the leaves. Each leaf is a univariate distribution which evaluates to the probability of one variable according to that distribution. Sum and product nodes evaluate to the weighted sum and product of their child nodes in the network, respectively. To compute the probability of a partial configuration, we need to sum out one or more variables. In an SPN, this is done by setting the values of all leaf distributions for those variables to 1. Conditional probabilities can then be computed as the ratio of two partial configurations. Thus, computing marginal and conditional probabilities can be done in linear time with respect to the size of the SPN, while these operations are often intractable in Bayesian and Markov networks with high treewidth.

2.3.1. Learning SPNs

Several different methods have recently been proposed for learning SPNs. Dennis and Ventura (2012) construct a region graph by first clustering the training instances and then repeatedly clustering the variables within each cluster to find smaller scopes. When creating new regions, if a region with that scope already exists, it is reused. Given the region graph, Dennis and Ventura (2012) convert this to an SPN by introducing sum nodes to represent mixtures within each region and product nodes to connect regions to sub-regions. Gens and Domingos (2013) also perform a top-down clustering, but they create the SPN directly through recursive partitioning of variables and instances rather than building a region graph first. The advantage of their approach is that it greedily optimizes log-likelihood; however, the resulting SPN always has a tree structure and does not reuse model

components. Peharz et al. (2013) propose a greedy bottom-up clustering approach for learning SPNs that merges small regions into larger regions.

2.4. Other tractable representations

There exists a plenty of other tractable representations such as thin junction tree, feature trees, and cutset networks. In this section we briefly describe them.

2.4.1. Thin junction trees

A general approach to address the tractability is to assume a restricted structure for the underlying network (graph) such that inference is tractable on the structure. It is well-understood that exact inference is tractable over tree structures (e.g. by running belief propagation), so intuitively, the degree of similarity of a graph to a tree can determine the complexity of exact inference over the graph. This degree is called treewidth, and inference is exponential in the treewidth of the underlying graph. This fact leads to interest in learning the structure of graphs with bounded treewidth.

For the class of bounded treewidth graphs, inference is also tractable, while it is NP-hard for general graphs. This tractability motivates numerous algorithms for learning bounded treewidth graphical models and thin-junction trees (Bach and Jordan, 2001; Chechetka and Guestrin, 2008; Elidan and Gould, 2008).

2.4.2. Feature trees

A feature tree (graph) Gogate et al. (2010) is an AND/OR search tree (graph) Dechter and Mateescu (2007), in which OR and AND nodes represent features and feature assignments, respectively. The assigned features of OR nodes,

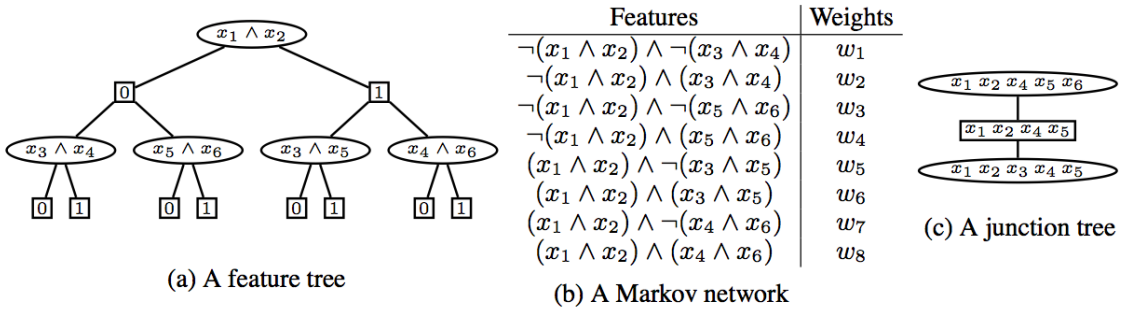


FIGURE 2.4. (Gogate et al., 2010) a) A simple feature tree. b) the features of Markov network this feature tree representing. c) the corresponding junction tree for the Markov network. The features of the Markov network are produced by conjoining the feature nodes in all paths from the root to the leaves.

also called feature nodes (or F-nodes), have bounded length. Nevertheless, feature trees can represent Markov networks with high treewidth. The AND nodes are called assignment nodes, or for short, A-nodes. Feature trees exploit context-specific independence to represent a compact model for Markov networks, and as a result, feature trees are able to offer tractable inference and closed-form weight learning. We can convert a feature tree to a Markov network. Conjoining the feature assignments along every path from root to leaves represent the features of the corresponding Markov network. Figure 2.4 shows a simple feature tree, the set of features of a Markov network represented by the feature tree, and the equivalent junction tree. The probability of a full variable assignment can be computed by traversing the feature tree once from leaves to the root; see Gogate et al. (2010); Dechter and Mateescu (2007) for more details.

To learn a feature tree, Gogate et al. (2010) introduce an algorithm called LEM, which searches for all features up to a feature length bound, and then finds a set of groups of variables that are approximately independent given every feature assignment (using the method of Chechetka and Guestrin (2008)). LEM scores the

features based on their inference complexity, and picks the best feature and the related set of groups of variables. It adds the feature to the feature tree (creating a new F-node), and then recurs for every assignment to the selected feature (creating new A-nodes) and for every group of variables which is independent given the assignment. Therefore, it builds the feature tree recursively. Gogate et al. (2010) provide performance guarantees on the accuracy of the feature trees learned using LEM.

2.4.3. Cutset networks

Recently introduced cutset networks (CNs) (Rahman et al., 2014a) are another tractable representation for probability distributions, which leverage both context-specific independence and determinism. The cutset network is an OR search tree (Dechter and Mateescu, 2007), in which every leaf is a Chow-Liu tree (Chow and Liu, 1968). The fundamental idea of CNs is to select a group of variables such that the joint probability of the remaining variables conditioned on the assignment to the selected variables is representable using a Chow-Liu tree. The complexity of inference is linear in the number of edges in the OR tree, which is exponential in the size (cardinality) of cutset variables. Therefore, bounding the size of the cutset variables leads to tractable inference given that inference is tractable at the leaves. Figure 2.5 shows an example of a CN which represents a probability distribution over six variables. Selecting the cutset variables is the challenging problem for learning CNs. Rahman et al. (2014a) propose to choose a variable to be in the cutset if it maximizes the expected reduction in the average entropy over individual variables (an approximation to the joint entropy), or in other words, select the variable that has the highest information gain.

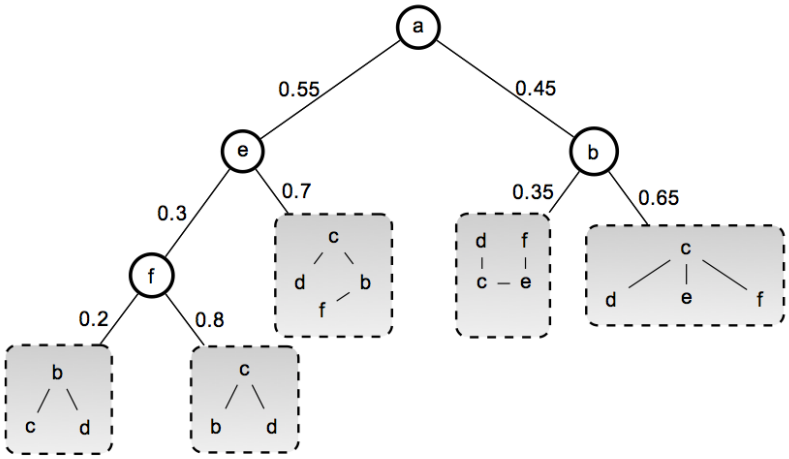


FIGURE 2.5. (Rahman et al., 2014a) A simple cutset network over 6 variables.

2.4.4. Sentential decision diagrams

Sentential decision diagrams (SDDs) (Darwiche, 2011) are a subset of negation normal forms (NNFs) which fulfill both decomposability and strong determinism. An NNF is a DAG representation of a Boolean function in which all internal nodes are conjunctions or disjunctions and the leaves are either constants ($\top : \mathit{true}, \bot : \mathit{false}$) or literals (variables or their negation). In NNFs, the negation only applies to the input variables. If the sets of variables appearing on the left and right subtrees of every conjunction have no variables in common, then the NNF is decomposable. Considering a decomposable NNF (DNNF), if any assignment to the variables only activates one child of every disjunction, then the DNNF is deterministic. In fact, SDDs are a strict subset of deterministic decomposable NNFs (d-DNNFs), which themselves are a subset of DNNFs, and DNNFs are a subset of NNFs. A *vtree* is a full rooted binary tree whose leaves are the set of variables, and every variable appears exactly once in the leaves (Pipatsrisawat and Darwiche, 2008). Figure 2.6.a demonstrates an example of a vtree for four

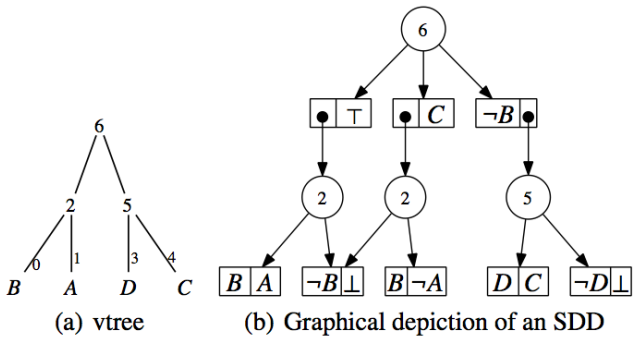


FIGURE 2.6. (Darwiche, 2011) An SDD that represents $f = (A \wedge B)(B \wedge C) \vee (C \wedge D)$. The SDD respects the vtree, every decision node (circle) in the SDD are related to one internal node in the vtree, and the variables in primes (subs) of the decision node appear in the left (right) subtrees of the corresponding node in the vtree.

variables. Darwiche (2011) defines a Boolean function $f(\mathcal{Z})$ over the set of variables $\mathcal{Z}$ as a $(\mathcal{X}, \mathcal{Y})$-decomposition, if $f(\mathcal{Z})$ can be written as:

$$f(\mathcal{Z}) = (p_1(\mathcal{X}) \wedge s_1(\mathcal{Y}) \vee (p_2(\mathcal{X}) \wedge s_2(\mathcal{Y}) \vee \cdots \vee (p_n(\mathcal{X}) \wedge s_n(\mathcal{Y}). \quad \text{(Equation 2.2)}$$

A structured d-DNNF respects a vtree if there exists an one-to-one mapping from d-DNNF decompositions $(p_1(\mathcal{X}) \wedge s_1(\mathcal{Y}) \vee (p_2(\mathcal{X}) \wedge s_2(\mathcal{Y}) \vee \cdots \vee (p_n(\mathcal{X}) \wedge s_n(\mathcal{Y})$ to vtree internal nodes v such that $\mathcal{X}$ and $\mathcal{Y}$ are subsets of variables in the left and right subtrees of v, respectively.

Assuming a given vtree, an SDD is a constant, literal, or an $(\mathcal{X}, \mathcal{Y})$-decomposition represented by a decision node, such that the decomposition respects a vtree, and $p_1 \cdots p_n, s_1 \cdots s_n$ are SDDs. p_is and s_is are called primes and subs, respectively, and each pair of a prime and a sub constitute an element. SDDs are strongly deterministic, which means that $p_i \wedge p_j = \mathit{false}$ for any $i \neq j$, and $\vee_i p_i = \mathit{true}$. Figure 2.6.b illustrates an SDD which respects the vtree in Figure 2.6.a, and

represents the Boolean function $f(A, B, C, D) = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. An important feature of SDDs is that they offer tractable weighted model counting, which can be used for inference in graphical models (Choi et al., 2013).

Probabilistic SDDs (PSDDs) (Kisa et al., 2014) are probabilistic counterparts for SDDs, in which the logical decision nodes in SDDs have been replaced with probabilistic decision nodes, and every outgoing edge from a decision node is labeled with a parameter. The subs in the elements also take parameters. If a sub element does not have any parameter, it means that the parameter is one. Figure 2.7 shows an example of a PSDD. Parameters of PSDDs can be learned using closed-form maximum likelihood parameter learning. Computing the probability of an evidence configuration can be done in one bottom-up pass of the diagram, and the computation of marginals given evidence needs two passes (one bottom-up and one top-down). PSDDs are able to represent local structures (context-specific independence and determinism) as well as conditional independencies.

To show the power of PSDDs for representing the local structures, Kisa et al. (2014) compile the disjunction of all data (including test data) as domain constraints (logical constraints) and then compile it into a PSDD. They learn parameters using training data and then compute the likelihood of the test data. The result shows a significant improvement in comparison to the state-of-the-art likelihood results for the compared datasets, although the experiments are not sound since they use test data for creating the structure of the PSDD.

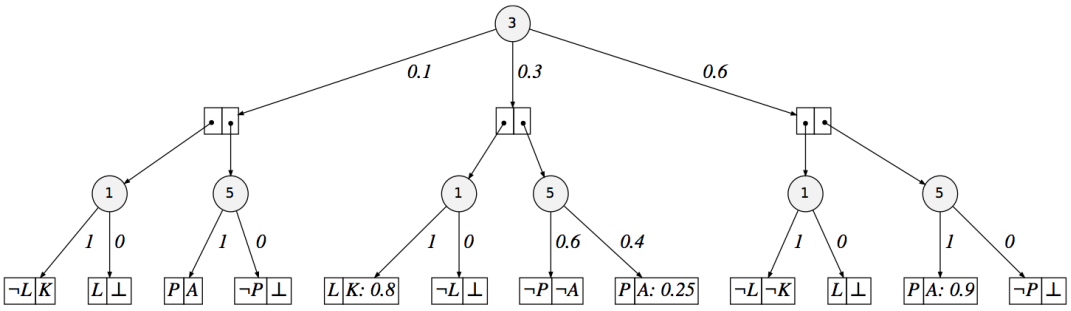


FIGURE 2.7. Kisa et al. (2014) A PSDD for set of variable (L, P, A, K). Given the PSDD, $Pr(P \wedge A) = 1 * 1 * 0.1 + 0.25 * 0.4 * 0.3 + 0.9 * 1 * 0.6$.

2.4.5. Restricted interactions

For general Markov networks, tractable exact inference is achievable through restricting the potentials. Here we discuss approaches that focus on finding the MAP state which is equal to minimizing the free energy[2].

A successful approach to minimize the free energy is graph cuts. To use the graph cut algorithm, the free energy should have a particular graph structure, so the graph cut based approaches assume a pairwise Markov network over a graph of variables, $\mathcal{G}(\mathcal{V}, \mathcal{E})$. In this case, the MAP inference reduces to minimizing the following energy function:

$$E(\mathcal{X}) = \sum_{i \in \mathcal{V}} \theta_i(x_i) + \sum_{(i,j) \in \mathcal{E}} \theta_{ij}(x_i, x_j), \qquad \text{(Equation 2.3)}$$

where $\theta(.)$ and $\theta(.,.)$ are unary and pairwise potentials, respectively. Greig et al. (1989) show that there exists a polynomial solution for minimizing Equation 2.3 using the graph cut algorithm if the underlying network is an Ising model, for which the variable x_is are binary and $x_i \in \{-1, 1\}$, and also the potentials have the

[2] $P(\mathcal{X}) = \frac{1}{Z} \exp(-E(\mathcal{X}))$ then maximizing $P(\mathcal{X})$ is equal to minimizing $-E(\mathcal{X})$

form of $\theta_{ij}(x_i, x_j) = w_{ij}x_i x_j$, where w_{ij} is a feature weight (Koller and Friedman, 2009). Kolmogorov and Zabin (2004) extend using the graph cut approach to minimize the free energy, Equation 2.3, with binary variables and submodular potentials:

$$\theta_{ij}(0,0) + \theta_{ij}(1,1) \leq \theta_{ij}(0,1) + \theta_{ij}(1,0). \qquad \text{(Equation 2.4)}$$

Ishikawa (2003) applies the graph cut approach to minimize Equation 2.3 for multi-value variables. In their formulation, potentials have to be convex functions of variable distances, and the variables are consecutive integer numbers. For examples, in the pixel labeling problem, the variables are the label indexes of pixels and potentials are convex functions of label differences. For more general non-convex functions, the graph cut approach returns a local solution or an approximation of the global minimum (Veksler, 2007).

Taskar et al. (2004) introduce associative Markov networks (AMNs), which are less restricted than the models used by Kolmogorov and Zabin (2004) and Greig et al. (1989). AMNs allow arbitrary topology with high order potentials, and different labels (variable values) can have different strengths (in the Ising model, different labels have the same strength), so the energy functions become:

$$E(\mathcal{X}) = \sum_i \theta_i(x_i) + \sum_{c \in \mathcal{C}} \theta_c(\mathbf{x}_c), \qquad \text{(Equation 2.5)}$$

where $\mathcal{C}$ is the set of cliques in the graph, and $\theta_c(\mathbf{x}_c)$ is equal to a non-negative parameter λ_c^k when all variables agree on the label k, and zero otherwise.

Taskar et al. (2004) model the MAP inference as an integer linear programming (ILP) problem, and approximate it using an LP relaxation. They

show that for binary variables (having only two labels), the LP formulation has integral solutions which means the approximated solution is the optimum value. For multi-value variables, the integral solutions are not guaranteed, but it works well in practice (Taskar et al., 2004).

Taskar et al. (2004) also introduce a max-margin approach for learning the parameters, in which the parameter learning is modeled as a Quadratic programming (QP) problem, which calls the MAP inference as a component. They show that the QP problem has an optimal solution when MAP inference is exact, which happens when the variables are binary (network has only two labels).

2.4.6. Exchangeable variable models

Recently, Niepert and Van den Broeck (2014) leverage the concept of exchangeability in statistics to introduce new tractable models. Exchangeability can enrich graphical models which merely use the notion of independence and conditional independence. A set of random variables is fully exchangeable if permuting the assignment to the random variables does not change the probability of the assignment. Similarly a set of random variables is partial exchangeable with respect to the sufficient statistic T if for every assignment $\mathbf{x}$ and $\mathbf{x}'$ to random variables, $T(\mathbf{x}) = T(\mathbf{x}')$ implies that the probability of $\mathbf{x}$ and $\mathbf{x}'$ is equal. Benefiting from the partial exchangeability, we can express the probability distribution as a mixture of uniform distributions (Niepert and Domingos, 2014):

$$P(\mathbf{x}) = \sum_{t \in \mathcal{T}} w_t U_t(\mathbf{x}), \qquad \text{(Equation 2.6)}$$

where $\mathcal{T}$ is the possible values of sufficient statistic T, and should be finite, and $w_t = P(T(\mathbf{x}) = t)$, and U_t is a uniform distribution of a set of assignments that satisfies $T(\mathbf{x}) = t$.

Niepert and Domingos (2014) prove that exchangeable variable models (EVMs) can compute the marginal probabilities and the MAP state in time polynomial in the number of variables. They also extend EVMs to the mixture of EVMs (MEVMs) which combine the conditional independencies with exchangeability, and is more expressive than simple mixture models such as the naive Bayes mixture model. MEVMs show promising results in problems that are not linearly separable like parity and counting. They also have more accurate models compared to tractable graphical models like Chow-Liu trees and latent tree models (Choi et al., 2011) and show competitive performance compared to more complex models such as SPNs (Gens and Domingos, 2013) and ACs (Lowd and Rooshenas, 2013). Finding the correct sufficient statistics which fulfill the assumption of partial exchangeability and incorporating EVMs in more complex models are still open problems.

CHAPTER III

LEARNING SUM-PRODUCT NETWORKS

Sum-product networks (SPNs) are a deep probabilistic representation that allows for efficient, exact inference. SPNs generalize many other tractable models, including thin junction trees, latent tree models, and many types of mixtures. Previous work on learning SPN structure has mainly focused on using top-down or bottom-up clustering to find mixtures, which capture variable interactions indirectly through implicit latent variables. In contrast, most work on learning graphical models, thin junction trees, and arithmetic circuits has focused on finding direct interactions among variables. In this chapter, we present ID-SPN, a new algorithm for learning SPN structure that unifies the two approaches.

3.1. Motivation and background

Previous work about learning sum-product networks (SPNs) has focused exclusively on the latent variable approach, using a complex hierarchy of mixtures to represent all interactions among the observable variables (Gens and Domingos, 2013; Dennis and Ventura, 2012). These SPN structures can be learned from data by recursively clustering instances and variables. Clustering over the instances is done to create a mixture, represented with a sum node. Clustering over the variables is done to find independencies within the cluster, represented with a product node. We refer to this approach as creating *indirect interactions* among the variables, since all dependencies among the observable variables are mitigated by the latent variables implicit in the mixtures.

This type of top-down clustering is good at representing clusters, but may have difficulty with discovering *direct interactions* among variables. For example, suppose the data in a domain is generated by a 6-by-6 grid-structured Markov network (MN) with binary-valued variables. This MN can be represented as a junction tree with treewidth 6, which is small enough to allow for exact inference. This can also be represented as an SPN that sums out 6 variables at a time in each sum node, representing each of the separator sets in the junction tree. However, learning this from data requires discovering the right set of 64 (2^6) clusters that happen to render the other regions of the grid independent from each other. Of all the possible clusterings that could be found, happening to find one of the separator sets is extremely unlikely. Learning a good structure for the next level of the SPN is even less likely, since it consists of 64 clustering problems, each working with 1/64th of the data.

In contrast, Markov network structure learning algorithms can easily learn a simple grid structure, but may do poorly if the data has natural clusters that require latent variables or a mixture. For example, consider a naive Bayes mixture model where the variables in each cluster are independent given the latent cluster variable. Representing this as a Markov network with no latent variables would require an exponential number of parameters.

In order to get the best of both worlds, we propose ID-SPN, a new method for learning SPN structures that can learn both indirect and direct interactions, including conditional and context-specific independencies. This unifies previous work on learning SPNs through top-down clustering (Dennis and Ventura, 2012; Gens and Domingos, 2013) with previous work on learning tractable Markov networks through greedy search (Lowd and Rooshenas, 2013).

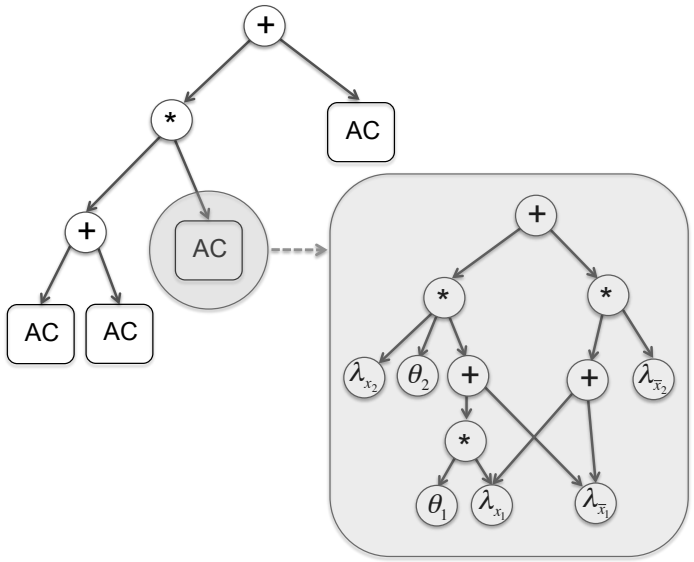


FIGURE 3.1. Example of an ID-SPN model. The upper layers are shown explicitly as sum and product nodes, while the lower layers are abbreviated with the nodes labeled "AC." The AC components encode graphical models over the observed variables represented as arithmetic circuits (AC), and may involve many nodes and edges.

ID-SPN combines top-down clustering with methods for learning tractable Markov networks to obtain the best of both worlds: indirect interactions through latent cluster variables in the upper levels of the SPN as well as direct interactions through the tractable Markov networks at the lower levels of the SPN. ID-SPN learns tractable Markov networks represented by ACs using the ACMN algorithm (Lowd and Rooshenas, 2013); however, this could be replaced with any algorithm that learns a tractable multivariate probability distribution that can be represented as an AC or SPN, including any thin junction tree learner.

For learning an SPN structure, LearnSPN (Gens and Domingos, 2013) recursively performs two operations to create a tree-structured SPN: partitioning the training data to create a sum node, representing a mixture over different clusters, or partitioning the variables to create a product node, representing groups

of independent variables within a cluster. The partitioning is chosen greedily to maximize the (regularized) likelihood of the training data. This process continues recursively, operating on fewer variables and examples until it reaches univariate distributions which become the leaves.

ID-SPN performs a similar top-down search, clustering instance and variables to create sum and product nodes, but it may stop this process before reaching univariate distributions and instead learn an AC to represent a tractable multivariate distribution with no latent variables. Thus, LearnSPN remains a special case of ID-SPN when the recursive clustering proceeds all the way to univariate distributions. ACMN is also a special case, when a tractable distribution is learned at the root and no clustering is performed. ID-SPN uses the likelihood of the training data to choose among these different operations.

Another way to view ID-SPN is that it learns SPNs where the leaves are tractable *multivariate* distributions rather than univariate distributions. As long as these leaf distributions can be represented as valid SPNs, the overall structure can be represented as a valid SPN as well. For ease of description, we refer to the structure learned by ID-SPN as a sum-product of arithmetic circuits (SPAC). A SPAC model consists of sum nodes, product nodes, and AC nodes. Every AC node is an encapsulation of an arithmetic circuit which itself includes many nodes and edges. Figure 3.1 shows an example of a SPAC model.

We can also relax the condition of learning a valid SPN structure and allow any tractable probabilistic models at the leaves. In this case, the SPAC model generalizes to a sum-product of tractable models. Such models may not be valid SPNs, but they retain the efficient inference properties that are the key advantage

Algorithm 1 Algorithm for learning a SPAC.

```
function ID-SPN(T)
input: Set of training examples, T; set of variables V
output: Learned SPAC model
n ← LearnAC(T, V).
SPAC ← n
N ← leaves of SPAC // which includes only n
while N ≠ ∅ do
    n ← remove a node from N
    (T_ji, V_j) ← set of samples and variables used for learning n
    subtree ← extend(n, T_ji, V_j)
    SPAC' ← replace n by subtree in SPAC
    if SPAC' has a better log-likelihood on T than SPAC then
        N ← N∪ leaves of SPAC'
        SPAC ← SPAC'
    end if
end while
return SPAC
```

of SPNs. Nevertheless, we leave this part for future exploration and continue with SPAC, which has a valid SPN structure.

3.2. ID-SPN algorithm

Algorithm 1 illustrates the pseudocode of ID-SPN. The ID-SPN algorithm begins with a SPAC structure that only contains one AC node, learned using ACMN on the complete data. In each iteration, ID-SPN attempts to *extend* the model by replacing one of the AC leaf nodes with a new SPAC subtree over the same variables. Figure 3.2 depicts the effect of such an extension on the SPAC model. If the extension increases the log-likelihood of the SPAC model on the training data, then ID-SPN updates the working model and adds any newly created AC leaves to the queue.

As described in Algorithm 2, the extend operation first attempts to partition the variables into independent sets to create a new product node. If a good

partition exists, the subroutine learns a new sum node for each child of the product node. If no good variable partition exists, the subroutine learns a single sum node. Each sum node is learned by clustering instances and learning a new AC leaf node for each data cluster. These AC nodes may be extended further in future iterations.

As a practical matter, in preliminary experiments we found that the root was always extended to create a mixture of AC nodes. Since learning an AC over all variables from all examples can be relatively slow, in our experiments we start by learning a sum node over AC nodes rather than a single AC node.

Every node n_{ji} in SPAC (including sum and product nodes) represents a valid probability distribution P_{ji} over a subset of variables V_j. To learn n_{ji}, ID-SPN uses a subset of the training set T_{ji}, which is a subset of training samples T_i projected into V_j. We call T_{ji} the *footprint* of node n_{ji} on the training set, or to be more concise, the footprint of n_{ji}.

A product node estimates the probability distribution over its scope as the product of approximately independent probability distributions over smaller scopes: $P(V) = \prod_j P(V_j)$ where V is the scope of the product node and V_js are the scope of its children. Therefore, to create a product node, we must partition variables into some sets that are approximately independent. We use pairwise mutual information, $\sum_{X_k,X_l \in j} \frac{C(X_k,X_l)}{|T_{ji}|} \log \frac{C(X_k,X_l)\cdot|T_{ji}|}{C(X_k)C(X_l)}$ where $C(.)$ counts the occurrences of the configuration in the footprint of the product node, to approximately measure the dependence among different sets of variables. A good variable partition is one where the variables within a partition have high mutual information, and the variables in different partitions have low mutual information. Thus, we create an adjacency matrix using pairwise mutual information such that two variables are connected if their empirical mutual information over the footprint of the product

Algorithm 2 Algorithm for extending SPAC structure.

function extend(n, T, V)
input: An AC node n, set of instances T and variables V used for learning n
output: An SPAC subtree representing a distribution over V learned from T
//learns a product node
p-success $\leftarrow$ Using T, partition V into approximately independent subsets V_j
if p-success **then**
  for each V_j **do**
    let T_j be the data samples projected into V_j
    //learns a sum node
    s-success $\leftarrow$ partition T_j into subsets of similar instances T_{ji}
    if s-success **then**
      for each T_{ji} **do** $n_{ji} \leftarrow \text{LearnAC}(T_{ji}, V_j)$
      end for
    else
      $n_j \leftarrow \text{LearnAC}(T_j, V_j)$
    end if
  end for
  subtree $\leftarrow \prod_j [(\sum_i \frac{|T_{ji}|}{|T_j|} \cdot n_{ji}) | n_j]$
else
  //learns a sum node
  s-success $\leftarrow$ partition T into subsets of similar instances T_i
  if s-success **then**
    $n_i \leftarrow \text{LearnAC}(T_i, V)$
    subtree $\leftarrow \sum_i \frac{|T_i|}{|T|} \cdot n_i$
  else
    fails the extension
  end if
end if
return subtree

node (not the whole training set) is larger than a predefined threshold. Then, we find the set of connected variables in the adjacency matrix as the independent set of variables. This operation fails if it only finds a single connected component or if the number of variables is less than a threshold.

A sum node represents a mixture of probability distributions with identical scopes: $P(V) = \sum_i w_i P^i(V)$ where the weights w_i sum to one. A sum node can also be interpreted as a latent variable that should be summed out: $\sum_c P(C =$

$c)P(V|C = c)$. To create a sum node, we partition instances by using the expectation maximization (EM) algorithm to learn a simple naive Bayes mixture model: $P(V) = \sum_i P(C_i) \prod_j P(X_j|C_i)$ where X_j is a random variable. To select the appropriate number of clusters, we rerun EM with different numbers of clusters and select the model that maximizes the penalized log-likelihood over the footprint of the sum node. To avoid overfitting, we penalize the log-likelihood with an exponential prior, $P(S) \propto e^{-\lambda C|V|}$ where C is the the number of clusters and λ is a tunable parameter. We use the clusters of this simple mixture model to partition the footprint, assigning each instance to its most likely cluster. We also tried using k-means, as done by Dennis and Ventura (2012), and obtained results of similar quality. We fail learning a sum node if the number of samples in the footprint of the sum node is less than a threshold.

To learn leaf distributions, AC nodes, we use the ACMN algorithm (Lowd and Rooshenas, 2013). ACMN learns a Markov network using a greedy, score-based search, but it uses the size of the corresponding arithmetic circuit as a learning bias. ACMN can exploit the context-specific independencies that naturally arise from sparse feature functions to compactly learn many high-treewidth distributions. The learned arithmetic circuit is a special case of an SPN where sum nodes always sum over mutually exclusive sets of variable states. Thus, the learned leaf distribution can be trivially incorporated into the overall SPN model. Other possible choices of leaf distributions include thin junction trees and the Markov networks learned by LEM (Gogate et al., 2010). We chose ACMN because it offers a particularly flexible representation (unlike LEM), exploits context-specific independence (unlike thin junction trees), and learns very accurate models on a range of structure learning benchmarks (Lowd and Rooshenas, 2013).

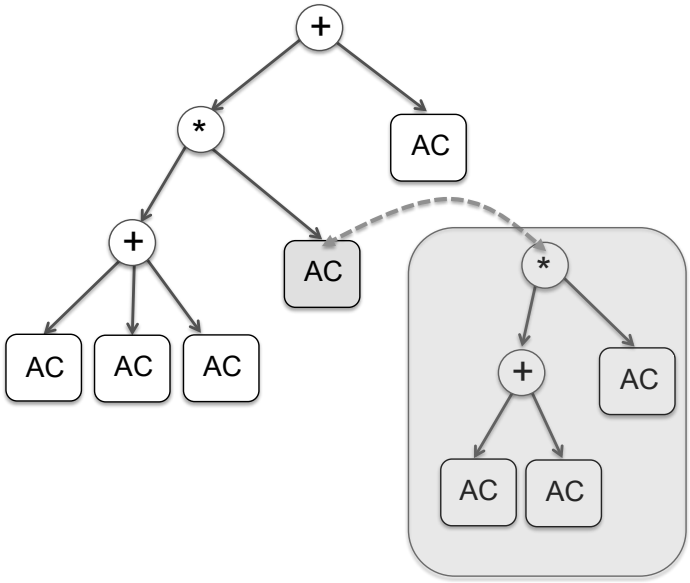


FIGURE 3.2. One iteration of ID-SPN: it tries to extends the SPAC model by replacing an AC node with a SPAC subtree over the same scope

The specific methods for partitioning instances, partitioning variables, and learning a leaf distribution are all flexible and can be adjusted to meet the characteristics of a particular application domain.

3.3. Relation of SPNs and ACs

In this section, We demonstrate the equivalence of the AC and SPN representations with the following two propositions.

Proposition 1. *For discrete domains, every decomposable and smooth AC can be represented as an equivalent SPN with fewer nodes and edges.*

Proof. We show this constructively. Given an AC, we can convert it to a valid SPN representing the same function in four steps:

1. If the root is not a sum node, set the root to a new sum node whose single child is the former root.

2. For each sum node, set the initial weights of all outgoing edges to 1.

3. For each parameter node, find the first sum node on each path to the root and multiply its outgoing edge weight along that path by the parameter value. (Do not multiply the same edge weight by any given parameter more than once, even if that edge occurs in multiple paths to the root. We assume that parameter nodes only occur as children of product nodes.)

4. Remove all parameter nodes from the network.

5. Replace each indicator node $\lambda_{X_i=v}$ with a deterministic univariate distribution, $P(X_i = v) = 1$.

Because the AC was decomposable, each product in the resulting SPN must be over disjoint scopes. Because the AC was smooth, each sum in the resulting SPN must be over identical scopes. Therefore, the SPN is valid. Since all indicator nodes are removed and at most one new node is added, the new SPN must have fewer nodes and edges than the original AC.

To prove that the SPN evaluates to the same function as the original AC, we use induction to show that each sum node evaluates to the same value as before. Since the root node is a sum, this suffices to show that the new SPN is equivalent.

Consider each outgoing edge from the sum node to one of its children. There are three cases to consider:

– If the child is a leaf, then the child's value is a deterministic distribution which is clearly identical to the parameter node in AC. The edge weight will be 1, since the leaf could not have had any parameter node descendants that were removed.

– If the child is another sum node, then by the inductive hypothesis, its value must be the same as in the AC. The weight of this edge must also be 1, since any parameter node descendant must have at least one sum node that is "closer," namely the child sum node.

– If the child is a product node, then its value might be different from the AC. Without loss of generality, consider a product node and all of its product node children, and all of their product ndoe children, etc. together as a single product. (This is valid because multiplication is commutative and associative.) The elements in this product are only sum nodes and leaf nodes, both of which have the same values as in the AC. One or more parameter nodes could have been removed from this product when constructing the SPN. These parameters have been incorporated into the edge weight, since the parent sum node is the first sum node on any path from the parameters to the root that passes through this edge. Therefore, the value of the product node times its edge weight is equal to the value of the product node in the AC.

Thus, each child of the sum node has the same value as in the original AC once it has been multiplied by the edge weight, so the sum node computes the same value as in the AC. For the base case of a sum node with no sum node descendants, the above arguments still suffice and no longer depend on the inductive hypothesis. Therefore, by structural induction, every sum node computes the same value as in the AC.

□

Proposition 2. *For discrete domains, every SPN can be represented as an AC with at most a linear increase in the number of edges.*

Proof. We again show this constructively. Given an SPN, we can convert it to a decomposable and smooth AC representing the same distribution in three steps:

1. Create indicator nodes for each variable/value combination.

2. Replace each univariate distribution with a sum of products of parameter and indicator nodes. Specifically, create one parameter node for each state of the target variable, representing the respective probability of that state. Create one product node for each state with the corresponding indicator node and new parameter node as children.

3. Replace each outgoing edge of each sum node with a product of the original child and a new parameter node representing the original weight from that edge.

Assuming the domain of each variable is bounded by a constant, each edge is replaced by at most a constant number of edges. Therefore, the number of edges in the resulting AC is linear in the number of edges in the original SPN.

We now use induction to show that, for each node in the SPN, there is a node in the AC that computes the same value. We have three types of nodes to consider:

- As the base case, each leaf in the SPN is represented by a new sum node created in the second step. By construction, this sum node clearly represents the same value as the leaf distribution in the SPN.

- Each product node in the SPN is unchanged in the AC conversion. Since, by the inductive hypothesis, its children compute the same values as in the SPN, so does the product.

– Each sum node in the SPN is represented by a similar node in the AC. The AC node is an unweighted sum over products of the SPN edge weights and nodes representing the corresponding SPN children. By the inductive hypothesis, these children compute the same values as their SPN counterparts, so the weighted sum performed by the AC node (with the help of new product and parameter nodes) is identical to the SPN sum node.

The root of the AC represents the root of the SPN, so by induction, they compute the same value and the two models represent the same distribution. □

3.4. Experimental results

We evaluated ID-SPN on 20 datasets illustrated in Table 3.1.a with 16 to 1556 binary-valued variables. These datasets or a subset of them also have been used previously (Davis and Domingos, 2010; Lowd and Davis, 2010; Haaren and Davis, 2012; Lowd and Rooshenas, 2013; Gens and Domingos, 2013). In order to show the accuracy of ID-SPN, we compared it with the state-of-the-art learning method for SPNs (LearnSPN) (Gens and Domingos, 2013), mixtures of trees (MT) (Meila and Jordan, 2000), ACMN, and latent tree models (LTM) (Choi et al., 2011). To evaluate these methods, we selected their hyper-parameters according to their accuracy on the validation sets.

For LearnSPN, we used the same learned SPN models presented in the original paper (Gens and Domingos, 2013). We used the WinMine toolkit (WM) (Chickering, 2002) for learning intractable Bayesian networks.

We slightly modified the original ACMN codes to incorporate both Gaussian and L_1 priors into the algorithm. The L_1 prior forces more weights to become zero, so the learning algorithm can prune more features from the split candidate list.

TABLE 3.1. Dataset characteristic and Log-likelihood comparison. • shows significantly better log-likelihood than ID-SPN, and ∘ indicates significantly worse log-likelihood than ID-SPN. † means that we do not have value for that entry.

	a) Datasets characteristics				b) Log-likelihood comparison					
Dataset	Var#	Train	Valid	Test	ID-SPN	ACMN	MT	WM	LearnSPN	LTM
NLTCS	16	16181	2157	3236	-6.02	• -6.00	-6.01	-6.02	-6.11	∘ -6.49
MSNBC	17	291326	38843	58265	-6.04	∘ -6.04	∘ -6.07	-6.04	∘ -6.11	∘ -6.52
KDDCup 2000	64	180092	19907	34955	-2.13	∘ -2.17	-2.13	∘ -2.16	-2.18	∘ -2.18
Plants	69	17412	2321	3482	-12.54	∘ -12.80	∘ -12.95	∘ -12.65	∘ -12.98	∘ -16.39
Audio	100	15000	2000	3000	-39.79	∘ -40.32	∘ -40.08	∘ -40.50	∘ -40.50	∘ -41.90
Jester	100	9000	1000	4116	-52.86	∘ -53.31	∘ -53.08	∘ -53.85	∘ -53.48	∘ -55.17
Netflix	100	15000	2000	3000	-56.36	∘ -57.22	∘ -56.74	∘ -57.03	∘ -57.33	∘ -58.53
Accidents	111	12758	1700	2551	-26.98	∘ -27.11	∘ -29.63	• -26.32	∘ -30.04	∘ -33.05
Retail	135	22041	2938	4408	-10.85	∘ -10.88	-10.83	∘ -10.87	-11.04	∘ -10.92
Pumsb-star	163	12262	1635	2452	-22.40	∘ -23.55	∘ -23.71	• -21.72	∘ -24.78	∘ -31.32
DNA	180	1600	400	1186	-81.21	• -80.03	∘ -85.14	• -80.65	∘ -82.52	∘ -87.60
Kosarek	190	33375	4450	6675	-10.60	∘ -10.84	-10.62	∘ -10.83	∘ -10.99	∘ -10.87
MSWeb	294	29441	32750	5000	-9.73	∘ -9.77	∘ -9.85	• -9.70	∘ -10.25	∘ -10.21
Book	500	8700	1159	1739	-34.14	∘ -35.56	∘ -34.63	∘ -36.41	-35.89	-34.22
EachMovie	500	4524	1002	591	-51.51	∘ -55.80	∘ -54.60	∘ -54.37	-52.49	†
WebKB	839	2803	558	838	-151.84	∘ -159.13	∘ -156.86	∘ -157.43	-158.20	∘ -156.84
Reuters-52	889	6532	1028	1540	-83.35	∘ -90.23	∘ -85.90	∘ -87.55	-85.07	∘ -91.23
20 Newsgroup	910	11293	3764	3764	151.47	∘ -161.13	∘ -154.24	∘ -158.95	∘ -155.93	∘ -156.77
BBC	1058	1670	225	330	-248.93	∘ -257.10	∘ -261.84	∘ -257.86	-250.69	∘ -255.76
Ad	1556	2461	327	491	-19.00	• -16.53	• -16.02	-18.35	-19.73	†

The modified version is as accurate as the original version, but it is considerably faster. For ACMN, we used an L_1 prior of 0.1, 1, and 5, and a Gaussian prior with a standard deviation of 0.1, 0.5, and 1. We also used a split penalty of 2, 5, 10 and maximum edge number of 2 million.

For LTM, we ran the authors' code[1] with its default EM configuration to create the models with different provided algorithm: CLRG, CLNJ, regCLRG, and regCLNJ. For MT, we used our own implementation and the number of components ranged from 2 to 30 with a step size of 2. To reduce variance, we re-ran each learning configuration 5 times.

For ID-SPN, we need specific parameters for learning each AC leaf node using ACMN. To avoid exponential growth in the parameter space, we selected the L_1 prior C^{ji}, split penalty SP^{ji}, and maximum edges ME^{ji} of each AC node to be proportional to the size of its footprint: $P^{ji} = max\{P\frac{|T_{ji}|}{|T|} \cdot \frac{|V_j|}{|V|}, P^{min}\}$ where parameter P can be C, SP or ME. When SPAC becomes deep, C^{min} and SP^{min} help avoid overfiting and ME^{min} permits ID-SPN to learn usefull leaf distributions. Since ID-SPN has a huge parameter setting, we used random search (Bergstra and Bengio, 2013) instead of grid search. In order to create a configuration, we defined a uniform distribution over a discrete parameter values, and sampled every parameter distribution independently.

For learning AC nodes, we selected C from 1.0, 2.0, 5.0, 8.0, 10.0, 15.0 and 20.0, and SP from 5, 8, 10, 15, 20, 25, and 30. We selected the pair setting of (C^{min}, SP^{min}) between (0.01, 1) and (1, 2), and ME and ME^{min} are 2M and 200k edges, respectively. We used similar Gaussian priors with a standard deviation of 0.1, 0.3, 0.5, 0.8, 1.0, or 2.0 for learning all AC nodes.

[1] http://people.csail.mit.edu/myungjin/latentTree.html

TABLE 3.2. Statistically significance comparison. Each table cell lists the number of datasets where the row's algorithm obtains significantly better log-likelihoods than the column's algorithm

	ID-SPN	LearnSPN	WM	ACMN	MT	LTM
ID-SPN	–	11	13	17	15	17
LearnSPN	0	–	0	1	2	10
WM	4	6	–	10	7	13
ACMN	3	7	7	–	9	13
MT	1	7	11	11	–	15
LTM	0	0	3	4	2	–

For learning sum nodes, the cluster penalty λ is selected from 0.1, 0.2, 0.4, 0.6 and 0.8, the number of clusters from 5, 10, and 20, and we restarted EM 5 times. When there were fewer than 50 samples, we did not learn additional sum nodes, and when there were fewer than 10 variables, we did not learn additional product nodes.

Finally, we limited the number of main iteration of ID-SPNs to 5, 10, or 15, which helps avoid overfitting and controls the learning time. Learning sum nodes and AC nodes is parallelized as much as it was possible using up to 6 cores simultaneously.

We bounded the learning time of all methods to 24 hours, and we ran our experiments, including learning, tuning, and testing, on an Intel(R) Xeon(R) CPU X5650@2.67GHz.

Table 3.1.b shows the average test set log-likelihood of every methods. We could not learn a model using LTM for EachMovie and Ad datasets, so we excluded these two datasets for all comparisons involving LTM. We use • to indicate that on

TABLE 3.3. Conditional (marginal) log-likelihood comparison, the bold numbers show significantly better values

	10% Query		30% Query		50% Query		70% Query		90% Query	
Dataset	ID-SPN	WM	ID-SPN	WM	ID-SPN	WM	ID-SPN	WM	ID-SPN	WM
NLTCS	**-0.3082**	-0.3157	**-0.3138**	-0.3193	**-0.2881**	-0.2940	**-0.3393**	-0.3563	**-0.3622**	-0.4093
MSNBC	**-0.2727**	-0.2733	**-0.3130**	-0.3149	**-0.3285**	-0.3333	**-0.3403**	-0.3522	**-0.3528**	-0.3906
KDDCup 2000	**-0.0322**	-0.0329	**-0.0318**	-0.0325	**-0.0309**	-0.0316	**-0.0323**	-0.0337	**-0.0329**	-0.0361
Plants	**-0.1297**	-0.1361	**-0.1348**	-0.1444	**-0.1418**	-0.1590	**-0.1500**	-0.1876	**-0.1656**	-0.2797
Audio	**-0.3471**	-0.3630	**-0.3699**	-0.3808	**-0.3766**	-0.3854	**-0.3801**	-0.3957	**-0.3870**	-0.4309
Jester	**-0.4651**	-0.4869	**-0.4964**	-0.5096	**-0.5042**	-0.5128	**-0.5090**	-0.5231	**-0.5158**	-0.5589
Netflix	**-0.4893**	-0.5079	**-0.5254**	-0.5366	**-0.5349**	-0.5451	**-0.5416**	-0.5637	**-0.5507**	-0.6012
Accidents	**-0.1316**	-0.1368	**-0.1532**	-0.2226	**-0.1748**	-0.3165	**-0.1993**	-0.4523	**-0.2274**	-0.5834
Retail	-0.0794	-0.0800	**-0.0783**	-0.0794	**-0.0788**	-0.0807	**-0.0796**	-0.0823	**-0.0803**	-0.0841
Pumsb-star	-0.0727	**-0.0675**	**-0.0809**	-0.1077	**-0.0906**	-0.1584	**-0.1023**	-0.2920	**-0.1193**	-0.7654
DNA	-0.3371	-0.3373	**-0.3815**	-0.3957	**-0.3942**	-0.4450	**-0.4156**	-0.4898	**-0.4374**	-0.5335
Kosarek	**-0.0482**	-0.0514	**-0.0507**	-0.0533	**-0.0520**	-0.0547	**-0.0529**	-0.0568	**-0.0544**	-0.0623
MSWeb	**-0.0294**	-0.0300	**-0.0305**	-0.0318	**-0.0307**	-0.0328	**-0.0317**	-0.0350	**-0.0326**	-0.0375
Book	**-0.0684**	-0.0771	**-0.0674**	-0.0736	**-0.0673**	-0.0720	**-0.0673**	-0.0711	**-0.0675**	-0.0734
EachMovie	**-0.0958**	-0.1078	**-0.0976**	-0.1046	**-0.0985**	-0.1008	**-0.0986**	-0.1012	**-0.1005**	-0.1140
WebKB	**-0.1744**	-0.1881	**-0.1766**	-0.1858	**-0.1766**	-0.1834	**-0.1778**	-0.1844	**-0.1795**	-0.1929
Reuters-52	**-0.0911**	-0.1015	**-0.0889**	-0.0963	**-0.0896**	-0.0954	**-0.0905**	-0.0968	**-0.0920**	-0.1054
20 Newsgroup	**-0.1633**	-0.1807	**-0.1641**	-0.1753	**-0.1644**	-0.1708	**-0.1646**	-0.1688	**-0.1653**	-0.1724
BBC	**-0.2353**	-0.2503	**-0.2360**	-0.2459	**-0.2340**	-0.2406	**-0.2338**	-0.2406	**-0.2337**	-0.2462
Ad	-0.0080	-0.0078	-0.0086	-0.0087	**-0.0087**	-0.0180	**-0.0090**	-0.0767	**-0.0104**	-0.2849
Avg. Time (ms)	159	203	164	621	162	1091	168	1554	171	2021
Max. Time (ms)	286	1219	310	3641	312	6255	319	8682	319	11102

a dataset, the corresponding method has significantly better test set log-likelihood than ID-SPN, and $\circ$ for the reverse. For significance testing, we performed a paired t-test with p=0.05. ID-SPN has better average log-likelihood on every single dataset than LearnSPN, and better average log-likelihood on 17 out of 20 datasets (with 1 tie) than ACMN. Thus, ID-SPN consistently outperforms the two methods it integrates. Table 3.2 shows the number of datasets for which a method has significantly better test set log-likelihood than another. ID-SPN is significantly better than WinMine on 13 datasets and significantly worse than it on only 4 datasets, which means that ID-SPN is achieving efficient exact inference without sacrificing accuracy. ID-SPN is signficantly better than LearnSPN, ACMN, MT and LTM on 11, 17, 15, 17 datasets, respectively.

We also evaluated the accuracy of ID-SPN and WinMine for answering queries by computing conditional log-likelihood (CLL): $\log P(X = x|E = e)$. For WinMine, we used the Gibbs sampler from the Libra toolkit[2] with 100 burn-in and 1000 sampling iterations. Since the Gibbs sampler can approximate marginal probablities better then joint probablities, we also computed CMLL: $\sum_i \log P(X_i = x_i|E = e)$. For WinMine, we reported the greater of CLL and CMLL; however, CMLL was higher for all but 12 settings. The reported values have been normalized by the number of query variables.

We generated queries from test sets by randomly selecting the query variables, using the rest of variables as evidence. Table 3.3 shows the C(M)LL values ranging from 10% query and 90% evidence variables to 90% query and 10% evidence variables. The bold numbers indicate statistically significance using a paired t-test

[2]http://libra.cs.uoregon.edu

with p=0.05. ID-SPN is significantly more accurate on 95 out of 100 settings and is significantly worse on only 1.

We also report the per-query average time and per-query maximum time, both in milliseconds. We computed the per-query average for each dataset, and then reported the average and the maximum of per-query averages. For WinMine, the per-query average time significantly varies with the number of variables. The per-query maximum time shows that even with only 1000 iteration, Gibbs sampling is still slower than exact inference in our ID-SPN models.

3.5. Summary

Most previous methods for learning tractable probabilistic models have focused on representing all interactions directly or indirectly. ID-SPN demonstrates that a combination of these two techniques is extremely effective, and can even be more accurate than intractable models. Interestingly, the second most accurate model in our experiments was often the mixture of trees model (MT) (Meila and Jordan, 2000), which also contains indirect interactions (through a single mixture) and direct interactions (through a Chow-Liu tree in each component). After ID-SPN, MT achieved the second-largest number of significant wins against other algorithms. ID-SPN goes well beyond MT by learning multiple layers of mixtures and using much richer leaf distributions, but the underlying principles are similar. Therefore, rather than focusing exclusively on mixtures or direct interaction terms, this suggests that the most effective probabilistic models need to use a combination of both techniques.

CHAPTER IV

DISCRIMINATIVE LEARNING OF ARITHMETIC CIRCUITS

In this chapter, we introduce conditional arithmetic circuits as a compact representation for conditional distributions, and we present the first discriminative structure learning algorithm for arithmetic circuits (ACs), DACLearn (Discriminative AC Learner). Like previous work on generative structure learning, DACLearn finds a log-linear model with conjunctive features, using the size of an equivalent AC representation as a learning bias. Unlike previous work, DACLearn optimizes conditional likelihood, resulting in a more accurate conditional distribution. DACLearn also learns much more compact ACs than generative methods, since it does not need to represent a consistent distribution over the evidence variables. To ensure efficiency, DACLearn uses novel initialization and search heuristics to drastically reduce the number of feature evaluations required to learn an accurate model. In experiments on 20 benchmark domains, we find that DACLearn learns models that are more accurate and compact than other tractable generative and discriminative methods.

4.1. Motivation and background

Probabilistic graphical models such as Bayesian networks, Markov networks, and conditional random fields are widely used for knowledge representation and reasoning in computational biology, social network analysis, information extraction, and many other fields. However, the problem of inference limits their effectiveness and broader applicability: in many real-world problems, exact inference is intractable and approximate inference can be unreliable and inaccurate. This

poses difficulties for parameter and structure learning as well, since most learning methods rely on inference.

A compelling alternative is to work with model classes where inference is efficient, such as bounded treewidth models (Bach and Jordan, 2001; Chechetka and Guestrin, 2008), mixtures of tractable models (Meila and Jordan, 2000; Rahman et al., 2014a), sum-product networks (SPNs) (Poon and Domingos, 2011; Gens and Domingos, 2013; Rooshenas and Lowd, 2014), and arithmetic circuits (ACs) (Darwiche, 2003; Lowd and Domingos, 2008; Lowd and Rooshenas, 2013). Previous work has demonstrated that these models can be learned from data and that they often meet or exceed the accuracy of intractable models. However, most of this work has focused on joint probability distributions over all variables. For discriminative tasks, the conditional distribution of query variables given evidence, $P(\mathcal{Y}|\mathcal{X})$, is usually more accurate and more compact than the joint distribution $P(\mathcal{Y}, \mathcal{X})$.

To the best of our knowledge, only a few algorithms address general tractable discriminative structure learning. These include learning tree conditional random fields (tree CRFs) (Bradley and Guestrin, 2010), learning junction trees using graph cuts (Shahaf et al., 2009), max-margin tree predictors (Meshi et al., 2013) and mixtures of conditional tree Bayesian networks (MCTBN) (Hong et al., 2014). The first three methods are limited to pairwise potentials over the query variables. MCTBN learns a mixture of trees, which is slightly more flexible but still performed poorly in our experiments.

In this chapter, we present DACLearn (Discriminative AC Learner), a flexible and powerful method for learning tractable discriminative models over discrete domains. DACLearn is built on ACs, a particularly flexible model class that

is equivalent to SPNs (Rooshenas and Lowd, 2014) and subsumes many other tractable model classes. DACLearn performs a search through the combinatorial space of conjunctive features, greedily selecting features that increase conditional likelihood. In order to keep the model compact, DACLearn uses the size of the AC as a learning bias. Since DACLearn is modeling a conditional distribution, its ACs can condition on arbitrary evidence variables without substantially increasing the size of the circuit, leading to much more compact models. DACLearn is similar to previous AC learning methods (Lowd and Domingos, 2008; Rooshenas and Lowd, 2013), but it discriminatively learns a conditional distribution instead of a full joint distribution. DACLearn also introduces new initialization and search heuristics that improve the performance of existing generative AC learning algorithms.

4.1.1. Conditional Random Fields

Consider sets of discrete variables $\mathcal{Y} = \{Y_1, Y_2, \cdots, Y_n\}$ and $\mathcal{X} = \{X_1, X_2, \cdots, X_m\}$. Conditional random fields[1] (CRFs) (Lafferty et al., 2001) are undirected graphical models that represent the conditional probability distribution of query variables $\mathcal{Y}$ given the evidence variables $\mathcal{X}$:

$$P(\mathcal{Y}|\mathcal{X}) = \frac{1}{Z(\mathcal{X})} \prod_c \phi_c(D_c), \qquad \text{(Equation 4.1)}$$

where each ϕ_c is a real-valued, non-negative function, known as a potential function, with scope $D_c \subset \mathcal{X} \cup \mathcal{Y}$. $Z(\mathcal{X})$ is a normalization function, also called the partition function, which only depends on evidence variables $\mathcal{X}$. A Markov network

[1]Also known as conditional Markov networks.

can be seen as a special case of a CRF with no evidence variables, so $\mathcal{X}$ is empty and the partition function Z is a constant.

If all potential functions of Equation 4.1 are positive, then we can represent the conditional probability distribution using an equivalent log-linear formulation:

$$\log P(\mathcal{Y}|\mathcal{X}) = \sum_i w_i f_i(D_i) - \log Z(\mathcal{X}), \qquad \text{(Equation 4.2)}$$

where f_i is a logical conjunction of variable states. For example, for three binary variables X_1, Y_1, and Y_2, we can define $f_1(Y_1, X_1) = x_1 \wedge \neg y_1$ and $f_1(Y_1, Y_2) = y_1 \wedge y_2$.

4.2. Conditional ACs

Inference in probabilistic graphical models such as CRFs is typically intractable. An appealing alternative is tractable probabilistic models, which can efficiently answer any marginal or conditional probability query. Our focus is on arithmetic circuits (ACs) (Darwiche, 2003), a particularly flexible tractable representation. An arithmetic circuit (AC) is a tractable probabilistic model over a set of discrete random variables, $P(\mathcal{X})$. An AC consists of a rooted, directed, acyclic graph in which interior nodes are sums and products. Each leaf is either a non-negative model parameter or an indicator variable that is set to one if a particular variable can take on a particular value.

For example, consider a simple Markov network over two binary variables with features $f_1 = y_1 \wedge y_2$ and $f_2 = y_2$:

$$P(Y_1, Y_2) = \frac{1}{Z} \exp(w_1 f_1 + w_2 f_2).$$

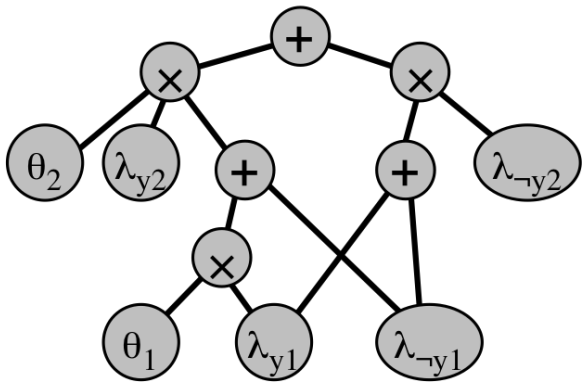


FIGURE 4.1. Simple arithmetic circuit that encodes a Markov network with two variables y_1 and y_2 and two features $f_1 = y_1 \wedge y_2$ and $f_2 = y_2$

Figure 4.1 represents this probability distribution as an AC, where $\theta_1 = e^{w_1}$ and $\theta_2 = e^{w_2}$ are parameters, and $\lambda_{y_1} = 1_{(y_1=1)}$ and $\lambda_{y_2} = 1_{(y_2=1)}$ are indicator variables.

In an AC, to compute the unnormalized probability of a complete configuration $\tilde{P}(\mathcal{X} = \mathbf{x})$, we first set the indicators variable leaves to one or zero depending on whether they are consistent or inconsistent with the values in $\mathbf{x}$. Then we evaluate each interior node from the bottom up, computing its value as a function of its children. The value of the root node is the unnormalized probability of the configuration. However, the real strength of ACs is their ability to efficiently marginalize over an exponential number of variable states. To compute the probability of a partial configuration, set all indicator variables for the marginalized variables to one and proceed as with a complete configuration. The normalization constant Z can similarly be computed by setting all indicator variables to one. Conditional probabilities can be computed as probability ratios. For example, for the AC in Figure 4.1, we can compute the unnormalized probability $\tilde{P}(y_1)$ by setting $\lambda_{\neg y1}$ to zero and all others to one, and then evaluating the root. To obtain the normalization constant, we set all indicator variables to one and again evaluate the root.

Sum-product networks (SPNs) (Poon and Domingos, 2011) are closely related to ACs – both represent probability distributions as a computation graph of sums and products, and both support linear-time inference. In discrete domains, SPNs can be efficiently converted to ACs and vice versa Rooshenas and Lowd (2014). For representing and manipulating tractable log-linear models, ACs are a better fit, since they represent parameters directly as parameter nodes rather than implicitly as edge weights.

This efficient marginalization relies on two properties of ACs (Darwiche, 2003; Lowd and Domingos, 2008): 1) An AC is *decomposable* if the children of a product node have no common descendant variable. 2) An AC is *smooth* if the children of a sum node have identical descendant variables. We say that an AC is *valid* if it satisfies both properties. Valid ACs can compactly represent probabilistic graphical models with low tree-width, sum-product networks (Poon and Domingos, 2011; Rooshenas and Lowd, 2014), and many high tree-width models with local structure (Chavira and Darwiche, 2005).

A valid AC can efficiently marginalize over any variables, but this comes at a cost: converting a Bayesian or Markov network to a valid AC could lead to an exponential blow-up in size. For discriminative tasks, however, we only need a *conditional* probability distribution, $P(\mathcal{Y}|\mathcal{X})$. In this case, we will never need to marginalize over any variables in $\mathcal{X}$, since we assume they are given as evidence. By relaxing the validity constraints over those variables, we can obtain a more compact AC.

Definition 4.1. An AC over query variables $\mathcal{Y}$ and evidence variables $\mathcal{X}$ is conditionally valid if it is smooth and decomposable over $\mathcal{Y}$.

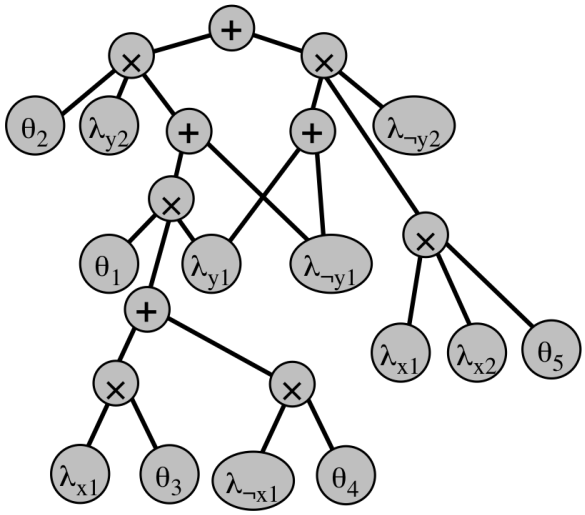


FIGURE 4.2. Conditional arithmetic circuit that encodes a conditional random field over two binary query variables y_1 and y_2, two binary evidence variables x_1 and x_2, and five features $f_1 = y_1 \wedge y_2$, $f_2 = y_2$, $f_3 = y_1 \wedge y_2 \wedge x_1$, $f_4 = f_3 = y_1 \wedge y_2 \wedge \neg x_1$, and $f_5 = \neg y_2 \wedge x_1 \wedge x_2$.

Note that replacing indicator variables for $\mathcal{X}$ with constants does not affect smoothness or decomposability over $\mathcal{Y}$. Therefore, after conditioning on any evidence $\mathbf{x}$ (that is, assigning values to indicator variables as described earlier), we are left with a valid AC over $\mathcal{Y}$. A conditionally valid AC therefore defines a *tractable conditional probability distribution*: for each configuration of evidence variables, it defines a tractable distribution over query variables.

Relaxing the definition of validity effectively allows features to be conditioned on arbitrary evidence without substantially increasing the complexity of inference. This ability to add complex dependencies on the evidence is also a stated benefit of CRFs. ACs offer the additional benefits of rich structure and tractable inference over the query variables. See Figure 4.2 for an example of how a conditionally valid AC can remain compact while including numerous dependencies on the evidence.

4.3. DAClearn

Learning CRFs includes structure learning, finding the set of feature functions f, and parameter learning, finding the optimal values for $\theta = e^w$ through joint optimization.

DACLearn builds on methods for learning tractable Markov networks, the ACMN algorithm in particular (Lowd and Rooshenas, 2013). As with learning a Markov network, DACLearn performs a greedy search through the combinatorial space of conjunctive features, using the size of the corresponding AC as a learning bias. However, rather than optimizing log-likelihood, DACLearn optimizes the conditional log-likelihood (CLL) of the training dataset $\mathcal{D}$:

$$\begin{aligned} CLL(\mathcal{D}) &= \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} \log P(\mathbf{y}|\mathbf{x}) \\ &= \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} \sum_j w_j f_j(\mathbf{d}_j) - \log Z(\mathbf{x}) \end{aligned} \qquad \text{(Equation 4.3)}$$

where $\mathbf{d}_j$ denotes the values of $\mathbf{y}$ and $\mathbf{x}$ for the variables that are in the scope of f_j. As mentioned earlier, the partition function Z depends on evidence, so in order to compute the CLL objective function, we need to run inference in the model for every example. Therefore, the complexity of evaluating the CLL objective function is $|\mathcal{D}|$ times larger than the complexity of inference in the model, which increases the importance of efficient inference.

Moreover, similar to ACMN, DACLearn updates the circuit that represents the CRF model as it adds features to the model. However, DACLearn maintains a conditionally valid AC to represent a conditional distribution.

These two changes allow us to learn arbitrary CRFs, where the conditional distribution over the query variables is always tractable. As with ACMN, these models may have high treewidth over the query variables yet remain tractable due to context-specific independence among the features.

In the following sections, we describe our procedures for structure search, parameter learning, and updating circuits in more detail.

4.3.1. Structure search

The goal of structure search is to find features that increase the value of the CLL objective function, Equation 5.3, without vastly increasing the complexity of inference in the model. Therefore, following Lowd and Domingos (2008) we optimize a modified objective that penalizes circuits with more edges:

$$\text{Score}(C, \mathcal{D}) = \log P(\mathcal{D}; C) - \gamma n_e(M) - \lambda n_p(M)$$

where $\log P(\mathcal{D}; C)$ is the CLL of the training data, n_e is the number of edges in the circuit, and n_p is the number of parameters, to help avoid overfitting. γ and λ are hyperparameters that adjust how much additional edges and parameters are penalized.

The value of adding a feature can thus be determined by its effect on this score function. However, rather than adding features individually, we have found it to be more effective to add groups of related features at once. We define a *candidate feature group* $\mathcal{F}(f, v)$ to be the result of extending an existing feature

f with all states of variable V:

$$\mathcal{F}(f, v) = \bigcup_{i=1}^{k} f \wedge v^i, \qquad \text{(Equation 4.4)}$$

where v^i denotes the ith state of some variable V with cardinality k. We score the candidate feature group as a whole instead of scoring each candidate feature separately:

$$Score(\mathcal{F}) = \Delta_{cll}(\mathcal{F}) - \gamma\Delta_e(\mathcal{F}) - \lambda|\mathcal{F}|, \qquad \text{(Equation 4.5)}$$

where Δ_{cll} and Δ_e denote the change in CLL and the number of edges, respectively, resulting from adding this set of features to the current circuit.

In order to compute the score of candidate feature group $\mathcal{F}$, we need to compute the increment in the likelihood, which requires optimizing the weight of each candidate feature in the group. For efficiency, while scoring a feature group we assume that the weights of the other features are fixed, an approach also used by McCallum (2003). This leads to an approximation of the CLL gain that can be optimized without rerunning inference in the model. Specifically, if we add a candidate feature group $\mathcal{F}(f, v)$ into the model while keeping the other parameters fixed, we can update the partition function using the following relation:

$$\Delta \log Z(\mathcal{X}) = \log(\sum_i \exp(\theta_i) P(f_i|\mathcal{X}) + P(\neg f|\mathcal{X})), \qquad \text{(Equation 4.6)}$$

where $f_i = f \wedge v_i$, and θ_i is the weight of f_i. P is the probability distribution represented by the current model. Using an AC, we can compute $P(f_i|\mathcal{X})$ for all

$f_i \in \mathcal{F}$ by running inference in the circuit: once to compute the partition function $Z(\mathcal{X})$ and once again to compute all unnormalized $\widetilde{P}(f_i)$. These unnormalized probabilities can be computed in parallel by differentiating the circuit (see Darwiche (2003) for details). For each feature f, we can re-use the expectation and partition function for candidate feature groups $F(f, v)$ for every variable v.

To find θ_i, we maximize the increment in the CLL function:

$$\Delta_{cll}(\mathcal{F}) = \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} (\sum_i (\theta_i \widetilde{P}(f_i|\mathbf{x})) - \Delta \log Z(\mathbf{x})), \qquad \text{(Equation 4.7)}$$

where $\widetilde{P}$ is the empirical probability distribution. The gradient of Equation 4.7 with respect to θ_i becomes:

$$\frac{\partial \Delta_{cll}(\mathcal{F})}{\partial \theta_i} = \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} \frac{\exp(\theta_i) P(f_i|\mathbf{x})}{\exp(\Delta \log Z(\mathbf{x}))} \qquad \text{(Equation 4.8)}$$

As a result, optimizing Equation 4.7 does not require inference in the candidate model.

Nevertheless, fixing the existing parameters is very restrictive since adding new features may affect the optimal weights of the current features; thus we have to relearn the parameters after adding each feature through joint parameter optimization. In our experiments, we found that it sufficed to perform this optimization incrementally, running just one step of gradient descent after adding each feature. This works because the optimal weights for most features do not change very much after adding one feature group, so even one step of gradient descent is sufficient to keep weights close to their optimal values throughout

Algorithm 3 DACLearn

```
C ← AC representing initial structure. //Section 4.3.1.1.
fs ← ∅ //candidate feature group max heap.
of ← ∅ //omitted candidates max heap.
fh ← feature max heap based on feature support.
//t is feature batch size.
while fs ≠ ∅ do
    F ← fs.pop()
    s ← Δ_cll(F) − λ|F|
    if s > γΔ_e(F) then
        Update C
        Joint parameter optimization
    else
        of.push(F)
    end if
    if size of C > max size then Stop.
end while
for i=1 to t do
    f ← fh.pop()
    if Support of f < min support then
        γ ← γ/2 //Shrink edge cost.
        if γ < γ_min then
            Stop
        else
            fs ← of; of ← ∅
        end if
        break
    else fs ← GenCandidates(f)
    end if
end fornot Stop
return C
```

structure learning. After structure learning is complete, we fine-tune all model parameters by running joint parameter optimization to convergence.

When we add a candidate feature group to the model, the scores of all the other candidate feature groups become obsolete, so we have to re-score them. If we have m candidate feature groups, we may re-score a candidate feature group $O(m)$ times. Therefore, if we initially generate all possible candidate feature groups we would end up with an exponential number of candidate feature groups that makes

re-scoring become the bottleneck of the learning process. To address this problem, we use a greedy approach, Algorithm 4, to have a small set of candidate feature groups (candidate set) at every point of the structure search.

This heuristic is based on the idea that interesting features may have more support in the data, a heuristic also used by Haaren and Davis (2012) to learn Markov networks. Although this heuristic will sometimes overlook higher-scoring feature groups, our experiments verify that, using this heuristic, the algorithm can better explore the feature space, resulting in more accurate models.

As shown in Algorithm 3, DACLearn maintains a set of candidate feature, groups ordered by Δ_{cll}. Each candidate feature group that increases the model score is added to the model. When the current set of feature groups has been exhausted, it selects the t current features with the most support in the data and uses them to generate new candidate feature groups. Algorithm 4 shows the process of generating candidate features groups, which consists of extending an existing feature with all possible variables and computing the conditional likelihood gain of each new group. Therefore, we need to optimize Equation 4.7 $O(t|V|)$ times for each round of candidate feature generation.

This process of adding feature groups, ordered by CLL, and generating new features groups, ordered by feature support, continues until no feature remains with more than minimal support or the model's size reaches a predefined maximum size limit.

During structure search, we may omit some candidate feature groups because of their edge costs. However, we may run out of good candidate feature groups while the size of the circuit is much smaller than the maximum circuit size. Therefore, to benefit from these candidate feature groups, we keep all the omitted

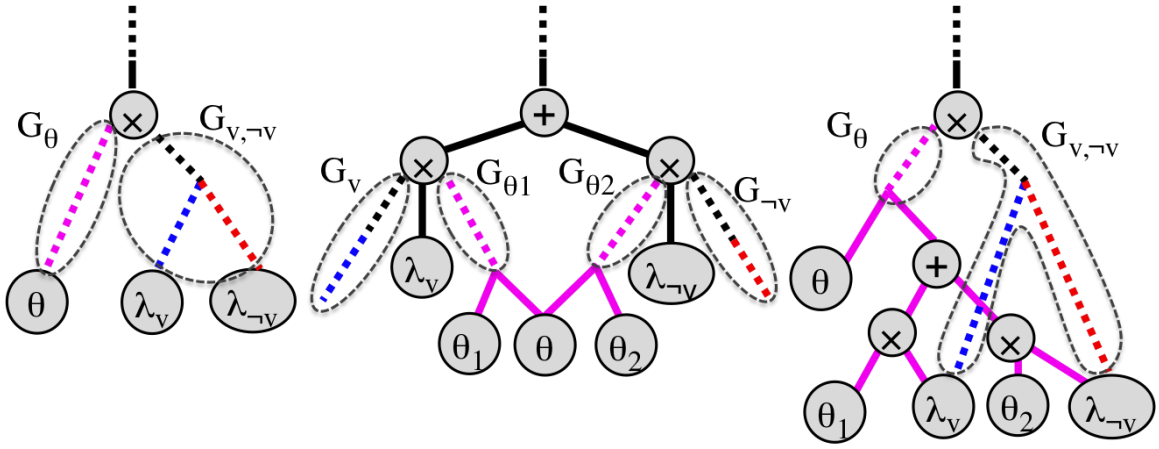


FIGURE 4.3. Updating circuits. Left: initial circuit, middle: circuit after a consistent split, and right: circuit after an inconsistent split.

Algorithm 4 Candidate feature group generator

```
procedure GenCandidates(f)
    //f is the feature used to generate candidate features.
    fs ← empty candidate feature group heap
    for each v in V do
        k ← cardinality of variable v
        F ← ⋃_{i=1}^{k} f ∧ v^i
        //λ is the feature penalty.
        if Δ_cll(F) > λ|F| then
            fs.push(F)
        end if
    end for
    return fs
end procedure
```

candidate feature groups in a priority heap. These candidates are pruned only because of the complexity penalty term, so when there are no more features to expand, we halve the edge penalty γ and re-score these candidate feature groups.

4.3.1.1. Initial structure

As discussed before, keeping the candidate set small is important to reduce the number of re-scores. Unfortunately, even scoring pairwise features would require

at least $\Omega(|\mathcal{V}^2|)$ calls to the score function, where $\mathcal{V}$ is the set of all variables. Therefore, we begin with a pre-defined initial structure consisting of heuristically chosen pairwise features. This allows us to consider more complex features given limited training time.

For each query variable Y_i, we introduce a set of potentials $\phi(Y_i, X_j)$ by selecting evidence variables X_j with high mutual information, $I(Y_i, X_j) = H(Y_i) - H(Y_i|X_j)$. Since the circuit does not have to be decomposable and smooth for the evidence variables, we are free to pick as many evidence variables as we want. The number of evidence variables chosen is a hyperparameter that we tune on validation data. We apply the same idea to the query variables. However, we have tractability constraints for query variables, so we only learn a Chow-Liu tree over the query variables. Finally, we exactly compile the initial structure to a conditionally valid AC that represent a CRF, and learn the weihts using convex optimization methods.

4.3.2. Parameter learning

We jointly optimize all parameters after compiling the initial structure, adding each feature, and after the end of the structure search. The CLL objective function is convex with respect to feature weights, so we use the L-BFGS algorithm to optimize it. We compute the gradient for all feature weights by differentiating the circuit once, which only requires two passes over the circuit (Darwiche, 2003). The partition function of Equation 5.3 depends on evidence variables, so we have to differentiate the circuit once for each example. This requires performing inference in the circuit $\Omega(|D|)$ times, which highlights the expense of joint optimization and importance of having efficient inference.

4.3.3. Updating circuits

Our circuit update method is based on the Split algorithm from ACMN. Given a feature f in the circuit with parameter θ and a binary variable V, the goal is to add two features to the circuit, $f \wedge v$ and $f \wedge \neg v$, with parameters θ_1 and θ_2, respectively. The left circuit in Figure 4.3 shows the initial circuit. G_θ indicates the sub-circuit between the common ancestor of indicator variables λ_v and $\lambda_{\neg v}$ and parameter node θ. Similarly, the sub-circuits between the common ancestor and indicator nodes λ_v and $\lambda_{\neg_v}$ are labeled G_v and $G_{\neg v}$, respectively. If V is a query variable, we have to duplicate the sub-circuit G_θ into $G_{\theta 1}$ and $G_{\theta 2}$, as shown in the middle circuit of Figure 4.3, and extend each sub-circuit using the new parameter nodes θ_1 and θ_2. Parameter node θ is attached to both sub-circuits, which ensures the decomposability and smoothness of the circuit. On the other hand, if V is an evidence variable, we have a more compact representation by avoiding the expensive duplication of G_θ. We refer to this as an *inconsistent split*. The right circuit of Figure 4.3 indicates the result of an inconsistent split, which is conditionally valid. For an inconsistent split, the number of new nodes and edges added to the circuit is significantly less than the size of G_θ that we need to duplicate for a consistent split.

4.4. Experiments

4.4.1. Datasets

We run our experiments using 20 datasets illustrated in Table 4.1 with 16 to 1556 binary-valued variables. These datasets are drawn from a variety of domains, including recommender systems, text analysis, census data, and plant

TABLE 4.1. Dataset characteristics

Dataset	Var#	Train	Dataset	Var#	Train
NLTCS	16	16181	DNA	180	1600
MSNBC	17	291326	Kosarek	190	33375
KDDCup 2000	64	180092	MSWeb	294	29441
Plants	69	17412	Book	500	8700
Audio	100	15000	EachMovie	500	4524
Jester	100	9000	WebKB	839	2803
Netflix	100	15000	Reuters-52	889	6532
Accidents	111	12758	20 Newsgroup	910	11293
Retail	135	22041	BBC	1058	1670
Pumsb-star	163	12262	Ad	1556	2461

species distribution, and have been extensively used in previous work (Davis and Domingos, 2010; Gens and Domingos, 2013; Rooshenas and Lowd, 2014; Rahman et al., 2014a).

To observe the performance of discriminative structure learning in the presence of variable number of query variables, we create two versions of these 20 datasets. In one, we label a randomly chosen 50% of the variables as evidence variables and the other half as query variables. We create the other version of the datasets by randomly selecting 80% of the variables as evidence variables while the remaining 20% are query variables.

4.4.2. Methods

For baselines, we compare to a state-of-the-art generative SPN learner, IDSPN, and a generative AC learner, ACMN. Our proposed heuristics for structure search can also help the generative ACMN algorithm to find a better structure. Therefore, we incorporate those heuristics into the ACMN algorithm, which we name efficient ACMN (EACMN). Based on our experiments on the 20 datasets, EACMN is significantly more accurate than ACMN on 13 datasets out of 20 datasets in terms of average log-likelihood of joint probability distribution, and not

significantly different on the remaining 7 datasets. Moreover, EACMN, on average, finds 1.7 times more features, while its circuits are 2.5 times more compact. These comparisons show the importance of our search heuristics. See the supplementary material for a more detailed comparison between ACMN and EACMN. By using EACMN as a baseline, we ensure that any performance gains demonstrated by DACLearn are attributable to discriminative learning, and are not simply an artifact of the new structure search efficiency heuristics.

To compare the effect of discriminative parameter learning, we also take the best models learned by EACMN, based on average log-likelihood on validation data, and relearn their parameters to maximize CLL. We call this method conditional ACMN (CAMCN). This idea is similar to discriminative learning of SPNs (Gens and Domingos, 2012), which supposes a predefined SPN structure and then applies a discriminative weight learning approach to learn the parameters.

As our last baseline, we choose MCTBN (Hong et al., 2014), which learns a mixture of conditional tree Bayesian networks. To find each tree, MCTBN needs to learn $O(n^2)$ logistic regression models, where n is the number of query variables, and each logistic regression does $O(|\mathcal{D}|)$ passes over training data. This makes MCTBN less practical when the number of query variables increases[2].

For all of the above methods, we learn the model using the training data and tune the hyper-parameters using the validation data, and we report the average CLL over the test data. To tune the hyper-parameters, we used a grid search over the parameter space. We used the original IDSPN models learned by the authors (Rooshenas and Lowd, 2014)[3].

[2]Another baseline would be learning tree CRFs (Bradley and Guestrin, 2010), however, its implementation is not usable due to a broken library dependency.

[3]`http://ix.cs.uoregon.edu/~pedram/ac_models.tar.gz`

For EACMN, CACMN, and DACLearn, we use an L1 prior of 0.1, 0.5, 1, and 2, and a Gaussian prior with a standard deviation of 0.1 and 0.5. For EACMN, we use feature penalties of 2, 5, and 10, an edge penalty of 0.1, a maximum circuit size of 2M edges, and a feature batch size of 2. For DACLearn, we use the same settings for feature penalty, edge penalty and feature batch size, but reduce the maximum circuit size to 1M edges. We also use 1, 10, 20, 30, and 40 as the number of initial evidence variables connected to each query variable. For MCTBN, we run the authors' code[4], but we tune the cost hyper-parameter on validation data, instead of using the default cross-validation. In our experiments, we found that using the validation data helps MCTBN avoid overfitting the training data. For the cost parameter we used values of 0.01, 0.05, 0.1, 0.5, 1, and 2. We also train MCTBN with 1, 2, 3, and 4 mixture components.

We bounded the learning time of all methods to 24 hours, and we ran our experiments on an Intel(R) Xeon(R) CPU X5650@2.67GHz.

Our implementations of DACLearn, EACMN, CACMN, and IDSPN are all available in the open-source Libra toolkit (Lowd and Rooshenas, 2015), available from `http://libra.cs.uoregon.edu/`.

4.4.3. Results

Table 4.2 shows the average CLL comparison of DACLearn and the other baselines on 20 datasets with 50% and 80% evidence variables[5]. We use $*$ to indicate that DACLearn has significantly better test set CLL than the corresponding method on the given dataset, and $\bullet$ for the reverse. We also use

[4] `https://github.com/charmgil/M-CTBN`

[5] For more detailed results, including timing information, see the online appendix at `http://ix.cs.uoregon.edu/~pedram/daclearn/`

TABLE 4.2. Average conditional log-likelihood (CLL) comparison. • shows significantly better CLL than DACLearn, ∗ indicates significantly worse CLL than DACLearn, and ∘ used when CLL is not significantly different. The bold numbers highlights the method that has the best CLL on each dataset. The last row, summarizes the number of wins (W), ties (T), and losses (L) of DACLearn comparing to the other baselines based on the significance results. † indicates the experiment has not finished given the 24 hour limit.

	50% Evidence variables					80% Evidence variables				
Dataset	IDSPN	EACMN	CACMN	DACL	MCTBN	IDSPN	EACMN	CACMN	DACL	MCTBN
NLTCS	-2.774∘	-2.781∗	-2.780 ∗	**-2.770**	-2.792 ∗	-1.262∘	-1.265 ∗	-1.262∗	**-1.255**	-1.263∗
MSNBC	-2.922∗	-2.925 ∗	-2.925 ∗	**-2.918**	-3.253 ∗	**-1.557**∘	-1.560 ∗	-1.560∗	**-1.557**	-1.614∗
KDDCup 2000	**-0.996**∘	-1.001 ∗	-0.999∘	-0.998	-1.009 ∗	-0.390∗	-0.387∘	**-0.386**∘	**-0.386**	-0.390∗
Plants	-4.759∗	-4.891 ∗	-4.794 ∗	**-4.655**	-4.866 ∗	-1.915 ∗	-1.928 ∗	-1.888∗	**-1.812**	-1.911∗
Audio	-19.372∗	19.647 ∗	-19.512 ∗	**-18.958**	-18.965∘	-6.645 ∗	-7.777 ∗	-7.647∗	**-7.337**	-7.343∘
Jester	-25.544∗	-25.597 ∗	-25.477 ∗	**-24.830**	-24.955 ∗	-10.437∗	-10.422∗	-10.351∗	**-9.998**	-10.004∘
Netflix	-27.051∗	-27.348 ∗	-27.282 ∗	**-26.245**	-26.309 ∗	-10.954∗	-11.065∗	-10.997∗	-10.482	**-10.476**∘
Accidents	-9.566•	-9.185•	**-9.143**•	-9.718	-10.198 ∗	-3.972∗	-3.722∗	†	**-3.493**	-3.711∗
Retail	-4.853∗	-4.845 ∗	-4.844 ∗	**-4.825**	-4.840 ∗	-1.705∗	-1.694∘	-1.691∘	-1.687	**-1.685**∘
Pumsb-star	-6.414∘	-6.844 ∗	-6.653 ∗	-6.363	**-6.002**•	-2.851∗	-3.281 ∗	†	**-2.594**	-2.661∗
DNA	-35.727∗	-34.561•	**-34.480**•	-34.737	-37.151 ∗	-12.727∗	-12.159∘	**-12.099**∘	12.116	-13.116∗
Kosarek	**-5.000**•	-5.098 ∗	-5.046∘	-5.053	-5.144 ∗	**-2.535**•	-2.594 ∗	-2.557∘	-2.549	-2.601∗
MSWeb	-5.658∘	-5.682 ∗	-5.681 ∗	**-5.653**	-5.788 ∗	-1.376∗	-1.363 ∗	-1.355∗	**-1.333**	-1.366∗
Book	**-16.530**•	-17.528 ∗	-17.115 ∗	-16.801	-16.764 ∘	-6.891∗	7.364 ∗	-7.047∗	**-6.817**	-6.979∗
EachMovie	-25.399∘	-27.354 ∗	-26.568 ∗	**-25.325**	-26.233 ∗	-9.573∘	-10.608∗	-10.029∗	**-9.403**	-9.996∗
WebKB	-74.473∗	-76.827 ∗	-75.840 ∗	-72.072	**-66.302**•	-29.127∗	-30.189∗	-29.522∗	**-28.087**	-29.891∗
Reuters-52	**-40.209**•	-43.129 ∗	-42.379 ∗	-41.544	†	**-16.853**•	-17.895∗	-17.529∗	-17.143	-17.252∘
20 Newsgroup	**-74.785**•	-78.228 ∗	-77.831 ∗	-76.063	†	-28.443∗	-29.674∗	-29.442∗	**-27.918**	-29.176∗
BBC	-121.798∗	-124.539∗	-123.504 ∗	-118.684	**-93.192**•	-46.116∗	-47.298∗	-46.381∗	**-44.811**	44.818∘
Ad	-7.349∗	-5.184 ∗	**-4.658**•	-4.893	†	-2.341∗	-1.658∗	-1.505∗	**-1.370**	-1.546∗
W/T/L	10/5/5	18/0/2	15/2/3	N/A	12/2/3	15/3/2	17/3/0	14/4/0	N/A	14/6/0

$\circ$ to show that two methods are not significantly different. The bold numbers only highlight which method out 5 has the better average CLL on the given dataset. Wins and losses are determined by two-tailed paired t-tests ($p < 0.05$). Based on the results, DACLearn is never significantly worse than MCTBN, CACMN, and EACMN, on the 20 datasets with 80% evidence variables, and only is significantly worse than IDSPN on two datasets. Furthermore, DACLearn has better average CLL than the other methods on 15 datasets. As we decrease the number of evidence variables, the benefits of discriminative learning diminish (as expected), but DACLearn still significantly outperforms the other methods on many datasets. MCTBN reaches the 24 hour limit without training all the needed $O(n^2)$ logistic regressions on 3 datasets. Table 4.2 also shows that discriminative parameter learning is no substitute for discriminative structure learning. The performance of CACMN is much closer to EACMN than to DACLearn. This is because DACLearn learns conditionally valid ACs, which allows DACLearn to consider many models that EACMN and CACMN cannot compactly represent as valid ACs. On two of the three datasets where CACMN is significantly better than DACLearn, EACMN is also significantly better. DACLearn also compares favorably to IDSPN Rooshenas and Lowd (2014), in spite of the fact that IDSPN learns models with hidden variables and DACLearn does not.

As discussed earlier, conditionally valid ACs representing $P(\mathcal{Y}|\mathcal{X})$ are more compact than valid ACs representing $P(\mathcal{Y}, \mathcal{X})$. We verify this empirically by measuring the size of circuits learned with each method. The average sizes of the circuits learned by IDSPN and EACMN are 2.2M and 1.1M edges, respectively, while the average size of the circuits learned by DACLearn is 55K edges when we have 50% evidence variables and 22K edges when we have 80% evidence variables.

This means that inference in conditionally valid ACs is 100 times faster than IDSPN when we have 80% evidence variables!

CACMN is actually less efficient than DACLearn overall, since it performs weight learning on the much larger ACs learned by EACMN. As a result, it runs out of time on two datasets. We can avoid this problem by restricting EACMN to learn smaller models, although this sacrifices accuracy. It is also informative that CACMN could finish learning using the same circuits when we have 50% evidence variables, because when we have less evidence it is more likely that more examples share the same evidence setting, and since the partition function only depends on evidence, we need to perform inference fewer times.

4.5. Summary

Tractable probabilistic models are a promising alternative to Bayesian networks, Markov networks, and other intractable models. DACLearn builds on previous successful methods for learning tractable probabilistic models, extending them to learning conditional probability distributions. By optimizing conditional likelihood and learning a conditionally valid AC, DACLearn obtains more accurate and more compact ACs than previous generative approaches.

DACLearn is limited to learning conjunctive features over the observed variables. Previous work with SPNs has shown that mixtures often lead to higher accuracy. For example, IDSPN uses hierarchical mixtures of tractable Markov networks to obtain consistently better results than tractable Markov networks alone (Rooshenas and Lowd, 2014). Learning tractable conditional distributions with latent variables remains an important open problem.

CHAPTER V

GENERALIZED ARITHMETIC CIRCUITS

Recently, there has been an emerging interest in using non-linear function estimation, especially deep neural networks, in many domains such as vision, speech recognition, and language modeling. Deep neural networks such as convolutional neural networks and recurrent neural networks achieve the state-of-the-art performance in many tasks such as image classification and image segmentation (Clevert et al., 2015; Lee et al., 2016). However, deep neural networks cannot represent functions with structured outputs without exponential blow-up in the size of the networks. Therefore, recently a combination of neural networks and graphical models has been introduced (Johnson et al., 2016; Chen et al., 2015; Zheng et al., 2015; Sohn et al., 2015). These models benefit from using graphical models to represent interactions among the output variables, while deep neural networks correlate input variables with the output variables. For applications such as structured output prediction, in which only the MAP state of variables is important, we can also learn the interaction of output variables using deep neural networks (Belanger and McCallum, 2016; Amos et al., 2016). The main disadvantage of these approaches is that exact inference is not tractable.

To the best of our knowledge, the combination of graphical models and neural networks have not been studied for representing tractable conditional distributions, which is the target of this chapter.

5.1. Motivation and background

Many prediction tasks such structured output prediction or multi-label classification can be cast as inference in conditional distributions. For example, in multi-label classification problems, we are interested in finding the set of most relevant labels L to a given input $\mathbf{x}$. Suppose we represent each label l_i with a binary variable y_i such that $y_i = 1$ for all $l_i \in L$ and $y_i = 0$ otherwise. Then the problem of finding the set of most relevant labels of input $\mathbf{x}$ is equal to finding the MAP state of the conditional distribution that relates output variables $\mathcal{Y}$ with input variables $\mathcal{X}$: $\mathbf{y}_{map} = \arg\max_{\mathbf{y}} P(Y|\mathbf{x})$, where $\mathbf{x}$ is an instantiation of input variables. Similarly, we can represent any output prediction problem with a fixed set of input and output variables as an inference problem in a conditional distribution. For example, in the problem of image denoising, input and output variables are the pixels of noisy images and denoised images, respectively.

Although training of conditional distributions using different intractable representations has been studied (Sohn et al., 2015; Peng et al., 2009; Mnih et al., 2012), only a few works address learning tractable representations (Hong et al., 2014; Bradley and Guestrin, 2010; Li et al., 2016). The main problem of learning tractable conditional distributions is the dependence of the partition function on the input variables, which significantly increases the complexity of structure learning. Moreover, the existing approaches do not learn or are not able to learn a rich structure over output variables, which makes them only suitable for simpler problems where the output variables are not highly correlated given the input.

Bradley and Guestrin (2010) introduce a method for learning tree conditional random fields. Their method is based on finding the maximum spanning tree over output variables given some heuristics for scoring each edge. These heuristics

suppose a known local dependence among output variables and input variables. Hong et al. (2014) introduce a mixture of conditional trees as a tractable representation for conditional distributions. They use a similar approach for learning each tree over output variables, but they approximate the score of each edge using logistic regression. As a result, their method has to train $|\mathcal{Y}|^2$ logistic regressions to learn each conditional tree. The approach by Li et al. (2016) also adopts mixture models, but it learns a mixture of conditional Bernoullis, which is more restricted than the mixture of conditional trees, but its learning complexity is linear in $|\mathcal{Y}|$. Although mixture models can be used to learn tractable representations for marginal and conditional queries, answering MAP queries remains intractable, thus requiring approximation techniques.

In the following section, we introduce generalized arithmetic circuits (GACs) as a tractable representation for conditional distributions. GACs are able to represent the prior work on learning tractable conditional distributions as well as representations with more complex structure over output variables.

5.2. Definition and properties

A generalized arithmetic circuits (GAC) is a rooted directed acyclic graph, in which intermediate nodes are operators and leaves are indicator variables, input variables and parameters. Operators include linear operators, sum and product, as well as non-linear operators such as sigmoid, exponentiation, or rectified linear unit.

For a discrete variable y_i, similar to ACs, an indicator variable $\lambda_{y_i}^k$ is a binary variable that depicts the state of variable y_i in a given instantiation $\mathbf{y}$ of variables $\mathcal{Y}$: $\lambda_{y_i}^k$ is one if $y_i = k$ or y_i does not appear in the instantiation $\mathbf{y}$; otherwise, $\lambda_{y_i}^k$ is zero. Therefore, we can assign values to all indicator variables given an

instantiation of variables. We define this assignment to indicator variables using $\Lambda_{\mathbf{y}}$, which is a matrix[1] such that $\Lambda_{\mathbf{y}}^{ik}$ is the corresponding value of indicator $\lambda_{y_i}^k$ given the instantiation $\mathbf{y}$ of all discrete variables:

- $\Lambda_\emptyset$ is a matrix whose all entries are one.
- If all variables appear in instantiation $\mathbf{y}$ then $\forall i, \sum_j \Lambda_{\mathbf{y}}^{ij} = 1$.

So far, we understand a GAC as a function G over two sets of variables $\mathcal{Y}$ and $\mathcal{X}$. However, we are interested in computing conditional distributions $P(\mathcal{Y}|\mathcal{X})$ given an unnormalized function G. Therefore, we need to compute the normalization constant or partition function: $Z(\mathbf{x}) = \sum_{\mathbf{y}} G(\Lambda_{\mathbf{y}}, \mathbf{x}; \Omega)$. However computing $Z(\mathbf{x})$ is intractable in general, but linear in the size of GAC given some constraints in the graph. To establish these constraints, we have to define some terms:

Definition 5.1. The *scope* of an intermediate node n over variables $\mathcal{Y}$, $s(n, \mathcal{Y})$, is a set of variables in $\mathcal{Y}$ whose corresponding indicator variables appear in the subgraph rooted at n.

Definition 5.2. A product node is *decomposable* over a set of variables $\mathcal{Y}$ if the scopes of its children over $\mathcal{Y}$ are disjoint.

Definition 5.3. A sum node is *complete* over a set of variables $\mathcal{Y}$ if the scopes of its children over $\mathcal{Y}$ are the same.

Definition 5.4. A non-linear node is *avoidable* for a set of variables $\mathcal{Y}$ if its scope over $\mathcal{Y}$ is empty.

Let $\mathcal{Y}_T \in \mathcal{Y}$ be the set of variables that are avoidable for all non-linear nodes and let $\mathcal{Y}_N = \mathcal{Y} - \mathcal{Y}_T$. .

[1] In general, $\Lambda_{\mathbf{y}}$ is a matrix if all the variables in $\mathbf{y}$ have similar dimensions, which we assume to simply the notations.

$G(\Lambda, \mathcal{X}, \Omega)$ is a tractable representation for $P(\mathcal{Y}_T|\mathcal{Y}_N, \mathcal{X}; \Omega)$ if:

- Every sum node is complete over $\mathcal{Y}_T$.
- Every product node is decomposable over $\mathcal{Y}_T$.
- Every non-linear node is avoidable for $\mathcal{Y}_T$.

Given a tractable representation GAC $G(\Lambda, \mathcal{X}; \Omega)$ for $P(\mathcal{Y}_T|\mathcal{Y}_N, \mathcal{X}; \Omega)$, the following quantities can be answered exactly:

- The partition function $Z(\mathbf{y}_N, \mathbf{x})$ using a feed-forward evaluation of G, i.e. evaluating all nodes in G from the leaves to the root:

$$Z(\mathbf{y}_N, \mathbf{x}) = G(\Lambda_{\mathbf{y}_N}, \mathbf{x}; \Omega)$$

- All conditional marginals $P(y_i|\mathbf{y}_N, \mathbf{x})$ for all $y_i \in \mathcal{Y}_T$ using a feed-forward[2] and a back-propagation pass over G.

Nevertheless, we can also compute $Z(\mathbf{x})$, but its computation is exponential in the number of vars in $\mathcal{Y}_N$, $|\mathcal{Y}_N|$:

$$Z(\mathbf{x}) = \sum_{\mathbf{y}_N \in Y_N} Z(\mathbf{y}_N, \mathbf{x}), \qquad \text{(Equation 5.1)}$$

where each term $Z(\mathbf{y}_N, \mathbf{x})$ is computed by assigning one possible instantiation to the variables $\mathcal{Y}_N$. Therefore, we refer to $\mathcal{Y}_T$ and $\mathcal{Y}_N$ as tractable and intractable variables, respectively.

[2]Feed-forward and back-propagation in GACs are similar to neural networks.

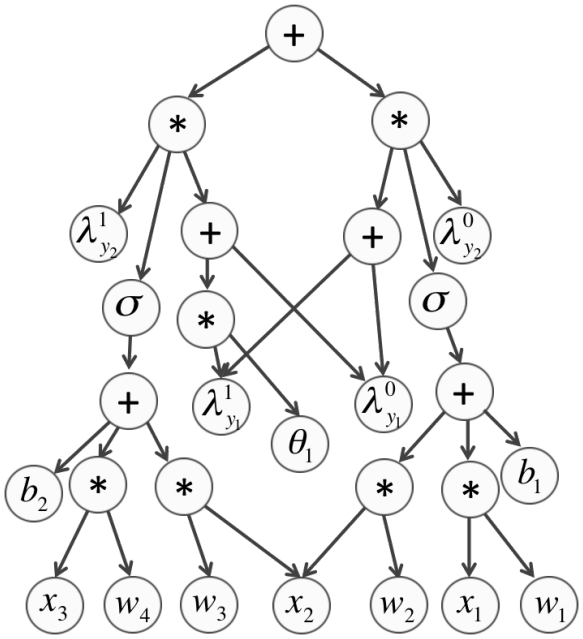


FIGURE 5.1. An example of a generalized arithmetic circuit with two tractable variables y_1 and y_2 and three input variables x_1, x_2, and x_3.

5.2.1. Representational power

GACs can represent different structures. GACs can represent any neural network by unrolling the multilayer perceptron into its sum, product, or non-linear functions. In general, the set of tractable variables of a GAC representation of any neural network is empty. Sum-product networks and arithmetic circuits can also be represented by GACs with no input variables, intractable variables, and non-linear nodes.

GACs can represent more complex structures such as mixture of conditional Bernoullis or a combination of neural networks and graphical models.

Conditional exponential families are the other models that GACs can represent. In general, a log-linear model with conjunctive feature functions

$F(\Lambda(\mathcal{Y}))$ and parameter functions $\Phi(X, \Omega)$ can be represented using GACs:

$$P(\mathcal{Y}|\mathcal{X}) = \exp(F(\Lambda)^T \Phi(X, \Omega) - \log Z(\mathcal{X}, \Omega)), \qquad \text{(Equation 5.2)}$$

For example, for univariate features $f_1 = \lambda^1_{y_2}$ and $f_2 = \lambda^0_{y_1}$, bivariate feature $f_3 = \lambda^1_{y_1} \wedge \lambda^1_{y_2}$, parameter functions $\phi_1(\mathcal{X}, W) = \sigma(w_1 x_1 + w_2 x_2 + b_1)$ and $\phi_2(\mathcal{X}, W) = \sigma(w_3 x_2 + w_4 x_3 + b_2)$, where σ is a sigmoid function, and constant parameter θ_1, the conditional distribution is defined by $\exp(f_1 \log \phi_1 + f_2 \log \phi_2 + f_3 \log \theta_1 - \log Z(\mathcal{X}, \Omega))$, where $\Omega = [w_1; w_2; w_3; w_4; \theta_1]$ is the set of parameters. The GAC that represents this distribution is depicted in Figure 5.1.

5.3. Learning

Learning GACs includes parameter learning as well as structure search. However, the main objective of these two phases can be same. We choose to maximize the conditional log-likelihood over the trainset $\mathcal{D}$:

$$CLL(\mathcal{D}) = \sum_{(\mathbf{y},\mathbf{x}) \in \mathcal{D}} \log \hat{P}(\mathbf{y}|\mathbf{x}) - \log Z(\mathbf{x}, \Omega), \qquad \text{(Equation 5.3)}$$

where $\hat{P}(\mathbf{y}|\mathbf{x}) = G(\Lambda_{\mathbf{y}}, \mathbf{x}, \Omega)$ and Z is the partition function.

To reduce the complexity of structure search, our initial learning algorithms focus on a specific subclass of GACs. We only explore GACs that represent the following exponential family:

$$GAC(\Lambda, \mathcal{X}; [W; \Theta]) = \exp\left(\sum_i \sum_{j=1}^{|y_i|} \Lambda^{ij} \log \phi_{ij}(\mathcal{X}, W) + \Theta^T \Psi(\Lambda)\right), \quad \text{(Equation 5.4)}$$

where Ψ is a set of multivariate conjunctive features over the indicator variables. Moreover, we require that all the output variables are tractable. We also suppose that ϕ_{ij} are the outputs of a standard multilayer perceptron (MLP) parametrized by W over input variables, unrolled as a subgraph of the GAC structure. However, the output non-linearity should be only positive; therefore, we used a biased sigmoid: $\sigma(x) = \epsilon + \frac{1}{1+\exp(-x)}$ for some $\epsilon > 0$. All the other non-linearities are smoothed rectified linear units: $\log(1 + \exp(.))$.

Therefore, the structure search reduces to finding the set of conjunctive features $\Psi(\Lambda)$. To find feature functions, we build on the DACLearn algorithm (Rooshenas and Lowd, 2016), which discriminatively learns a conditional AC.

5.3.1. Structure search

The goal of structure search is to find a set of feature functions Ψ that maximize the conditional log-likelihood Equation 5.3.

Since univariate potentials Φ and high-order features Ψ are correlated through output variables, changing Ψ also affects Φ. Nevertheless, to simplify the structure search we fix W and Φ throughout the structure search, and only update Θ and Ψ.

Building on DACLearn (Rooshenas and Lowd, 2016), we use a greedy search in the space of all possible conjunctive features. We first construct a candidate feature group for each feature in set Ψ: $\mathcal{F}(f, v) = \cup_{i=1}^{|v|} f \wedge v^i$, where $v \in \mathcal{Y}$.

We score each candidate feature group using the increase in the conditional log-likelihood, Δ_{cll}, penalized by the increase in the inference complexity,Δ_e:

$$Score(\mathcal{F}) = \Delta_{cll}(\mathcal{F}) - \gamma\Delta_e(\mathcal{F}) - \alpha|\mathcal{F}|, \qquad \text{(Equation 5.5)}$$

where γ is the edge penalty and α is the parameter penalty[3].

After scoring all candidate features, we pick the candidate feature groups with the highest score and update the GAC in order to represent it.

The new features are used to generate new candidate feature groups. However, adding one candidate feature group to the model invalidates the score of previously scored candidate feature groups. Therefore, finding the next best candidate feature groups requires re-scoring the whole set of candidate feature groups, which is intractable in practice. In order to increase the efficiency of this process, we keep the candidate feature groups in a priority group ordered by their scores and re-score candidate features groups. As long as we find a candidate feature group such that its new score is greater than or equal to its previous score, we add the candidate feature group to the model. This process only approximates the greedy structure search algorithm since it is possible that the score of a previously low-score candidate feature group increases significantly after adding one feature, relative to other candidate feature groups. To alleviate this effect, we periodically re-score the whole set of candidate feature groups.

Updating circuits is based on the procedure described in ACMN (Lowd and Rooshenas, 2013).

5.4. Experiments

In this section, we discuss the properties of GACs using extensive experiments on different problems.

[3]See Section 4.3 for computing Δ_{cll} and Δ_e.

	ID-SPN	DACLearn	MCTBN	GACLearn
WebKB	-29.127	-28.087	-29.891	**-27.860**
Reuters-52	-16.853	-17.143	-17.252	**-16.841**
20 Newsgroups	-28.443	-27.918	-29.176	**-27.399**
BBC	-46.116	-44.811	-44.818	**-43.418**

TABLE 5.1. Average conditional log-likelihood comparison of GACLearn and the baselines.

5.4.1. Inference complexity

We train GACs on the Jester dataset (Goldberg et al., 2001) for which we randomly select 80% of the variables as input variables and the others as output variables.

We train two models: one with no hidden layer and the other with one hidden layer of 100 nodes. Figure 5.2 shows the conditional log-likelihood of two models as we add high-order features. The GAC with no hidden layer starts with a lower CLL compared to the model with one hidden layer, meaning that the hidden layer helps more in the pretraining phase to learn better univariate potentials that better describe the data. As we add more features to the model with no hidden layer, the difference in CLL decrease since the high-order potentials model the interaction of variables that was not captured during the pretraining phase. As illustrated by Figure 5.2, both models finally converge to the same CLL; however, the model with no hidden layer needs more structure over the output variables. As shown in Figure 5.3, this causes an exponential increase in the size of the circuit without a hidden layer. This experiment suggests that we can have more compact GAC models if we can learn better univariate potentials.

FIGURE 5.2. Trainset negative conditional log-likelihood versus the number of high-order features for GACs with multilayer perceptron with no hidden layer and one hidden layer.

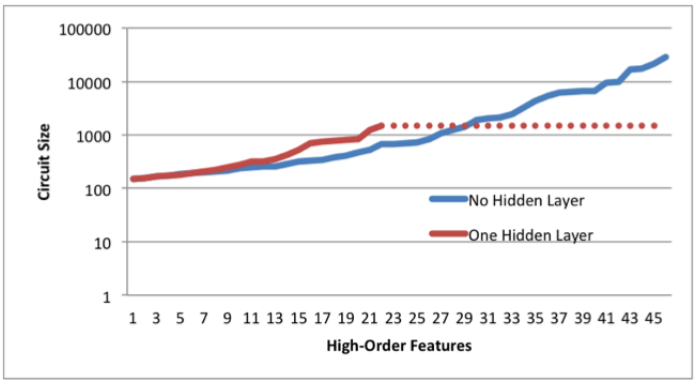


FIGURE 5.3. The number of edges in the circuit versus the number of high-order features for GACs with multilayer perceptron with no hidden layer and one hidden layer.

5.4.2. Conditional log-likelihood

To understand how GACLearn compares to other tractable probabilistic models, we compare GACLearn with ID-SPN (Rooshenas and Lowd, 2014), DACLearn (Rooshenas and Lowd, 2016) and MCTBN (Hong et al., 2014). ID-SPN is a state-of-the-art SPN learner. DACLearn discriminatively learns the structure of conditional arithmetic circuits, which is a special case of GACs with no non-linear nodes and no input variables. MCTBN is another tractable model that learns a mixture of conditional trees.

TABLE 5.2. Comparison of structural SVM, SPEN, and GACLearn using hamming loss on the Yeast dataset.

Exact	LP	LBP	SPEN	GACLearn
20.2 ± 0.5	20.5 ± 0.5	24.3 ± 0.6	20.0 ± 0.3	$\mathbf{19.9 \pm 0.1}$

We compare these algorithms on a set of text datasets: Reuters-52, 20 Newsgroups, BBC, and WebKB. In these datasets, each sample is a bag-of-words representation of a document. We randomly choose 80% of the words to be given as input and compare the algorithms based on the conditional log-likelihood of the other 20% of words. Table 5.1 shows the average conditional log-likelihood of the 20% output words given the 80% input words, based on which we can conclude that GACLearn is constantly better than the other baselines.

5.4.3. Multi-label classification

We compare GACLearn, which learns a tractable GAC representation, with SPEN (Belanger and McCallum, 2016) and a structural SVM with exact and approximate inference on a multi-label classification task on the Yeast dataset (Elisseeff and Weston, 2001), which includes 103 input features of gene expressions with 1500 genes in the training set and 917 genes in the test set. Each gene can be labeled with one or more of 14 possible groups, which defines a multi-label classification problem.

The GAC structure has only one layer of 103 hidden variables.

Table 5.2 shows that GACLearn performs at least as well as a structural SVM with exact inference. This achievement is mostly attributable to the potential of GACLearn for learning high-order feature functions, while the structural SVM only

benefits from univariate and pairwise features. Moreover, GACLearn is also as good as SPEN, due to its ability to run exact inference.

5.5. Summary

Learning tractable conditional distributions is important in many real-world problems. In this chapter, we introduced generalized arithmetic circuits (GACs), which can represent complex structures over input and output variables, while providing tractable exact inference. We also have introduced GACLearn for learning GAC representations from data, and experimentally shown that GACLearn can learn more expressive representations than tractable and intractable baselines.

CHAPTER VI

CONCLUSION AND FUTURE DIRECTIONS

In this dissertation, we have studied the problem of learning tractable representations for probability distributions. These tractable models benefit from local structures to compactly represent relations among random variables. More specifically, we have introduced ID-SPN for learning sum-product networks (SPNs). ID-SPNs represent direct interaction among random variables through Markov networks represented using arithmetic circuits and represent indirect interaction using mixture models. As we have shown the final representation is a valid SPN that outperforms other methods for learning tractable models as well as the state-of-the-art method for learning intractable Bayesian networks.

Discriminative structure learning is the other problem that we have discussed in this dissertation. As we have shown, when a set of variables always appear as evidence, then discriminatively learning conditional distributions given the evidence set (instead of learning joint distributions over all query and evidence variables) results in more compact representations. Therefore, we have studied the approaches for discriminatively learning conditional distributions using tractable representations. We have introduced two tractable representations: conditional ACs and generalized ACs. The former is a more compact representation for conditional tractable distributions comparing to ordinary ACs. The latter, on the other hand, extends the operations of ACs to non-linear operations, which enables us to learn a more expressive representation over the evidence variables while we still have a tractable representation over query variables. We have also introduced DACLearn and GACLearn for learning conditional ACs and generalized ACs, respectively.

6.1. Future directions

Despite the recent developments in algorithms for learning tractable representation, especially arithmetic circuits and sum-product networks, there exist many potential problems to be addressed. Here we describe some of the more relevant future directions regarding the focus of this dissertation.

6.1.1. Max-margin structure learning

In many real-world problems, such as gene annotation or multi-label image and document classification, we care about the most probable state (MAP state) of a set of interacting labels. The models for answering such queries can be learned from data using different formulations such as conditional random fields or max-margin training. Max-margin training is a powerful formulation that has achieved many state-of-the-art results in recent years. In general, max-margin training includes minimizing a regularized hinge loss objective function, which depends on running MAP inference over a loss-augmented dependence structure of labels and evidence variables. However, MAP inference becomes intractable for high treewidth structures, so finding an appropriate dependence structure that offers tractable MAP inference is important. To the best of our knowledge, only a few works address learning the dependence structure for max-margin training, mainly for low treewidth structures. SPNs and ACs are able to learn high treewidth models, in which exact MAP inference is efficient. Therefore, SPNs and ACs are reasonable candidates to represent the loss-augmented dependence structure over variables. In general, to find a near-optimal model, we need to generate and score candidate structures, which are very expensive operations in a max-margin setting. Therefore,

we could focus on exploring the structures that are beneficial for optimizing the regularized hinge loss objective function.

6.1.2. Approximate sum-product networks

Many high treewidth graphical models do not have a tractable representation in SPNs. To represent such intractable models, we need to relax consistency and completeness conditions of SPNs, which results in networks that are no longer valid representations of probability distributions. For example, in a consistent SPN, the children of every product node must have disjoint scopes, defined as the set of variables appearing in the univariate distributions of their descendants. If we relax this constraint and let the children of a product node have some common variables, then it is possible that the parents of the common variables do not agree on their beliefs over the common variables. In this situation, we need to incorporate belief propagation into the SPN regular inference procedure, so we can ensure that the parents agree on their beliefs over the common variables. This relaxation is interesting since it allows SPNs to represent higher treewidth models with more compact structures. However, we need to measure the effectiveness of this relaxation and address its error bounds. We also need to propose an algorithm for finding near-optimal relaxed SPN structures, which balances the approximation error and expressiveness of the models.

6.1.3. Learning conditional sum-product networks

Compared to generative models, discriminative models make fewer independence assumption about the random variables, so they can represent conditional distributions more compactly. However, discriminative structure

learning is more complex than generative structure learning. Gens and Domingos (2013) show that discriminative parameter learning of SPNs with a predefined structure achieves notable results on real-world image datasets, but in general, these predefined structures are not always available. Therefore, we need structure learning algorithms to capture a near-optimal SPN structure. To learn a conditional SPN, we can either adopt LearnSPN and substitute logistic regression for leaf distributions, or similar to ID-SPN, learn a sum-product of conditional ACs. However, in either case, we have fewer samples for learning nodes of the network as we increase its depth. As a result, discriminatively learned leaf distributions may not fit data well when the number of samples is small. Therefore, we need to incorporate a hybrid approach of generative and discriminative learning to address this problem. Moreover, for conditional SPNs, learning sum and product nodes is more challenging. Each sum node represent a weighted distribution of its children, and in a conditional setting, these weights depend on evidence variables and must be learned generatively or discriminatively. To learn product nodes, we need practical heuristics to find independent groups of variables, which is more difficult in a conditional setting. For example, using mutual information to determine independence in a generative setting is very efficient, but computing conditional mutual information is very expensive.

REFERENCES CITED

Amer, M. and Todorovic, S. (2012). Sum-product networks for modeling activities with stochastic structure. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1314–1321.

Amos, B., Xu, L., and Kolter, J. Z. (2016). Input convex neural networks. *arXiv preprint arXiv:1609.07152*.

Bach, F. R. and Jordan, M. I. (2001). Thin junction trees. *Advances in Neural Information Processing Systems*, 14:569–576.

Belanger, D. and McCallum, A. (2016). Structured prediction energy networks. In *Proceedings of the International Conference on Machine Learning*.

Bergstra, J. and Bengio, Y. (2013). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305.

Boutilier, C., Friedman, N., Goldszmidt, M., and Koller, D. (1996). Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 115–123, Portland, OR. Morgan Kaufmann.

Bradley, J. K. and Guestrin, C. (2010). Learning tree conditional random fields. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 127–134.

Chavira, M. and Darwiche, A. (2005). Compiling Bayesian networks with local structure. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1306–1312.

Chavira, M. and Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799.

Chechetka, A. and Guestrin, C. (2008). Efficient principled learning of thin junction trees. In Platt, J., Koller, D., Singer, Y., and Roweis, S., editors, *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA.

Chen, L.-C., Schwing, A. G., Yuille, A. L., and Urtasun, R. (2015). Learning deep structured models. In *ICML*, pages 1785–1794.

Cheng, W.-C., Kok, S., Pham, H. V., Chieu, H. L., and Chai, K. M. A. (2014). Language modeling with sum-product networks. In *Annual Conference of the International Speech Communication Association 15 (INTERSPEECH'14)*.

Chickering, D. M. (2002). The WinMine toolkit. Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA.

Choi, A., Kisa, D., and Darwiche, A. (2013). Compiling probabilistic graphical models using sentential decision diagrams. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 121–132. Springer.

Choi, M. J., Tan, V., Anandkumar, A., and Willsky, A. (2011). Learning latent tree graphical models. *Journal of Machine Learning Research*, 12:1771–1812.

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

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Darwiche, A. (2003). A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305.

Darwiche, A. (2011). Sdd: A new canonical representation of propositional knowledge bases. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 819.

Davis, J. and Domingos, P. (2010). Bottom-up learning of Markov network structure. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning*, Haifa, Israel. ACM Press.

Dechter, R. and Mateescu, R. (2007). AND/OR search spaces for graphical models. *Artificial Intelligence*, 171:73–106.

Della Pietra, S., Della Pietra, V., and Lafferty, J. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:380–392.

Dennis, A. and Ventura, D. (2012). Learning the architecture of sum-product networks using clustering on variables. In *Advances in Neural Information Processing Systems 25*.

Elidan, G. and Gould, S. (2008). Learning bounded treewidth Bayesian networks. *Journal of Machine Learning Research*, 9(2699-2731):122.

Elisseeff, A. and Weston, J. (2001). A kernel method for multi-labelled classification. In *Advances in Neural Information Processing Systems*, volume 14, pages 681–687.

Gens, R. and Domingos, P. (2012). Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems*, pages 3248–3256.

Gens, R. and Domingos, P. (2013). Learning the structure of sum-product networks. In *Proceedings of the Thirtieth International Conference on Machine Learning*.

Globerson, A. and Jaakkola, T. (2007). Approximate inference using conditional entropy decompositions. In *International Conference on Artificial Intelligence and Statistics*, pages 130–138.

Gogate, V., Webb, W., and Domingos, P. (2010). Learning efficient Markov networks. In *Proceedings of the 24th conference on Neural Information Processing Systems (NIPS'10)*.

Goldberg, K., Roeder, T., Gupta, D., and Perkins, C. (2001). Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151.

Greig, D., Porteous, B., and Seheult, A. H. (1989). Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 271–279.

Haaren, J. V. and Davis, J. (2012). Markov network structure learning: A randomized feature generation approach. In *Proceedings of the Twenty-Sixth National Conference on Artificial Intelligence*. AAAI Press.

Heskes, T., Albers, K., and Kappen, B. (2002). Approximate inference and constrained optimization. In *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*, pages 313–320. Morgan Kaufmann Publishers Inc.

Hong, C., Batal, I., and Hauskrecht, M. (2014). A mixtures-of-trees framework for multi-label classification. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 211–220. ACM.

Ishikawa, H. (2003). Exact optimization for markov random fields with convex priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(10):1333–1336.

Johnson, M., Duvenaud, D. K., Wiltschko, A., Adams, R. P., and Datta, S. R. (2016). Composing graphical models with neural networks for structured representations and fast inference. In *Advances in Neural Information Processing Systems*, pages 2946–2954.

Kisa, D., Van den Broeck, G., Choi, A., and Darwiche, A. (2014). Probabilistic sentential decision diagrams. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA.

Kolmogorov, V. and Zabin, R. (2004). What energy functions can be minimized via graph cuts? *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(2):147–159.

Korhonen, J. and Parviainen, P. (2013). Exact learning of bounded tree-width bayesian networks. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics (AISTATS'13)*, pages 370–378.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, Williamstown, MA. Morgan Kaufmann.

Lee, C.-Y., Gallagher, P. W., and Tu, Z. (2016). Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. In *International conference on artificial intelligence and statistics (AISTATS'16)*.

Li, C., Wang, B., Pavlu, V., and Aslam, J. (2016). Conditional bernoulli mixtures for multi-label classification. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 2482–2491.

Lowd, D. and Davis, J. (2010). Learning Markov network structure with decision trees. In *Proceedings of the 10th IEEE International Conference on Data Mining (ICDM)*, Sydney, Australia. IEEE Computer Society Press.

Lowd, D. and Domingos, P. (2005). Naive Bayes models for probability estimation. In *Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 529–536, Bonn, Germany.

Lowd, D. and Domingos, P. (2008). Learning arithmetic circuits. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, Helsinki, Finland. AUAI Press.

Lowd, D. and Rooshenas, A. (2013). Learning Markov networks with arithmetic circuits. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2013)*, Scottsdale, AZ.

Lowd, D. and Rooshenas, A. (2015). The libra toolkit for probabilistic models. *Journal of Machine Learning Research*, 16:2459–2463.

McCallum, A. (2003). Efficiently inducing features of conditional random fields. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, Acapulco, Mexico. Morgan Kaufmann.

Meila, M. and Jordan, M. I. (2000). Learning with mixtures of trees. *Journal of Machine Learning Research*, 1:1–48.

Meshi, O., Eban, E., Elidan, G., and Globerson, A. (2013). Learning max-margin tree predictors. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence (UAI2013)*. AUAI Press.

Mnih, V., Larochelle, H., and Hinton, G. E. (2012). Conditional restricted boltzmann machines for structured output prediction. *arXiv preprint arXiv:1202.3748*.

Murphy, K., Weiss, Y., and Jordan, M. I. (1999). Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, Stockholm, Sweden.

Niepert, M. and Domingos, P. (2014). Exchangeable variable models. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*.

Niepert, M. and Van den Broeck, G. (2014). Tractability through exchangeability: A new perspective on efficient probabilistic inference. *arXiv preprint arXiv:1401.1247*.

Peharz, R., Geiger, B. C., and Pernkopf, F. (2013). Greedy part-wise learning of sum-product networks. In *Machine Learning and Knowledge Discovery in Databases*, volume 8189 of *Lecture Notes in Computer Science*, pages 612–627. Springer Berlin Heidelberg.

Peharz, R., Kapeller, G., Mowlaee, P., and Pernkopf, F. (2014). Modeling speech with sum-product networks: Application to bandwidth extension. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 3699–3703.

Peng, J., Bo, L., and Xu, J. (2009). Conditional neural fields. In Bengio, Y., Schuurmans, D., Lafferty, J. D., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22*, pages 1419–1427. Curran Associates, Inc.

Pipatsrisawat, K. and Darwiche, A. (2008). New compilation languages based on structured decomposability. In *AAAI*, volume 8, pages 517–522.

Poon, H. and Domingos, P. (2011). Sum-product networks: A new deep architecture. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI-11)*, Barcelona, Spain. AUAI Press.

Rahman, T., Kothalkar, P., and Gogate, V. (2014a). Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Machine Learning and Knowledge Discovery in Databases*, pages 630–645. Springer.

Rahman, T., Kothalkar, P., and Gogate, V. (2014b). Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Machine Learning and Knowledge Discovery in Databases*, volume 8725 of *Lecture Notes in Computer Science*, pages 630–645. Springer Berlin Heidelberg.

Rooshenas, A. and Lowd, D. (2013). Learning tractable graphical models using mixture of arithmetic circuits. In *AAAI (Late-Breaking Developments)*.

Rooshenas, A. and Lowd, D. (2014). Learning sum-product networks with direct and indirect variable interactions. In *Proceedings of The 31st International Conference on Machine Learning*, pages 710–718.

Rooshenas, A. and Lowd, D. (2016). Discriminative structure learning of arithmetic circuits. In *Proceedings of the Nineteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2016)*, Cadiz, Spain.

Shahaf, D., Chechetka, A., and Guestrin, C. (2009). Learning thin junction trees via graph cuts. In *International Conference on Artificial Intelligence and Statistics*, pages 113–120.

Sohn, K., Lee, H., and Yan, X. (2015). Learning structured output representation using deep conditional generative models. In *Advances in Neural Information Processing Systems*, pages 3483–3491.

Sontag, D., Globerson, A., and Jaakkola, T. (2011). Introduction to dual decomposition for inference. *Optimization for Machine Learning*, 1:219–254.

Taskar, B., Chatalbashev, V., and Koller, D. (2004). Learning associative Markov networks. In *Proceedings of the twenty-first international conference on machine learning*. ACM Press.

Veksler, O. (2007). Graph cut based optimization for mrfs with truncated convex priors. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE.

Wainwright, M. J. and Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2):1–305.

Zheng, S., Jayasumana, S., Romera-Paredes, B., Vineet, V., Su, Z., Du, D., Huang, C., and Torr, P. H. (2015). Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1529–1537.