University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Daniel Lowd

and have found that it is complete and satisfactory in all respects, and that any and all revisions required by the final examining committee have been made.

Chair of the Supervisory Committee:

Pedro Domingos

Reading Committee:

Pedro Domingos

Jeff Bilmes

Marina Meila

Mark Handcock

Date:

University of Washington

Abstract

Efficient Learning and Inference in Rich Statistical Representations

Daniel Lowd

Chair of the Supervisory Committee:
Professor Pedro Domingos
Computer Science and Engineering

Rich statistical representations such as Markov logic networks are essential for solving hard problems in artificial intelligence and machine learning. However, the increased complexity of learning and inference often limits their effectiveness in practice. In this dissertation, we make several contributions towards richer representations and the algorithms to support them. We introduce recursive Markov logic, a "deep" generalization of Markov logic that introduces uncertainty into every level of a first-order knowledge base. We also develop improved weight learning algorithms for Markov logic, leading to more accurate models in less time. Finally, we use arithmetic circuits to address the problem of inference in graphical models in two ways. First, we present an algorithm that learns Bayesian networks with fast inference by using inference complexity as a learning bias. Second, we show how to use arithmetic circuits in an extremely flexible form of variational inference.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Chapter 1

INTRODUCTION

Achieving human-level artificial intelligence will require a powerful representation with efficient learning and inference algorithms. Such a representation would also have a huge impact on current applications in natural language processing, bioinformatics, robotics, and more. More powerful representations can represent more powerful concepts, leading to better models of complex domains. However, learning these models from data or reasoning about them is much more difficult. For example:

- Backpropagation neural networks are more powerful than perceptrons, but training them is considerably more difficult.

- Loopy Bayesian networks are more powerful than tree-structured networks, but inference is #P-complete instead of polynomial.

- First-order logic is much more powerful than propositional logic, but is only semidecidable.

These greater challenges must be addressed by efficient algorithms that exploit the structure of the problem whenever possible. Even if the worst-case is hard, the common case may be tractable in practice. Furthermore, if an optimal or exact solution is intractable, an efficient approximation may be good enough. Humans solve intractable problems every day without thinking about it. Therefore, richer representations and better algorithms go hand in hand: algorithmic improvements make richer representations more effective in more domains.

One of the most powerful representation languages to date is Markov logic, recently introduced by Richardson and Domingos [39]. Markov logic generalizes both first-order logic and probabilistic graphical models by attaching a weight to each formula in a first-order knowledge base and using them as features in a log-linear model. Though conceptually simple, Markov logic makes it easy to write down state-of-the-art solutions for many challenging AI problems. Successful applications including bioinformatics [40, 29], robot mapping [50], natural language processing [41, 38], and more. The standard weight learning algorithm is the voted perceptron algorithm [45]; structure learning is done using top-down [27] or bottom-up [**?**] search strategies; and inference is done using the MaxWalkSAT [**?**] or MC-SAT [37] algorithm.

A Markov logic network (MLN) can be viewed as a probabilistic "softening" of a deterministic knowledge base: worlds that violate one or more formulas are less probable, not impossible. The formulas themselves remains purely logical; each one is either true or false. If a formula is a conjunction or universally quantified, we can choose to soften it by splitting it into multiple formulas, making worlds less likely when they violate more "pieces" of the formula. However, disjunctions, existential quantifiers, and nested formulas remain deterministic. In this dissertation, we resolve this inconsistency by introducing recursive Markov logic networks, which we also refer to as recursive random fields (RRFs). A RRF uses weights to softens *every* connective or quantifier in every formula, rather than just the top-level conjunctions and universal quantifiers. RRFs retain the full power of MLNs while representing many important distributions, such as m-of-n concepts, exponentially more compactly. As an added bonus, given a sufficiently expressive network structure, RRF weight learning is more powerful than MLN structure learning.

While experts can often create a good knowledge base by hand, choosing good MLN weights is very difficult. When training data is available, learning weights automatically is almost always the better choice. In this dissertation, we develop and explore several different weight learning approaches that learn better weights or are orders of magnitude faster than the previous state-of-the-art. This makes many applications of Markov logic both easier and more effective.

The biggest hurdle for applying Markov logic to even more problems is the complexity of inference. Much of this difficulty is inherited from the probabilistic graphical models MLNs are based on. Therefore, we make several contributions to efficient inference in graphical models in general. In this dissertation, we show how to learn graphical models with efficient inference by using inference cost as a learning bias. We measure inference cost using arithmetic circuits, a representation in which inference takes linear time. (Of course, the arithmetic circuit required to represent a given graphical model might be exponentially larger.) Since our models allow for exact inference instead of approximate, our methods are more accurate as well as faster than traditional approaches.

Finally, we present a variational inference method based on arithmetic circuits. The key idea is to find a small arithmetic circuit that approximates a given graphical model as well as possible. Once we have found such a circuit, we can answer any query efficiently. The answers are often much more accurate than those of mean field or sampling-based methods.

The outline of this dissertation is as follows. In Chapter 2, we provide background on probabilistic graphical models, first-order logic and Markov logic. In Chapter 3, we describe recursive Markov logic and its richer representational capabilities over Markov logic. In Chapter 4, we explore a variety of different methods for Markov logic weight learning and demonstrate orders of magnitude improvement over the previous state-of-the-art method. In Chapter 5, we discuss methods for learning models with efficient inference by using inference complexity as a learning bias. In Chapter 6, we show the ideas in Chapter 5 can be extended into a general purpose variational inference algorithm with significantly more flexibility than classic mean field and structured mean field methods. Finally, we conclude in Chapter 7 by reviewing the contributions of this dissertation and discussing important areas of future work.

Chapter 2

BACKGROUND

In this chapter, we present background on the statistical representations that will be discussed in this dissertation. We begin by describing probabilistic graphical models, including Bayesian networks and Markov networks. We then describe Markov logic and how it extends these models using first-order logic.

2.1 Probabilistic Graphical Models

2.1.1 Bayesian Networks

A *Bayesian network* encodes the joint probability distribution of a set of n variables, $\{X_1, \ldots, X_n\}$, as a directed acyclic graph and a set of conditional probability distributions (CPDs) [35]. Each node corresponds to a variable, and the CPD associated with it gives the probability of each state of the variable given every possible combination of states of its parents. The set of parents of X_i, denoted Π_i, is the set of nodes with an arc to X_i in the graph. The structure of the network encodes the assertion that each node is conditionally independent of its non-descendants given its parents. The joint distribution of the variables is thus given by $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|\Pi_i)$.

For discrete domains, the simplest form of CPD is a conditional probability table. When the structure of the network is known, learning reduces to estimating CPD parameters. When the structure is unknown, it can be learned by starting with an empty or prior network and greedily adding, deleting and reversing arcs to optimize some score function [20]. The score function is usually log-likelihood plus a complexity penalty or a Bayesian score (product of prior and marginal likelihood).

The goal of inference in Bayesian networks is to answer arbitrary marginal and conditional queries (i.e., to compute the marginal distribution of a set of query variables, possibly conditioned on the values of a set of evidence variables). One common method is to construct a *junction tree* from the Bayesian network and pass messages from the leaves of this tree to the root and back. A junction tree is constructed by connecting parents of the same variable, removing arrows, and triangulating the resulting undirected graph (i.e., ensuring that all cycles of length four or more have a chord). Each node in the junction tree corresponds to a *clique* (maximal completely connected subset of variables) in the triangulated graph. Ordering cliques by the highest-ranked variable they contain, each clique is connected to a predecessor sharing the highest number of variables with it. The intersection of the variables in two adjacent cliques is called the *separator* of the two cliques. A junction tree satisfies two important properties: each variable in the Bayesian network appears in some clique with all of its parents; and if a variable appears in two cliques, it appears in all the cliques on the path between them (the *running intersection property*). The *treewidth* of a junction tree is one less than the maximum clique size. The complexity of inference is exponential in the treewidth. Finding the minimum-treewidth junction tree is NP-hard [3]. Inference in Bayesian networks is #P-complete [42].

Because exact inference is intractable, approximate methods are often used, of which the most popular is *Gibbs sampling*, a form of Markov chain Monte Carlo [19]. A Gibbs sampler proceeds by sampling each non-evidence variable in turn conditioned on its Markov blanket (parents, children and parents of children). The distribution of the query variables is then approximated by computing, for each possible state of the variables, the fraction of samples in which it occurs. Gibbs sampling can be very slow to converge, and many MCMC variations have been developed, but choosing and tuning one for a given application remains a difficult, labor-intensive task. Diagnosing convergence is also difficult.

2.1.2 Markov Networks

Like a Bayesian network, a *Markov network* is a factorized representation of a probability distribution over a set of variables. The key difference is that, in a Markov network, the factors are not constrained to be conditional probability distributions. The general form of the probability distribution is as follows:

$$P(X = x) = \frac{1}{Z} \prod_{j=1} k\pi_j(x_{\{j\}}) \tag{2.1}$$

where π_j represents the jth *potential* function, which assigns a non-negative number to each configuration of its arguments, denoted by $x_{\{j\}}$. Z is the *partition function*, a constant that ensures that the probabilities of all states sum to one:

$$Z = \sum_x \prod_{j=1} k\pi_j(x_{\{j\}}) \tag{2.2}$$

For Bayesian networks, the validity of the probability distribution is ensured by the form of the potential functions (i.e., conditional probability distributions), so this constant is only needed for Markov networks. Markov networks are sometimes represented by an exponentiated sum of weighted features rather than as a product of potentials:

$$P(X = x) = \frac{1}{Z} \exp\left(\sum_i w_i f_i(x)\right) \tag{2.3}$$

If the Markov network represents a positive distribution (that is, every x has a non-zero probability), then it can always be written in this second form. Depending on the structure of the potential functions, the feature-based representation may be much more compact.

Most Bayesian network inference methods are easily adapted to Markov networks. Learning methods, however, are somewhat different. While Bayesian network CPDs can be easily estimated in closed form using the counts in the data, but optimizing the weights or potentials in a Markov network requires an iterative procedure such as gradient descent. As in Bayesian networks, structure learning can be done by greedy search, but the search space is typically over features rather than the parents of each variable [15, 31].

2.2 Markov Logic

A Markov logic network (MLN) consists of a set of formulas in first-order logic and their weights, $\{(w_i, f_i)\}$. Intuitively, a formula represents a noisy relational rule, and its weight represents the relative strength or importance of that rule. Given a finite set of constants, we can instantiate an MLN as a Markov network in which each node is a grounding of a predicate (atom) and each feature is a grounding of one of the formulas (clauses). This leads to the following joint probability distribution for all atoms:

$$P(\mathrm{X} = \mathrm{x}) = \frac{1}{Z} \exp\left(\sum_i w_i n_i(x)\right)$$

where n_i is the number of times the ith formula is satisfied by the state of the world x and Z is a normalization constant, required to make the probabilities of all worlds to sum to one.

The formulas in an MLN are typically specified by an expert, or they can be obtained (or refined) by inductive logic programming or MLN structure learning [27]. Many complex models, and in particular many non-i.i.d. ones, can be very compactly specified using MLNs.

Exact inference in MLNs is intractable. Instead, we can perform approximate inference using Markov chain Monte Carlo (MCMC), and in particular Gibbs sampling [19]. However, when weights are large convergence can be very slow, and when they are infinite (corresponding to deterministic dependencies) ergodicity breaks down. This remains true even for more sophisticated alternatives like simulated tempering. A much more efficient alternative, which also preserves ergodicity in the presence of determinism, is the MC-SAT algorithm, recently introduced by Poon and Domingos [37]. MC-SAT is a "slice sampling" MCMC algorithm that uses a modified satisfiability solver to sample from the slice. The solver is able to find isolated modes in the distribution very efficiently, and as a result the Markov chain mixes very rapidly. The slice sampling scheme ensures that detailed balance is (approximately) preserved.

For a thorough introduction to Markov logic, its algorithms, and its applications, see Domingos and Lowd [16].

Chapter 3
RECURSIVE MARKOV LOGIC

3.1 Introduction

A Markov logic network can be viewed as a probabilistic "softening" of a deterministic knowledge base. A formula in first-order logic can be viewed as a tree, with a logical connective at each node, and a knowledge base can be viewed as a tree whose root is a conjunction. Markov logic makes this conjunction probabilistic, as well as the universal quantifiers directly under it, but the rest of the tree remains purely logical. This causes an asymmetry in the treatment of conjunctions and disjunctions, and of universal and existential quantifiers.

For example, an MLN with the formula $R(X) \wedge S(X)$ can treat worlds that violate both $R(X)$ and $S(X)$ as worse than worlds that only violate one. Since an MLN acts as a soft conjunction, the groundings of $R(X)$ and $S(X)$ simply appear as distinct formulas. (MLNs often convert the knowledge base to CNF before performing learning or inference.) This is not possible for the disjunction $R(X) \vee S(X)$: no distinction is made between satisfying both $R(X)$ and $S(X)$ and satisfying just one. Since a universally quantified variable is effectively a conjunction over all groundings, while an existentially quantified variable is a disjunction over all groundings, this leads to the two quantifiers being handled differently.

In this chapter, we overcome this limitation with a new representation, called recursive Markov logic or recursive random fields (RRFs). RRFs use a "deep" multi-level representation to soften disjunction and existential quantification in the same way that Markov logic softens conjunction and universal quantification. RRFs have many desirable properties, including the ability to represent distributions like noisy DNF, rules with exceptions, and m-of-all quantifiers much more compactly than MLNs. RRFs also allow more flexibilty in revising first-order theories to maximize data likelihood. Standard methods for inference in Markov random fields are easily extended to RRFs, and weight learning can be carried out efficiently using a variant of the backpropagation algorithm.

RRF theory revision can be viewed as a first-order probabilistic analog of the KBANN algorithm, which initializes a neural network with a propositional theory and uses back-propagation to improve its fit to data [48]. A propositional RRF (where all predicates have zero arity) differs from a multilayer perceptron in that its output is the joint probability of its inputs, not the regression of a variable on others (or, in the probabilistic version, its conditional probability). Propositional RRFs are an alternative to Boltzmann machines, with nested features playing the role of hidden variables. Because the nested features are deterministic functions of the inputs, learning does not require EM, and inference does not require marginalizing out variables.

3.2 Recursive Random Fields

A recursive random field (RRF) is a log-linear model in which each feature is either an observable variable or the output of another recursive random field. To build up intuition, we first describe the propositional case, then generalize it to the more interesting relational case. A concrete example is given in Section 3.2.3, and illustrated in Figure 3.1.

3.2.1 Propositional RRFs

While our primary goal is solving relational problems, RRFs may be interesting in propositional domains as well. Propositional RRFs extend Markov random fields and Boltzmann machines in the same way multilayer perceptrons extend single-layer ones. The extension is very simple in principle, but allows RRFs to compactly represent important concepts, such as m-of-n. It also allows RRFs to learn features via weight learning, which could be more effective than current feature-search methods for Markov random fields.

The probability distribution of a propositional RRF is as follows:

$$P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z_0} \exp\left(\sum_i w_i f_i(\mathbf{x})\right)$$

where Z_0 is a normalization constant, to ensure that the probabilities of all possible states $\mathbf{x}$ sum to 1. What makes this different from a standard Markov random field is that the features can be built up from other subfeatures to an arbitrary number of levels. Specifically, each feature is either:

$$\begin{aligned} f_i(\mathbf{x}) &= x_j \quad \text{(base case), or} \\ f_i(\mathbf{x}) &= \frac{1}{Z_i} \exp\left(\sum_j w_{ij} f_j(\mathbf{x}) \right) \quad \text{(recursive case)} \end{aligned}$$

In the recursive case, the summation is over all features f_j referenced by the "parent" feature f_i. A child feature, f_j, can appear in more than one parent feature, and thus an RRF can be viewed as a directed acyclic graph of features. The attribute values are at the leaves, and the probability of their configuration is given by the root. (Note that the probabilistic graphical model it represents is still undirected.)

Since the overall distribution is simply a recursive feature, we can also write the probability distribution as follows:

$$P(\mathbf{X} = \mathbf{x}) = f_0(\mathbf{x})$$

Except for Z_0 (the normalization of the root feature, f_0), the per-feature normalization constants Z_i can be absorbed into the corresponding feature weights w_{ki} in their parent features f_k. Therefore, the user is free to choose any convenient normalization or even no normalization at all.

It is easy to show that this generalizes Markov random fields with conjunctive or disjunctive features. Each f_i approximates a conjunction when weights w_{ij} are very large. In the limit, f_i will be 1 iff the conjunct is true. f_i can also represent disjuncts using large negative weights, along with a negative weight for a "parent" feature f_k, w_{ki}. The negative weight w_{ki} turns the conjunction into a disjunction just as negation does in De Morgan's laws. However, one can also move beyond conjunction and disjunction to represent m-of-n concepts, or even more complex distributions where the feature values are given non-uniform weights.

Note that features with small absolute weights have little effect. Therefore, instead of using heuristics or search to determine which attributes should appear in which feature, we can include *all* predicates and let weight learning sort out which attributes are relevant for which feature. This is similar to learning a neural network by initializing it with small, random values. Since the network can represent any logical formula, there is no need to commit to a specific structure ahead of time. This is an attractive alternative to the traditional inductive methods used for learning MRF features.

An RRF can be seen as a type of multi-layer neural network, in which the node function is exponential (rather than sigmoidal) and the network is trained to maximize joint likelihood. Unlike in neural networks, the random variables are all inputs, not inputs or outputs. The output is the likelihood of the random variables' joint configuration. In other ways, an RRF resembles a Boltzmann machine, but with the greater flexibility of multiple layers and learnable using a variant of the back-propagation algorithm. RRFs have no hidden variables to sum out, since all nodes in the network have deterministic values.

3.2.2 Relational RRFs

In the relational case, relations over an arbitrary number of objects take the place of a fixed number of variables. To allow parameter tying across different groundings, we use parameterized features, or *parfeatures*. We represent the parameter tuple as a vector, $\vec{g}$, whose size depends on the arity of the parfeature. (Note that $\vec{g}$ is a vector of logical variables—i.e., arguments to predicates—as opposed to the random Boolean variables $\mathbf{x}$—ground atoms—that represent a state of the world.) We use subscripts to distinguish among parfeatures with different parameterizations, e.g. $f_{i,\vec{g}}(\vec{x})$ and $f_{i,\vec{g}'}(\vec{x})$ represent different groundings of the ith parfeature.

Each RRF parfeature is defined in one of two ways:

$$\begin{aligned} f_{i,\vec{g}}(\mathbf{x}) &= R_i(g_{i_1}, \ldots, g_{i_k}) \quad \text{(base case)} \\ f_{i,\vec{g}}(\mathbf{x}) &= \frac{1}{Z_i} \exp\left(\sum_j w_{ij} \sum_{\vec{g}'} f_{j,\vec{g},\vec{g}'}(\mathbf{x}) \right) \quad \text{(recursive case)} \end{aligned}$$

The base case is straightforward: it simply represents the value of a ground relation (as specified by $\mathbf{x}$). The grounding of the relation depends on the parameters of the parfeature. The recursive case sums the weighted values of all child parfeatures. Each parameter g_i of a child parfeature is either a parameter of the parent feature ($g_i \in \vec{g}$) or a parameter of a child feature that is summed out and does not appear in the parent feature ($g_i \in \vec{g'}$). (These $\vec{g'}$ parameters are analogous to the parameters that appear in the body but not the head of a Horn clause.) Just as sums of child features act as conjunctions, the summations over $\vec{g'}$ parameters act as universal quantifiers with Markov logic semantics. In fact, these generalized quantifiers can represent m-of-all concepts, just as the simple feature sums can represent m-of-n concepts.

The relational version of a recursive random field is therefore defined as follows:

$$P(\mathbf{X} = \mathbf{x}) = f_0(\mathbf{x})$$

where X is the set of all ground relations (e.g., $R(A, B)$, $S(A)$), $\mathbf{x}$ is an assignment of ground relations to truth values, f_0 is the root recursive parfeature (which, being the root, has no parameters). Since f_0 is a recursive parfeature, it is normalized by the constant Z_0 to ensure a valid probability distribution. (As in the propositional case, all other Z_i's can be absorbed into the weights of their parent features, and may therefore be normalized in any convenient way.)

Any relational RRF can be converted into a propositional RRF by grounding all parfeatures and expanding all summations. Each distinct grounding of a parfeature becomes a distinct feature, but with shared weights.

3.2.3 RRF Example

To clarify these ideas, let us take the example knowledge base from Richardson and Domingos [2006]. The domain consists of three predicates: Smokes(g) (g is a smoker); Cancer(g) (g has cancer); and Friends(g, h) (g is a friend of h). We abbreviate these predicates as Sm(g), Ca(g), and Fr(g, h), respectively.

We wish to represent three beliefs: (i) smoking causes cancer; (ii) friends of friends are friends (transitivity of friendship); and (iii) everyone has at least one friend who smokes. (The most interesting belief from Richardson and Domingos [2006], that people smoke if and only if their friends do, is omitted here for simplicity.) We demonstrate how to represent these beliefs by first converting them to first-order logic, and then converting to an RRF.

One can represent the first belief, “smoking causes cancer,” in first-order logic as a universally quantified implication: $\forall g : \text{Sm}(g) \Rightarrow \text{Ca}(g)$. This implication can be rewritten as a disjunction: $\neg\text{Sm}(g) \vee \text{Ca}(g)$. From De Morgan’s laws, this is equivalent to: $\neg(\text{Sm}(g) \wedge \neg\text{Ca}(g))$, which can be represented as an RRF feature:

$$f_{1,g}(\mathbf{x}) = \frac{1}{Z_1} \exp(w_{1,1}\text{Sm}(g) + w_{1,2}\text{Ca}(g))$$

where $w_{1,1}$ is positive, $w_{1,2}$ is negative, and the feature weight $w_{0,1}$ is negative (not shown above). In general, since RRF features can model conjunction and disjunction, any CNF knowledge base can be represented as an RRF. A similar approach works for the second belief, “friends of people are friends.”

The first two beliefs are also handled well by Markov logic networks. The key advantage of recursive random fields is in representing more complex formulas. The third belief, “everyone has at least one friend who smokes,” is naturally represented by nested quantifiers: $\forall g : \exists h : \text{Fr}(g, h) \wedge \text{Sm}(h)$. This is best represented as an RRF feature that references a secondary feature:

$$f_{3,g}(\mathbf{x}) = \frac{1}{Z_3} \exp\left(\sum_h w_{3,1} f_{4,g,h}(\mathbf{x})\right)$$

$$f_{4,g,h}(\mathbf{x}) = \frac{1}{Z_4} \exp(w_{4,1}\text{Fr}(g, h) + w_{4,2}\text{Sm}(h))$$

Note that in RRFs this feature can also represent a distribution over the number of smoking friends each person has, depending on the assigned weights. It’s possible that, while almost everyone has at least one smoking friend, many people have at least two or three. With an RRF, we can actually learn this distribution from data.

This third belief is very problematic for an MLN. First of all, in an MLN it is purely logical: there’s no change in probability with the number of smoking friends once that number exceeds one. Secondly, MLNs do not represent the belief efficiently. In an MLN, the existential quantifier is converted to a very large disjunction:

$$(\text{Fr}(g, A) \wedge \text{Sm}(A)) \vee (\text{Fr}(g, B) \wedge \text{Sm}(B)) \vee \cdots$$

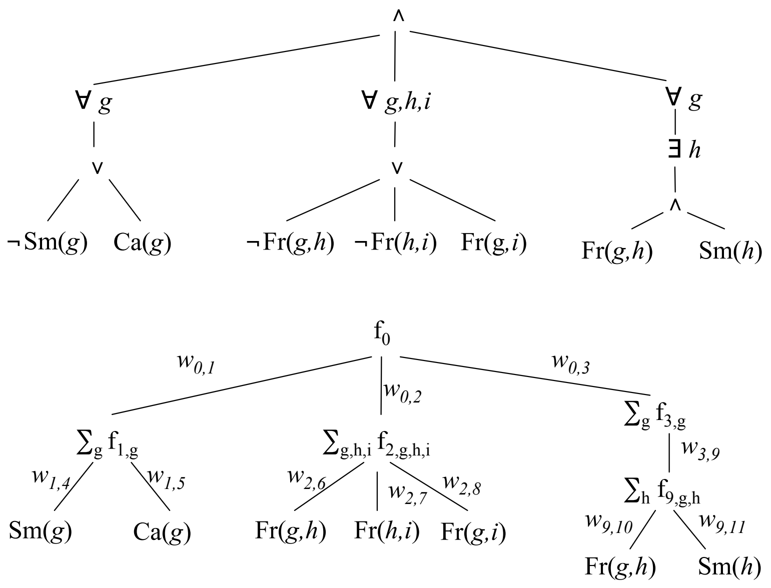

Figure 3.1: Comparison of first-order logic (above) and RRF structures (below). The RRF structure closely mirrors that of first-order logic, but connectives and quantifiers are replaced by weighted sums.

If there are 1000 objects in the database, then this disjunction is over 1000 conjunctions. Further, the MLN will convert this DNF into CNF form, leading to 2^{1000} CNF clauses from each grounding of this rule.

These features define a full joint distribution as follows:

$$P(\mathbf{X}=\mathbf{x}) = \frac{1}{Z_0}\exp\ \Big(\sum_g w_{0,1} f_{1,g}(\mathbf{x}) + \\ \sum_{g,h,i} w_{0,2} f_{2,g,h,i}(\mathbf{x}) \\ \sum_g w_{0,3} f_{3,g}(\mathbf{x}))$$

Figure 3.1 diagrams the first-order knowledge base containing all of these beliefs, along with the corresponding RRF.

3.3 Inference

Since RRFs generalize MLNs, which in turn generalize finite first-order logic and Markov random fields, exact inference is intractable. Instead, we use MCMC inference, in particular Gibbs sampling. This is straightforward: we sample each unknown ground predicate in turn, conditioned on all other ground predicates. The probability of a particular ground predicate may easily be computed by evaluating the relative likelihoods when the predicate is true and when it is false.

We speed this up significantly by caching feature sums. When a predicate is updated, it notifies its parents of the change so that only the necessary values are recomputed.

Our current implementation of MAP inference uses iterated conditional modes (ICM) [5], a simple method for finding a mode of a distribution. Starting from a random configuration, ICM sets each variable in turn to its most likely value, conditioned on all other variables. This procedure continues until no single-variable change will further improve the likelihood. ICM is easy to implement, fast to run, and guaranteed to converge. Unfortunately, it has no guarantee of converging to the most likely overall configuration. Possible improvements include random restarts, simulated annealing, or other ways of adding noise.

We also use ICM to find an initial state for the Gibbs sampler. By starting at a mode, we significantly reduce the burn-in time and achieve better predictions sooner.

3.4 Learning

Given a particular RRF structure and initial set of weights, we learn weights using a novel variant of the back-propagation algorithm. As in traditional back-propagation, the goal is to efficiently compute the derivative of the loss function with respect to each weight in the model. In this case, the loss function is not the error in predicting the output variables, but rather the joint log-likelihood of all variables. We must also consider the partition function for the root feature, Z_0. For these computations, we extract the $1/Z_0$ term from f_0, and use f_0 refer to the unnormalized feature value.

We begin by discussing the simpler, propositional case. We abbreviate $f_i(\mathbf{x})$ as f_i for these arguments. The derivative of the log-likelihood with respect to a weight w_{ij} consists of two terms:

$$\frac{\partial \log P(\mathbf{x})}{\partial w_{ij}} = \frac{\partial \log(1/Z_0 f_0)}{\partial w_{ij}} = \frac{\partial \log(f_0)}{\partial w_{ij}} - \frac{\partial \log(Z_0)}{\partial w_{ij}}$$

The first term can be evaluated with the chain rule:

$$\frac{\partial \log(f_0)}{\partial w_{ij}} = \frac{\partial \log(f_0)}{\partial f_i} \frac{\partial f_i}{\partial w_{ij}}$$

From the definition of f_i (including the normalization Z_i):

$$\frac{\partial f_i}{\partial w_{ij}} = f_i \left(f_j - \frac{1}{Z_i} \frac{\partial Z_i}{\partial w_{ij}} \right)$$

From repeated applications of the chain rule, the $\partial \log(f_0)/\partial f_i$ term is the sum of all derivatives along all paths through the network from f_0 to f_i. Given a path in the feature graph $\{f_0, f_a, \ldots, f_k, f_i\}$, the derivative along that path takes the form $f_0 w_a f_a w_b f_b \cdots w_k f_k w_i$. We can efficiently compute the sum of all paths by caching the per-feature partials, $\partial f_0/\partial f_i$, analogous to back-propagation.

The second term, $\partial \log(Z_0)/\partial w_{ij}$, is the expected value of the first term, evaluated over all possible inputs $\mathbf{x}'$. Therefore, the complete partial derivative is:

$$\frac{\partial \log P(\mathbf{x})}{\partial w_{ij}} = \frac{\partial \log(f_0(\mathbf{x}))}{\partial w_{ij}} - E_{\mathbf{x}'} \left[\frac{\partial \log(f_0(\mathbf{x}'))}{\partial w_{ij}} \right]$$

where the individual components are evaluated as above.

Computing the expectation is typically intractable, but it can be approximated using Gibbs sampling. A rough approximation of only a few iterations usually suffices to determine the rough direction of the gradient, a method Hinton [2002] refers to as "contrastive divergence."

A more efficient alternative, used by Richardson and Domingos [2006], is to instead optimize the pseudo-log-likelihood (P^*):

$$P^*(X{=}x) = \sum_{t=1}^{n} \log P(X_t = x_t)$$

Since it is a consistent estimator, pseudo-log-likelihood will converge to the same parameters as log-likelihood given infinite data. Pseudo-log-likelihood can perform poorly when long chains of inference are required, but worked quite well in our test domain.

The expression for the gradient of the pseudo-log-likelihood of a propositional RRF is as follows:

$$\frac{\partial}{\partial w_i} P^*(X{=}x) = \sum_{t=1}^{n} P(X_t = \neg x_t) \left(\frac{\partial f_0}{\partial w_i} - \frac{\partial f_{0[X_t=\neg x_t]}}{\partial w_i} \right)$$

We can compute this by iterating over all query predicates, toggling each one in turn, and getting the relative likelihood and unnormalized likelihood gradient for that permuted state. Note that we compute the gradient of the unnormalized log-likelihood as a subroutine in computing the gradient of the pseudo-log-likelihood. However, we no longer need to approximate the intractable normalization term, Z.

To learn a relational RRF, we use the domain to instantiate a propositional RRF with tied weights. The number of features as well as the number of children per feature will depend on the number of objects in the domain. Instead of a weight being attached to a single feature, it is now attached to a set of groundings of a parfeature. The partial derivative with respect to a weight is therefore the sum of the partial derivatives with respect to each instantiation of the shared weight.

3.5 RRFs vs. MLNs

Both RRFs and MLNs subsume probabilistic models and first-order logic in finite domains. Both can be trained generatively or discriminatively using gradient descent, either to optimize log-likelihood or pseudo-likelihood. For both, when optimizing log-likelihood, the normalization constant Z_0 is approximated using the most-likely explanation or MCMC.

Any MLN can be converted into a relational RRF by translating each clause into an equivalent parfeature. With sufficiently large weights, a parfeature approximates a hard conjunction or disjunction over its children. However, when its weights are sufficiently distinct, a parfeature can take on a different value for each configuration of its children. This allows RRFs to compactly represent distributions that would require an exponential number of clauses in an MLN.

Any RRF can be converted to an MLN by flattening the model, but this will typically require an exponential number of clauses. Such an MLN would be intractable for learning or inference. RRFs are therefore much better at modeling soft disjunction, existential quantification, and nested formulas.

In addition to being "softer" than an MLN clause, an RRF parfeature can represent many different MLN clauses simply by adjusting its weights. This makes RRF weight learning more powerful than MLN structure learning: an RRF with $n + 1$ recursive parfeatures (one for the root) can represent any MLN structure with up to n clauses, as well as many distributions that an n-clause MLN cannot represent.

This leads to new alternatives for structure learning and theory revision. In a domain where little background knowledge is available, an RRF could be initialized with small random weights and still converge to a good statistical model. This is potentially much better than MLN structure learning, which constructs clauses one predicate at a time, and must adjust weights to evaluate every candidate clause.

When background knowledge is available, we can begin by initializing the RRF to the background theory, just as in MLNs. However, in addition to the known dependencies, we can also add dependencies on other parfeatures or predicates with very small weights. Weight learning can learn large weights for relevant dependencies and negligible weights for irrelevant dependencies. This is analagous to what the KBANN system does using neural networks [48]. In contrast, MLNs can only do theory revision through discrete search.

3.6 Experiments: Probabilistic Integrity Constraints

Integrity constraints are statements in first-order logic that are used to detect and repair database errors [1]. Logical statements work well when errors are few and critical, but are increasingly impractical with noisy databases such as those that arise from the integration of multiple databases, information extraction from the web, etc. We want to make these constraints *probabilistic*, so we can statistically infer the types of errors and their sources. To date, there has been very little work on this problem. (See [2, 23] for two early approaches.) This is a natural domain for MLNs and RRFs, since both use first-order formulas to construct probability distributions over worlds, or databases.

The two most common types of integrity constraints are inclusion constraints and func-

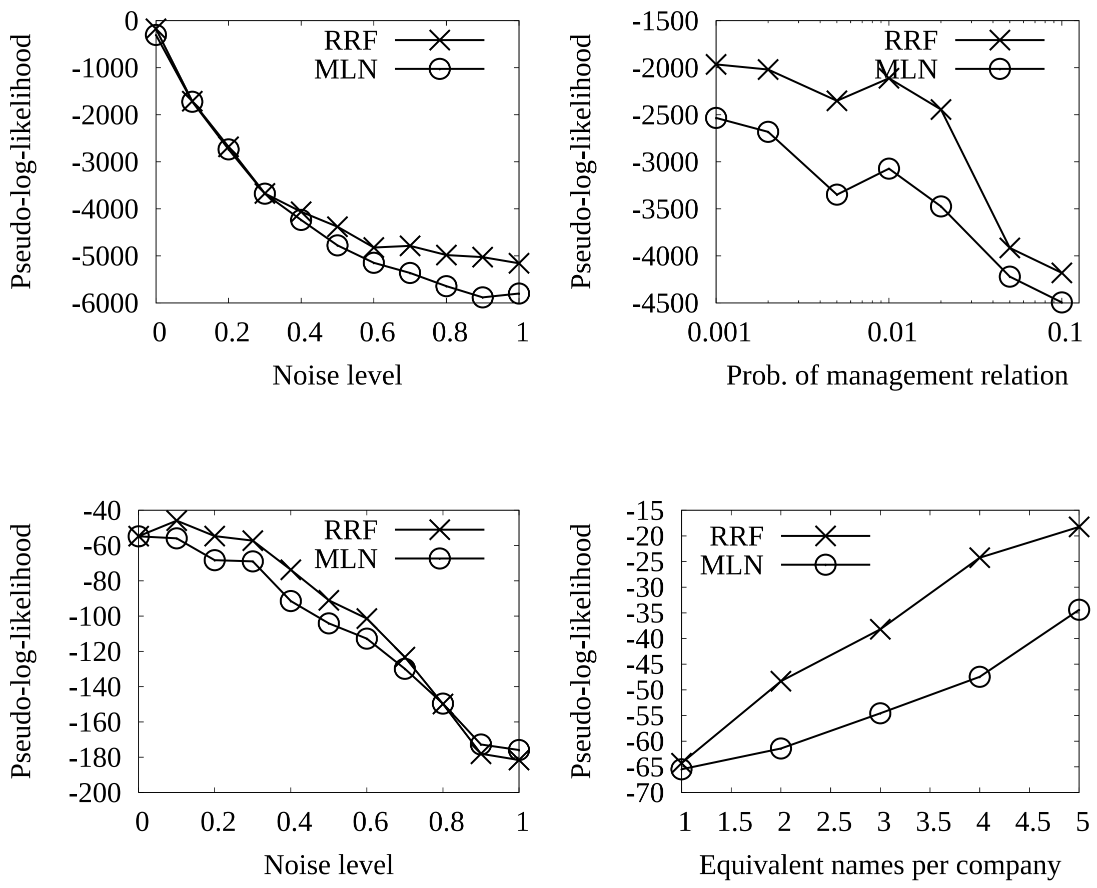

Figure 3.2: Pseudo-log-likelihood of RRFs and MLNs for the inclusion constraints and functional dependencies. For inclusion constraints, we vary the probability of corrupting each observed relation, $ProjectLead'(x, y)$ and $Manages'(x, z)$ (upper-left) and the probability of adding extra $Manages(x, z)$ tuples (upper-right). For functional dependencies, we vary the probability of corrupting the company name of a supplier (lower-left) and the number of equivalent names per company (lower-right).

tional dependencies. Inclusion constraints are of the form:

$$\forall x.(\exists y.R(x, y)) \Rightarrow (\exists z.S(x, z))$$

For example, in Company X, R could be the relation "ProjectLead" (x is in charge of project y) and S could be the relation "ManagerOf" (x manages employee z). This constraint says that every project leader manages at least one other worker. Of course, some employees could manage other employees without being the lead on any project.

To evaluate MLNs and RRFs on inclusion constraints, we generated domains consisting of 100 people and 100 projects. With probability 0.25, a person x is a project leader, and leads (or co-leads) each project y with probability 0.1. For each project x leads, x manages employee z with probability 0.05. Additional managing relationships are generated with a probability that we vary from 0.001 to 0.1. However, the actual leadership and management relationships are unobserved: at test time, we only see noisy versions, which are corrupted with a probability that we vary from 0.0 to 1.0.

We converted the constraint formula into an MLN and an RRF as described in previous sections and added the implications $ProjectLead(x, y) \Rightarrow ProjectLead'(x, y)$ and $ManagerOf(x, z) \Rightarrow ManagerOf'(x, z)$. We found that both MLNs and RRFs worked better when given both directions of the constraint, since project leaders manage people and managers lead projects. We learned weights to optimize pseudo-log-likelihood in all models. The results are shown in Figure 3.2. Each data point is an average over 10 train/test set pairs. RRFs show a consistent advantage over MLNs, because they can better represent the fact that an employee who manages many people is probably a project leader, while an employee who manages few people may not be.

The second type of integrity constraints, functional dependencies, are of the form:

$$\forall x, y_1, y_2.(\exists z_1, z_2.R(x, y_1, z_2) \wedge R(x, y_2, z_2)) \Rightarrow y_1 = y_2$$

In a functional dependency, each x determines a unique y (or an equivalence set of ys). For example, suppose Company X has a table of parts suppliers represented by the relation

Supplier(TaxID, CompanyName, PartType). A supplier that supplies multiple types of parts may appear multiple times, but each tax ID should always be associated with the same company name or set of equivalent company names (e.g., “Microsoft,” “Microsoft, Inc.,” and “MSFT” are all equivalent). If there is noise in the database, logic alone cannot say which company names are equivalent and which are errors.

For evaluating functional dependencies, we generated a database with 30 tax IDs, 30 company names (partitioned into equivalence sets of size k), and 30 part types. Each tax ID is associated with one set of equivalent company names, uniformly selected from the $30/k$ name groups. For each company name a tax ID uses, we generated part types z that the company supplies with probability 0.25. This simulates the merging of k distinct databases, each with its own company naming scheme but with shared identifiers for the other fields. As a final step, we randomly corrupted company names with a probability that we varied from 0.0 to 1.0.

The task was to predict which pairs of company names were equivalent, given the supplier table. The results are shown in Figure 3.2. With multiple databases at moderate levels of noise, RRFs outperform MLNs. We attribute this to RRFs’ ability to represent the fact that two company names are more likely to be equivalent if they both appear multiple times with the same tax ID, and to learn the appropriate form of this dependence.

3.7 Conclusion

Recursive random fields overcome some salient limitations of Markov logic. While MLNs only model uncertainty over conjunctions and universal quantifiers, RRFs also model uncertainty over disjunctions and existentials, and thus achieve a deeper integration of logic and probability. Inference in RRFs can be carried out using Gibbs sampling and iterated conditional modes, and weights can be learned using a variant of the back-propagation algorithm.

The main disadvantage of RRFs relative to MLNs is reduced understandability. One possibility is to extract MLNs from RRFs with techniques similar to those used to extract propositional theories from KBANN models. Another important problem for future work is scalability. Here we plan to adapt many of the MLN optimizations to RRFs.

Most importantly, we intend to apply RRFs to real datasets to better understand how they work in practice, and to see if their greater representational power yields better models.

Chapter 4

EFFICIENT WEIGHT LEARNING IN MARKOV LOGIC NETWORKS

4.1 Introduction

The previous state-of-the-art method for learning MLN weights is Singla and Domingos' voted perceptron algorithm [45], based on Collins' earlier one [11] for hidden Markov models. Voted perceptron uses gradient descent to approximately optimize the conditional likelihood of the query atoms given the evidence.

Weight learning in Markov logic is a convex optimization problem, and thus gradient descent is guaranteed to find the global optimum. However, convergence to this optimum may be extremely slow. MLNs are exponential models, and their sufficient statistics are the numbers of times each clause is true in the data. Because this number can easily vary by orders of magnitude from one clause to another, a learning rate that is small enough to avoid divergence in some weights is too small for fast convergence in others. This is an instance of the well-known problem of ill-conditioning in numerical optimization, and many candidate solutions for it exist [34]. However, the most common ones are not easily applicable to MLNs because of the nature of the function being optimized. As in Markov random fields, computing the likelihood in MLNs requires computing the partition function, which is generally intractable. This makes it difficult to apply methods that require performing line searches, which involve computing the function as well as its gradient. These include most conjugate gradient and quasi-Newton methods (e.g., L-BFGS). Two exceptions to this are scaled conjugate gradient [33] and Newton's method with a diagonalized Hessian [4]. In this chapter we show how they can be applied to MLN learning, and verify empirically that they greatly speed up convergence. We also obtain good results with a simpler method: per-weight learning rates, with a weight's learning rate being the global one divided by the corresponding clause's empirical number of true groundings.

Voted perceptron approximates the expected sufficient statistics in the gradient by computing them at the MPE state (i.e., the most likely state of the non-evidence atoms given the evidence ones, or most probable explanation). Since in an MLN the conditional distribution can contain many modes, this may not be a good approximation. Also, using second-order methods requires computing the Hessian (matrix of second-order partial derivatives), and for this the MPE approximation is no longer sufficient. We address both of these problems by instead computing expected counts using MC-SAT, a very fast Markov chain Monte Carlo (MCMC) algorithm for Markov logic [37].

4.2 Weight Learning for MLNs

Given a set of formulas and a database of atoms, we wish to find the formulas' maximum *a posteriori* (MAP) weights, i.e., the weights that maximize the product of their prior probability and the data likelihood. Since optimization is typically posed as function minimization, we will equivalently minimize the negative log-likelihood.

Richardson and Domingos [39] originally proposed learning weights generatively using pseudo-likelihood [5]. Pseudo-likelihood is the product of the conditional likelihood of each variable given the values of its neighbors in the data. While efficient for learning, it can give poor results when long chains of inference are required at query time. Singla and Domingos [45] showed that pseudo-likelihood is consistently outperformed by discriminative training, which minimizes the negative conditional likelihood of the query predicates given the evidence ones. Thus, in this chapter we focus on this type of learning.[1]

4.2.1 Voted Perceptron

Gradient descent algorithms use the gradient, $\mathbf{g}$, scaled by a learning rate, η, to update the weight vector $\mathbf{w}$ in each step:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{g}$$

In an MLN, the derivative of the negative conditional log-likelihood (CLL) with respect to a weight is the difference of the expected number of true groundings of the corresponding clause and the actual number according to the data:

$$\frac{\partial}{\partial w_i} - \log P(Y{=}y|X{=}x) = E_w[n_i] - n_i$$

where y is the state of the non-evidence atoms in the data, and x is the state of the evidence.

[1]For simplicity, we omit prior terms throughout; in our experiments, we use a zero-mean Gaussian prior on all weights with all algorithms.

The basic idea of the voted perceptron (VP) algorithm [11] is to approximate the intractable expectations $E_w[n_i]$ with the counts in the most probable explanation (MPE) state, which is the most probable state of non-evidence atoms given the evidence. To combat overfitting, instead of returning the final weights, VP returns the average of the weights from all iterations of gradient descent.

Collins originally proposed VP for training hidden Markov models discriminatively, and in this case the MPE state is unique and can be computed exactly in polynomial time using the Viterbi algorithm. In MLNs, MPE inference is intractable but can be reduced to solving a weighted maximum satisfiability problem, for which efficient algorithms exist such as MaxWalkSAT [25]. Singla and Domingos [45] use this approach and discuss how the resulting algorithm can be viewed as approximately optimizing log-likelihood. However, the use of voted perceptron in MLNs is potentially complicated by the fact that the MPE state may no longer be unique, and MaxWalkSAT is not guaranteed to find it.

4.2.2 Contrastive Divergence

The contrastive divergence (CD) algorithm is identical to VP, except that it approximates the expectations $E_w[n_i]$ from a small number of MCMC samples instead of using the MPE state. Using MCMC is presumably more accurate and stable, since it converges to the true expectations in the limit. While running an MCMC algorithm to convergence at each iteration of gradient descent is infeasibly slow, Hinton [21] has shown that a few iterations of MCMC yield enough information to choose a good direction for gradient descent. Hinton named this method *contrastive divergence*, because it can be interpreted as optimizing a difference of Kullback-Leibler divergences. Contrastive divergence can also be seen as an efficient way to approximately optimize log-likelihood.

The MCMC algorithm typically used with contrastive divergence is Gibbs sampling, but for MLNs the much faster alternative of MC-SAT is available. Because successive samples in MC-SAT are much less correlated than successive sweeps in Gibbs sampling, they carry more information and are likely to yield a better descent direction. In particular, the different samples are likely to be from different modes, reducing the error and potential instability associated with choosing a single mode.

In our experiments, we found that five samples were sufficient, and additional samples were not worth the time: any increased accuracy that 10 or 100 samples might bring was offset by the increased time per iteration. We avoid the need for burn-in by starting at the last state sampled in the previous iteration of gradient descent. (This differs from Hinton's approach, which always starts at the true values in the training data.)

4.2.3 Per-Weight Learning Rates

VP and CD are both simple gradient descent procedures, and as a result highly vulnerable to the problem of ill-conditioning. Ill-conditioning occurs when the *condition number*, the ratio between the largest and smallest absolute eigenvalues of the Hessian, is far from one. On ill-conditioned problems, gradient descent is very slow, because no single learning rate is appropriate for all weights. In MLNs, the Hessian is the negative covariance matrix of the clause counts. Because some clauses can have vastly greater numbers of true groundings than others, the variances of their counts can be correspondingly larger, and ill-conditioning becomes a serious issue.

One solution is to modify both algorithms to have a different learning rate for each weight. Since tuning every learning rate separately is impractical, we use a simple heuristic to assign a learning rate to each weight:

$$\eta_i = \frac{\eta}{n_i}$$

where η is the user-specified global learning rate and n_i is the number of true groundings of the ith formula. (To avoid dividing by zero, if $n_i = 0$ then $\eta_i = \eta$.) When computing this number, we ignore the groundings that are satisfied by the evidence (e.g., $A \Rightarrow B$ when A is false). This is because, being fixed, they cannot contribute to the variance.

We refer to the modified versions of VP and CD as VP-PW and CD-PW.

4.2.4 Diagonal Newton

When the function being optimized is quadratic, Newton's method can move to the global minimum or maximum in a single step. It does so by multiplying the gradient, $\mathbf{g}$, by the inverse Hessian, $\mathbf{H}^{-1}$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{H}^{-1}\mathbf{g}$$

When there are many weights, using the full Hessian becomes infeasible. A common approximation is to use the *diagonal* Newton (DN) method, which uses the inverse of the diagonalized Hessian in place of the inverse Hessian. DN typically uses a smaller step size than the full Newton method. This is important when applying the algorithm to non-quadratic functions, such as MLN negative CLL, where the quadratic approximation is only good within a local region.

The Hessian of the negative CLL for an MLN is simply the covariance matrix:

$$\frac{\partial}{\partial w_i \partial w_j} - \log P(Y=y|X=x) = E_w[n_i n_j] - E_w[n_i]E_w[n_j]$$

Like the gradient, this can be estimated using samples from MC-SAT. In each iteration, we take a step in the diagonalized Newton direction:

$$w_i = w_i - \alpha \frac{E_w[n_i] - n_i}{E_w[n_i^2] - (E_w[n_i])^2}$$

The step size α could be computed in a number of ways, including keeping it fixed, but we achieved the best results using the following method. Given a search direction $\mathbf{d}$ and Hessian matrix $\mathbf{H}$, we compute the step size as follows:

$$\alpha = \frac{-\mathbf{d}^T\mathbf{g}}{\mathbf{d}^T\mathbf{H}\mathbf{d} + \lambda\mathbf{d}^T\mathbf{d}}$$

where $\mathbf{d}$ is the search direction. For a quadratic function and $\lambda = 0$, this step size would move to the minimum function value along $\mathbf{d}$. Since our function is not quadratic, a non-zero λ term serves to limit the size of the step to a region in which our quadratic approximation is good. After each step, we adjust λ to increase or decrease the size of the so-called *model* *trust region* based on how well the approximation matched the function. Let Δ_{actual} be the actual change in the function value, and let Δ_{pred} be the predicted change in the function value from the previous gradient and Hessian and our last step, $\mathbf{d}_{t-1}$:

$$\Delta_{pred} = \mathbf{d}_{t-1}^T \mathbf{g}_{t-1} + 1/2\, \mathbf{d}_{t-1}^T \mathbf{H}_{t-1} \mathbf{d}_{t-1}$$

A standard method for adjusting λ is as follows [17]:

$$\begin{array}{lll} \text{if } (\Delta_{actual}/\Delta_{pred} > 0.75) & \text{then} & \lambda_{t+1} = \lambda_t/2 \\ \text{if } (\Delta_{actual}/\Delta_{pred} < 0.25) & \text{then} & \lambda_{t+1} = 4\lambda_t \end{array}$$

Since we cannot efficiently compute the actual change in negative CLL, we approximate it as the product of the step we just took and the gradient after taking it: $\Delta_{actual} = \mathbf{d}_{t-1}^T \mathbf{g}_t$. Since the negative CLL is a convex function, this product is an upper bound on the actual change. When this value is positive our CLL may be worse than before, so the step is rejected and redone after adjusting λ.

In models with thousands of weights or more, storing the entire Hessian matrix becomes impractical. However, when the Hessian appears only inside a quadratic form, as above, the value of this form can be computed simply as:

$$\mathbf{d}^T \mathbf{H} \mathbf{d} = E_w[(\textstyle\sum_i d_i n_i)^2] - (E_w[\textstyle\sum_i d_i n_i])^2$$

The product of the Hessian by a vector can also be computed compactly [36]. Note that α is computed using the full Hessian matrix, but the step direction is computed from the diagonalized approximation which is easier to invert.

Our per-weight learning rates can actually be seen as a crude approximation of the diagonal Newton method. The number of true groundings not satisfied by evidence is a heuristic approximation to the count variance, which the diagonal Newton method uses to rescale each dimension of the gradient. The diagonal Newton method, however, can adapt to changes in the second derivative at different points in the weight space. Its main limitation is that clauses can be far from uncorrelated. The next method addresses this issue.

4.2.5 Scaled Conjugate Gradient

Gradient descent can be sped up by, instead of taking a small step of constant size at each iteration, performing a line search to find the optimum along the chosen descent direction. However, on ill-conditioned problems this is still inefficient, because line searches along successive directions tend to partly undo the effect of each other: each line search makes the gradient along its direction zero, but the next line search will generally make it non-zero again. In long narrow valleys, instead of moving quickly to the optimum, gradient descent zigzags.

A solution to this is to impose at each step the condition that the gradient along previous directions remain zero. The directions chosen in this way are called *conjugate*, and the method *conjugate gradient* [44]. We used the Polak-Ribiere method for choosing conjugate gradients since it has generally been found to be the best-performing one. Conjugate gradient methods are some of the most efficient available, on a par with quasi-Newton ones. Unfortunately, applying them to MLNs is difficult, because line searches require computing the objective function, and therefore the partition function Z, which is highly intractable. (Computing Z is equivalent to computing all moments of the MLN, of which the gradient and Hessian are the first two.)

Fortunately, we can use the Hessian instead of a line search to choose a step size. This method is known as *scaled conjugate gradient* (SCG), and was originally proposed by Møller [33] for training neural networks. In our implementation, we choose a step size the same way as in diagonal Newton.

Conjugate gradient is usually more effective with a preconditioner, a linear transformation that attempts to reduce the condition number of the problem (e.g., [43]). Good preconditioners approximate the inverse Hessian. We use the inverse diagonal Hessian as our preconditioner. We refer to SCG with the preconditioner as PSCG.

4.3 Experiments

4.3.1 Datasets

Our experiments used two standard relational datasets representing two important relational tasks: Cora for entity resolution, and WebKB for collective classification.

The Cora dataset consists of 1295 citations of 132 different computer science papers, drawn from the Cora Computer Science Research Paper Engine. This dataset was originally labeled by Andrew McCallum[2]. We used a cleaned version from Singla and Domingos [46], with five splits for cross-validation.

The task on Cora is to predict which citations refer to the same paper, given the words in their author, title, and venue fields. The labeled data also specifies which pairs of author, title, and venue fields refer to the same entities. In our experiments, we evaluated the ability of the model to deduplicate fields as well as citations. Since the number of possible equivalances is very large, we used the canopies found by Singla and Domingos [46] to make this problem tractable.

The MLN we used for this is very similar to the "MLN(B+C+T)" model used by Singla and Domingos [46]. Its formulas link words to citation identity, words to field identity, and field identity to citation identity. In this way, word co-occurrence affects the probability that two citations are the same both indirectly, through field similarities, and directly. These rules are repeated for each word appearing in the database so that individualized weights can be learned, representing the relative importance of each word in each context. This model also features transitive closure for all equivalence predicates.

We did two things differently from Singla and Domingos. First, we added rules that relate words to field identity but apply equally to all words. Because these rules are not specific to particular words, they can potentially improve generalization and reduce overfitting. Secondly, we learned weights for all rules. Singla and Domingos set the weights for all word-specific rules using a naive Bayes model, and only learned the other rules' weights using VP. Our learning problem is therefore much harder and more ill-conditioned, but our

[2]http://www.cs.umass.edu/∼mccallum/data/cora-refs.tar.gz

more powerful algorithms enabled us to achieve the best results to date on Cora.

In our version, the total number of weights is 6141. During learning, the number of ground clauses exceeded 3 million.

The WebKB dataset consists of labeled web pages from the computer science departments of four universities. We used the relational version of the dataset from Craven and Slattery [12], which features 4165 web pages and 10,935 web links, along with the words on the webpages, anchors of the links, and neighborhoods around each link.

Each web page is marked with some subset of the categories: person, student, faculty, professor, department, research project, and course. Our goal is to predict these categories from the web pages' words and link structures.

We used a very simple MLN for this model, consisting only of formulas linking words to page classes, and page classes to the classes of linked pages. The "word-class" rules were of the following form:

$$\begin{aligned} \texttt{Has(page,word)} &\Rightarrow \texttt{Class(page,class)} \\ \neg\texttt{Has(page,word)} &\Rightarrow \texttt{Class(page,class)} \end{aligned}$$

We learned a separate weight for each of these rules for each `(word, class)` pair. Classes of linked pages were related by the formula:

$$\texttt{Class(page1,class1)} \wedge \texttt{LinksTo(page1,page2)} \Rightarrow \texttt{Class(page2,class2)}$$

We learned a separate weight for this rule for each pair of classes. When instantiated for each word and class, the model contained 10,891 weights. While simple to write, this model represents a complex, non-i.i.d. probability distribution in which query predicates are linked in a large graph. During learning, the number of ground clauses exceeded 300,000.

We estimated the condition number for both Cora and WebKB at the point where all weights are zero. (Because our learning problem is not quadratic, the condition number depends on the current weights.) The size of these problems makes computing the condition number of the full Hessian matrix difficult, but we can easily compute the condition number of the diagonalized Hessian, which is simply the largest ratio of two clause variances. For Cora, this was over 600,000, while for WebKB it was approximately 7000. This indicates that both learning problems are ill-conditioned, but Cora is much worse than WebKB.

4.3.2 Metrics

To score our models, we ran MC-SAT for 100 burn-in and 1000 sampling iterations on the test data. The marginal conditional probability of each query atom is the fraction of samples in which the atom was true with a small prior to prevent zero counts.

From these marginal probabilities, we estimate conditional log-likelihood (CLL) by averaging the log marginal probabilities of the true values of the query predicates. CLL is the metric all of the algorithms attempt to optimize. However, in cases such as entity resolution where the class distribution is highly skewed, CLL can be a poor metric. For this reason, we also look at AUC, the area under the precision-recall curve. The disadvantage of AUC is that it ignores calibration: AUC only considers whether true atoms are given higher probability than false atoms.

4.3.3 Methodology

We ran our experiments using five-way cross-validation for Cora and four-way cross-validation for WebKB. For each train/test split, one of the training datasets was selected as a validation set and the remaining ones formed the tuning set. The tuning procedure consisted of training each algorithm for four hours on the tuning sets with various values of the learning rate. For each algorithm on each split, we chose the learning rates that worked best on the corresponding validation set for each evaluation metric.

We used the implementation of voted perceptron for MLNs in the Alchemy package [28], and implemented the other algorithms as extensions of Alchemy. For DN, SCG, and PSCG, we started with $\lambda = 1$ and let the algorithm adjust it automatically. For algorithms based on MC-SAT, we used 5 samples of MC-SAT for each iteration of the learning algorithm. The width of the Gaussian prior was set for each dataset based on preliminary experiments.

After tuning all algorithms, we reran them for 10 hours with their respective training sets,

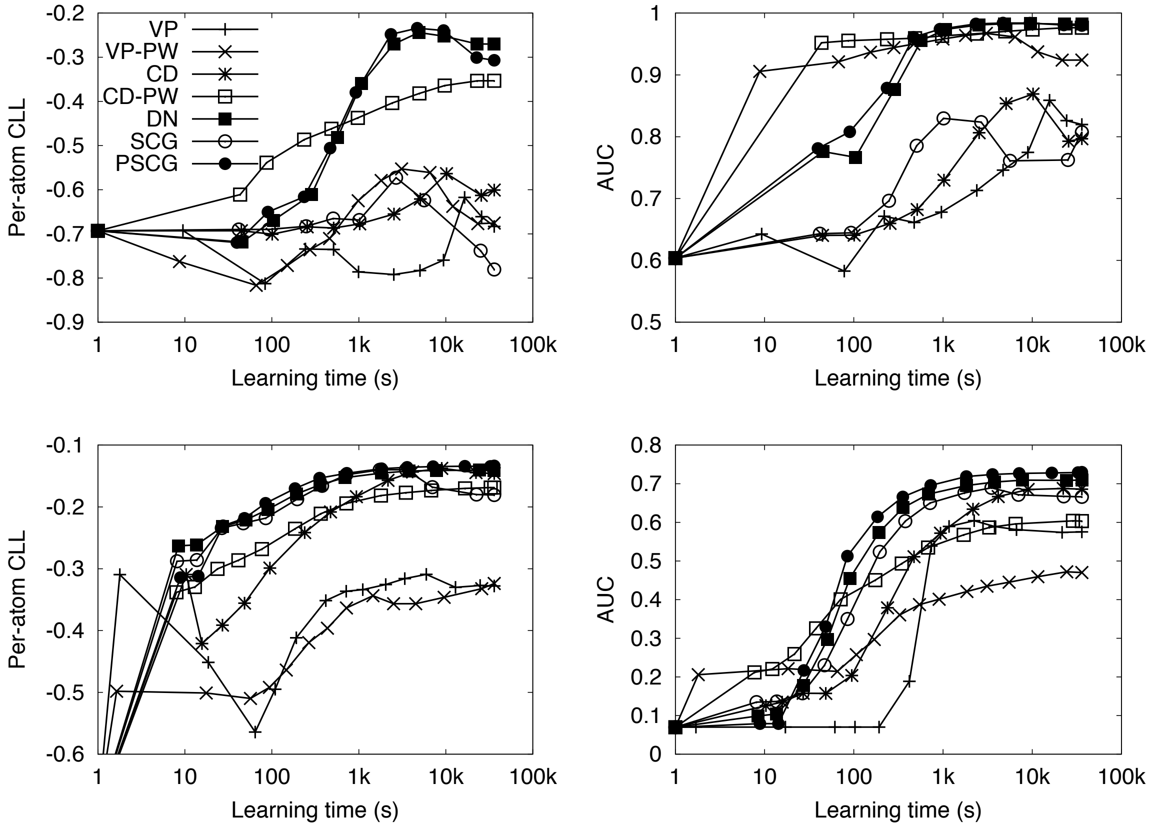

Figure 4.1: CLL and AUC for Cora (above) and WebKB (below). Learning times are shown on a logarithmic scale.

including the held-out validation data. For the gradient descent algorithms, we averaged the weights from all iterations.

4.3.4 Results

Our results for the Cora and WebKB datasets are shown in Figure 4.1. Error bars are omitted for clarity; at the final data point, all differences exceed twice the standard error. For AUC, we computed the standard deviation using the technique of Richardson and Domingos [39].

PSCG is the most accurate of all the algorithms compared, obtaining the best CLL and AUC on both Cora and WebKB. It converges relatively quickly as well: on WebKB, the PSCG learning curve dominates all others after 2 minutes; on Cora, it dominates after 15 minutes. DN is consistently close behind PSCG in CLL and AUC, briefly doing better when PSCG starts to overfit. In contrast, VP and CD consistently converge more slowly to worse AUC and CLL.

On Cora, the algorithms that adjust the search direction using true clause counts or count variance do much better than those that do not. This suggests that these techniques help greatly in cases of extreme ill-conditioning. Without a preconditioner, even SCG does poorly. This is because, like VP and CD, the first step it takes is in the direction of the gradient. On a very ill-conditioned dataset like Cora, the gradient is a very poor choice of search direction.

The AUC results we show for Cora are for all query predicates—`SameAuthor`, `SameVenue`, `SameTitle`, and `SameBib`. When computing the AUC for just the `SameBib` predicate, PSCG reaches a high of 0.992 but ends at 0.990 after overfitting slightly. DN and CD-PW do about the same, ending at AUCs of 0.992 and 0.991, respectively. All of these algorithms exceed the 0.988 AUC reported by Singla and Domingos [46], the best previously published result on this dataset, and they do so by more than twice the standard error.

On WebKB, the ill-conditioning is less of an issue. PSCG still does better than SCG, but not drastically better. VP-PW and CD-PW actually do worse than VP and CD. This is because the per-weight learning rates are much smaller for the relational rules than the word-specific rules. This makes the relational rules converge much more slowly than they should.

The performance of some of the algorithms sometimes degrades with additional learning time. For some of the algorithms, such as PSCG, DN, and VP-PW on Cora, this is simply a symptom of overfitting. More careful tuning or a better prior could help correct this. But for other algorithms, such as SCG and VP on Cora, the later models perform worse on training data as well. For SCG, this seems to be the result of noisy inference and very ill-conditioned problems, which cause even a slight error in the step direction to potentially have a significant effect. Our lower bound on the improvement in log-likelihood prevents this in theory, but in practice a noisy gradient may still cause us to take bad steps. PSCG suffers much less from this effect, since the preconditioning makes the learning problem better behaved. For VP and CD, the most likely cause is learning rates that are too high.

Our tuning experiments selected the learning rates that worked best after four hours on a smaller set of data. The increased amount of data in the test scenario increased the magnitude of the gradients, making these learning rates less stable than they were in the tuning scenario. This extreme sensitivity to learning rate makes learning good models with VP and CD much more difficult. We also experimented with the stochastic meta-descent algorithm [49], which automatically adjusts learning rates in each dimension, but found it to be too unstable for these domains.

In sum, the MLN weight learning methods we have introduced in this chapter greatly outperform the voted perceptron. Given similar learning time, they learn much more accurate models; and, judging from the curves in Figure 4.1, running VP until it reaches the same accuracy as the better algorithms would take an extremely long time.

4.4 Conclusion

Weight learning for Markov logic networks can be extremely ill-conditioned, making simple gradient descent-style algorithms very slow to converge. In this chapter we studied a number of more sophisticated alternatives, of which the best-performing one is preconditioned scaled conjugate gradient. This can be attributed to its effective use of second-order information. However, the simple heuristic of dividing the learning rate by the true clause counts for each weight can sometimes give very good results. Using one of these methods instead of gradient descent can yield a much better model in less time.

Chapter 5

LEARNING ARITHMETIC CIRCUITS

5.1 Abstract

Graphical models are usually learned without regard to the cost of doing inference with them. As a result, even if a good model is learned, it may perform poorly at prediction, because it requires approximate inference. We propose an alternative: learning models with a score function that directly penalizes the cost of inference. Specifically, we learn arithmetic circuits with a penalty on the number of edges in the circuit (in which the cost of inference is linear). Our algorithm is equivalent to learning a Bayesian network with context-specific independence by greedily splitting conditional distributions, at each step scoring the candidates by compiling the resulting network into an arithmetic circuit, and using its size as the penalty. We show how this can be done efficiently, without compiling a circuit from scratch for each candidate. Experiments on several real-world domains show that our algorithm is able to learn tractable models with very large treewidth, and yields more accurate predictions than a standard context-specific Bayesian network learner, in far less time.

5.2 Introduction

Probabilistic graphical models, such as Bayesian and Markov networks, are capable of compactly representing very complex dependences. Unfortunately, the compactness of the representation does not necessarily translate into efficient inference. Networks with relatively few edges per node can still require exponential inference time. As a consequence, approximate inference methods must often be used, but these can yield poor and unreliable results. If the network represents manually encoded expert knowledge, this is perhaps inevitable. But when the network is learned from data, the cost of inference can potentially be greatly reduced, without compromising accuracy, by suitably directing the learning process.

Bayesian networks can be learned using local search to maximize a likelihood or Bayesian score, with operators like edge addition, deletion and reversal [20]. Typically, the number of parameters or edges in the network is penalized to avoid overfitting, but this is only very indirectly related to the cost of inference. Two edge additions that produce the same improvement in likelihood can result in vastly difference inference costs. In this case, it seems reasonable to prefer the edge yielding the lowest inference cost. In this chapter, we propose a learning method that accomplishes this, by directly penalizing the cost of inference in the score function.

Our method takes advantage of recent advances in exact inference by compilation to arithmetic circuits [14]. An arithmetic circuit is a representation of a Bayesian network capable of answering arbitrary marginal and conditional queries, with the property that the cost of inference is linear in the size of the circuit. When context-specific independences are present, arithmetic circuits can be much more compact than the corresponding junction trees. We take advantage of this by learning arithmetic circuits that are equivalent to Bayesian networks with context-specific independence, using likelihood plus a penalty on the circuit size as the score function. Arithmetic circuits can also take advantage of other structural properties such as deterministic dependencies and latent variables; utilizing these in addition to context-specific independence is an important item of future work.

Previous work on learning graphical models with the explicit goal of limiting the complexity of inference falls into two main classes: mixture models with polynomial-time inference (e.g., [32, 30]) and graphical models with thin junction trees (e.g. [47, 8]). The former are limited in the range of distributions that they can compactly represent. The latter are computationally viable (at both learning and inference time) only for very low treewidths. Our approach can flexibly and compactly learn a wide variety of models, including models with very large treewidth, while guaranteeing efficient inference, by taking advantage of the properties of arithmetic circuits.

The prior work most closely related to ours is Jaeger *et al.*'s [24]. Jaeger *et al.* define probabilistic decision graphs, a new language related to binary decision diagrams. In contrast, we use standard arithmetic circuits, and our models are equivalent to standard Bayesian networks. Jaeger *et al.* speculate that learning arithmetic circuits directly from data would be very difficult. In this chapter we propose one approach to doing this.

5.3 Local Structure in Bayesian Networks

Table CPDs require exponential space in the number of parents of the variable. A more scalable approach is to use *decision trees* as CPDs, taking advantage of context-specific independencies (i.e., a child variable is independent of some of its parents given some values of the others) [7, 18, 9]. The algorithm we present in this chapter learns arithmetic circuits that are equivalent to this type of Bayesian network.

In a decision tree CPD for variable X_i, each interior node is labeled with one of the parent variables, and each of its outgoing edges is labeled with a value of that variable.[1] Each leaf node is a multinomial representing the marginal distribution of X_i conditioned on the parent variable values specified by its ancestor nodes and edges in the tree.

The following two definitions will be useful in describing our algorithm.

Definition 1. *For leaf node D and k-valued variable X_j, the* split *$S(D, X_j)$ replaces D with k new leaves, each conditioned on a particular value of X_j in addition to the parent values on the path to D.*

Definition 2. *Let D be a leaf from the tree CPD for X_i. Split $S(D, X_j)$ is* valid *iff X_j is not a descendant of X_i in the Bayesian network and no decision tree ancestor of D is labeled with X_j*

The first definition describes a structural update to the Bayesian network; the second one gives the conditions necessary for that update to be consistent and meaningful.

A Bayesian network can now be learned by greedily applying the best valid splits according to some criterion, such as the likelihood of the data penalized by the number of parameters. This is one version of Chickering et al.'s algorithm [9]. A number of other methods have also been proposed, such as merging leaves to obtain decision graphs [9] or searching through Bayesian network structures and inducing decision trees conditioned on the global structure [18].

[1]In general, each outgoing edge can be labeled with any subset of the variable's values, as long as the sets of labels assigned to all child edges include every variable value and are disjoint with each other. For simplicity, we limit our discussion to the case in which each edge has a single label, which Chickering et al. [9] refer to as a *complete split.* For Boolean variables, as in our experiments, all types of splits are equivalent.

5.4 Arithmetic Circuits

The probability distribution represented by a Bayesian network can be equivalently represented by a multilinear function known as the *network polynomial* [14]:

$$\begin{aligned} & P(X_1 = x_1, \ldots, X_n = x_n) \\ = & \sum_{\mathbf{X}} \prod_{i=1}^{n} I(X_i = x_i) P(X_i = x_i | \Pi_i = \pi_i) \end{aligned}$$

where the sum ranges over all possible instantiations of the variables, $I()$ is the indicator function (1 if the argument is true, 0 otherwise), and the $P(X_i|\Pi_i)$ are the parameters of the Bayesian network. The probability of any partial instantiation of the variables can now be computed simply by setting to 1 all indicators consistent with the instantiation, and to 0 all others. This allows arbitrary marginal and conditional queries to be answered in time linear in the size of the polynomial.

Unfortunately, the size of the network polynomial is exponential in the number of variables, but it can be more compactly represented using an *arithmetic circuit.* An arithmetic circuit is a rooted, directed acyclic graph whose leaves are numeric constants or variables, and whose interior nodes are addition and multiplication operations. The value of the function for an input tuple is computed by setting the variable leaves to the corresponding values and computing the value of each node from the values of its children, starting at the leaves. In the case of the network polynomial, the leaves are the indicators and network parameters. The arithmetic circuit avoids the redundancy present in the network polynomial, and can be exponentially more compact. Figure 5.1 shows a simple circuit that represents the simple probability distribution $P(A, B) = P(A)P(B)$

Every junction tree has a corresponding arithmetic circuit, with an addition node for every instantiation of a separator, a multiplication node for every instantiation of a clique, and an addition node as the root. Thus one way to compile a Bayesian network into an arithmetic circuit is via a junction tree. However, when the network contains context-specific independences, a much more compact circuit can be obtained. Darwiche [14] describes one way to do this, by encoding the network into a special logical form, factoring the logical

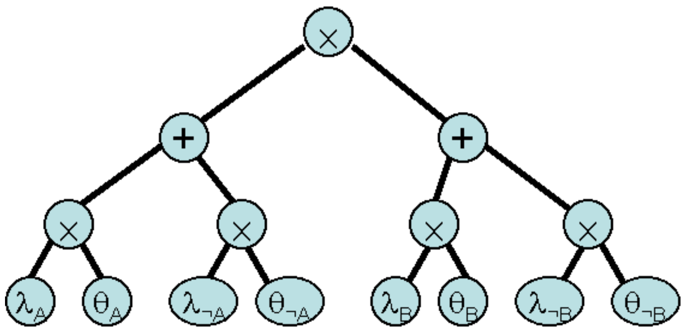

Figure 5.1: Arithmetic circuit to represent $P(A, B) = P(A)P(B)$, a product of marginals distribution over two variables. Indicator functions $I(X)$ and parameters $P(Y|X)$ are written as λ_X and $\theta_{Y|X}$, following Darwiche [14].

form, and extracting the corresponding arithmetic circuit.

5.5 *Learning Arithmetic Circuits*

5.5.1 Scoring and Searching

Instead of learning a Bayesian network and then compiling it into a circuit, we induce an arithmetic circuit directly from data using a score function that penalizes circuits with more edges. The score of an arithmetic circuit C on an i.i.d. training sample T is

$$\text{score}(C, T) = \log P(T|C) - k_e n_e(C) - k_p n_p(C)$$

where the first term is the log-likelihood of the training data, $P(T|C) = \prod_{X \in T} P(X|C)$, $k_e \geq 0$ is the per-edge penalty, $n_e(C)$ is the number of edges in the circuit, $k_p \geq 0$ is the per-parameter penalty, and $n_p(C)$ is the number of parameters in the circuit. The last two allow us to easily combine our inference-cost penalty with a more traditional one based on model complexity.

We use this formulation for simplicity; our algorithm would work equally well with a Bayesian Dirichlet score [20], with a prior of the form $\exp(-k_e n_e(C) - k_p n_p(C))$, since the computation of the marginal likelihood would be the same as in standard Bayesian network

Table 5.1: Greedy algorithm for learning arithmetic circuits.

```
function LearnAC($T$)
initialize circuit $C$ as product of marginals
loop
  $C_{best} \leftarrow C$
  for each valid split $S(D, V)$ do
    $C' \leftarrow$ SplitAC($C, S(D, V)$)
    if score($C', T$) $>$ score($C_{best}, T$) then
      $C_{best} \leftarrow C'$
    end if
  end for
  if score($C_{best}, T$) $>$ score($C, T$) then
    $C \leftarrow C_{best}$
  else
    return $C$
  end if
end loop
```

learning. Aside from its practical utility, a prior penalizing inference cost is meaningful if we believe the inference task being modeled can be carried out quickly, for example because humans do it. Either way, the main difficulty is that the penalty (or prior) is no longer node-decomposable, and repeatedly computing it might be very expensive. Reducing this cost is one of the key technical issues addressed in this chapter.

Arithmetic circuits can be learned in the same way as Bayesian networks with local structure, by starting with an empty network and greedily applying the best splits, except that candidate structures are scored by compiling them into arithmetic circuits. However, compiling an arithmetic circuit can be computationally costly, and doing so for every candidate structure would be prohibitive. A better approach is to incrementally compile the circuit as splits are applied. Table 5.1 shows pseudo-code for this algorithm.

The algorithm begins by constructing the initial arithmetic circuit C as a product of marginal distributions:

$$C = \prod_i \sum_j I(X_i = x_{ij}) P(X_i = x_{ij})$$

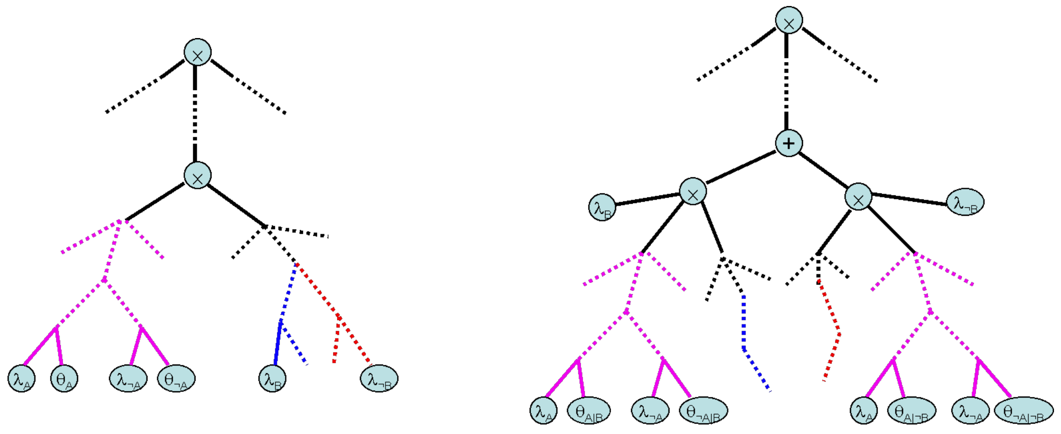

Figure 5.2: Visual representation of applying SplitAC to a circuit. On the left is the original circuit. On the right is the circuit after the distribution over variable A has been split on variable B. Regions of the circuit are colored to show analogous regions in the circuit. Dotted lines indicate portions of the circuit that are not shown in full detail.

This initial circuit is equivalent to a Bayesian network with no edges. In each iteration, the algorithm greedily chooses and applies the best valid split, where split validity is defined according to the equivalent Bayesian network. Each split is scored by applying it to the current circuit and counting the edges and parameters.[2] Learning ends when the algorithm reaches a local optimum, where no valid split improves the score.

5.5.2 Splitting Distributions

The key subroutine is SplitAC, which updates an arithmetic circuit without recompiling it from scratch. Given an arithmetic circuit C that is equivalent to a Bayesian network B and a valid split $S(D, V)$, SplitAC returns a modified circuit C' that is equivalent to B after applying split $S(D, V)$. We will use the following notation to refer to distributions, parameter nodes, and indicator nodes:

d_j**:** Parameter node corresponding to the jth probability in the multinomial distribution D.

[2]All model parameters are MAP estimates, using a Dirichlet prior with all hyperparameters $\alpha_{ijk} = 1$, where k ranges over the leaves of the decision tree for variable X_i.

D_i: Leaf distribution resulting from split $S(D, V)$ that replaces D when $V = i$.

d_{ij}: Parameter node corresponding to the jth probability in D_i.

v_i: Indicator node $I(V = i)$.

Table 5.2 contains pseudo-code for the splitting algorithm. Figure 5.2 uses a visual representation to more intuitively illustrates the operations SplitAC performs. It might at first appear that to split D on V it suffices to replace references to each d_j with a sum of products, $\sum_i d_{ij} v_i$. However, the resulting circuit would then be correct only when V is fixed to a particular value, and summing out V would produce inconsistent results. Intuitively, the circuit must maintain the running intersection property of the corresponding junction tree, so that no variable can take on different values in different subcircuits. SplitAC maintains a consistent probability distribution by preserving three properties, analogous to those defined by [13] for logical circuits.

Definition 3. *For an arithmetic circuit, C:*

- *C is* smooth *if, for each addition node, all children are ancestors of indicator nodes for the same variables and parameter nodes from distributions of the same variables.*

- *C is* decomposable *if, for each multiplication node, no two children are ancestors of indicator nodes for the same variable or parameter nodes from distributions of the same variable.*

- *C is* deterministic *if, for each addition node, there is a variable V such that each child is the ancestor of some non-empty set of indicator nodes for V, and their sets are disjoint.*

The network polynomial for a Bayesian network contains one term for each configuration of its variables; each term includes exactly one indicator variable and one conditional probability parameter per variable. Intuitively, if C is not smooth, then some terms in the polynomial it computes may not have an indicator variable and a conditional probability parameter for every variable. If C is not decomposable, then some terms in the polynomial may have more than one indicator variable or conditional probability parameter for some variable. If C is not deterministic, then there may be multiple terms for the same set of indicator variables.

Definition 4. *We define three special types of node in the circuit as follows:*

- *A* D-ancestor *is any leaf d_j corresponding to a parameter of D, or any parent of a D-ancestor.*
- *A* V-ancestor *is any leaf v_i corresponding to an indicator of V, or any parent of a V-ancestor.*
- *A* mutual ancestor (MA) of D and V *is a node that is both a D-ancestor and a V-ancestor, and has no child that is both a D-ancestor and a V-ancestor.*

Note that every MA must be a multiplication node, or the circuit would not be smooth. Furthermore, from decomposability, each MA must have exactly one D-ancestor child, n_D, and one V-ancestor child, n_V. Naively replacing d_j with $\sum_i d_{ij} v_i$ would cause both n_V and n_D to be ancestors of v_i, violating decomposability.

To avoid this, SplitAC duplicates the subcircuits between the MAs and the parameter nodes d_j, and between the MAs and the indicator nodes v_i, "conditioning" each copy on a different value of V. Each n_V and n_D are replaced by a new addition node, n_+, that sums over products of v_i and copies of n_V and n_D conditioned on v_i. This duplication of subcircuits is the reason different splits can have widely different edge costs. We now describe the details of which nodes are duplicated and how they are connected.

Let N be the set of all D-ancestors and V-ancestors that are also descendants of a mutual ancestor. These are all the nodes "in between" D and V that must agree on the value of V. For each value i in the domain of V, SplitAC creates a copy N_i of the nodes in N.

Let $n_i \in N_i$ be the copy of node $n \in N$. SplitAC inserts edges from n_i to its children as follows. If n has a child $c \in N$, then it inserts an edge from n_i to the corresponding copy c_i.

If n has a child $c \notin N$, then it inserts an edge from n_i to c. This minimizes node duplication by linking to existing nodes or copies whenever possible.

A few additional changes are required for N_i to properly depend on v_i. If $n_i \in N_i$ has some parameter node d_j as a child, SplitAC replaces it with d_{ij}. This is how the new leaf distributions, conditioned on V, are integrated into the circuit. Secondly, if n_i has v_i as a child, it should be omitted: every node in N_i will depend on v_i, so this is redundant. Finally, if n_i has a child that is an ancestor of some v_j but not of v_i, then that child is inconsistent with conditioning on v_i and must be removed.

Finally, SplitAC connects each mutual ancestor, m, to a sum over these copies. SplitAC removes the D-ancestor, n_D, and the V-ancestor, n_V, as children of m and replaces them with an addition node with one child for each value of V. The ith child of the addition node is a product of v_i, the copy of n_D from N_i, and the copy of n_V from N_i. (If m was an ancestor of only certain values of V, the addition node sums only over those values.)

Intuitively, the resulting circuit represents the correct probability distribution because D has been replaced with the split distributions D_i, each conditioned on v_i, and because the circuit satisfies the running intersection property, since all nodes between V and D now depend on V.

Theorem 1. *After each iteration of LearnAC, C computes the network polynomial of a Bayesian network constructed by starting with an empty network and applying the same splits that were applied to C up to that iteration.*

The proof can be found in Section 5.6.

5.5.3 Optimizations

We now discuss optimizations necessary to make this algorithm practical for real-world datasets with many variables.

Consider once again the high-level overview in Table 5.1. Scoring every possible circuit in every iteration would be very expensive. Choosing the split that leads to the best scoring circuit is equivalent to choosing the split that leads to the greatest increase in score, so we can store changes in score instead. The improvement in log-likelihood is not affected by other splits, and so this only needs to be computed once for each potential split. Unfortunately, the number of edges that a split adds to the circuit can increase or decrease due to other splits. For convenience, we will refer to the number of edges added by the application of a split as its *edge cost*.

As a simple example, consider a chain-structured junction tree of 5 variables: AB-BC-CD-DE-EF. If we add an arc from A to F, then A is added to every other cluster: AB-ABC-ACD-ADE-AEF. However, this also reduces the cost of adding an arc from A to E, since the two variables now appear together in a cluster. As a second example, suppose that we instead added an arc from B to F: AB-BC-BCD-BDE-BEF. Now the cost of adding an arc from A to F is greatly increased, since adding a variable to a larger cluster costs more edges than adding a variable to a smaller cluster.

Evaluating the edge cost of every potential split in every iteration is expensive. The number of potential splits is linear in the number of splits that have been performed so far, leading to a time complexity that is at least quadratic in the total number of splits. Further, computing the edge cost for a single candidate may be linear in the size of the current circuit. With a non-zero edge cost, circuit size tends to be linear in the number of iterations, leading to an $O(n^3)$ algorithm. While this is still polynomial, it makes learning models with thousands of splits intractable in practice.

Fortunately, most splits only change a fraction of edge costs. Determining exactly which costs need to be updated is difficult, but we can rule out many splits whose costs do not need to be updated using the following conservative rule. Applying one split may change the edge cost of another split $S(D, V)$ if the applied split changes a node that is an ancestor of D and not V, or of V and not D. This covers all nodes that lie between D or V and their mutual ancestors, and thus all nodes that are copied by the splitting procedure. An applied split changes a node when it copies that node or reduces the number of children it has. In practice, this single heuristic lets us avoid recomputing over 95% of the edge costs.

As an alternative to this optimization, we have found a heuristic that leads to even larger speed-ups, but at the cost of no longer being perfectly greedy. We noticed that when edge costs changed, they rarely decreased. If a split's last computed edge cost was always a valid lower bound on the true value, then we could ignore any split whose total estimated score was worse than the best split found so far in this iteration. This assumption is often not valid in practice, but it lets us learn models that are nearly as effective in an order of magnitude less time.

Two other optimizations combine well with either of the above to offer further gains. First, we can reduce the number of computations by placing potential splits in order of decreasing likelihood gain, so that we consider the splits with the highest possible scores first. Since the likelihood gain is an upper bound on the score gain, once the score of the best split found so far is greater than the next likelihood gain, this split is guaranteed to be the highest-scoring one overall.

Second, we can exit the edge calculation procedure once we know that the edge cost is sufficient to make the overall score negative. It is also possible to exit once we know that the score of the current split will be worse than the best split so far, but this interferes with the other optimizations. If we only compute an upper bound on the score, we will often have to recompute the edge cost when the next iteration requires a slightly lower upper bound.

5.6 Proofs

In this section, we present a full proof of Theorem 1, which establishes the correctness of our learning algorithm. The proof is built up from a number of lemmas that describe properties of arithmetic circuits, properties of isomorphic logical circuits, and invariants maintained by LearnAC in every iteration. The general outline of the proof is as follows.

Our proof rests heavily on Theorem 1 from Darwiche [13], which says that a logical circuit satisfying the properties of smoothness, determinism, and decomposability, and whose models are the terms of the network polynomial, can be converted to an arithmetic circuit that computes the network polynomial. We begin by defining analogous properties of smoothness, determinism, and decomposability on arithmetic circuits and showing that they are satisfied at every step in our algorithm. We use these properties to prove properties of mutual ancestors along the way. We then define a mapping from arithmetic to logical circuits and show that smooth, deterministic, and decomposable arithmetic circuits are mapped to logical circuits with similar properties. Finally, we show that the models of these logical circuits correspond to the terms of the appropriate network polynomial at every stage in the algorithm. This allows us to apply Darwiche's Theorem 1 to conclude that our arithmetic circuits always compute the appropriate network polynomials.

5.6.1 Background

We begin by restating the central theorem from Darwiche, along with the necessary supporting definitions:

Theorem 2 (Theorem 1 in (Darwiche, 2002))**.** *Let Δ be a smooth d-DNNF which encodes a multi-linear function f. The arithmetic circuit encoded by Δ implements the function f.*

A negated normal form (NNF) is a rooted, directed acyclic graph in which each leaf node is labeled with a literal, *true* or *false*, and each internal node is labeled with a conjunction $\wedge$ or a disjunction $\vee$ [13]. To highlight the correspondence between arithmetic circuits and NNFs, we will sometimes refer to NNFs as "logical circuits."

For a node n in an NNF, $Vars(n)$ refers to the set of all propositional variables that are descendants of n, and $\Delta(n)$ refers to the formula represented by n and its descendants.

A smooth d-DNNF is an NNF satisfying the three following properties, taken directly from Darwiche [13]:

- Smoothness holds when $Vars(n_i) = Vars(n_j)$ for any two children n_i and n_j of an or-node n.

- Determinism holds when $\Delta(n_i) \wedge \Delta(n_j)$ is logically inconsistent for any two children n_i and n_j of an or-node n.

- Decomposability holds when $Vars(n_i) \cap Vars(n_j) = \emptyset$ for any two children n_i and n_j of an and-node n.

The multi-linear function encoded by Δ is a polynomial in which each term corresponds to a satisfying assignment, or model, of Δ. Each term is constructed as the product of all true variables in the assignment.

The arithmetic circuit encoded by Δ refers to the circuit obtained by replacing each conjunction with multiplication, each disjunction with addition, and each negative literal with 1.

5.6.2 Properties of Arithmetic Circuits

We now prove certain properties of arithmetic circuits which are necessary for proper operation of the algorithm as well as other later proofs. These properties are analogous to the logical circuit properties defined in the previous section.

From Section 5.4, recall that our arithmetic circuits are rooted, directed acyclic graphs in which leaf nodes are indicators or network parameters, and internal nodes are addition or multiplication operations.

We use $IN(n, V, C)$ to denote the set of indicator nodes associated with variable V that are descended from node n in circuit C. Similarly, $PN(n, V, C)$ denotes the set of parameter nodes associated with variable V (possibly from many different conditional distributions) that are descended from node n in circuit C.. We further define $IV(n, C)$ and $PV(n, C)$ as the sets of variables corresponding to the indicator and parameter nodes descended from n:

$$IV(n, C) = \{V | IN(n, V, C) \neq \emptyset\}$$
$$PV(n, C) = \{V | PN(n, V, C) \neq \emptyset\}$$

These function are parameterized with a circuit as well as a node in order to allow for distinctions between the properties of a node before and after a call to SplitAC.

For each of the stated NNF properties, we can define an analogous property on an arithmetic circuit, C:

- Smoothness holds when $IV(n_i, C) = IV(n_j, C)$ and $PV(n_i, C) = PV(n_j, C)$ for any two children n_i and n_j of an addition node n.

- Determinism holds when there is some V such that $IN(n_i, V, C) \neq \emptyset$, $IN(n_j, V, C) \neq \emptyset$, and $IN(n_i, V, C) \cap IN(n_j, V, C) = \emptyset$ for any two children n_i and n_j of an addition node n.

- Decomposability holds when $IV(n_i, C) \cap IV(n_j, C) = \emptyset$ and $PV(n_i, C) \cap PV(n_j, C) = \emptyset$ for any two children n_i and n_j of a multiplication node n.

A node is smooth, deterministic, and decomposable if the corresponding properties are satisfied for the particular node rather. Thus, a circuit is smooth, deterministic, and decomposable if and only if each of its nodes is smooth, deterministic, and decomposable.

Since each iteration of LearnAC implicitly depends on the operation of SplitAC, we must first prove that mutual ancestors satisfy the properties SplitAC assumes.

Lemma 1. *If C is smooth and decomposable, then every mutual ancestor (MA) of variable V and distribution D is a multiplication node with one child that is a V-ancestor and one that is a D-ancestor.*

Proof. By Definition 4, any MA n must be an ancestor of both V and D. Let n_V by a child of n that is also an ancestor of V and let n_D be a child of n that is also an ancestor of D. If n was an addition node, then n_D would be a V-ancestor, violating the condition that no child of an MA is an ancestor of both V and D. Therefore, every MA is a multiplication node.

By decomposability, n can have at most one child that is an ancestor of D and one child that is an ancestor of V. By definition, every MA must have at least one that is an ancestor of each, and the two cannot be the same. □

Let C' be the circuit that results from calling SplitAC($C, S(D, V)$). The following lemma describes how the indicator and parameter nodes descended from nodes in C' relate to those descended from similar nodes in C. This is important for later proving that the arithmetic circuits are always smooth, deterministic, and decomposable.

Lemma 2. *If C is smooth, deterministic, and decomposable, then for each node $n \in C$:*

If $n \in C'$, then $PV(n, C') = PV(n, C)$; $IV(n, C') = IV(n, C)$; and for any domain variable U, $IN(n, U, C') = IN(n, U, C)$.

If n was copied, then for each copy $n' \in C'$, $PV(n', C') = PV(n, C)$; $IV(n', C') = IV(n, C) \setminus V$; and for any domain variable $U \neq V$, $IN(n', U, C') = IN(n, U, C)$.

Proof. We prove this lemma by induction on the structure of C', showing that if the lemma is true for all children of n, then it is also true for n.

Base case: If $n \in C$ has no children and $n \in C'$, then n clearly has identical descendants in C' as in C, so all conditions are satisfied. If $n \notin C'$, then the lemma does not apply, since SplitAC never copies nodes without children.

Inductive step: Suppose that the lemma is true for each child of $n \in C$ and that n has one or more children, n_i.

Case 1: If $n \in C'$ and is not an MA, then it was not directly changed by SplitAC. n has the same children in C and C', and for each child n_i, $n_i \in C$. We begin by writing $IV(n, C')$ recursively in terms of its children:

$$IV(n, C') = \bigcup_i IV(n_i, C')$$

By the inductive hypothesis:

$$IV(n, C') = \bigcup_i IV(n_i, C) = IV(n, C)$$

An identical argument applies to show that $PV(n, C') = PV(n, C)$.

For an arbitrary variable, U:

$$\begin{aligned} IN(n, U, C') &= \bigcup_i IN(n_i, U, C') \\ &= \bigcup_i IN(n_i, U, C) \\ &= IN(n, U, C) \end{aligned}$$

where the second equality follows from the inductive hypothesis.

Case 2: If a copy of n is present in C', then let $n' \in C'$ be an arbitrary copy of n. From the operation of SplitAC, the children of n can be partitioned into three disjoint sets: N_{same}, nodes that are also children of n'; N_{copy}, nodes of which a copy is a child of n'; and N_{none}, nodes which are excluded from the children of n'. The children of n' can similarly be partitioned into two sets: N'_{copy}, copies of nodes that are children of n, and N_{same}.

We handle the PV/IV and IN conditions as two separate sub-cases.

We prove the IN condition first. For an arbitrary variable, $U \neq V$:

$$\begin{aligned} IN(n, U, C') =& \bigcup_i IN(n_i, U, C') \\ =& \bigcup_i IN(n_i, U, C) \\ =& IN(n, U, C) \end{aligned}$$

where the second equality follows from the inductive hypothesis.

Addition: Suppose n' is an addition node. By smoothness, $IV(n'_i, C') = IV(n'_j, C')$ for any two children of n'. Therefore, $IV(n', C') = IV(n'_i, C')$. $n \in C$ must also be an addition node, so $IV(n, C) = IV(n_i, C)$ for any child of n, n_i.

Let n'_i be an arbitrary child of n'. If $n'_i \in N'_{copy}$, then let $n_i \in N_{copy}$ be the node of which n'_i is a copy. By the inductive hypothesis, $IV(n'_i, C') = IV(n_i, C) \setminus V$, so by substitution, $IV(n', C') = IV(n, C) \setminus V$.

Otherwise $n'_i \in N_{same}$, so by the inductive hypothesis, $IV(n'_i, C') = IV(n'_i, C)$. In the operation of SplitAC, children of copied nodes that are ancestors of V are also copied, so we can conclude that $IV(n'_i, C) = IV(n'_i, C) \setminus V$. By substitution, $IV(n', C) = IV(n, C) \setminus V$. An identical argument applies to show that $PV(n', C) = PV(n, C)$.

Multiplication: Now suppose, instead, that n' is a multiplication node. We first prove, by contradiction, that $N_{none} = \emptyset$. Suppose, to the contrary, that there exists some $n_i \in N_{none}$. From the operation of SplitAC, there are indicator nodes v_i and v_j such that $v_j \in IN(n_i, V, C)$ and $v_i \notin IN(n_i, V, C)$. By decomposability and the definition of IV, since $v_j \in IN(n_i, V, C)$, $v_i \notin IN(n_j, V, C)$ for any other child of n. Therefore, $v_i \notin IN(n, V, C)$ and $v_j \in IN(n, V, C)$, so n should be omitted from this copy as well. Since we originally assumed n' exists, $N_{none} = \emptyset$.

$IV(n', C')$ can be written recursively as follows:

$$IV(n', C') = \bigcup_{n'_i \in N_{same} \cup N'_{copy}} IV(n'_i, C')$$

As pointed out in the addition case, children of copied nodes that are ancestors of V are also copied, so for $n_i \in N_{same}$, $IV(n_i, C) = IV(n_i, C) \setminus V$. Combined with the inductive hypothesis, we may conclude:

$$IV(n', C') = \bigcup_{n_i \in N_{same} \cup N_{copy}} IV(n_i, C) \setminus V$$

Since $N_{none} = \emptyset$, adding it to a union changes nothing:

$$IV(n', C') = \bigcup_{n_i \in N_{same} \cup N_{copy} \cup N_{empty}} IV(n_i, C) \setminus V$$

Since every child of n is an element of N_{same}, N_{copy}, or N_{none}, this is the recursive statement of $IV(n, C) \setminus V$:

$$IV(n', C') = IV(n, C) \setminus V$$

An identical argument applies to show that $PV(n', C) = PV(n, C)$.

Case 3: If n is an MA for the split, then from Lemma 1 we know n is a multiplication node and its children include exactly one ancestor of D, n_D; exactly one ancestor of V, n_V; and a set of other children which we will call N_o.

From the operation of SplitAC, every $n_i \in N_o$ is unchanged by the algorithm, and therefore $n_i \in C'$. n_D and n_V, however, are replaced by an addition node, n'_+. The children of n_+ are products of some indicator node for V, v_i; a copy of n_V, $n'_{V,i}$; and a copy of n_D, $n'_{D,i}$.

We can write $PV(n, C')$ as:

$$PV(n, C') = PV(n'_+, C') \cup \bigcup_{n_i \in N_o} PV(n_o, C')$$

We describe $PV(n'_+, C')$ in terms of n'_+'s grandchildren:

$$\bigcup_i PV(n'_{D,i}, C') \cup PV(n'_{V,i}, C') \cup V$$

V is included in the union since v_i is a child of the ith child of n_+. By the inductive hypothesis, since every $n'_{D,i}$ and $n'_{V,i}$ is a copy, $PV(n'_{D,i}, C') = PV(n_D, C)$ and $PV(n'_{V,i}, C') = PV(n_V, C)$. We can therefore substitute and simplify to obtain:

$$IN(n'_+, U, C') = PV(n_D, C) \cup PV(n_V, C)$$

Substituting into our previous expresion for $PV(n, C')$:

$$\begin{aligned} PV(n, C') = & PV(n_V, C) \cup PV(n_D, C) \\ & \cup \bigcup_{n_i \in N_o} PV(n_o, C') \end{aligned}$$

By the inductive hypothesis, $PV(n_o, C') = PV(n_o, C)$, so this reduces to the recursive description of $PV(n, C)$.

Our procedure for proving that $IN(n, U, C') = IN(n, U, C)$ is nearly identical. First, we handle the case where $U \neq V$.

$$IN(n, U, C') = IN(n'_+, U, C') \cup \bigcup_{n_i \in N_o} IN(n_o, U, C')$$

As before, we describe $IN(n'_+, U, C')$ in terms of the grandchildren of n'_+:

$$\bigcup_i IN(n'_{D,i}, U, C') \cup IN(n'_{V,i}, U, C') \cup v_i$$

Note that we can safely ignore the v_i's, since $U \neq V$. By applying the inductive hypothesis and simplifying, we obtain:

$$IN(n'_+, U, C') = IN(n_D, U, C) \cup IN(n_V, U, C)$$

We can substitute this into the original expression:

$$\begin{aligned} IN(n, U, C') =& IN(n_D, U, C) \cup IN(n_V, U, C) \\ & \cup \bigcup_{n_i \in N_o} IN(n_o, U, C') \end{aligned}$$

which is equivalent to $IN(n, U, C)$. Since $IN(n, U, C') = IN(n, U, C)$ for all U, it follows that $IV(n, C') = IV(n, C)$.

For the case where $U = V$, SplitAC explicitly includes the indicator node v_i as a grandchild of n_+ for each v_i that is a descendant of n. Therefore, the lemma holds for this case as well. □

Lemma 3. *At every iteration of LearnAC, C is smooth, decomposable, and deterministic.*

Proof. By induction.

Base case: We first show that the initial circuit is smooth, deterministic, and decomposable. The initial circuit is a product of sums of products. The leaves are indicator nodes for each value of each variable and parameter nodes for the marginal probability of each variable value. Above this are multiplication nodes, each the product of one parameter node and one indicator node, clearly satisfying decomposability. Above this are summations, each summing over the values and probabilities of a single variable. Since each child of a given addition node is a parent of values and parameters for the same variable, all addition nodes satisfy smoothness. Since the values being summed out are mutually exclusive, they also satisfy determinism. The top level multiplication is over marginal distributions for different variables, so it satisfies decomposability.

Inductive step: Let C be the circuit after the last iteration of LearnAC and let C' be the circuit that results from calling SplitAC$(C, S(D, V))$. Assuming the circuit, C, was smooth, deterministic, and decomposable after the last iteration of LearnAC, we must show that C' is also smooth, deterministic, and decomposable.

We demonstrate this for each node $n' \in C'$. If $n' \in C$ and n' is not an MA, then each of its children n_i is also in C, since every path to a copied or created node leads through an MA. By Lemma 2, $PV(n_i, C') = PV(n_i, C)$, $IV(n_i, C') = IV(n_i, C)$, and

$IN(n_i, U, C') = IN(n_i, U, C)$. By the inductive hypothesis, n' satisfied smoothness and decomposability before the split. Since PV, IV, and IN remain the same for all children, n must still satisfy smoothness and decomposability.

If n' is a copy of some $n \in C$, then each of its children n'_i is also a copy of a child of n, $n_i \in C$. Consider two children of n, n'_i and n'_j, and the corresponding children of n, n_i and n_j.

Suppose n' and n are addition nodes. By Lemma 2, $IV(n'_i, C') = IV(n_i, C) \setminus V$. By smoothness, $IV(n_i, C) \setminus V = IV(n_j, C) \setminus V$. By Lemma 2, $IV(n_j, C) \setminus V = IV(n'_j, C')$, so by transitivity $IV(n'_i, C') = IV(n'_j, C')$. The same argument can be applied to show that $PV(n'_i, C') = PV(n'_j, C')$, so n' is smooth.

Since n is deterministic, there must be some $U \in IV(n_i, C)$ such that $IN(n_j, U, C) \cap IN(n_i, U, C) = \emptyset$. If $U \neq V$, then by applying Lemma 2 and transitivity we can infer that n' is deterministic. If $U = V$, then let v_i be the value of V that the particular copy n' is conditioned on during the operation of SplitAC. From determinism and our choice of V, v_i cannot be a descendant of both n_i and n_j for $n_i \neq n_j$. Since both n_i and n_j are ancestors of some indicator of V, but both are not ancestors of v_i, at most one child is copied. Therefore, n' only has one child and determinism is trivially satisfied.

This leaves the case of the MAs and newly created nodes. If n' is a newly created multiplication node, then its children are some indicator node v_i and copies $n'_{D,i}$ and $n'_{V,i}$ of two children the MA, n_D and n_V. By the inductive hypothesis, the MA satisfied decomposability before the split, so $IV(n_V, C) \cap IV(n_D, C) = \emptyset$ and $PV(n_V, C) \cap PV(n_D, C) = \emptyset$. By Lemma 2, $IV(n'_V, C') = IV(n_V, C) \setminus V$, $IV(n'_D, C') = IV(n_D, C) \setminus V$ (and $IV(v_i, C') = V$). The PV are similarly disjoint, demonstrating that n' satisfies decomposability.

If n' is a newly created addition node, then its children are newly created multiplication nodes, n'_i. By their construction in SplitAC:

$$IV(n'_i, C') = V \cup IV(n'_{D,i}, C') \cup IV(n'_{V,i}, C')$$

From Lemma 2, since each $n'_{D,i}$ and $n'_{V,i}$ is a copy of the same n_D and n_V, their IV are identical. Therefore, $IV(n'_i, C') = IV(n'_j, C')$, so n' is smooth. By construction,

$IN(n'_i, V, C') = v_i$, so $IN(n'_j, V, C') \cap IN(n'_i, V, C') = \emptyset$ and decomposability is satisfied.

Finally, if n' is an MA node, then two of its children have been replaced by a new addition node, n'_+: $IV(n'_+, C') = \prod_i IV(n'_i, C')$, where each n'_i is a new multiplication node. Since we already showed that n'_+ is smooth:

$$\begin{aligned} IV(n'_+, C') &= IV(n'_i, C') \\ &= V \cup IV(n'_{D,i}, C') \cup IV(n'_{V,i}, C') \\ &= V \cup (IV(n_D, C) - V) \\ &\quad \cup (IV(n_V, C') - V) \\ &= IV(n_D, C) \cup IV(n_V, C). \end{aligned}$$

The second equality follows from Lemma 2. Applying the same argument demonstrates that $PV(n'_+, C') = PV(n_D, C) \cup PV(n_V, C)$.

Given another child of n', n'_i, from Lemma 2 we know that $IV(n'_i, C') = IV(n_i, C)$. Therefore, we can write the intersection as:

$$\begin{aligned} & IV(n'_i, C') \cap IV(n'_+) \\ &= IV(n_i, C) \cap (IV(n_D, C) \cup IV(n_V, C)) \\ &= (IV(n_i, C) \cap IV(n_D, C)) \\ &\quad \cup (IV(n_i, C) \cap IV(n_V, C)) \end{aligned}$$

The second equality is an application of the distributive law. Applying the inductive hypothesis that n is decomposable, we can conclude:

$$IV(n'_i, C') \cap IV(n'_+) = \emptyset \cup \emptyset = \emptyset$$

The same result holds for PV. Given n'_i and n'_j where neither is n'_+, Lemma 2 and the inductive hypothesis show that their respective IV and PV are also disjoint. Therefore, n' satisfies decomposability. $\square$

5.6.3 Properties of the Logical Image

For an arithmetic circuit, C, that represents some Bayesian network over the variables $X_1, \ldots, X_n$, the *logical image* of C, $\mathcal{L}(C)$, is obtained by replacing addition with disjunction and multiplication with conjunction. In order to make the different values of each variable mutually exclusive, we replace indicator nodes v_i with conjunctions of v_i and the negation of every other v_j for $j \neq i$. We make the conditional probability parameters for each variable mutually exclusive with each other using an analagous transformation. We use $\mathcal{L}(n, C)$ to refer to the node in $\mathcal{L}(C)$ that replaces $n \in C$. For non-leaf $n' \in \mathcal{L}(C)$, we use $\mathcal{L}^{-}1(n', \mathcal{L}(C))$ to refer to the node in C that n' replaced.

If we take the logical image of C and replace conjunction with multiplication, disjunction with addition, and negated variables with 1, then we recover an arithmetic circuit that is equivalent to C. Therefore, from our earlier definitions, $\mathcal{L}(C)$ encodes the arithmetic circuit C.

Recall that $\Delta(n)$ refers to the logical formula represented by n and its descendants. We modify this notation as $\Delta(n, L)$, to specify the logical circuit L from which this subformula is drawn. For notational convenience, we will abbreviate $\Delta(\mathcal{L}(n, C))$ as $\Delta(n, C)$, where C is an arithmetic circuit.

The following lemma states necessary conditions for $\Delta(n, C)$ to be true. These will be used several times in later lemmas.

Lemma 4. *If $\mathcal{L}(C)$ is smooth, then $\Delta(n, C)$ is false unless:*

- *For each $V \in IV(n, C)$, the literal for exactly one indicator v_i of V is true, and $v_i \in IN(n, V, C)$*

- *For each $V \in PV(n, C)$, the literal for exactly one parameter d_i of V is true, and $d_i \in PN(n, V, C)$*

Proof. Suppose that $V \in IV(n, C)$, and that in a particular truth assignment, more than one literal for V is true or no $v_i \in IN(n, V, C)$ is true. By induction over the structure of C

and $\mathcal{L}(C)$, we show that $\Delta(n, C))$ must be false. The exact same argument can be applied for the second condition, substituting PV and PN for IV and IN.

Base case: If node n has no children and $V \in IV(n, C)$, then n must be an indicator node v_i. Recall that $\mathcal{L}(v_i, C)$ is a conjunction over the literals for each value of variable V, where only v_i appears non-negated. By the pigeon-hole principle, if more than one literal of V is true, then one of them must be negated in the conjunction, so $\Delta(v_i, C)$ is false. In the second case, if no $v_j \in IN(v_i, V, C)$ is true, then v_i is false. Since $\mathcal{L}(v_i, C)$ is a conjunction that includes v_i, this implies that $\Delta(v_i, C)$ is false.

Inductive step: Suppose the lemma holds for all children of n. Consider first the case where $\mathcal{L}(n, C)$ is a conjunction. Since $V \in IV(n, C)$, n must have some child n_i such that $V \in IV(n_i, C)$. If no $v_i \in IN(n, V, C)$ is true, then no $v_i \in IN(n_i, V, C)$ is true either. By the inductive hypothesis, $\Delta(n_i, C)$ is false, so the conjunction $\Delta(n, C)$ is false. If more than one literal for V is true, then by the inductive hypothesis, $\Delta(n_i, C)$ is false, and hence $\Delta(n, C)$ is false.

Otherwise, $\mathcal{L}(n, C)$ is a disjunction. Since $V \in IV(n, C)$ and $\mathcal{L}(C)$ is smooth, for every child n_i, $V \in IV(n_i, C)$. If more than one literal for V is true, then by the inductive hypothesis, every child $\Delta(n_i, C)$ is false, so the disjunction $\Delta(n, C)$ is false. If no literal $v_i \in IN(n, V, C)$ is true, then no $v_i \in IN(n_i, V, C)$ is true either. By the inductive hypothesis, every $\Delta(n_i, C)$ is false, so $\Delta(n, C)$ is also false. $\square$

Lemma 5. *The logical image of a smooth, deterministic, and decomposable arithmetic circuit is a smooth d-DNNF.*

Proof. Given an arithmetic circuit, C, $\mathcal{L}(C)$ is clearly an NNF. To show that it is a smooth d-DNNF, we must show that $\mathcal{L}(C)$ satisfies smoothness, determinism, and decomposability.

Every conjunction or disjunction $n' \in \mathcal{L}(C)$ is a replacement for some node $n \in C$. For each $n \in C$, we will demonstrate that the replacement $n' \in \mathcal{L}(C)$ satisfies smoothness, determinism, and decomposability.

If $n \in C$ has fewer than two children, then $n' \in \mathcal{L}(C)$ must either be a conjunction of literals representing all indicator or parameter nodes for a particular variable, or it must be a disjunction or conjunction with fewer than two children. In the former case, no two children represent the same literal so decomposability is satisfied and neither smoothness nor determinism applies. In the latter case, smoothness, determinism, and decomposability are all trivially satisfied by n'.

Otherwise, let n_i and n_j be two arbitrary children of n, and let n'_i and n'_j be the corresponding replacements in $\mathcal{L}(C)$.

Suppose that an indicator v_k (or parameter d_k) of variable V is in $Vars(n'_i)$, the set of all literals in the subgraph rooted at n'_i. n'_i must be a parent or ancestor of a parent of v_k (or d_k). Parents of literal nodes are conjunctions that were handled earlier, in the case where n has fewer than two children. Therefore, n'_i is an ancestor of a parent of v_k's literal (or d_k's), which is a conjunction that replaced some v_l (or d_l) in C. In the original circuit, n_i is therefore an ancestor of v_l (or d_l). We use this fact as a starting point for proving decomposability, smoothness, and determinism.

If n is a multiplication node, then by decomposability, $V \notin IV(n_j)$ (or $PV(n_j)$), since $V \in IV(n_i)$ (or $IV(n_i)$). From how the logical image is constructed, v_k (or d_k) $\notin Vars(n'_j)$, where v_k (d_k) is any literal for an indicator (parameter) of V. Since the child nodes and descendant of n'_i were arbitrary, $Vars(n'_i) \cap Vars(n'_j) = \emptyset$, and n' satisfies decomposability.

Otherwise, n is an addition node. By smoothness, $V \in IV(n_j)$ (or $PV(n_j)$). From the construction of the logical image, n'_i and n'_j must both be ancestors of every indicator (or parameter) node for V. Since the child node was arbitrary, $Vars(n'_i) \subset Vars(n'_j)$. Since n'_i and n'_j are arbitrary, by symmetry $Vars(n'_j) \subset Vars(n'_i)$. Therefore, $Vars(n'_i) = Vars(n'_j)$, so n' satisfies smoothness.

From determinism, there must be some variable V such that n_i and n_j are both ancestors of one or more indicator nodes for V, but no indicator node v_k is the descendant of both. Suppose n'_i is true for a particular assignment of truth values to logical variables. By Lemma 4 and determinism, some descendant v_k of n_i must be true, where v_k is not a descendant of c_j. By Lemma 4, n'_j can only be true when exactly one indicator variable of V is true. Since we have selected v_k to be true and v_k is not a descendant of n_j, by Lemma 4 n'_j is false. Therefore, n'_j is false whenever n'_i is true, so n'_i and c'_j are logically inconsistent. Since the child nodes were arbitrary, n' satisfies determinism. □

Lemma 6. *After each iteration of LearnAC, the models of $\mathcal{L}(C)$ are the terms of the network polynomial for a Bayesian network constructed by starting with an empty network and applying the same splits that were applied to C up to that iteration.*

For a logical formula Δ, we will use the notation $\Delta[x/y]$ to refer to the logical formula obtained by replacing every occurrence of x in Δ with y.

Proof. (By induction.)

Base case: The initial circuit is a product of marginal distributions, equivalent to a Bayesian network with no arcs.

Inductive step: Suppose that, after k iterations, the models of $\mathcal{L}(C)$ are the terms of the network polynomial for some Bayesian network, B. In the $(k+1)$st iteration, LearnAC selects and applies some split $S(D,V)$. Let B' and C' be the resulting Bayesian network and arithmetic circuit. The network polynomial of B' is identical to that of B except that in every term containing an indicator of V, v_i, and a parameter of distribution D, d_j, the parameter d_j is replaced by $d_{i,j}$. We will demonstrate that the models of $\mathcal{L}(C')$ are exactly these terms.

Let m be a mutual ancestor of D and V in C. From Lemma 1 and the logical image transformation,

$$\Delta(m,C) = \Delta(n_D,C) \wedge \Delta(n_V,C) \wedge (\bigwedge_{o \in O} \Delta(o,C))$$

where n_D is the D-ancestor, n_V is the V-ancestor, and O is the set of all other children. Now consider how m is changed by calling SplitAC$(C, S(D,V))$. By construction,

$$\Delta(m,C') = \bigvee_j (\Delta(v_j,C') \wedge \Delta(n'_{D_j},C') \wedge \Delta(n'_{V_j},C')) \\ \wedge \left(\bigwedge_{o \in O} \Delta(o,C') \right)$$

where $n'_{D,j}$ and $n'_{V,j}$ are copies of n_D and n_V, respectively. Since neither $o \in O$ nor any descendent of o is changed by the split, $\bigwedge_{o \in O} \Delta(o,C') = \bigwedge_{o \in O} \Delta(o,C)$. We will abbreviate this disjunction as $\Delta(O)$.

From Lemma 4, the literal of exactly one indicator for V is true in every model of $\Delta(m, C)$ and $\Delta(m, C')$. Let us restrict m to the case where a particular v_i is true. Note that we only need to apply the substitution to ancestors of v_i:

$$\begin{aligned}\Delta(m, C)[v_i/\top] &= \Delta(O) \wedge \Delta(n_D, C) \wedge \Delta(n_V, C)[v_i/\top] \\ \Delta(m, C')[v_i/\top] &= \Delta(O) \wedge \Delta(n'_{D_i}, C') \wedge \Delta(n'_{V_i}, C')\end{aligned}$$

Note that we have excluded the children of n_+ that are conditioned on other values of V, $v_j \neq v_i$, since the indicators are mutually exclusive.

From the operation of SplitAC, the subtree rooted at n'_{V_i} is a copy of the subtree rooted at n_V, excluding v_i and every node that is an ancestor of some v_j but not v_i. When we apply the substitution $[v_i/\top]$, v_i becomes redundant. From Lemma 4, the other excluded nodes are all false when v_i is true. From decomposability, the other excluded nodes are never children of conjunctions or the parent conjunction would also be excluded. Therefore, the other excluded nodes are false children of disjunctions, which means they can be ignored when v_i is true. Since every excluded node is either redundant or irrelevant when v_i is true, $\Delta(n'_{V_i}, C') = \Delta(n_V, C)[v_i/\top]$.

By construction $\Delta(n_{D_i}, C')$ is identical to $\Delta(n_D, C)$ except with the parameter nodes of D replaced by those of D_i. Since n_D is the only child of m that is an ancestor of D, we can conclude:

$$\begin{aligned}&\Delta(m, C')[v_i/\top] \\ &= \Delta(O) \wedge \Delta(n'_{D_i}, C') \wedge \Delta(n_V, C)[v_i/\top] \\ &= \Delta(m, C)[v_i/\top, d_j/d_{i,j}]\end{aligned}$$

Since every path from the root to D goes through some m, and since only mutual ancestors and their descendants were changed by SplitAC,

$$\mathcal{L}(C')[v_i/\top] = \mathcal{L}(C)[v_i/\top, d_j/d_{i,j}]$$

Since some v_i is true in every model of $\mathcal{L}(C)$ and $\mathcal{L}(C')$ (by Lemmas 3, 4, and 5), the models of $\mathcal{L}(C')$ are the terms of the updated network polynomial. $\square$

5.6.4 Proof of Theorem 1

We now prove the following theorem, first stated in Section 5.5:

Theorem 1. *After each iteration of LearnAC, C computes the network polynomial of a Bayesian network constructed by starting with an empty network and applying the same splits that were applied to C up to that iteration.*

Proof. By Lemma 3, C' is smooth, deterministic, and decomposable. From Lemma 5, we know that its logical image $\mathcal{L}(C')$ is a smooth d-DNNF. By Lemma 6, the models of $\mathcal{L}(C')$ are the terms of the updated network polynomial. It follows from the previous two statements and Theorem 1 from Darwiche [13] that C' computes the network polynomial of B with the split $S(D, V)$. $\square$

5.7 Experiments

5.7.1 Datasets

We evaluated our methods on three widely used real-world datasets. The KDD Cup 2000 clickstream prediction dataset [26] consists of web session data taken from an online retailer. Using the subset of Hulten and Domingos [22], each example consists of 65 Boolean variables, corresponding to whether or not a particular session visited a web page matching a certain category. Anonymous MSWeb is visit data for 294 areas (Vroots) of the Microsoft web site, collected during one week in February 1998. It can be found in the UCI machine learning repository [6]. EachMovie[3] is a collaborative filtering dataset in which users rate movies they have seen. We took a 10% sample of the original dataset, focused on the 500 most-rated movies, and reduced each variable to “rated” or “not rated”. For KDD Cup and MSWeb, we used the training and test partitions provided with the datasets. For EachMovie, we randomly selected 10% of the data for the test set and used the remainder for training.

[3]Provided by Compaq at http://research.compaq.com/SRC/eachmovie/; no longer available for download, as of October 2004.

Basic statistics for each dataset are shown in Table 5.3. Density refers to the fraction of non-zero entries across all examples and all variables.

5.7.2 Learning

For each dataset, we randomly split the training data into tuning and validation sets, corresponding to 90% and 10% of the training data, respectively. All parameters were tuned by training models on the tuning data and selecting the parameter sets that led to the highest log likelihood of the validation set. Finally, models were retrained using the full training set. All experiments were run on CPUs with 4 GB of RAM running at 2.8 GHz.

We used two versions of the algorithm for learning arithmetic circuits from Section 4: AC-Greedy, which guarantees that we pick the best split in each iteration, and AC-Quick, which uses a heuristic to avoid recomputing edge costs but may sometimes choose worse splits. We varied the per-edge cost k_e from 1.0 to 0.01. Not surprisingly, our models were most accurate on the validation set with low per-edge costs (0.01 or 0.02). We also tuned the per-parameter cost k_p. For KDD Cup, the best cost was 0.0; for MSWeb and EachMovie, the best costs were 1.0 for greedy ACs and 0.5 for quick ACs.

We used the WinMine Toolkit [10] as a baseline. WinMine implements the algorithm for learning Bayesian networks with local structure described in Section 2 [9], and has a number of other state-of-the-art features. We tuned WinMine's multiplicative per-parameter penalty κ; the best values were: 1 (no penalty) for KDD Cup, 0.1 for MSWeb, and 0.01 for EachMovie. We looked into using thin junction trees as a second baseline, but they do not scale to datasets of these dimensions.

A summary of the learned models appears in Table 5.4. For each dataset, we report the log-likelihood per example on the test data, the number of edges in the arithmetic circuit, the number of leaves across all decision trees, the average and maximum number of parents across all variables, the treewidth (estimated using a min-fill heuristic), the number of edges generated by compiling the Bayesian network using c2d[4], and the training time. On each model for which c2d ran out of memory, we obtained a lower bound by compiling a model

[4]Available at http://reasoning.cs.ucla.edu/c2d/. We also tried using the ACE package, but it does not support decision tree CPDs and, for our models, tabular CPDs would be prohibitively large.

with fewer splits, obtained by halting the learning process early. We varied the number of splits until we found the most complex sub-model that could still be compiled, within 10 splits. For WinMine, the chosen sub-models had less than one quarter of the original splits.

The test set log-likelihoods of the AC learners and WinMine are very similar, with WinMine having a slight edge. This is not surprising, given that WinMine is free to choose expensive splits. Perhaps more remarkable is that this freedom translates to very little improvement in likelihood. The difference in accuracy between quick and greedy ACs is negligible except in the case of EachMovie, where the greedy AC is actually less accurate because it did not converge in the allowed time (72h).

Not surprisingly, WinMine is much faster than the AC learners. It is worth noting that the cost of learning is only incurred once, while the cost of inference is incurred many times. Also, the AC learner directly outputs an arithmetic circuit, while WinMine's Bayesian network would still have to be compiled into one, which can be very time-consuming. Finally, the quick heuristic offers up to an order-of-magnitude speedup with similar accuracy; additional heuristics might offer additional improvements.

5.7.3 Inference

For each dataset, we used the test data to generate queries with varied numbers of randomly selected query and evidence variables. Each query asked the probability of the configuration of the query variables in the test example conditioned on the configuration of the evidence variables in the same test example.

We estimate inference accuracy as the mean log probability of the test examples's configuration across all test examples. This is an approximation (up to an additive constant) of the Kullback-Leibler divergence between the inferred distribution and the true one, estimated using the test samples. For KDD Cup and MSWeb, we generated queries from 1000 test examples; for EachMovie, we generated queries from all 593 test examples.

For the arithmetic circuits, we used exact inference. For the Bayesian networks learned using WinMine, we used Gibbs sampling. We initialized the sampler to a random state,

[5] AC-Greedy did not finish running in the maximum allowed time of 72h. As a result, it has fewer edges and lower log-likelihood than AC-Quick.

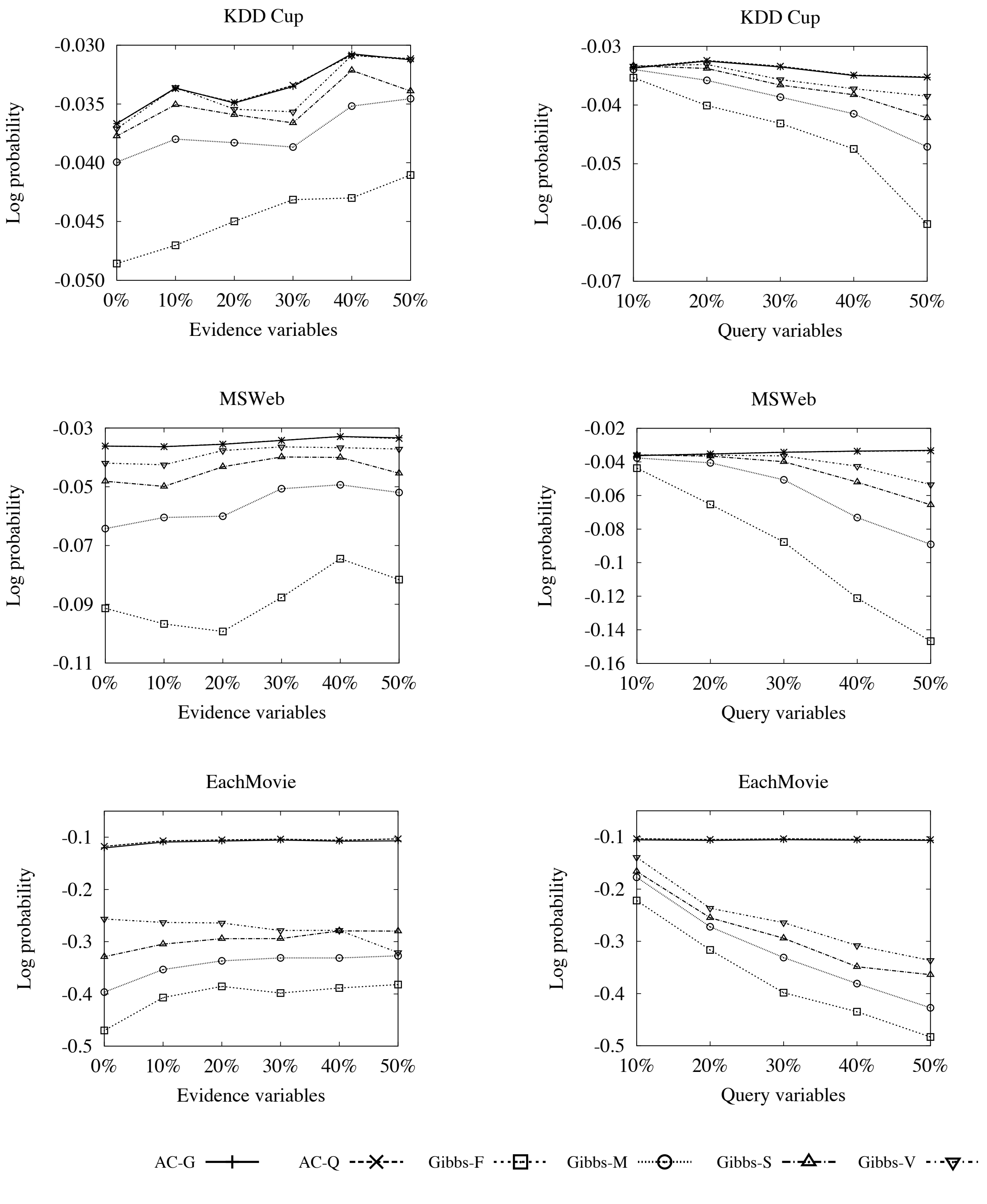

Figure 5.3: Conditional log probability per query variable, per query. In the legend, AC-G refers to AC-Greedy and AC-Q refers to AC-Quick. Gibbs-F, Gibbs-M, Gibbs-S and Gibbs-V refer to the fast, medium, slow, and very slow Gibbs sampling scenarios, respectively.

ran it for a burn-in period, and then collected samples to estimate the probability of the queried marginal or conditional event. All estimates were smoothed by uniformly distributing a count of 1 across all states of the query variables. Since convergence is difficult to diagnose and may take prohibitively long, we ran Gibbs sampling in four scenarios: fast (one chain, 100 burn-in iterations, 1000 sampling iterations); medium (ten chains, 100 burn-in iterations, 1000 sampling iterations); slow (ten chains, 1000 burn-in iterations, 10,000 sampling iterations); and very slow (ten chains, 10,000 burn-in iterations, 100,000 sampling iterations).

Figure 5.3 shows the relative accuracy of the different methods on each dataset. Per-variable query log-likelihood is on the y axis. In the graphs on the left, each query included 30% of the variables in the domain, conditioned on 0% to 50% of the domain variables as evidence. In the graphs on the right, the number of query variables varies from 10% to 50%, conditioned on 30% of the variables in the domain as evidence. Inference times (averaged over all queries) are listed in Table 5.5. Note that AC inference times are in milliseconds, while Gibbs inference times are in seconds.

The ACs were roughly one order of magnitude faster than the fastest runs of Gibbs sampling, and three orders of magnitude faster than the slowest. Except when the number of query variables is very small, the ACs also easily dominate even the slowest runs of Gibbs sampling on accuracy. Because of the approximate inference, the slightly higher test-set log-likelihood of WinMine's models does not translate into higher accuracy in answering queries. Presumably, given enough time Gibbs sampling will eventually catch up with the ACs in accuracy, but by then it will be many orders of magnitude slower. Further, Gibbs sampling (like other approximate inference methods) requires tuning for best results, and we can never be sure that it has converged. In contrast, the AC inference is reliable, the time it takes is predetermined, and the time is short enough for online or interactive use.

5.8 Conclusion

In the past, work on learning and inference in graphical models has been largely separate. This has had the somewhat paradoxical result that much computational effort is often expended to learn accurate models, only to result in less accurate predictions when approximate inference becomes necessary. Our work seeks to ameliorate this by more closely integrating learning and inference. In particular, we presented an algorithm for learning arithmetic circuits by maximizing likelihood with a penalty on circuit size. This ensures efficient inference while still providing great modeling flexibility. In experiments on real-world domains, our algorithm outperformed standard Bayesian network learning on both accuracy of query answers and speed of inference.

Table 5.2: Subroutine that updates an arithmetic circuit C by splitting distribution D on variable V.

```
function SplitAC($C, S(D, V)$)
let $M$ be the set of mutual ancestors of $D$ and $V$
let $N$ be the set of nodes between $M$ and $V$ or $D$
for $i \in$ Domain($V$) do
  create new parameter nodes $d_{ij}$
  $N_i \leftarrow$ copy of all nodes in $N$
  for each $n \in N$ do
    let $n_i$ be the copy of $n$ in $N_i$
    for each child $c$ of $n$ do
      if $c = v_i$ or $c$ is inconsistent with $v_i$ then
        skip
      else if $c$ is some parameter node $d_j$ then
        insert edge from $n_i$ to $d_{ij}$
      else if $c \in N$ then
        let $c_i$ be the copy of $c$ in $N_i$
        insert edge from $n_i$ to $c_i$
      else
        insert edge from $n_i$ to $c$
      end if
    end for
  end for
end for
for $m \in M$ do
  let $n_V$ be the child of $m$ that is a $V$-ancestor
  let $n_D$ be the child of $m$ that is a $D$-ancestor
  for $i \in$ Domain($V$) do
    let $n'_V$ be the copy of $n_V$ in $N_i$
    let $n'_D$ be the copy of $n_D$ in $N_i$
    create $n_{\times_i} := v_i \times n'_V \times n'_D$
  end for
  create $n_+ := \sum_i n_{\times_i}$
  replace $m$'s children $n_V$ and $n_D$ with $n_+$
end for
delete unreachable nodes, including all $d_j$
```

Table 5.3: Summary of experimental datasets.

Domain	Vars.	Train Exs.	Test Exs.	Density
KDD Cup	65	199,999	34,955	0.0079
MSWeb	294	32,711	5,000	0.0102
EachMovie	500	6,117	591	0.0581

Table 5.4: Summary of Learned Models

KDD Cup	AC-Greedy	AC-Quick	WinMine
Log-likelih.	-2.16	-2.16	-2.16
Edges	382K	365K	
Leaves	4574	4463	2267
Avg. parents	13.2	13.0	16.3
Max. parents	37	36	35
Treewidth	38	38	53
c2d edges	>18.2M	3664k	>39.5M
Time	50h	3h	3m

MSWeb	AC-Greedy	AC-Quick	WinMine
Log-likelih.	-9.85	-9.85	-9.69
Edges	204K	256K	
Leaves	1353	1870	1710
Avg. parents	2.5	3.1	5.2
Max. parents	114	127	94
Treewidth	114	127	118
c2d edges	>23.5M	>44.6M	>63.5M
Time	8h	3h	2m

EachMovie	AC-Greedy	AC-Quick	WinMine
Log-likelih.	-55.7	-54.9	-53.7
Edges	155K	372K	
Leaves	4070	6521	4830
Avg. parents	5.0	6.5	8.0
Max. parents	13	17	27
Treewidth	35	54	281
c2d edges	207k	855k	>27.3M
Time	>72h[5]	22h	3m

Table 5.5: Average inference time per query.

Algorithm	KDD Cup	MSWeb	EachMovie
AC-Greedy	194ms	91ms	62ms
AC-Quick	198ms	115ms	162ms
Gibbs-Fast	1.46s	1.89s	7.22s
Gibbs-Medium	11.3s	15.6s	42.5s
Gibbs-Slow	106s	154s	452s
Gibbs-VerySlow	1124s	1556s	3912s

Chapter 6

VARIATIONAL ARITHMETIC CIRCUITS

Chapter 7

CONCLUSION

The contributions of this dissertation are as follows.

- We introduced RRFs, one of the most powerful statistical relational representation to date. We showed how they resolve inconsistencies in the MLN representation and allow for probabilistic disjunctions, existentials, and nested formulas. More flexible representations such as RRFs will be key to solving human-level AI and other hard problems.

- We developed algorithms that vastly improve MLN weight learning, making Markov logic a much more practical and effective modeling language.

- We presented two new methods for efficient inference using arithmetic circuits. In the first method, we learn an efficient model from data by using inference cost as a learning bias. In the second, we use an arithmetic circuit to approximate an existing model.

These methods advance the state-of-the-art in representation, learning, and inference. However, much remains to be done. In addition to extending and improving to the specific methods, we would like to combine them.

The greatest challenge facing recursive random fields is weight learning. Current methods optimize pseudo-likelihood, as done by the earliest MLN weight learning methods. Since optimizing conditional likelihood works much better for MLNs, we expect similar benefits for RRFs. Therefore, we would like to extend our MLN weight learning methods to support RRFs as well. RRFs face the additional challenges of local optima (some of which could be very bad) and slower inference, since the multi-layered formulation requires more computation.

MLNs could benefit from more efficient inference as well. In addition to producing answers from learned models, inference is used in each step of weight learning for computing the gradient. The inference methods discussed int his dissertation work with Bayesian networks. A natural next step is to extend them to Markov networks, and eventually to MLNs and RRFs.

For learning arithmetic circuits, the difference between Markov networks and Bayesian networks is minimal. We can perform a structure search over Markov networks in much the same way that we search over Bayesian networks, except that the maximum likelihood and MAP parameter estimates no longer have closed form solutions. But since our models always permit exact inference, it should still be possible to compute these parameters efficiently. Extending this work to relational representations such as MLNs and RRFs poses a greater challenge. This is because MLN formulas are instantiated for each set of objects in the domain. Even relatively simple formulas, such as transitivity, lead to intractable exact inference. Bayesian networks do not have this problem, since all structure is "local" – adding an edge in a BN only directly affects a single factor. Therefore, any extension to relational models would be restricted to very simple structures (e.g., trees), only approximate the distribution with a circuit, or do something else to manage the complexity.

Using approximate circuits seems like the best bet for applying arithmetic circuits to relational representations. Extending our work on variational arithmetic circuits to relational models faces several challenges. First, since our methods begin by generating samples, they cannot be directly applied to representations based on Markov networks. A possible solution is to generate the samples using MCMC. Additional some corrections might be necessary in order to correct for the correlations between samples. Once we had a general approximate inference method for Markov networks, it could be applied directly to MLNs or RRFs. The final step would be to find additional ways to exploit MLN or RRF structure.

BIBLIOGRAPHY

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Periklis Andritsos, Ariel Fuxman, and Renée J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, page 30, 2006.

[3] S. Arnborg, D. W. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algebraic and Discrete Methods*, 8:277–284, 1987.

[4] S. Becker and Y. Le Cun. Improving the convergence of back-propagation learning with second order methods. In *Proceedings of the 1988 Connectionist Models Summer School*, pages 29–37, San Mateo, CA, 1989. Morgan Kaufmann.

[5] J. Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society, Series B*, 48:259–302, 1986.

[6] C. Blake and C. J. Merz. UCI repository of machine learning databases. Machine-readable data repository, Department of Information and Computer Science, University of California at Irvine, Irvine, CA, 2000. http://www.ics.uci.edu/∼mlearn/-MLRepository.html.

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

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

[9] D. Chickering, D. Heckerman, and C. Meek. A Bayesian approach to learning bayesian networks with local structure. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 80–89, Providence, RI, 1997. Morgan Kaufmann.

[10] D. M. Chickering. The WinMine toolkit. Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA, 2002.

[11] M. Collins. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on*

Empirical Methods in Natural Language Processing, pages 1–8, Philadelphia, PA, 2002. ACL.

[12] M. Craven and S. Slattery. Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, 43(1/2):97–119, 2001.

[13] A. Darwiche. A logical approach to factoring belief networks. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning*, pages 409–420, Toulouse, France, 2002.

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

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

[16] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for AI.* Morgan & Claypool, San Rafael, CA, 2009.

[17] R. Fletcher. *Practical Methods of Optimization.* Wiley-Interscience, New York, NY, second edition, 1987.

[18] N. Friedman and M. Goldszmidt. Learning Bayesian networks with local structure. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 252–262, Portland, OR, 1996. Morgan Kaufmann.

[19] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice.* Chapman and Hall, London, UK, 1996.

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

[21] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[22] G. Hulten and P. Domingos. Mining complex models from arbitrarily large databases in constant time. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 525–531, Edmonton, Canada, 2002. ACM Press.

[23] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658, New York, NY, USA, 2004. ACM Press.

[24] M. Jaeger, J. Nielsen, and T. Silander. Learning probabilistic decision graphs. *International Journal of Approximate Reasoning*, 42:84–100, 2006.

[25] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In D. Gu, J. Du, and P. Pardalos, editors, *The Satisfiability Problem: Theory and Applications*, pages 573–586. American Mathematical Society, New York, NY, 1997.

[26] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

[27] S. Kok and P. Domingos. Learning the structure of Markov logic networks. In *Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 441–448, Bonn, Germany, 2005. ACM Press.

[28] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2007. http://alchemy.cs.washington.edu.

[29] M. Lippi and P. Frasconi. Markov logic improves protein β-partners prediction. In *Proceedings of the Sixth International Workshop on Mining and Learning with Graphs*, Helsinki, Finland, 2008.

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

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

[32] M. Meila and M. Jordan. Learning with mixtures of trees. *Journal of Machine Learning Research*, 1:1–48, 2000.

[33] M. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[34] J. Nocedal and S. Wright. *Numerical Optimization.* Springer, New York, NY, 2006.

[35] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Francisco, CA, 1988.

[36] B. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 1994.

[37] H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 458–463, Boston, MA, 2006. AAAI Press.

[38] H. Poon and P. Domingos. Joint unsupervised coreference resolution with Markov logic. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 650–659, Honolulu, HI, 2008. ACL.

[39] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[40] S. Riedel and E. Klein. Genic interaction extraction with semantic and syntactic chains. In *Proceedings of the Fourth Workshop on Learning Language in Logic*, pages 69–74, Bonn, Germany, 2005. IMLS.

[41] S. Riedel and I. Meza-Ruiz. Collective semantic role labelling with Markov logic. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 188–192, Honolulu, HI, 2008. ACL.

[42] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.

[43] F. Sha and F. Pereira. Shallow parsing with conditional random fields. In *Proceedings of the 2003 Human Language Technology Conference and North American Chapter of the Association for Computational Linguistics*, pages 134–141. Association for Computational Linguistics, 2003.

[44] J. Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, 1994.

[45] P. Singla and P. Domingos. Discriminative training of Markov logic networks. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 868–873, Pittsburgh, PA, 2005. AAAI Press.

[46] P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 488–493, Boston, MA, 2006. AAAI Press.

[47] N. Srebro. Maximum likelihood Markov networks: An algorithmic approach. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2000.

[48] G. G. Towell and J. W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70:119–165, 1994.

[49] S. Vishwanathan, N. Schraudolph, M. Schmidt, and K. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In *Proceedings of the Twenty-Third International Conference on Machine Learning*, Pittsburgh, PA, 2006.

[50] J. Wang and P. Domingos. Hybrid Markov logic networks. In *Proceedings of the Twenty-Third National Conference on Artificial Intelligence*, pages 1106–1111, Chicago, IL, 2008. AAAI Press.