Learning Markov Networks With Arithmetic Circuits

Daniel Lowd LOWD@CS.UOREGON.EDU
Amirmohammad Rooshenas PEDRAM@CS.UOREGON.EDU
Department of Computer and Information Science, University of Oregon, Eugene, OR 97403

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 (Roth, 1996). 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 (Bach & Jordan, 2001; Elidan & Gould, 2008; Chechetka & Guestrin, 2008), which is a sufficient condition for efficient inference, but not a necessary one. Another approach is to learn a tree of features (Gogate et al., 2010), 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 (Lowd & Domingos, 2005), in which the variables are conditionally independent given a single latent variable. Other examples include mixtures of trees (Meila & Jordan, 2000) and latent tree models (Wang et al., 2008; Choi et al., 2011). Sum-product networks that use a carefully structured network of latent variables have been very successful at certain computer vision applications (Poon & Domingos, 2011). 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 learning 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

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28. Copyright 2013 by the author(s).

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 (Lowd & Domingos, 2008), except that it learns an MN rather than a Bayesian network. Bayesian networks are a less flexible representation than MNs, 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 Section 2, we present additional background on ACs. In Section 3, we present the details of ACMN. We compare ACMN empirically to a variety of baseline algorithms in Section 4, and conclude in Section 5.

2. Arithmetic Circuits

Consider a set of n discrete random variables, $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$. For simplicity, we assume that each X_i is Boolean, with states x_i (meaning X_i is true) and $\neg x_i$ (X_i is false). However, these methods can be generalized to multi-valued variables as well. For positive distributions, a *Markov network* (MN) over these variables can be represented as a log-linear model, $\log P(\mathcal{X}) = \sum_i w_i f_i(D_i) - \log Z$, where Z is the partition function, each f_i is a real-valued feature function with domain $D_i \subset \mathcal{X}$, and w_i is a real-valued weight. We focus on the special case where each f_i is a logical conjunction of variable tests that evaluates to 1 if the expression is satisfied and 0 otherwise.

The *network polynomial* for an MN is a polynomial with an exponential number of terms, one for each possible state of the random variables (Darwiche, 2003). 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 an MN over X_1 and X_2 with two conjunctive features, $f_1 = x_1 \wedge x_2$ and $f_2 = x_2$. Since the weights are in log-space, we define $\theta_1 = e^{w_1}$ 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 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 (2003) 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 (Darwiche, 2003). An *arithmetic circuit* (AC) is a rooted, directed acyclic graph in which leaves contain numerical values, such as parameters or indicator variables, and interior nodes are addition and multiplication operations. 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) (Poon & Domingos, 2011). 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.

Lowd and Domingos (2008) demonstrated that an AC could be learned from data by adapting a Bayesian network structure learning algorithm to use the size of the corresponding circuit as a learning bias. Rather than compiling the AC from scratch each time, LearnAC evaluates candidate structure modifications by performing equivalent modifications to the AC. In the following section, we will show how this idea can be extended to learning arbitrary MNs, which is our main contribution in this paper.

3. 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. (1997) and McCallum (2003). The initial structure is the set of all single-variable features, which define a product of marginals distribution. 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 $f \wedge \neg v$. 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 instead use 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. (1997) and McCallum (2003) for efficiently computing feature gains.

$\Delta_e(s)$ denotes the number of edges that would be added to the AC if this split were included. Computing this has similar complexity to performing the split. γ gives 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.

3.1. 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.

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.
  $s \leftarrow Q$.pop()   *// Select best split*
  $(M, C, f, \theta, f', \theta') \leftarrow$ ACMN-Split(s, M, C)
  $(M, C) \leftarrow$ OptimizeWeights(M, C, T)
  for $V \in \mathcal{X}$ **do**
    Add new splits (f, θ, V) and (f', θ', V) to Q.
  end for
end loop
return (M, C)

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 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.

3.2. 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.

Pseudocode is present in Algorithm 2, followed by an illustration of the basic operation in Figure 1. 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

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, $\lambda_{\neg v}$).
Let G_θ be the subcircuit between θ and A.
Let $G_{v,\neg v}$ be the subcircuit between $\{\lambda_v, \lambda_{\neg v}\}$ and A.
$A \leftarrow$ mutual ancestors of θ and V
$G_v \leftarrow$ Clone$(G_{v,\neg v})[0/\lambda_{\neg v}]$
$G_{\neg v} \leftarrow$ Clone$(G_{v,\neg v})[0/\lambda_v]$
$G_{\theta_1} \leftarrow$ Clone(G_θ)[Prod$(\theta_1, \theta)/\theta$]
$G_{\theta_2} \leftarrow$ Clone(G_θ)[Prod$(\theta_2, \theta)/\theta$]
$A' \leftarrow$ Sum(Prod$(\lambda_v, G_v, G_{\theta_1})$, Prod$(\lambda_{\neg v}, G_{\neg v}, G_{\theta_2})$)
Let $g = f \wedge v$, $g' = f \wedge \neg v$
return $(M \cup \{g, g'\}, C[A'/A], g, \theta_1, g', \theta_2)$

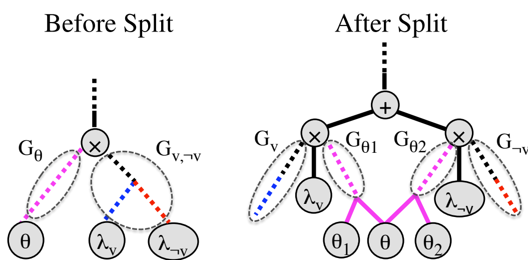

Figure 1. Illustration of the operation of the ACMN-Split subroutine, splitting a feature with parameter node θ on variable V. Dashed lines indicate sections of the circuit where details have been omitted.

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. *Clone* creates a copy of a subcircuit that maintains the same connectivity both within the subcircuit and to external nodes. 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, Sum and Prod construct new addition and multiplication nodes.

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.

4. Experiments

4.1. Methods

To evaluate the accuracy of ACMN, we compared it to four state-of-the-art algorithms, two for learning MNs and two for learning other forms of tractable graphical models. Our MN baselines are GSSL (Haaren & Davis, 2012) and L_1-regularized logistic regression (L1) (Ravikumar et al., 2009), 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) (Choi et al., 2011) and the LearnAC algorithm (Lowd & Domingos, 2008). 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.

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 our evaluation, we used 12 binary variable datasets, which have been used by several previous papers on MN structure learning (Davis & Domingos, 2010; Lowd & Davis, 2010; Haaren & Davis, 2012).

Table 1. 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

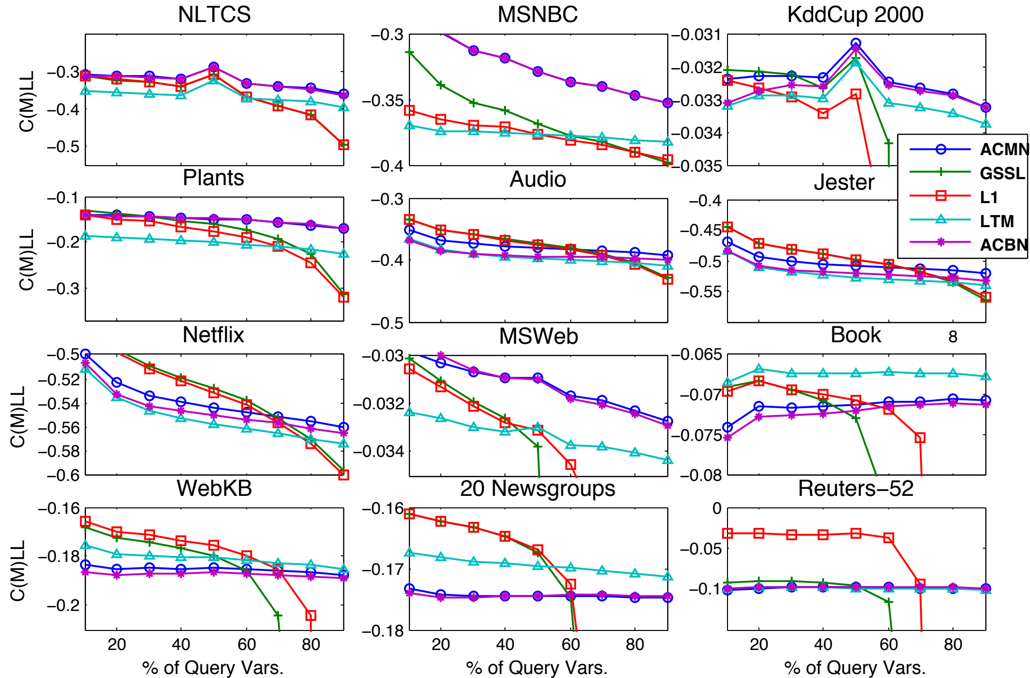

Figure 2. Normalized CLL and CMLL vs. fraction of query variables. CLL is reported for all tractable models (ACMN, LTM, ACBN) while CMLL is reported for the rest.

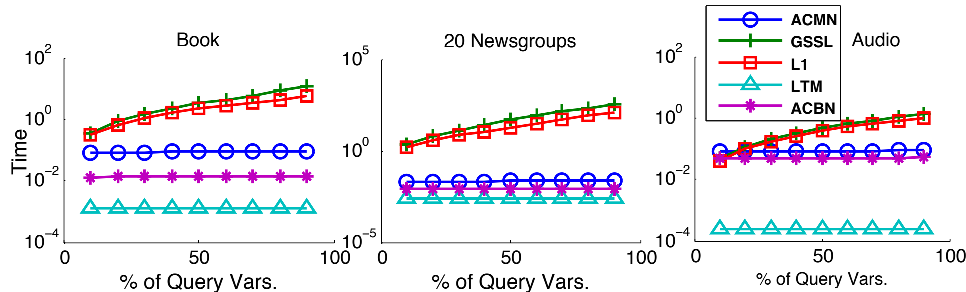

Figure 3. Query time for different percentages of query variables

marginal log-likelihood (CMLL) instead, a popular alternative when rare joint probabilities are hard to estimate (Lee et al., 2007). 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 (NCLL) and normalized CMLL (NCMLL), respectively.

4.2. Results

Table 1 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 used sampling to estimate the KL divergence between ACBN and ACMN models, and found that it was often much larger than the difference in log-likelihood. Furthermore, the number of nodes and edges was often very different as well.

Figure 2 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-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 3. 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.

5. 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.

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

Bach, F.R. and Jordan, M.I. Thin junction trees. *Advances in Neural Information Processing Systems*, 2001.

Chechetka, A. and Guestrin, C. Efficient principled learning of thin junction trees. *Advances in Neural Information Processing Systems*. 2008.

Choi, M. J., Tan, V., Anandkumar, A., and Willsky, A. Learning latent tree graphical models. *JMLR*, 2011.

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

Davis, J. and Domingos, P. Bottom-up learning of Markov network structure. *ICML*, 2010.

Della Pietra, S., Della Pietra, V., and Lafferty, J. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1997.

Elidan, G. and Gould, S. Learning bounded treewidth Bayesian networks. *JMLR*, 2008.

Gogate, V., Webb, W., and Domingos, P. Learning efficient Markov networks. *NIPS'10*, 2010.

Haaren, J. Van and Davis, J. Markov network structure learning: A randomized feature generation approach. *AAAI*, 2012.

Lee, S.-I., Ganapathi, V., and Koller, D. Efficient structure learning of Markov networks using L1-regularization. *Advances in Neural Information Processing Systems 19*, 2007.

Lowd, D. and Davis, J. Learning Markov network structure with decision trees. *ICDM*, 2010.

Lowd, D. and Domingos, P. Naive Bayes models for probability estimation. *ICML*, 2005.

Lowd, D. and Domingos, P. Learning arithmetic circuits. *UAI*, 2008.

McCallum, A. Efficiently inducing features of conditional random fields. *UAI*, 2003.

Meila, M. and Jordan, M. Learning with mixtures of trees. *JMLR*, 2000.

Poon, H. and Domingos, P. Sum-product networks: A new deep architecture. *UAI*, 2011.

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

Roth, D. On the hardness of approximate reasoning. *Artificial Intelligence*, 1996.

Wang, Y., Zhang, N. L., and Chen, T. Latent tree models and approximate inference in Bayesian networks. *Journal of Artificial Intelligence Research*, 2008.