Learning Markov Networks With Arithmetic Circuits

Daniel Lowd and **Amirmohammad Rooshenas**
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403
{lowd,pedram}@cs.uoregon.edu

Abstract

Markov networks are an effective way to represent complex probability distributions. However, learning their structure and parameters or using them to answer queries is typically intractable. One approach to making learning and inference tractable is to use approximations, such as pseudo-likelihood or approximate inference. An alternate approach is to use a restricted class of models where exact inference is always efficient. Previous work has explored low treewidth models, models with tree-structured features, and latent variable models. In this paper, we introduce ACMN, the first ever method for learning efficient Markov networks with arbitrary conjunctive features. The secret to ACMN's greater flexibility is its use of arithmetic circuits, a linear-time inference representation that can handle many high treewidth models by exploiting local structure. ACMN uses the size of the corresponding arithmetic circuit as a learning bias, allowing it to trade off accuracy and inference complexity. In experiments on 12 standard datasets, the tractable models learned by ACMN are more accurate than both tractable models learned by other algorithms and approximate inference in intractable models.

1 INTRODUCTION

Markov networks (MNs) are one of the most effective ways to compactly represent a complex probability distribution over a set of random variables. Unfortunately, answering marginal or conditional queries in an MN is #P-complete in general [25]. Learning MN parameters and structure is also intractable in the general case, since computing the gradient of the log-likelihood requires running inference in the model.

As a result, most applications of MNs use approximate methods for learning and inference. For example, parameter and structure learning are often done by optimizing pseudo-likelihood instead of log-likelihood, or by using approximate inference to compute gradients. Many approximate inference algorithms have been developed, but, depending on the problem and the algorithm, the approximation may be inaccurate or unacceptably slow.

The key to making MNs more useful is to make exact inference efficient. Even though inference is #P-complete in the worst case, there are many interesting special cases where exact inference remains tractable. Previous work has investigated methods for learning MNs with low treewidth [1, 12, 6], which is a sufficient condition for efficient inference, but not a necessary one. Another approach is to learn a tree of features [14], which may have high treewidth but still admits efficient inference. However, this approach leads to many very long features, with lengths proportional to the depth of the tree.

Another method for learning tractable graphical models is to use mixture models with latent variables. The simplest example is a latent class model [18], in which the variables are conditionally independent given a single latent variable. Other examples include mixtures of trees [22] and latent tree models [28, 8]. Sum-product networks that use a carefully structured network of latent variables have been very successful at certain computer vision applications [23]. Latent variable models excel when there are natural clusters present in the domain, but may do worse when such structure is not present. Another limitation of latent variable models is that they cannot efficiently compute the most likely configuration of the observable variables conditioned on evidence (the MPE state), since summing out the latent variables makes the maximization problem hard.

In this paper, we propose ACMN, a new method for learn-

Appearing in Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS) 2013, Scottsdale, AZ, USA. Volume 31 of JMLR: W&CP 31. Copyright 2013 by the authors.

ing the structure and parameters of tractable MNs over discrete variables. Our method represents the network structure as a set of conjunctive features, each of which is a logical rule that evaluates to 1 if the specified variables take on their given values and 0 otherwise. Unlike previous work, there are neither latent variables nor explicit restrictions on the treewidth or structure of these features, as long as they admit a model with efficient inference.

To ensure efficient inference, ACMN simultaneously learns an arithmetic circuit (AC) that encodes the same distribution as the MN. An AC is a compact representation with linear time inference. ACs are similar to junction trees, but can be exponentially more compact by exploiting local structure or determinism. Thus, as long as the AC is relatively small, inference can be done quickly in the MN. ACMN exploits this directly by performing a greedy search in the space of possible structures, using the size of the AC as a learning bias.

ACMN is similar to the LearnAC algorithm [19], except that it learns an MN rather than a Bayesian network. Bayesian networks are a less flexible representation than MN, since every probability distribution that can be encoded as a compact Bayesian network can also be encoded as an MN, but the converse is not true. The disadvantage of MNs is that the likelihood is no longer node decomposable and parameter estimation can no longer be done in closed form. ACMN overcomes these challenges with intelligent heuristics to minimize the cost of scoring candidate moves. Even so, ACMN is more computationally expensive than LearnAC, but offers the benefits of a more flexible representation and thus more accurate models.

The rest of the paper is organized as follows. In Sections 2 and 3, we present additional background on MNs and ACs. In Section 4, we present the details of ACMN. We compare ACMN empirically to a variety of baseline algorithms in Section 5, and conclude in Section 6.

2 MARKOV NETWORKS

Consider a set of n random variables, $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$. We focus on the case where each X_i is discrete: $Val(X_i) = \{x_i^1, x_i^2, \ldots, x_i^k\}$. For the special case of Boolean-valued variables ($k = 2$), we refer to the two states as x_i (X_i is true) and $\neg x_i$ (X_i is false). For simplicity, we will assume that X_i is Boolean, but our methods can be generalized to multi-valued variables as well.

A *Markov network* (MN) represents a probability distribution as a normalized product of factors:

$$P(\mathcal{X}) = \frac{1}{Z} \prod_c \phi_c(D_c)$$

where each ϕ_c is a non-negative, real-valued function, $D_c \subset \mathcal{X}$ represents the variables in the domain of factor c, and Z is a normalization constant called the *partition function*. If all probabilities are positive, the distribution can be represented as an equivalent log-linear model:

$$\log P(\mathcal{X}) = \sum_i w_i f_i(D_i) - \log Z$$

where each f_i is a real-valued feature function with domain D_i and w_i is a real-valued weight. A common special case is where each f_i is a logical conjunction of variable tests that evaluates to 1 if the expression is satisfied and 0 otherwise. For example: $f_1(X_1, X_3, X_8) = x_1 \wedge \neg x_3 \wedge x_8$. Compared to a table-based representation, conjunctive features can be much more compact when the tables have repeated values or other kinds of local structure.

MNs can be used to answer probabilistic queries such as the marginal or joint probabilities of a set of query variables given the values of some evidence variables. In some cases, such as when the MN has low treewidth or other special structure, this can be done exactly. In general, however, computing exact probabilities is usually intractable, so approximate inference techniques are used instead. One of the simplest and most widely used inference methods is Gibbs sampling, which repeatedly samples each variable given the current state of all other variables in the network. The probability of a query is then estimated as the fraction of the samples that satisfy the query. For positive distributions, the sample distribution will eventually converge to the true network distribution, although convergence can be very slow in practice.

MN parameters are typically learned to maximize the penalized log-likelihood or pseudo-log-likelihood (PLL) of the training data. The most common regularization penalties are an L_1 or L_2 norm of the weights, which is equivalent to placing a Laplacian or Gaussian prior on each parameter. When the training data is fully observed, both objective functions are convex. Computing the log-likelihood or its gradient requires performing exact or approximate inference in the current model, which may be slow or inaccurate in the general case. A compelling alternative is pseudo-likelihood [2], which is the product of the conditional probability of each variable given the others. Pseudo-likelihood is a popular choice because it is a consistent estimator and can be computed efficiently. Models trained using pseudo-likelihood tend to do well on queries with large amounts of evidence, since this closely mirrors the objective function, but worse on queries with less evidence.

2.1 Learning Markov Network Structure

Markov network structure can be learned to maximize an objective function such as penalized log-likelihood or PLL. Combinatorial search approaches perform a direct search through the space of possible conjunctive features, scoring

candidate features using approximate inference or pseudo-likelihood. For top-down search, the initial state is all single-variable features ($\{x_1, \neg x_1, x_2, \neg x_2, \ldots\}$) and the search operations are adding a new feature that is the conjunction of two existing features [11, 21]. For bottom-up search, the initial state contains one very specific feature for each instance, and the search operations are merging and simplifying features to make them more general [10]. The recent GSSL algorithm improves on bottom-up search by replacing the expensive searching and scoring procedure with randomness and heuristics, allowing it to learn better models in less time [15].

Another approach to structure learning is to learn a local model to predict each variable given all others and then combine them into a single joint model. Ravikumar et al. [24] use L_1-regularized logistic regression to learn the local models, and then construct a global model with a feature for each pair of variables that had a non-zero interaction weight in the logistic regression. The DTSL algorithm [17] uses probability trees as the local models and constructs the global model by creating a conjunctive feature for each leaf in each probability tree. The weights in the global model can be learned by optimizing PLL or by directly translating the conditional distributions with the DN2MN method [20]. Instead of learning local models first, optimizing PLL with L_1 regularization can also be used to select features directly, as long as the features are enumerated in advance [16] or restricted to a certain hierarchical structure [26].

A different class of MN structure learning algorithms attempts to directly identify the independencies in the model [27, 3]. These methods tend to place more emphasis on discovering the true structure of the domain, and less emphasis on simply learning an accurate distribution.

3 ARITHMETIC CIRCUITS

The *network polynomial* for a Bayesian or Markov network is a polynomial with an exponential number of terms, one for each possible state of the random variables [9]. Each term is a product of indicator variables (λ_{x_i}) for the states of the random variables and the parameters (θ_j) of all features satisfied by that state. For example, consider a Markov network over Boolean variables X_1 and X_2 with features $f_1 = x_1 \wedge x_2$ and $f_2 = x_2$ and their weights w_1 and w_2. Since the weights are in log-space, we must define an alternate parameterization, $\theta_1 = e^{w_1}$ and $\theta_2 = e^{w_2}$. Now we can construct the network polynomial, which is multilinear in the λ and θ variables:

$$\lambda_{x_1}\lambda_{x_2}\theta_1\theta_2 + \lambda_{x_1}\lambda_{\neg x_2} + \lambda_{\neg x_1}\lambda_{x_2}\theta_2 + \lambda_{\neg x_1}\lambda_{\neg x_2}$$

When all indicator variables λ are set to 1, the network polynomial computes the partition function of the MN. The

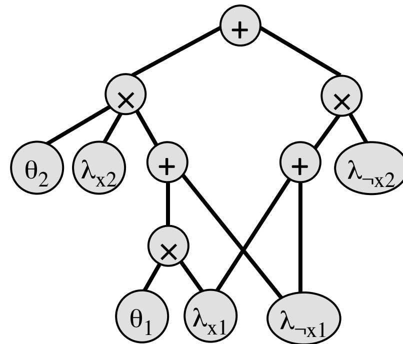

Figure 1: Simple arithmetic circuit that encodes the network polynomial of a Markov network with two variables and two features.

network can be conditioned on evidence by setting the appropriate indicator variables to zero. For example, conditioning the network on $X_1 = \neg x_1$ can be done by setting λ_{x_1} to zero, so that all terms involving λ_{x_1} evaluate to zero. Marginals of variables and features can also be computed by differentiating the network polynomial. See Darwiche [9] for more details.

Since the network polynomial has exponential size, working with it directly is intractable. However, in some cases, it can be represented compactly as an arithmetic circuit [9]. An *arithmetic circuit* (AC) is a rooted, directed acyclic graph in leaves contain numerical values, such as parameters or indicator variables, and interior nodes are addition and multiplication operations. ACs exploit structure by representing repeated computations just once in the graph and referencing them multiple times. See Figure 1 for an example AC that encodes the example two-variable Markov network from above. Evaluating or differentiating the AC with or without evidence can be done in linear time in the size of the circuit. Therefore, we can perform efficient inference in any MN if we have a compact representation of it as an AC.

ACs are very closely related to sum-product networks (SPNs) [23]. In fact, every AC can be compactly represented as an SPN and vice versa. The key difference is that SPNs attach weights to the outgoing edges of sum nodes, while ACs represent the same operation using additional product and parameter nodes. The specific structures used by Poon and Domingos involved a complex arrangement of implicit latent variables, while we focus on learning MNs with no latent variables.

3.1 Learning Arithmetic Circuits

Most previous work with ACs has focused on compiling Bayesian networks (BNs) to compact circuits [4, 5]. However, for many networks, no compact circuit exists. An alternate approach is to learn a Bayesian network and circuit

simultaneously, as done by the LearnAC algorithm [19]. The LearnAC algorithm greedily learns a Bayesian network with context-specific independence, similar to the method of Chickering et al. [7], but using the size of the corresponding AC as a learning bias. Rather than compiling the AC from scratch each time, it evaluates candidate structure modifications by performing equivalent modifications to the AC. Since LearnAC is still learning a BN from fully observed data, all parameters can be estimated in closed form, and the likelihood component of the score function is easy to compute.

4 THE ACMN ALGORITHM

We now describe ACMN, our proposed method for learning an MN with conjunctive features and its corresponding compact AC.

ACMN performs a greedy search through structure space, similar to the methods of Della Pietra et al. [11] and McCallum [21]. The initial structure is the set of all single-variable features. The search operations are to take an existing feature in the model, f, and combine it with another variable, V, creating two new features: $f \wedge v$ and $f \wedge \neg v$.[1] We refer to this operation as a "split," since it takes an existing feature and splits it into three: the original feature and two new ones that condition on the value of V.

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

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

Here, Δ_{ll} is a measure of how much the split will increase the log-likelihood. Measuring the exact effect would require jointly optimizing all model parameters along with the parameters for the two new features. To make split scoring more efficient, we measure this as the log-likelihood gain from modifying only the weights of the two new features, keeping all others fixed. This gives a lower bound on the actual log-likelihood gain. This gain can be computed by solving a simple two-dimensional convex optimization problem, which depends only on the empirical counts of the new features in the data and their expected counts in the model, requiring performing inference just once to compute these expectations. A similar technique was used by Della Pietra et al. [11] and McCallum [21] for efficiently computing feature gains.

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

ACMN is similar to the LearnAC [19], which also starts with a product of marginals and repeatedly select the highest scoring structure operation until convergence. The difference is that instead of the probability distribution being a BN where each conditional probability distribution (CPD) is a tree, the distribution is a log-linear model where each feature is a conjunction of feature tests. Every BN with tree CPDs can easily be expressed as a set of conjunctive features, but the converse is not true. Therefore, ACMN should be more expressive than LearnAC and able to learn better models.

4.1 The Overall Algorithm

ACMN makes one additional approximation that leads to a much faster implementation. ACMN assumes that, as learning progresses, the score of any given split decreases monotonically. The score of a split can decrease for two reasons. First, a split's likelihood gain $\Delta_{ll}(s)$ may decrease as other similar splits are performed, making s increasingly redundant. Second, as the circuit grows in size, the edge costs typically increase, since there are more edges that may need to be duplicated when performing a split. While this assumption does not always hold, it allows us to evaluate only a small fraction of the available splits in each iteration, rather than rescoring every single one.

A high-level view of our algorithm is shown in Algorithm 1. This simple description assumes that every split is rescored in every iteration. To achieve reasonable running times, our actual implementation of ACMN uses a priority queue which ranks splits based on their most recently computed score. The split at the front of the queue is therefore the most promising candidate. We repeatedly remove the split from the front of the queue and recompute its likelihood gain or edge gain if either is out of date. Since computing likelihood gain is usually cheaper, we compute it first and reinsert the split into the priority queue with the updated score, since a bad likelihood may be enough to rule it out. If both gains are up-to-date, then the split is better than any other split in the priority queue, as long as we assume that the scores for other splits in the queue have not increased since they were inserted.

One final optimization is that we can compute many of the expectations we need in parallel using the AC. Specifically, by conditioning on feature f and differentiating the circuit with respect to the indicator variables, we can compute the expectations $E[f \wedge x_i]$ and $E[f \wedge \neg x_i]$ for all variables X_i *in a single pass*, which takes linear time in the size of the circuit. In other words, the time to estimate the likelihood gain for all splits of a single feature can be done in $O(e)$ time rather than $O(ne)$ time, where e is the number of

[1] For convenience and clarity of exposition, we assume binary-valued variables, but our method also applies to multi-valued variables with only minor modifications.

Algorithm 1 Greedy algorithm for learning MN ACs.

```
function ACMN(T)
  initialize model M and circuit C as product of marginals
  initialize priority queue Q with initial candidate splits
  loop
    Update edge and likelihood gain for each split in Q.
    s ← Q.pop()        // Select best split
    (M, C, f, θ, f′, θ′) ← ACMN-Split(s, M, C)
    (M, C) ← OptimizeWeights(M, C, T)
    for V ∈ 𝒳 do
      Add new splits (f, θ, V) and (f′, θ′, V) to Q.
    end for
  end loop
  return (M, C)
```

Algorithm 2 Subroutine that updates an arithmetic circuit C by adding two new features, $g = f \wedge v$ and $g' = f \wedge \neg v$.

```
function ACMN-Split(s, M, C)
  Let θ = s.paramNode, V = s.varNodes
  Let A be the mutual ancestors of the parameter node (θ)
  and the variable nodes (λ_v, λ_¬v).
  Let G_θ be the subcircuit between θ and A.
  Let G_{v,¬v} be the subcircuit between {λ_v, λ_¬v} and A.
  A ← mutual ancestors of θ and V
  G_v ← Clone(G_{v,¬v})[0/λ_¬v]
  G_¬v ← Clone(G_{v,¬v})[0/λ_v]
  G_θ1 ← Clone(G_θ)[Prod(θ_1, θ)/θ]
  G_θ2 ← Clone(G_θ)[Prod(θ_2, θ)/θ]
  A′ ← Sum(Prod(λ_v, G_v, G_θ1), Prod(λ_¬v, G_¬v, G_θ2))
  Let g = f ∧ v, g′ = f ∧ ¬v
  return (M ∪ {g, g′}, C[A′/A], g, θ_1, g′, θ_2)
```

edges in the circuit. We can use this same technique when recomputing likelihood gains, by caching the expectations for all of a feature's splits when we compute the first one.

As described, ACMN uses a scoring function with a fixed, user-specified parameter of γ. In order to meet particular efficiency goals, we may wish to place a limit on the total number of edges in the circuit instead. To do this, we run the ACMN algorithm with a very large initial value of γ. If no splits remain with positive score and the edge budget has not been exhausted, then we divide γ by 2 and continue learning. This can be repeated with smaller and smaller values of γ until the edge budget has been used up. This is a conservative approach that tends to select low-cost splits early on, in order to avoid exhausting the edge budget too quickly.

4.2 Updating the Circuit

One of the key subroutines in ACMN is ACMN-Split, which updates an AC without recompiling it from scratch. (A very similar procedure is also used for ComputeEdgeGain, which computes exactly how many edges ACMN-Split would add.) Given a circuit C that is equivalent to an MN M and a valid split s, SplitAC returns a modified circuit C' that is equivalent to M after applying split s, along with the new features and parameters.

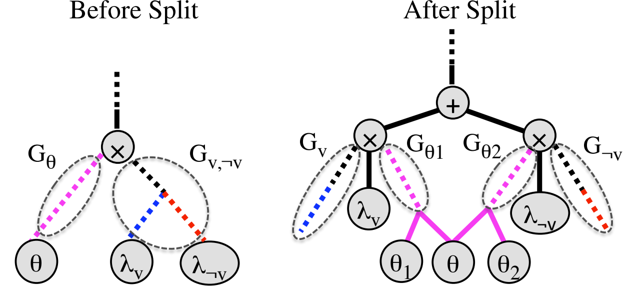

Figure 2: Illustration of the operation of the ACMN-Split subroutine, splitting a feature with parameter node θ on variable V. Dashed lines indicate sections of the circuit where details have been omitted.

Pseudocode is present in Algorithm 2, followed by an illustration of the basic operation in Figure 2. To clearly explain how it works, we must first define some additional terminology. In an AC, the *mutual ancestors* of two sets of nodes N and M are the nodes that are ancestors of at least one node in each set, and that have no children that are mutual ancestors of N and M. The *subcircuit between N and M* consists of all nodes in the circuit that are ancestors of a node in N and descendants of a node in M. The *Clone* function creates a copy of a subcircuit. If two nodes in the original subcircuit are connected by an arc, then their clones in the new subcircuit are connected by an arc as well. For external links, if a node in the original subcircuit has a child outside of the subcircuit, then its clone will have an arc to the exact same child node. We also define *(sub)circuit substitution* syntax as follows: $C[n'/n]$ represents a new circuit where all nodes that had node n as a child now have n' as a child instead. Finally, the functions Sum and Prod construct new addition and multiplication nodes, respectively.

To split a feature f on a variable V, we must introduce two new parameters for the new features, so that all terms in the network polynomial that satisfy one of the new features will include the appropriate parameter. In other words, whenever f is satisfied and $V = v$, we must multiply by both θ, the parameter for f, and θ_1, the parameter for the new feature $f \wedge v$. For the AC to be consistent, we must "sum out" V only once. The logical place to do this is at the mutual ancestors of the parameter node θ and the variable nodes $\{\lambda_v, \lambda_{\neg v}\}$. This allows us to condition θ on V, without invalidating the existing portions of the AC that already depend on V.

5 EXPERIMENTS

5.1 Datasets

We used 12 binary variable datasets, listed in Table 1 in increasing order by number of variables. These datasets have been used by several previous papers on MN structure learning [10, 17, 15]. Each dataset consists of three parts: a training set, which we used as input to each learning algorithm; a validation set, which we used to select the best tuning parameters; and a test set, which we used for evaluation.

Table 1: Datasets characteristics

Dataset	Train set	Valid. set	Test set	Num. vars.	Density
NLTCS	16,181	2,157	3,236	16	0.332
MSNBC	291,326	38,843	58,265	17	0.166
KDDCup 2000	180,092	19,907	34,955	64	0.008
Plants	17,412	2,321	3,482	69	0.180
Audio	15,000	2,000	3,000	100	0.198
Jester	9,000	1,000	4,116	100	0.610
Netflix	15,000	2,000	3,000	100	0.541
MSWeb	29,441	3,270	5,000	294	0.010
Book	8,700	1,159	1,739	500	0.016
WebKB	2,803	558	838	839	0.063
Reuters-52	6,532	1,028	1,540	889	0.037
20 Newsgroups	11,293	3,764	3.764	910	0.049

5.2 Methods

To evaluate the accuracy of ACMN, we compared it to four state-of-the-art algorithms, two for learning Markov networks and two for learning other forms of tractable graphical models. Our MN baselines are GSSL [15] and L_1-regularized logistic regression (L1) [24], which have shown good performance on these datasets in previous work. Our two tractable baselines are a recent method for learning latent tree models (LTM) [8] and the LearnAC algorithm [19]. We refer to LearnAC as ACBN since, like ACMN, it learns an AC and graphical model through greedy combinatorial search, but it searches through BN structures rather than MNs. The objective function of GSSL and L1 is pseudo log-likelihood while ACMN, ACBN, and LTM optimize log-likelihood.[2]

For all baseline methods, we used publicly available code and replicated recommended tuning procedures as closely as possible. For the tractable models (ACMN, ACBN, LTM) all options and parameters were tuned to maximize log-likelihood on the validation set; for GSSL and L1, the pseudo-likelihood of the validation set was used instead.

For GSSL, we tried generated feature sizes of 0.5, 1, 2, and 5 million; pruning thresholds of 1, 5, and 10; Gaussian standard deviations of 0.1, 0.5, and 1; and L_1 priors of 1, 5, and 10, for a total of 108 configurations. We used the original authors' implementation of GSSL [3] to select the features and the Libra toolkit[4] to learn weights, as done in the original paper [15]. To learn MNs using L1, we used the Libra toolkit to learn the logistic regression distributions and convert them to an MN with DN2MN [20]. We used L_1 prior values of 0.1, 0.5, 1, 2, 5, 10, 15, and 20. For ACBN, we used *aclearnstruct* from the Libra toolkit with split penalties of 1, 5, and 10 and 0.1, 0.5, 1, and 2 million maximum edges, resulting in 12 different configurations. We used the same set of configurations for ACMN as well. For LTM, we ran the authors' code[5] with its default EM configuration to create latent tree models using four different methods that they provided: CLRG, CLNJ, regCLRG and regCLNJ.

Table 2: Log-likelihood comparison

Dataset	ACMN	ACBN	LTM
NLTCS	**-6.01**	-6.02	-6.49
MSNBC	**-6.04**	**-6.04**	-6.52
KDDCup 2000	**-2.15**	-2.16	-2.18
Plants	-12.89	**-12.85**	-16.39
Audio	**-40.32**	-41.13	-41.90
Jester	**-53.35**	-54.43	-55.17
Netflix	**-57.26**	-57.75	-58.53
MSWeb	**-9.77**	-9.81	-10.21
Book	-35.62	-36.02	**-34.22**
WebKB	-161.30	-159.85	**-156.84**
Reuters-52	-89.54	**-89.27**	-91.23
20 Newsgroup	-159.56	-159.65	**-156.77**

To evaluate the effectiveness of each method at answering queries, we used the test set to generate probabilistic queries with varying amounts of evidence, ranging from 90% to 10% of the variables in the domain. The evidence variables were randomly selected separately for each test query. All non-evidence variables were query variables. For LTM, ACBN, and ACMN, we computed the exact conditional log-likelihood (CLL) of the query variables given the evidence ($\log P(X = x|E = e)$). For L1 and GSSL, we computed the conditional marginal log-likelihood (CMLL) instead, a popular alternative when rare joint probabilities are hard to estimate [16, 10, 17, 15]. CMLL is similar to CLL, but the log-likelihood of the query variables is computed using their conditional marginals rather than their joint probability: $\sum_i \log P(X_i = x_i|E = e)$. Marginals were computed by running Gibbs sampling with 100 burn-in and 1000 sampling iterations; results using belief propagation were similar. To make the results with different amounts of evidence more comparable, we divided the CLL and CMLL by the number of query variables to obtain normalized CLL

[2]Other natural baselines would be LEM [14] and SPNs [23]. However, the LEM code is unavailable, due to a broken library dependency, and previous SPN learning methods assume a two-dimensional structure not present in these datasets.

[3]http://dtai.cs.kuleuven.be/ml/systems/gssl

[4]The open-source Libra toolkit is available online at http://libra.cs.uoregon.edu.

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

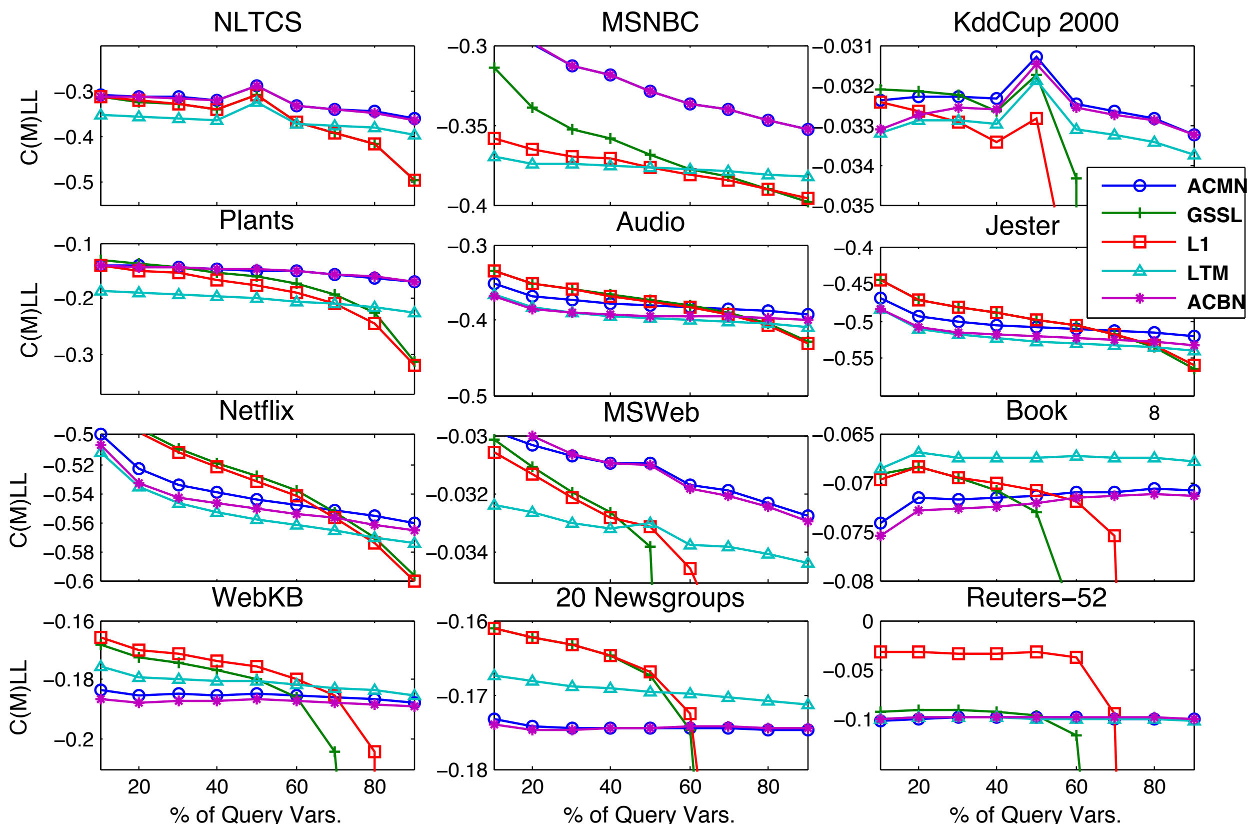

Figure 3: Normalized CLL and CMLL vs. fraction of query variables. CLL is reported for all tractable models (ACMN, LTM, ACBN) while CMLL is reported for the rest.

(NCLL) and normalized CMLL (NCMLL), respectively.

We ran all processes, including learning, tuning, and testing, on an *Intel(R) Xeon(R) CPU X5650@2.67GHz*.

5.3 Results

Table 2 shows the log-likelihoods of LTM, ACBN, and ACMN on each of the 12 datasets. (Computing log-likelihoods for GSSL or L1 is intractable.) ACMN is the most accurate algorithm on 6 of the 12 datasets, beating ACBN on 8 (plus 1 tie) and LTM on 9.

Since ACMN and ACBN often have very similar log-likelihoods, we wanted to determine whether or not their actual distributions were similar as well. We did this by generating samples from the ACBN models to estimate the KL divergence between the ACBN and ACMN models. Our results are in Table 3. The results demonstrate that the KL divergence between the two (KLD) is often much larger than their difference in log-likelihood on the test data (ΔLL), suggesting that their distributions are indeed different. The ACs learned by ACMN and ACBN also have different topological properties. For example, on MSNBC, the AC learned by ACBN has 167k nodes, 349k edges, and 5.4k features, while ACMN's has 952k nodes, 2.0 million edges, and 7.6k features.

Table 3: Comparison of BN and MN ACs.

Dataset	ΔLL	KLD
NLTCS	0.01	0.08
MSNBC	0.00	0.02
KDDCup 2000	0.01	0.08
Plants	0.04	1.43
Audio	0.81	2.46
Jester	1.08	3.14
Netflix	0.49	2.57
MSWeb	1.45	0.56
Book	0.09	3.73
WebKB	1.45	14.57
Reuters-52	0.27	6.98
20 Newsgroups	0.09	10.13

Figure 3 shows NCMLL and NCLL values for each dataset with different fractions of query variables. GSSL and L1 more or less follow the same trend in all datasets. They perform well when there are few query variables (and a lot of evidence), but their performance quickly degrades with more query variables and less evidence. This trend is consistent with the properties of optimizing the pseudo-

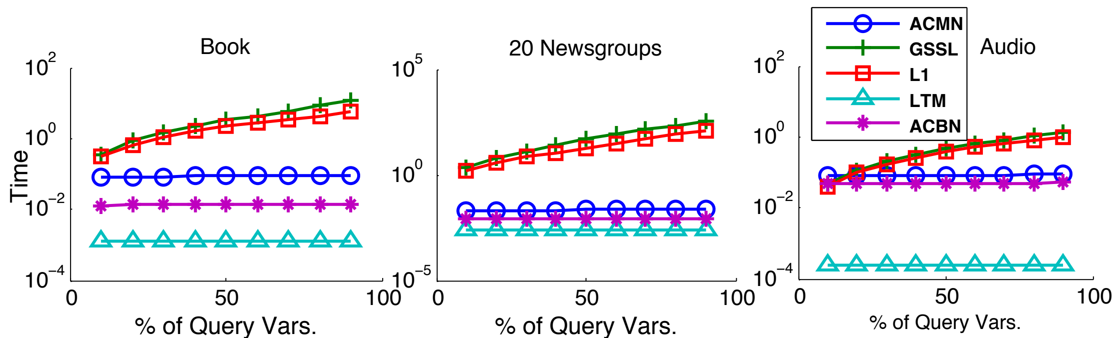

Figure 4: Query time for different percentages of query variables

likelihood objective, which is suitable for queries with a small number of variable conditioned on a large amount of evidence. In high dimensional datasets such as Reuters-52, WebKB, Book, and 20-Newsgroups, the very large number of query variables exacerbate the condition by preventing the Gibbs sampler from converging in the given number of iterations. Increasing the number of iterations might lead to improved performance, but Gibbs sampling is already quite slow on these domains, taking over 9 days for GSSL to compute the queries for 20 Newsgroups conditioned on 20 percent of the variables. When the queries are conditioned on only 10 percent of variables, the query time for the whole dataset goes up to 15 days.

ACMN, ACBN, and LTM, on the other hand, are less sensitive to the number of query variables, since they optimize log-likelihood and can perform exact inference. LTM shows better performance on 20-Newsgroups, WebKB, and Book, the same datasets where it has the largest log-likelihood. For the most part, ACMN dominates ACBN; in the few cases where ACBN has higher NCLL, the difference is very small.

Finally, we measured the query time of each method on each dataset, and show the result for three representative datasets in Figure 4. Note that the Y-axis is on a log-scale. Among these algorithms, LTM is considerably faster than the others because the LTM models can be represented as ACs with relatively few edges. For example, the LTM model for Book can be expressed as an AC with 1428 edges while the ACs learned with ACMN and ACBN each had over 1 million edges. The ACMN and ACBN inference times could be reduced somewhat by lowering the maximum number of allowed edges, although this would also reduce accuracy by a very small amount. Even so, the relatively large circuits selected by ACMN and ACBN are still more efficient than running Gibbs sampling in L1 or GSSL models, especially when there is less evidence. If Gibbs sampling were run for longer to obtain higher accuracy, then L1 and GSSL would be even further behind.

6 CONCLUSION

Overall, ACMN is less accurate than pseudo-likelihood based methods when there is a large amount of evidence available, but easily dominates them in both inference speed and accuracy when there is less evidence available. Compared to other tractable graphical models, ACMN is more accurate a majority of the time on these datasets. Therefore, ACMN is a excellent choice for applications that require reliable speed and accuracy with lesser amounts of evidence.

The biggest downside to ACMN is its high computational complexity. Even with our optimizations, ACMN remains significantly slower than the other learning algorithms. An important area for future work is to explore how to make it more efficient. One possibility is to restrict ACMN to a set of candidate features found by a separate, faster algorithm such as GSSL. By ruling out many features before they are scored, ACMN might be sped up significantly. Another option is to apply L1 regularization and prune features whose weights drop to zero. If learning could be made sufficiently efficient, then ACs could be used to learn conditional random fields (CRFs) as well. The challenge of CRFs is that inference must be done once for each evidence configuration in the training data, which could make learning orders of magnitude slower.

A final direction is learning ACs with latent variables. Ideally, this could give ACMN the advantages of LTM on datasets where clustering structure was present, while maintaining the flexibility of the unrestricted conjunctive feature representation.

Acknowledgments

This research was partly funded by ARO grant W911NF-08-1-0242, NSF grant IIS-1118050, and NSF grant OCI-0960354. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ARO, NSF, or the U.S. Government.

References

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

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

[3] F. Bromberg, D. Margaritis, and V. Honavar. Efficient Markov network structure discovery using independence tests. *Journal of Artificial Intelligence Research*, 35(2):449, 2009.

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

[5] M. Chavira and A. Darwiche. Compiling Bayesian networks using variable elimination. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2443–2449, 2007.

[6] 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.

[7] 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.

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

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

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

[11] 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.

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

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

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

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

[16] S.-I. Lee, V. Ganapathi, and D. Koller. Efficient structure learning of Markov networks using L1-regularization. In *Advances in Neural Information Processing Systems 19*, pages 817–824. MIT Press, 2007.

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

[18] 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.

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

[20] Daniel Lowd. Closed-form learning of Markov networks from dependency networks. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, Catalina Island, CA, 2012. AUAI Press.

[21] 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.

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

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

[24] P. Ravikumar, M. J. Wainwright, and J. Lafferty. High-dimensional ising model selection using L1-regularized logistic regression. *Annals of Statistics*, 2009.

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

[26] M. Schmidt and K. Murphy. Convex structure learning in log-linear models: Beyond pairwise potentials. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

[27] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Springer, New York, NY, 1993.

[28] Y. Wang, N. L. Zhang, and T. Chen. Latent tree models and approximate inference in Bayesian networks. *Journal of Artificial Intelligence Research*, 32:879–900, 2008.