Discriminative Structure Learning of Arithmetic Circuits

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

Abstract

The biggest limitation of probabilistic graphical models is the complexity of inference, which is often intractable. An appealing alternative is to use tractable probabilistic models, such as arithmetic circuits (ACs) and sum-product networks (SPNs), in which marginal and conditional queries can be answered efficiently. In this paper, we present the first discriminative structure learning algorithm for ACs, DACLearn (Discriminative AC Learner). Like previous work on generative structure learning, DACLearn finds a log-linear model with conjunctive features, using the size of an equivalent AC representation as a learning bias. Unlike previous work, DACLearn optimizes conditional likelihood, resulting in a more accurate conditional distribution. DACLearn also learns much more compact ACs than generative methods, since it does not need to represent a consistent distribution over the evidence variables. To ensure efficiency, DACLearn uses novel initialization and search heuristics to drastically reduce the number of feature evaluations required to learn an accurate model. In experiments on 20 benchmark domains, we find that our DACLearn learns models that are more accurate and compact than other tractable generative and discriminative methods.

Appearing in Proceedings of the 19^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2016, Cadiz, Spain. JMLR: W&CP volume 51.

1 Introduction

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

A compelling alternative is to work with model classes where inference is efficient, such as bounded treewidth models [1, 4], mixtures of tractable models [16, 19], sum-product networks (SPNs) [18, 8, 20], and arithmetic circuits (ACs) [5, 12, 13]. Previous work has demonstrated that these models can be learned from data and that they often meet or exceed the accuracy of intractable models. However, most of this work has focused on joint probability distributions over all variables. For discriminative tasks, the conditional distribution of query variables given evidence, $P(\mathcal{Y}|\mathcal{X})$, is usually more accurate and more compact than the joint distribution $P(\mathcal{Y}, \mathcal{X})$.

To the best of our knowledge, only a few algorithms address general tractable discriminative structure learning. These include learning tree conditional random fields (tree CRFs) [2], learning junction trees using graph cuts [22], max-margin tree predictors [17] and mixtures of conditional tree Bayesian networks (MCTBN) [10]. The first three methods are limited to pairwise potentials over the query variables. MCTBN learns a mixture of trees, which is slightly more flexible but still performed poorly in our experiments.

In this paper, we present DACLearn (Discriminative AC Learner), a flexible and powerful method for learning tractable discriminative models over discrete do-

mains. DACLearn is built on ACs, a particularly flexible model class that is equivalent to SPNs [20] and subsumes many other tractable model classes. DACLearn performs a search through the combinatorial space of conjunctive features, greedily selecting features that increase conditional likelihood. In order to keep the model compact, DACLearn uses the size of the AC as a learning bias. Since DACLearn is modeling a conditional distribution, its ACs can condition on arbitrary evidence variables without substantially increasing the size of the circuit, leading to much more compact models. DACLearn is similar to previous AC learning methods [12, 21], but it discriminatively learns a conditional distribution instead of a full joint distribution. DACLearn also introduces new initialization and search heuristics that improve the performance of existing generative AC learning algorithms.

In experiments on 20 benchmark datasets, we find that DACLearn is better at learning conditional distributions than several generative and discriminative baselines. We also find that its running time is similar to or better than other methods, in spite of the additional complexity of optimizing conditional likelihood.

2 Conditional Random Fields

Consider sets of discrete variables $\mathcal{Y} = \{Y_1, Y_2, \cdots, Y_n\}$ and $\mathcal{X} = \{X_1, X_2, \cdots, X_m\}$. Conditional random fields[1] (CRFs) [11] are undirected graphical models that represent the conditional probability distribution of query variables $\mathcal{Y}$ given the evidence variables $\mathcal{X}$:

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

where each ϕ_c is a real-valued, non-negative function, known as a potential function, with scope $D_c \subset \mathcal{X} \cup \mathcal{Y}$. $Z(\mathcal{X})$ is a normalization function, also called the partition function, which only depends on evidence variables $\mathcal{X}$. A Markov network can be seen as a special case of a CRF with no evidence variables, so $\mathcal{X}$ is empty and the partition function Z is a constant.

If all potential functions of Relation 1 are positive, then we can represent the conditional probability distribution using an equivalent log-linear formulation:

$$\log P(\mathcal{Y}|\mathcal{X}) = \sum_i w_i f_i(D_i) - \log Z(\mathcal{X}), \tag{2}$$

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

[1] Also known as conditional Markov networks.

3 Arithmetic Circuits

Inference in probabilistic graphical models such as CRFs is typically intractable. An appealing alternative is tractable probabilistic models, which can efficiently answer any marginal or conditional probability query. Our focus is on arithmetic circuits (ACs) [5], a particularly flexible tractable representation. An arithmetic circuit (AC) [5] is a tractable probabilistic model over a set of discrete random variables, $P(\mathcal{X})$. An AC consists of a rooted, directed, acyclic graph in which interior nodes are sums and products. Each leaf is either a non-negative model parameter or an indicator variable that is set to one if a particular variable can take on a particular value.

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

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

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

In an AC, to compute the unnormalized probability of a complete configuration $\tilde{P}(\mathcal{X} = \mathbf{x})$, we first set the indicators variable leaves to one or zero depending on whether they are consistent or inconsistent with the values in $\mathbf{x}$. Then we evaluate each interior node from the bottom up, computing its value as a function of its children. The value of the root node is the unnormalized probability of the configuration. However, the real strength of ACs is their ability to efficiently marginalize over an exponential number of variable states. To compute the probability of a partial configuration, set all indicator variables for the marginalized variables to one and proceed as with a complete configuration. The normalization constant Z can similarly be computed by setting all indicator variables to one. Conditional probabilities can be computed as probability ratios. For example, for the AC in Figure 1, we can compute the unnormalized probability $\tilde{P}(y_1)$ by setting $\lambda_{\neg y1}$ to zero and all others to one, and then evaluating the root. To obtain the normalization constant, we set all indicator variables to one and again evaluate the root.

Sum-product networks (SPNs) [18] are closely related to ACs – both represent probability distributions as a computation graph of sums and products, and both support linear-time inference. In discrete domains, SPNs can be efficiently converted to ACs and vice versa [20]. For representing and manipulating tractable log-linear models, ACs are a better fit, since they represent parameters directly as parameter nodes

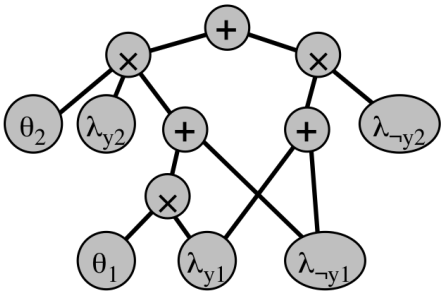

Figure 1: Simple arithmetic circuit that encodes a Markov network with two variables y_1 and y_2 and two features $f_1 = y_1 \wedge y_2$ and $f_2 = y_2$

rather than implicitly as edge weights.

3.1 Conditional ACs

This efficient marginalization relies on two properties of ACs [5, 12]: 1) An AC is *decomposable* if the children of a product node have no common descendant variable. 2) An AC is *smooth* if the children of a sum node have identical descendant variables. We say that an AC is *valid* if it satisfies both properties. Valid ACs can compactly represent probabilistic graphical models with low tree-width, sum-product networks [18, 20], and many high tree-width models with local structure [3].

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

Definition 1 *An AC over query variables $\mathcal{Y}$ and evidence variables $\mathcal{X}$ is conditionally valid if it is smooth and decomposable over $\mathcal{Y}$.*

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

Relaxing the definition of validity effectively allows features to be conditioned on arbitrary evidence without substantially increasing the complexity of inference. This ability to add complex dependencies on the evidence is also a stated benefit of CRFs. ACs offer the additional benefits of rich structure and tractable inference over the query variables. See Figure 2 for an example of how a conditionally valid AC can remain compact while including numerous dependencies on the evidence.

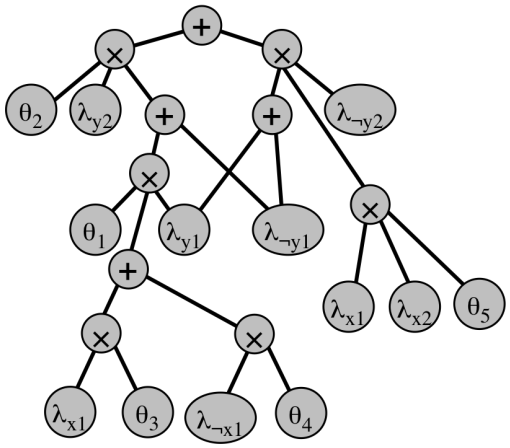

Figure 2: Conditional arithmetic circuit that encodes a conditional random field over two binary query variables y_1 and y_2, two binary evidence variables x_1 and x_2, and five features $f_1 = y_1 \wedge y_2$, $f_2 = y_2$, $f_3 = y_1 \wedge y_2 \wedge x_1$, $f_4 = f_3 = y_1 \wedge y_2 \wedge \neg x_1$, and $f_5 = \neg y_2 \wedge x_1 \wedge x_2$.

4 DACLearn

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

DACLearn builds on methods for learning tractable Markov networks, the ACMN algorithm in particular [13]. As with learning a Markov network, DACLearn performs a greedy search through the combinatorial space of conjunctive features, using the size of the corresponding AC as a learning bias. However, rather than optimizing log-likelihood, DACLearn optimizes the conditional log-likelihood (CLL) of the training dataset $\mathcal{D}$:

$$\begin{aligned} CLL(\mathcal{D}) &= \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} \log P(\mathbf{y}|\mathbf{x}) \\ &= \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} \sum_j w_j f_j(\mathbf{d}_j) - \log Z(\mathbf{x}) \end{aligned} \quad (3)$$

where $\mathbf{d}_j$ denotes the values of $\mathbf{y}$ and $\mathbf{x}$ for the vari-

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

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

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

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

4.1 Structure search

The goal of structure search is to find features that increase the value of the CLL objective function, Relation 3, without vastly increasing the complexity of inference in the model. Therefore, following Lowd and Domingos [12] we optimize a modified objective that penalizes circuits with more edges:

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

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

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

$$\mathcal{F}(f, v) = \bigcup_{i=1}^{k} f \wedge v^i, \tag{4}$$

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

$$Score(\mathcal{F}) = \Delta_{cll}(\mathcal{F}) - \gamma\Delta_e(\mathcal{F}) - \lambda|\mathcal{F}|, \tag{5}$$

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

In order to compute the score of candidate feature group $\mathcal{F}$, we need to compute the increment in the likelihood, which requires optimizing the weight of each candidate feature in the group. For efficiency, while scoring a feature group we assume that the weights of the other features are fixed, an approach also used by McCallum [15]. This leads to an approximation of the CLL gain that can be optimized without rerunning inference in the model. Specifically, if we add a candidate feature group $\mathcal{F}(f, v)$ into the model while keeping the other parameters fixed, we can update the partition function using the following relation:

$$\Delta \log Z(\mathcal{X}) = \log(\sum_i \exp(\theta_i) P(f_i|\mathcal{X}) + P(\neg f|\mathcal{X})), \tag{6}$$

where $f_i = f \wedge v_i$, and θ_i is the weight of f_i. P is the probability distribution represented by the current model. Using an AC, we can compute $P(f_i|\mathcal{X})$ for all $f_i \in \mathcal{F}$ by running inference in the circuit: once to compute the partition function $Z(\mathcal{X})$ and once again to compute all unnormalized $\widetilde{P}(f_i)$. These unnormalized probabilities can be computed in parallel by differentiating the circuit (see Darwiche [5] for details). For each feature f, we can re-use the expectation and partition function for candidate feature groups $F(f, v)$ for every variable v.

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

$$\Delta_{cll}(\mathcal{F}) = \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} (\sum_i (\theta_i \widetilde{P}(f_i|\mathbf{x})) - \Delta \log Z(\mathbf{x})), \tag{7}$$

where $\widetilde{P}$ is the empirical probability distribution. The gradient of Relation 7 with respect to θ_i becomes:

$$\frac{\partial \Delta_{cll}(\mathcal{F})}{\partial \theta_i} = \sum_{(\mathbf{y},\mathbf{x})\in\mathcal{D}} \frac{\exp(\theta_i) P(f_i|\mathbf{x})}{\exp(\Delta \log Z(\mathbf{x}))} \tag{8}$$

As a result, optimizing Relation 7 does not require inference in the candidate model.

Nevertheless, fixing the existing parameters is very restrictive since adding new features may affect the optimal weights of the current features; thus we have to relearn the parameters after adding each feature through joint parameter optimization. In our experiments, we found that it sufficed to perform this optimization incrementally, running just one step of gradient descent

Algorithm 1 DACLearn

```
C ← AC representing initial structure. //Section 4.1.1
fs ← ∅ //candidate feature group max heap.
of ← ∅ //omitted candidates max heap.
fh ← feature max heap based on feature support.
//t is feature batch size.
do
    while fs ≠ ∅ do
        F ← fs.pop()
        s ← Δ_cll(F) − λ|F|
        if s > γΔ_e(F) then
            Update C
            Joint parameter optimization
        else
            of.push(F)
        end if
        if size of C > max size then Stop.
    end while
    for i=1 to t do
        f ← fh.pop()
        if Support of f < min support then
            γ ← γ/2 //Shrink edge cost.
            if γ < γ_min then
                Stop
            else
                fs ← of; of ← ∅
            end if
            break
        else fs ← GenCandidates(f)
        end if
    end for
while not Stop
return C
```

after adding each feature. This works because the optimal weights for most features do not change very much after adding one feature group, so even one step of gradient descent is sufficient to keep weights close to their optimal values throughout structure learning. After structure learning is complete, we fine-tune all model parameters by running joint parameter optimization to convergence.

When we add a candidate feature group to the model, the scores of all the other candidate feature groups become obsolete, so we have to re-score them. If we have m candidate feature groups, we may re-score a candidate feature group $O(m)$ times. Therefore, if we initially generate all possible candidate feature groups we would end up with an exponential number of candidate feature groups that makes re-scoring become the bottleneck of the learning process. To address this problem, we use a greedy approach, Algorithm 2, to have a small set of candidate feature groups (candidate set) at every point of the structure search.

This heuristic is based on the idea that interesting features may have more support in the data, a heuristic also used by [9] to learn Markov networks. Although this heuristic will sometimes overlook higher-scoring feature groups, our experiments verify that, using this

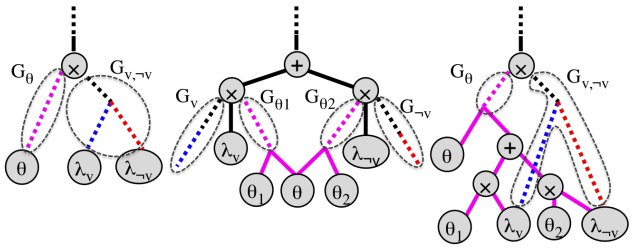

Figure 3: Updating circuits. Left: initial circuit, middle: circuit after a consistent split, and right: circuit after an inconsistent split.

heuristic, the algorithm can better explore the feature space, resulting in more accurate models.

As shown in Algorithm 1, DACLearn maintains a set of candidate feature, groups ordered by Δ_{cll}. Each candidate feature group that increases the model score is added to the model. When the current set of feature groups has been exhausted, it selects the t current features with the most support in the data and uses them to generate new candidate feature groups. Algorithm 2 shows the process of generating candidate features groups, which consists of extending an existing feature with all possible variables and computing the conditional likelihood gain of each new group. Therefore, we need to optimize Relation 7 $O(t|V|)$ times for each round of candidate feature generation.

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

During structure search, we may omit some candidate feature groups because of their edge costs. However, we may run out of good candidate feature groups while the size of the circuit is much smaller than the maximum circuit size. Therefore, to benefit from these candidate feature groups, we keep all the omitted candidate feature groups in a priority heap. These candidates are pruned only because of the complexity penalty term, so when there are no more features to expand, we halve the edge penalty γ and re-score these candidate feature groups.

4.1.1 Initial structure

As discussed before, keeping the candidate set small is important to reduce the number of re-scores. Unfortunately, even scoring pairwise features would require at least $\Omega(|\mathcal{V}^2|)$ calls to the score function, where $\mathcal{V}$ is the set of all variables. Therefore, we begin with a pre-defined initial structure consisting of heuristically

Algorithm 2 Candidate feature group generator

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

chosen pairwise features. This allows us to consider more complex features given limited training time.

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

4.2 Parameter learning

We jointly optimize all parameters after compiling the initial structure, adding each feature, and after the end of the structure search. The CLL objective function is convex with respect to feature weights, so we use the L-BFGS algorithm to optimize it. We compute the gradient for all feature weights by differentiating the circuit once, which only requires two passes over the circuit [5]. The partition function of Relation 3 depends on evidence variables, so we have to differentiate the circuit once for each example. This requires performing inference in the circuit $\Omega(|D|)$ times, which highlights the expense of joint optimization and importance of having efficient inference.

4.3 Updating circuits

Our circuit update method is based on the Split algorithm from ACMN. Given a feature f in the circuit with parameter θ and a binary variable V, the goal is to add two features to the circuit, $f \wedge v$ and $f \wedge \neg v$, with parameters θ_1 and θ_2, respectively. The left circuit in Figure 3 shows the initial circuit. G_θ indicates the sub-circuit between the common ancestor of indicator variables λ_v and $\lambda_{\neg v}$ and parameter node θ. Similarly, the sub-circuits between the common ancestor and indicator nodes λ_v and $\lambda_{\neg_v}$ are labeled G_v and $G_{\neg v}$, respectively. If V is a query variable, we have to duplicate the sub-circuit G_θ into $G_{\theta 1}$ and $G_{\theta 2}$, as shown in the middle circuit of Figure 3, and extend each sub-circuit using the new parameter nodes θ_1 and θ_2. Parameter node θ is attached to both sub-circuits, which ensures the decomposability and smoothness of the circuit. On the other hand, if V is an evidence variable, we have a more compact representation by avoiding the expensive duplication of G_θ. We refer to this as an *inconsistent split*. The right circuit of Figure 3 indicates the result of an inconsistent split, which is conditionally valid. For an inconsistent split, the number of new nodes and edges added to the circuit is significantly less than the size of G_θ that we need to duplicate for a consistent split.

Table 1: Dataset characteristics

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

5 Experiments

5.1 Datasets

We run our experiments using 20 datasets illustrated in Table 1 with 16 to 1556 binary-valued variables. These datasets are drawn from a variety of domains, including recommender systems, text analysis, census data, and plant species distribution, and have been extensively used in previous work [6, 8, 20, 19].

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

5.2 Methods

For baselines, we compare to a state-of-the-art generative SPN learner, IDSPN, and a generative AC learner,

ACMN. Our proposed heuristics for structure search can also help the generative ACMN algorithm to find a better structure. Therefore, we incorporate those heuristics into the ACMN algorithm, which we name efficient ACMN (EACMN). Based on our experiments on the 20 datasets, EACMN is significantly more accurate than ACMN on 13 datasets out of 20 datasets in terms of average log-likelihood of joint probability distribution, and not significantly different on the remaining 7 datasets. Moreover, EACMN, on average, finds 1.7 times more features, while its circuits are 2.5 times more compact. These comparisons show the importance of our search heuristics. See the supplementary material for a more detailed comparison between ACMN and EACMN. By using EACMN as a baseline, we ensure that any performance gains demonstrated by DACLearn are attributable to discriminative learning, and are not simply an artifact of the new structure search efficiency heuristics.

To compare the effect of discriminative parameter learning, we also take the best models learned by EACMN, based on average log-likelihood on validation data, and relearn their parameters to maximize CLL. We call this method conditional ACMN (CAMCN). This idea is similar to discriminative learning of SPNs [7], which supposes a predefined SPN structure and then applies a discriminative weight learning approach to learn the parameters.

As our last baseline, we choose MCTBN [10], which learns a mixture of conditional tree Bayesian networks. To find each tree, MCTBN needs to learn $O(n^2)$ logistic regression models, where n is the number of query variables, and each logistic regression does $O(|\mathcal{D}|)$ passes over training data. This makes MCTBN less practical when the number of query variables increases[2].

For all of the above methods, we learn the model using the training data and tune the hyper-parameters using the validation data, and we report the average CLL over the test data. To tune the hyper-parameters, we used a grid search over the parameter space. We used the original IDSPN models learned by the authors[20][3].

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

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

Our implementations of DACLearn, EACMN, CACMN, and IDSPN are all available in the open-source Libra toolkit [14], available from `http://libra.cs.uoregon.edu/`.

5.3 Results

Table 2 shows the average CLL comparison of DACLearn and the other baselines on 20 datasets with 50% and 80% evidence variables[5]. We use $*$ to indicate that DACLearn has significantly better test set CLL than the corresponding method on the given dataset, and $\bullet$ for the reverse. We also use $\circ$ to show that two methods are not significantly different. The bold numbers only highlight which method out 5 has the better average CLL on the given dataset. Wins and losses are determined by two-tailed paired t-tests ($p < 0.05$). Based on the results, DACLearn is never significantly worse than MCTBN, CACMN, and EACMN, on the 20 datasets with 80% evidence variables, and only is significantly worse than IDSPN on two datasets. Furthermore, DACLearn has better average CLL than the other methods on 15 datasets. As we decrease the number of evidence variables, the benefits of discriminative learning diminish (as expected), but DACLearn still significantly outperforms the other methods on many datasets. MCTBN reaches the 24 hour limit without training all the needed $O(n^2)$ logistic regressions on 3 datasets. Table 2 also shows that discriminative parameter learning is no substitute for discriminative structure learning. The performance of CACMN is much closer to EACMN than to DACLearn. This is because DACLearn learns conditionally valid ACs, which allows DACLearn to consider many models that EACMN and CACMN can-

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

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

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

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

Table 2: Average conditional log-likelihood (CLL) comparison. • shows significantly better CLL than DACLearn, ∗ indicates significantly worse CLL than DACLearn, and ∘ used when CLL is not significantly different. The bold numbers highlights the method that has the best CLL on each dataset. The last row, summarizes the number of wins (W), ties (T), and losses (L) of DACLearn comparing to the other baselines based on the significance results. † indicates the experiment has not finished given the 24 hour limit.

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

not compactly represent as valid ACs. On two of the three datasets where CACMN is significantly better than DACLearn, EACMN is also significantly better. DACLearn also compares favorably to IDSPN [20], in spite of the fact that IDSPN learns models with hidden variables and DACLearn does not.

As discussed earlier, conditionally valid ACs representing $P(\mathcal{Y}|\mathcal{X})$ are more compact than valid ACs representing $P(\mathcal{Y}, \mathcal{X})$. We verify this empirically by measuring the size of circuits learned with each method. The average sizes of the circuits learned by IDSPN and EACMN are 2.2M and 1.1M edges, respectively, while the average size of the circuits learned by DACLearn is 55K edges when we have 50% evidence variables and 22K edges when we have 80% evidence variables. This means that inference in conditionally valid ACs is 100 times faster than IDSPN when we have 80% evidence variables!

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

6 Conclusion

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

DACLearn is limited to learning conjunctive features over the observed variables. Previous work with SPNs has shown that mixtures often lead to higher accuracy. For example, IDSPN uses hierarchical mixtures of tractable Markov networks to obtain consistently better results than tractable Markov networks alone [20]. Learning tractable conditional distributions with latent variables remains an important open problem.

Acknowledgments

This research was supported by NSF grant IIS-1451453 and a Google Faculty Research Award.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

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

[14] Daniel Lowd and Amirmohammad Rooshenas. The libra toolkit for probabilistic models. *Journal of Machine Learning Research*, 16:2459–2463, 2015.

[15] 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.

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

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

[18] 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.

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

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

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

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