VACA: Variational Arithmetic Circuit Approximation

Daniel Lowd Pedro Domingos

June 6, 2008

1 Overview

Our goal is to approximate a Bayesian network, possibly conditioned on evidence, with a model that allows arbitrary queries to be answered efficiently. We focus on Bayesian networks with tree-structured conditional probability distributions (CPDs) because they can represent many complex dependencies much more efficiently than table CPDs, which must have one parameter for every configuration of the parent variables. The probability distribution of a Bayesian network conditioned on evidence can be represented as a Markov network in which each CPD is replaced by a clique over the unobserved variables of the corresponding family. A cliques can be represented as a set of features, one feature for each path through the corresponding decision tree.

We approximate these Markov network distributions with arithmetic circuits that compute simpler Markov network distributions, obtained by pruning the feature tree in each clique. At one extreme, this is equivalent to the mean field variational approximation when each tree is pruned to a stump representing the marginal distribution of a single variable. At the other extreme, this is equivalent to exact inference when the trees are not pruned at all. By adjusting algorithm parameters, our method allows the user to trade off computational cost and approximation quality.

The advantage of arithmetic circuits over other probabilistic representations such as junction trees is that arithmetic circuits can take advantage of structure such as context-specific independence. Inference is linear in arithmetic circuit size, and many queries can be computed in parallel, such as all single-variable marginals conditioned on arbitrary evidence. These strengths have been well-documented and exploited for exact inference in Bayesian networks. Recently, we demonstrated how arithmetic circuits could be used to learn Bayesian networks with high treewidth that still allowed for exact inference.

Our approximate inference algorithm works similarly to Bayesian network structure learning, except that we are trying to approximate a Bayesian network rather than an empirical distribution. We begin with a product of marginals distribution, equivalent to mean field. In each iteration, we greedily split a set of features in two by adding a literal or its negation to the conjunction. Viewing the features in each clique as a tree, each iteration can be seen as splitting a leaf in one of the trees. We use the Bayesian network structure to guide us, restricting our splits to those that are present in the original Bayesian network CPDs. The cliques in our approximating Markov network are therefore pruned versions of the trees in the Bayesian network.

The key details are how the Markov network is represented as an arithmetic circuit, how we pick which split to apply in each iteration, and how we set the parameters.

2 Arithmetic Circuit Representation

We start with a product of marginals distribution, represented as a multiplication node with one addition node per variable, summing over its values. Each value of each variable has a multiplication node over two leaves, one for the indicator variable of that variable/value combaination and one for its marginal probability. For variables that are known in the evidence, we include a null feature, consisting of a single probability leaf [Not done yet!]. We update this distribution using the SplitAC subroutine from Lowd and Domingos (2008).

3 Kullback-Leibler Divergence

Since our approximating model is not a Bayesian network, the parameters may not represent conditional probabilities and cannot be efficiently computed analytically. Instead, we select parameters to minimize the Kullback-Leibler divergence (KLD) of the Bayesian network distribution, P, from the approximating arithmetic circuit distribution, Q:

$$D_{KL}(Q \parallel P) = \sum_x Q(x) \log \frac{Q(x)}{P(x)}$$

We can represent both probability distributions as log linear models:

$$\begin{aligned} P(x) &= \exp(\sum_i w_i f_i(x)) \\ Q(x) &= \exp(\sum_k w_k g_k(x)) \end{aligned}$$

We can represent the KLD in terms of the features:

$$D_{KL}(Q \parallel P) = \sum_i w_i Q(f_i) - \log Z - \sum_k w_k Q(g_k)$$

where $Q(f_i) = \sum_x f_i(x)Q(x)$ and $Q(g_k) = \sum_x g_k(x)Q(x)$, representing the expected values of the respective features. Its gradient can be computed as follows:

$$\frac{\partial}{\partial w_j} D_{KL}(Q \parallel P) = \sum_i w_i (P(f_i f_j) - P(f_i)P(f_j)) - \sum_k w_k (P(f_i g_k) - P(g_k)P(f_j))$$

In an arithmetic circuit, the complexity of computing any marginal or conditional probability is linear in circuit size. Let p and q be the number of features in the Bayesian network and the approximating arithmetic circuit, respectively. Since all of our features are conjunctions of variable values, each probability can be computed in a single circuit evaluation, a total of $p + q$ for the K-L divergence or $q(p + q)$ for the gradient. Since the features in each clique form a subtree of the corresponding CPD in the Bayesian network, each features probability $P(f_i)$ can be written as the sum of one or more $P(g_k)$. For example, $P(A) = P(A \wedge B) + P(A \wedge \neg B)$. This allows us to compute the KLD in q circuit evaluations and its derivative in pq evaluations. To compute the derivative of the KLD in a mere p evaluations, we differentiate the circuit with respect to each feature f_i conditioned on each feature g_k. Differentiation is also linear in circuit size, allowing us to compute the probabilities $P(f_i \wedge g_k)$ for all f_i with a particular g_k in one upward and downward pass through the circuit. This method allows us to use a numeric solver, such as L-BFGS, to optimize the weights.

4 Evaluating Splits

As in the LearnAC algorithm (Lowd and Domingos, 2008) , we score splits with a metric that trades off how much we expect them to improve our accuracy with how much they cost in terms of increasing circuit size. We greedily add the best split in every iteration until no remaining split has positive score. For the original LearnAC algorithm, the improvement in accuracy was the change in log likelihood of the training data, an easy value to compute. For approximate inference, measuring the reduction in KLD from introducing a split is much more expensive because we don't know the optimal parameters.

The exact way to compute the reduction would be to add the split, optimize all parameters using L-BFGS, and recompute the K-L divergence. This requires p differentiations in every iteration of L-BFGS for every candidate split. Since the gain may increase or decrease as we add other splits, every split must be reevaluated in every iteration in order to be completely accurate. Naturally, this is extremely slow, possibly requiring millions of circuit differentiations in every iteration.

4.1 Inference-based Methods

There are several possible approaches to reducing this cost. The first is to do a one-dimensional optimization instead of an n-dimensional optimization, since we expect that most feature weights would change little when we add a new feature. This is a lower bound on the gain we would get by optimizing all weights. The advantage of the one-dimensional case is that, given all $Q(f_j \wedge g_k)$ for some f_j and all g_k, we can compute the KLD (and its derivative with respect to w_j) at any weight offset $w'_j = w_j + \delta_j$. We do this by computing updated marginals $Q(f_i)$, $Q(g_k)$, $Q(f_j \wedge f_i)$, and $Q(f_j \wedge g_k)$:

$$\begin{aligned} Z' &= Z(Q(f_j)(\exp(\delta_j) - 1) + 1) \\ Q'(f_i) &= (Q(f_i \wedge f_j)(\exp(\delta_j) - 1) + Q(f_i))Z/Z' \\ Q'(f_i \wedge f_j) &= Q(f_i \wedge f_j)\exp(\delta_j)Z/Z' \end{aligned}$$

The primed values represent the normalization constant or probabilities with the new weight w'_j. $Q'(g_k)$ and $Q'(g_k \wedge f_j)$ are computed analogously. This allows us to compute a lower bound on the KLD gain of a feature with just p circuit differentiations.

A useful approximation is to score a feature only when it is first introduced, rather than rescoring it in every iteration. Note that each new feature is an extension of an existing feature, $f_j = f_i \wedge x$ for some variable value x. Therefore, in order to score all f_j, it suffices to know $P(f_i \wedge x \wedge g_k)$ for all f_i, x, and g_k. When there are many more candidate features than variable/value pairs, we can do this more efficiently by differentiating the circuit conditioned on each $x \wedge g_k$ pair. This is most effective when we score features in batches: First score all initial features and add as many as look worthwhile, then score extensions to those features and add them, etc. Unfortunately, this overall approach is still quite slow.

4.2 Tree-based Methods

We adopt an alternate approach that allows us to score all features without doing any additional inference at all. Suppose that every variable in the Bayesian network was represented by a marginal distribution except for one. We can approximate this marginal distribution by using our initial mean field model. We score the gain of each split in the decision tree as the reduction in KLD between the approximate model without that split and the true model where all other variables CPDs are marginal distributions. This can be computed as:

$$\text{gain}(D, V) = Q(D)(H(D) - Q(V)H(D|V) - Q(\neg V)H(D|\neg V))$$

where D represents the distribution obtained by pruning the decision tree to this node, V is the variable being split on, $Q(D)$ represents the probability of the variable conjunction including all variables from the root of the tree to this node, H is the entropy function, and $H(D|V)$ and $H(D|\neg V)$ represent the entropies of the distribution conditioned on V and $\neg V$, respectively. All distributions are weighted averages of the leaf distributions, where the weights are determined using the mean field marginals. [TODO: Make sure this formula is correctly implemented in the code.]

The biggest problem with this is that when a variable value is nearly certain, a split's gain may be small even if its child splits are very useful. For this reason, it may be helpful to look at the total gain of performing a split along with all its descendants before entirely rejecting it. If a split has a high cost in terms of the number of edges it would add to the circuit but its descendants have large gains, it may be useful to consider skipping past a split, selecting the child branch with the largest total gain. [TODO: The value of these ideas is still inconclusive.]

A final modification to this scoring procedure is to rescore splits every iteration or periodically. As long as the marginals $Q(g_k)$ are known, we can easily construct the weighted average distributions and compute the gains based on the current approximation instead of the initial mean field marginals. Since this still does not require additional inference in the network, it should be relatively fast. [TODO: Test this out.]

5 Parameter Optimization

Our discussion of K-L divergence has already touched on how to optimize parameters using L-BFGS and relatively efficient gradient computations. We now discuss the options for when parameters should be optimized and how we can make this more efficient.

If features are scored using an inference-based method involving the current approximation, then we must optimize the weights of newly added features before scoring extensions. A one-dimensional optimization after adding each feature usually suffices. Periodic full runs of L-BFGS are only worth the overhead if infrequent, such as before scoring the next "level" of candidate features.

If features are scored using the tree-based method with mean field marginals, then we don't need to optimize any feature until the end. However, successive one-dimensional optimizations sometimes lead to a model with a much higher score than only optimizing all weights at once. Even with one-dimensional optimizations, a final n-dimensional pass at the end can make a large difference.

The cost of p differentiations of the circuit can still be too expensive. With a few assumptions, we can replace the exact probabilities with approximate ones that are much faster to compute. Consider a feature f_j with weight w_j that we are hoping to optimize. We can compute all $P(f_i)$ in a single differentiation, conditioned on no evidence. We can compute all $P(f_j \wedge f_i)$ in a second differentiation, conditioned on f_i. However, we cannot compute all $P(f_j \wedge g_k)$ in a single pass, since most g_k are not features in the network. Instead, we can approximate them as follows:

$$P(f_j \wedge g_k) = P(f_j|f_i)P(f_i|g_k)P(g_k)$$

All $P(f_j|f_i)$ are known from the first two differentiations. $P(g_k)$ can be cached from the last inference we ran, since our one-dimensional optimization always generates new marginals, $P(g_k)$. In general, $P(f_i|g_k)$ is not known. Assuming we never "skip" past splits, there exists some f_i which is a subset of the conjunction g_k, so $P(f_i|g_k) = 1$. This allows us to compute $P(f_j \wedge g_k)$ without doing any additional inference in the network at all. This approximation is likely to be bad when f_j and g_k have variables in their conjunctions that are not present in f_i. For these cases we compute the probability manually, and update $P(g_k)$ at the same time.

By using tree-based methods for scoring and one-dimensional optimization with approximate probabilities, we can construct a reasonably good approximate model with a minimal number of circuit differentiations. [TODO: Investigate ways of using approximate probabilities with inference-based scoring.] However, the final global optimization remains slow.

We can extend the one-dimensional optimization method to optimize n dimensions at once, as long as the corresponding features are mutually exclusive. This works because when there are no higher-order interactions among features, their weights have independent effects on the probabilities. We compute the adjusted probabilities as follows:

$$\begin{aligned}
Z' &= Z(1 + \sum_j Q(f_j)(\exp(\delta_j) - 1)) \\
Q'(f_i) &= (Q(f_i) + \sum_j Q(f_i \wedge f_j)(\exp(\delta_j) - 1))Z/Z' \\
Q'(f_i \wedge f_j) &= Q(f_i \wedge f_j)\exp(\delta_j)Z/Z'
\end{aligned}$$

where j now ranges over all features f_j in the mutually exclusive set. Note that one simple candidate set is all weights in one clique, since the tree-structured features are guaranteed to be mutually exclusive. The set can be expanded to include any other features whose pairwise probability with all others in the set is zero. We can further relax this condition to include features whose pairwise probability is nearly zero with all others in the set. L-BFGS can be used to find the best values for one subset of the weights, then another, then another until convergence. When combined with approximate probabilities, this could speed-up the final optimization by several orders of magnitude. [TODO: Try this out!] There might also be circumstances in which it is beneficial to optimize weights partway through constructing the approximating model.