PGM Notes

Amit Indap

October 24, 2012

Abstract

1 Introduction

1.1 Joint Distribution and CPDs

Say you are given a list of n random variables $X_1 \ldots X_N$ and for simplicitys sake each of these are binary valued. The join distribution, P , of these n variables would require the specification of 2n ? 1 numbers. Save for the smallest values of n, the joint distribution is cumbersome to represent and manipulate computationally. To represent joint distributions more compactly Bayesian Networks (BNs) take advantage of independence properties of variables and use the directed acyclic graph (DAG) as a general purpose data structure to encode, manipulate, and calculate probabilities of high dimensional distributions.

Taking the example from Section 3.1.2 in [1], consider a company trying to higher recent graduates. They want to hire gifted employees, but there is no direct way to assess intelligence. Rather, they use a students SAT score as a proxy. We can formalize this probabilistically by two random variables, Intelligence (I) and SAT (S). To represent the joint distribution of S and I, we can factorize the joint in the following way: $P(I, S) = P(I)P(S|I)$. Mathematically, P(I) represents the prior distribution over I while P(S—I is a conditional probability distribution (CPD) of S given I. Figure 1 shows how this can be shown as a DAG, representing each of the random variables as nodes and edges representing dependencies between variables.

1.2 Factors

2 Representing factors in Python/Matlab

Representing graphical models and performing inference is done by creating and manipulating factors. A factor is function ..

The Coursera class used Matlab/Octave for its programming assignments. As far as I know there is now Matlab library to do I/0 on BAM files. Therefore, I needed to implement factor representation in Python, using the PGM Matlab code as a template.

3 Variable Elimination and Exact Inference

The common feature of any inference techniques with PGMs are the manipulation of factors. A factor ϕ over a $Scope[\phi] = X$, which is a function $\phi : Val(X) \mapsto R$ The underlying operation when computing the probability of some variable in a PGM is marginalizing out variables from a distribution. We can view the as computation on a factor. Let X be a set of variables and $Y \notin X$ and $\psi(X, Y)$ be a factor. Marginalizing out Y generates a new factor ψ over X:

$$\psi(X) = \sum_X \phi(X, Y) \tag{1}$$

A key trick in doing inference on PGMs is exchanging a summation and a product if $X \notin Scope[\phi_1]$:

$$\sum_X (\phi_1 \phi_2) = \phi_1 \sum_X (\phi_2) \tag{2}$$

A marginal probability computation involves taking the product of CPDs and doing a summation on all the variables except the query variables (the variables you want to find the posterior for). So in general, the inference task involves taking a *sum-product* of the form

$$\sum_Z \prod_{\phi \in \Phi} \phi \tag{3}$$

The sum product variable elimination (VE) pseudo code is given on page 298. Each variable is summed out one at a time, given particular elimination order. When a variable is summed out, all factors that contain that variable in its scope are multiplied, generating a product factor. Then the variable to be eliminated is summed out of this product factor. Again, let X be a set of variables and Φ be a set of factors such that for each $\phi \in \Phi$, $Scope[\phi] \subseteq X$ Let $Y \subset X$ be a set of query variables and the remaining variables be $Z = X - Y$. Then for any elimination ordering of the set of variables, then the sum product variable elimination returns a new factor $\phi^*(Y)$:

$$\phi^*(Y) = \sum_{Z} \prod_{\phi \in \Phi} \phi \tag{4}$$

3.1 Graph theoretic view of Variable Elimination

The sum product VE algorithm is agnostic about the type of graph on which it operates. But the manipulation of factors can equivalently be seen as a series of graph transformations with an associated set of factors. Define an undirected graph H whose nodes are variables in the $Scope[\Phi]$ and where there is an edge between nodes if there exists a factor $\phi \in \Phi$ such that X_i and $X_j \in Scope[\phi]$ In other words, the undirected graph H_Φ is a fully connected sub-graph over the scope of each factor $\phi \in \Phi$. In the case of no evidence the graph H_Φ is a moralized graph of the Bayesian Network G.

In the process of eliminating a variable a new factor ψ is created with X and all the other variables $\mathbf{Y}$ that appear with it in factors. Then X is summed out, creating a new factor τ that contains all the variables $\mathbf{Y}$ but not X. Let Φ_X be the resulting set of factors. When the factor ψ is creates edges between all the variables $Y \in \mathbf{Y}$. Some may have been in the original graph H_Φ, others are introduced as fill edges. When the factor τ is created, X is removed and all its incident edges are removed. The elimination order than reflects a series of graphs and every factor that appears in the steps of the VE sum product algorithm is a clique. The set of factors generated in VE is a clique in the *induced* graph. Figure 9.11 on page 309 shows the induced graph and the resulting clique tree for the VE algorithm.

4 Clique Trees and Exact Inference

In section 3 we describe the VE algorithm which sums out variables one at a time. In Chapter 10, Koller and Friedman describe how to use a cluster graph/ clique tree as a global data structure to eliminate larger sets of variables.

4.1 Cluster Graphs to Clique Trees

A *cluster graph* is a data structure to help track the factor manipulation process at the heart of inference calculations in PGMs. Each node is a *cluster* of variables and undirected edges connect clusters that have a non-empty intersection of variables. Performing variable elimination defines the structure of the cluster graph. In VE, once a variable is eliminated, it doesn't appear in any computations, so the cluster graph induced by variable elimination is a *tree*. The order of VE defines a direction to the flow of messages between clusters, hence we can define a root. If cluster C_i is on the path from C_j to the root, then C_i is upstream from C_j and C_j is downstream from C_i. So the tree defined by VE defines what you call the *running intersection property*: Let T be a cluster tree over a set of factors Φ. Its nodes and edges are defined as V_T and E_T. The tree T has **running intersection property** whenever there is a variable X such that $X \in C_i$ and $X \in C_j$, then X is in every cluster in the (unique) path in T between C_i and C_j

This makes sense since in VE a variable appears in every factor from the time its first multiplied in (by a factor whose scope contains the variable) till the time is summed out. Here is another proposition from Koller and Friedman: Let T be a cluster tree induced by variable elimination over some set of factor Phi. Let C_i and C_j be neighboring clusters such that C_i passes a message τ_i to C_j. The scope of this message is the intersection of variables: $C_i \cap C_j$ So the running intersection property is quite helpful. Deriving from the RIP of cluster trees, we define a **clique tree**: Let Psi be a set of factors over X. A cluster tree over Psi satisfying the *running intersection property* is a *clique tree* (also called a junction tree or join tree). In the case of a clique tree, the clusters are also called cliques.

4.2 Variable Elimination and Clique Trees

Recall again in each step in VE a factor ψ_i is created by multiplying together factors and a variable is eliminated from ψ_i to create a new factor τ_i. This process is continued till the algorithm is finished. The generation of factors can be seen as *message passing* where a factor ψ_i takes incoming message τ_j generated by factors ψ_j, then generates its own message τ_i which in turn is passed onto another factor ψ_l. Figure 10.1 on page 346 shows a *cluster graph* , which is a data structure for message passing. Each node in the cluster graph are a set of variables and whose edges have variable scopes with a non-empty intersection. The cluster graph is a more general data structure than a *clique tree*, defined later in chapter 10, but the cluster graph is again discussed in chapter 11.

A cluster graph U is a for a set of factors Φ over X is an undirected graph whose nodes are associated with a subset $C_i \subseteq X$ The cluster graph must be *family preserving* such that each factor $\phi \in \Phi$ is associated with a cluster C_i, denoted as $\alpha(\phi)$, such that the $Scope[\phi] \subseteq C_i$ Each edge between clusters C_i and C_j is associated with the sepset $S_{i,j} \subseteq C_i \cap C_j$.

An execution of variable elimination defines a cluster graph Compare figure 10.1 on page 346 to the elimination process in table 9.1 on page 302. When an intermediate factor τ_i is generated, its used at most only once: ϕ_i is used to create ψ_j, its removed front the set of factors Φ, and cannot be used again. The cluster graph induced by VE is a tree. The tree defined by an execution of variable elimination satisfies the *running intersection property (RIP)*: Let T be a cluster tree over a set of factors Ψ. T has the RIP whenever there is a variable X such that $X \in C_i$ and $X \in C_j$ then X is in every cluster in the unique path between T between C_i and C_j Now when in a cluster tree induced by a variable elimination algorithm over a set of factors Φ, a cluster C_i passes a message τ_i to cluster C_j, the scope of the message is the intersection between C_i and $C_j : C_i \cap C_j$. If cluster tree over Φ that satisfies the RIP property is called a *clique tree* (also known as a junction tree or join tree). In a clique tree the clusters are called cliques.

4.3 Sum Product Message Passing

An execution of VE results in a clique tree. But you can start with a clique tree and use it as a data structure to perform variable elimination. The same clique tree can be used multiple times for different executions of VE. So given a tree that satisfies family preservation and the RIP property, you can do can use it in several different ways to do inference with graphical models. The clique tree can be used as a data structure for caching computations so you can do multiple variable eliminations rather than performing VE separately. Using the CT, each clique takes incoming messages and passes outgoing messages to another clique.

Step 1: Calculate initial potentials by multiplying factors assigned to a clique

A clique can only send messages once its received all its incoming messages and multiplied it with its own initial potential. In the clique tree message passing algorithm, a node in the CT is selected as a root. Then for each of the other nodes, define its neighbors on the path to the root. Each clique performs its message passing computation and sends its message to an upstream neighbor. The passing proceeds up the tree culminating at the root. When the root gets all its messages, it multiples them by its initial potential producing the final belief. Then from the final belief you can extract out your query nodes by summing out variables from the final belief factor.

4.4 Clique Tree Calibration

The clique tree can be used to calculate the probability of any variable in X. In many cases we want to calculate the probability of a large number of variables. (e.g. medical diagnosis of several possible diseases compatible with the same set of symptoms). Also we we would like to calculate the probability of unobserved variables and their parents when learning from Bayesian networks.
The clique tree MP algorithm can compute the probability of any variable in X. But what if you have several variables you want measure? You can do inference on each variable separately. Or you can run the algorithm making each node in the CT the root. But there is a better way! A technique called *sum-product belief propagation* can asynchronously perform inference.

Given a CT , T, a clique C_i is ready to transmit its message to a neighbor C_j once C_i has all its messages from its neighbors except C_j BP on the CT consists of one upward pass by picking a root and passing a message up to the root. When this finished, it can send messages to all its children (till all the leaves are reached). This second part is the downward pass. The root then is the first clique that gets all its messages from its neighbors. At the end of the algorithm, you can calculate final beliefs for *all* cliques in the tree. The cost of this inference is $2c$ versus Kc where K is the total number cliques.

Now at the end of this 2 pass calibration, if a variable X appears in two cliques, the cliques should agree on the marginal probability of the variable. A CT is calibrated if fore every pair of neighbors, the cliques are agree on septet beliefs. On page 357 there is algorithm pseudo-code for calibrating a clique tree (i.e. sum-product belief propogation). The main advantage of sum-product clique tree calibration algorithm is it computes the posterior probability of all variables using only twice the computation of the upward pass of the same tree. For a clique tree, there are two edges associated with it: incoming and outgoing. So if there are c cliques, there are then $c - 1$ edges in the CT, so we have $2(c - 1)$ edges to calculated. Now if if we doing a separate calculation for each variable $2c$ and Kc if we treat each of the K cliques as the root. In general, the clique tree algorithm is the best way to calculate posterior probability of multiple query variables.
Based on the the above section on Clique Tree calibration, we can use the clique tree sum product belief propagation as more general and efficient data structure to perform inference. Variable elimination are at the heart of both methods. But as far as I know, to compute an individual variable posterior genotype, you would have to employ VE separately each time. In the process, you would re-compute terms. Defining a clique tree first, then performing calibration, would be more efficient as you are minimizing the number of redundant computations. So using clique trees is the best way to calculate posterior probability of multiple query variables.