CS838-1 Advanced NLP: Inference in Graphical Models

Xiaojin Zhu

2007
Send comments to jerryzhu@cs.wisc.edu

"Inference" is the problem of computing the posterior distribution of hidden nodes given observed nodes in a graphical model. In particular, we are interested in the marginal distribution of each hidden node. The graphical model can be directed or undirected.

A graphical model defines a joint distribution $p(x_{1:N})$. Let's assume there is no observed node, and we want the marginal $p(x_n)$ on node n. By definition,

$$p(x_n) = \sum_{x_1} \dots \sum_{x_{n-1}} \sum_{x_{n+1}} \dots \sum_{x_N} p(x_{1:N}). \tag{1}$$

However, there are an exponential number of terms! This naive approach, although correct in theory, will not work in practice.

We can take advantage of the graph structure (which specifies conditional independence relations among nodes) to greatly speed up inference. There are several techniques, include variable elimination, junction tree and the sum-product algorithm. We focus on the sum-product algorithm because it is widely used in practice.

1 Factor Graph

It is convenient to introduce *factor graph*, which unifies directed and undirected graph with the same representation. The joint probability is written as a product of factors $f_s(x_s)$, where x_s is the set of nodes involved in the factor,

$$p(x_{1:N}) = \prod_s f_s(x_s). \tag{2}$$

In directed graph, the factors can be local conditional distributions of each node. In undirected graph, the factors can be the potential functions, with the normalization term $1/Z$ being a special factor with zero nodes.

There are two types of nodes in a factor graph: the set of original nodes, and the set of factors, forming a bipartite graph.

2 The Sum-Product Algorithm

The sum-product algorithm is also known as belief propagation. It can compute the marginals of all nodes efficiently and exactly, if the factor graph is a tree (i.e., if there is only one path between any two nodes). The algorithm involves passing *messages* on the factor graph. A message is a vector of length K, where K is the number of possible states a node can take. It is an unnormalized 'belief'.

There are two types of messages:

1. A message from a factor node f to a variable node x, denoted as $\mu_{f \to x}$. Note it is a vector of length K, and we write the x-th element (a slight abuse of notation, $x = 1 \dots K$) as $\mu_{f \to x}(x)$.

2. A message from a variable node x to a factor node f, denoted as $\mu_{x \to f}$. It is also a vector of length K, with elements $\mu_{x \to f}(x)$.

The messages are defined recursively. In particular, consider a factor f_s that involves (connects to) a particular variable x. Denote the other variables involved in f_s by $x_{1:M}$. We have

$$\mu_{f_s \to x}(x) = \sum_{x_1} \dots \sum_{x_M} f_s(x, x_1, \dots, x_M) \prod_{m=1}^{M} \mu_{x_m \to f_s}(x_m), \tag{3}$$

and

$$\mu_{x_m \to f_s}(x_m) = \prod_{f \in ne(x_m) \backslash f_s} \mu_{f \to x_m}(x_m), \tag{4}$$

where $ne(x_m) \backslash f_s$ is the set of factors connected to x_m, excluding f_s.

The recursion is initialized as follows. Since we assumed the factor graph is a tree, we can pick an arbitrary node and call it the root. This defines all the leaf nodes, which we start all the messages. If a leaf is a variable node x, its message to a factor node f is

$$\mu_{x \to f}(x) = 1. \tag{5}$$

If a leaf is a factor node f, its message to a variable node x is

$$\mu_{f \to x}(x) = f(x). \tag{6}$$

A node (factor or variable) can send out a message if all the necessary incoming messages have arrived. This will eventually happen for tree structured factor graph.

Once all messages have been sent, one can compute the desired marginal probabilities as

$$p(x) \propto \prod_{f \in ne(x)} \mu_{f \to x}(x). \tag{7}$$

One can also compute the marginal of the *set of* variables x_s involved in a factor f_s

$$p(x_s) \propto f_s(x_s) \prod_{x \in ne(f)} \mu_{x \to f}(x). \tag{8}$$

If a variable x is observed $x = v$, it is a constant in all neighboring factors. Its message $\mu_{x \to f}(x)$ is set to zero for all $x \neq v$. Alternatively, we can eliminate observed nodes by absorbing them (with their observed constant values) into the corresponding factors. Let X_o be the set of observed variables. With this modification, we get the *joint* probability (NB. not the conditional $p(x|X_o)$) of a single node x and all the observed nodes when we multiply the incoming messages to x:

$$p(x, X_o) \propto \prod_{f \in ne(x)} \mu_{f \to x}(x). \tag{9}$$

The conditional is easily obtained by normalization afterwards

$$p(x|X_o) = \frac{p(x, X_o)}{\sum_{x'} p(x', X_o)}. \tag{10}$$

When the factor graph contains loops (not a tree), there is no longer guarantee that the algorithm will even converge. However, people find in practice that it still works quite well. This way of applying the sum-product algorithm is known as loopy belief propagation (loopy BP).

3 The Max-Sum Algorithm

Sometimes it is important to know the 'best states' $z_{1:N}$ corresponding to the observation $x_{1:N}$. There are at least two senses of 'best':

1. With the sum-product algorithm we can compute the marginal $p(z_n|x_{1:N})$ for each node. We can define 'best' to be the state with the highest marginal probability

   $$z_n^* = \arg\max_k p(z_n = k|x_{1:N}), \tag{11}$$

   and we will have a set of most likely states $z_{1:N}^*$. Each time step is the best individual, however $z_{1:N}^*$ as a whole may not be the most likely *state configuration*. In fact it can even be an invalid configuration with zero probability, depending on the model!

2. The alternative is to find

   $$z_{1:N}^* = \arg\max_{z_{1:N}} p(z_{1:N}|x_{1:N}). \tag{12}$$

   It finds the most likely *state configuration* as a whole. The max-sum algorithm addresses this problem efficiently.

We first modify the sum-product algorithm to obtain the *max-product* algorithm. The idea is very simple: replace $\sum$ with max in the messages. In fact only factor-to-variable messages are affected:

$$\begin{aligned}
\mu_{f_s \to x}(x) &= \max_{x_1} \ldots \max_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{m=1}^{M} \mu_{x_m \to f_s}(x_m) && (13)\\
\mu_{x_m \to f_s}(x_m) &= \prod_{f \in ne(x_m) \setminus f_s} \mu_{f \to x_m}(x_m) && (14)\\
\mu_{x_{\text{leaf}} \to f}(x) &= 1 && (15)\\
\mu_{f_{\text{leaf}} \to x}(x) &= f(x). && (16)
\end{aligned}$$

As before, we specify an arbitrary variable node x as the root, and pass messages from leaves until they reach the root. At the root, we multiply all incoming messages to obtain the maximum probability

$$p^{\max} = \max_x \left(\prod_{f \in ne(x)} \mu_{f \to x}(x) \right). \qquad (17)$$

This is the probability of the most likely state configuration. But we have not specified how to identify the configuration itself. Note unlike the sum-product algorithm, we do not pass messages back from root to leaves. Instead, we keep *back pointers* whenever we perform the max operation. In particular, when we create the message

$$\mu_{f_s \to x}(x) = \max_{x_1} \ldots \max_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{m=1}^{M} \mu_{x_m \to f_s}(x_m), \qquad (18)$$

for each x value, we separately create M pointers back to the values of $x_1, \ldots, x_M$ that achieve the maximum. When at the root, we back trace the pointers from the value x that achieve $p^{\max}$. This eventually gives us the complete most likely state configuration.

The *max-sum algorithm* is equivalent to the max-product algorithm, but work in log space, to avoid potential underflow problem. In particular, the messages are

$$\begin{aligned}
\mu_{f_s \to x}(x) &= \max_{x_1} \ldots \max_{x_M} \log f_s(x, x_1, \ldots, x_M) + \sum_{m=1}^{M} \mu_{x_m \to f_s}(x_m) && (19)\\
\mu_{x_m \to f_s}(x_m) &= \sum_{f \in ne(x_m) \setminus f_s} \mu_{f \to x_m}(x_m) && (20)\\
\mu_{x_{\text{leaf}} \to f}(x) &= 0 && (21)\\
\mu_{f_{\text{leaf}} \to x}(x) &= \log f(x). && (22)
\end{aligned}$$

When at the root,

$$\log p^{\max} = \max_x \left(\sum_{f \in ne(x)} \mu_{f \to x}(x) \right). \tag{23}$$

The back pointers are the same. The max-product or max-sum algorithm, when applied to HMMs, is known as the Viterbi algorithm.