Probabilistic Graphical Models Programming Assignment #5: Approximate Inference

1 Introduction

Last week, we focused on implementing exact inference methods. Unfortunately, sometimes performing exact inference is intractable (exact inference in general networks is NP-hard). Luckily, there are a number of approximate inference methods and in this programming assignment we will investigate two of them: loopy belief propagation and Markov chain Monte Carlo (MCMC).

As you develop and test your code, you will run tests on a simple pairwise Markov network that we have provided. This Markov net is a 4 x 4 grid network of binary variables, parameterized by a set of singleton factors over each variable and a set of pairwise factors over each edge in the grid. This network is created by the function **ConstructToyNetwork.m**, and in this assignment, you will change some of its parameters and observe the effect this has on different inference techniques.

Before starting on the programming portion of this assignment, it might be useful to look at the section at the end entitled "Starter Code Reference". There is also a quiz associated with this assignment, due simultaneously with the programming portion. Access the quiz online at Coursera under the "Assignment Questions" section.

2 Loopy Belief Propagation

The first approximate inference method that we will implement is loopy belief propagation (LBP). This algorithm takes a cluster graph, a set of factors, and a list of evidence, and outputs the *approximate* posterior marginals for each variable. *The message passing framework you implemented for clique trees in PA4 should directly generalize to the case of cluster graphs,* so you should have to write relatively little further code for LBP. In LBP, we do not have a root and a specific up-down message passing order relative to that root. In particular, we can order messages by an arbitrary criterion. However, we want you to experiment with two different message passing orders.

You will first implement a naive message passing order that blindly iterates through all messages without considering other criteria. You will subsequently experiment with alternative message passing orders and analyze their impact on both the convergence of LBP and the values of the marginals at convergence.

- **NaiveGetNextClusters.m (5 points)** — This function should find a clique that is ready to transmit a message to its neighbor. It should return the indices of the two cliques that the message is ready to be passed between. In this naive implementation we simply iterate over the cluster pairs. Details on this ordering are given within the code file.

Now we can begin the message passing process to calibrate the cluster, but we'll need some infrastructure to make this work. For example, we need a criterion to tell us when we've converged and we need to create a cluster graph in order to run LBP.

- **CreateClusterGraph.m (5 points)** – Given a list of factors, this function will create a Bethe cluster graph with nodes representing single variable clusters and pairwise clusters

(see the lecture slides on Belief Propagation on the structure of a Bethe cluster graph). Note that the ClusterGraph data structure is essentially the same as the CliqueTree data structure from PA4 (the only difference is that cliqueList is renamed clusterList).

- **CheckConvergence.m (2 points)** – Given your current set of messages and the set of messages that immediately preceded each of our current messages, this function will determine if the cluster graph has converged, returning 1 if so and 0 otherwise. In this case, we say that the messages have converged if no messages have changed significantly, where "significantly" means by a value of over 10^{-6} in any entry (note: for theoretic calibration you would actually wait until the difference was 0).
- **ClusterGraphCalibrate.m (10 points)** — This function should perform loopy belief propagation over a given cluster graph. It should return an array of final beliefs (factors) for each cluster.

We are now ready to bring together all the steps described above to compute approximate marginals for the variables in our network. You should call the appropriate functions in the file **ComputeApproxMarginalsBP.m** to run approximate inference.

- **ComputeApproxMarginalsBP.m (5 points)** – This function should take a set of initial factors and vector of evidence and compute the approximate marginal probability distribution for each variable in the network. You should be able to reuse much of the code you wrote for **ComputeExactMarginalsBP.m** for assignment 4 in this function.

2.1 LBP Investigation Questions

After completing this part of the assignment, you should be able to answer questions 1-3 in the PA5 quiz.

2.2 Comments on Improving Message Passing

Consider the analysis of the naive message passing order that you performed in the quiz – can you think of ways to improve the message passing order? Perhaps you could use the magnitude of the changes in the marginals before and after message passing to help indicate which messages you should prioritize passing. There exist better message passing schedules that improve the convergence of the algorithm, but that is beyond the scope of this assignment. For more information, we refer the interested reader to [Elidan, McGraw, and Koller 06] [1].

3 MCMC

Now that we've looked at belief propagation, we will investigate a second class of inference methods based on Markov Chain Monte Carlo (MCMC) sampling. As a reminder, a Markov chain defines a transition model $T(x \rightarrow x')$ between different states x and x'. When used for probabilistic inference, each state is an assignment to all unobserved variables in the joint distribution. By running the Markov Chain with some transition model, we generate a sequence of joint variable assignments. Ideally, after running the chain for some time (the "mixing time"), sampling from this sequence (i.e., sampling from the stationary distribution of the Markov Chain) should closely approximate sampling from the actual posterior joint distribution, which is the

[1] `http://www.robotics.stanford.edu/~koller/Papers/Elidan+al:UAI06.pdf`

quantity we are trying to estimate. Obviously, not every transition model satisfies this criterion. In this section, however, we'll investigate Gibbs and Metropolis-Hastings sampling, two MCMC methods that allow us to construct Markov chains whose stationary distributions are equivalent to the joint posterior distribution.

In any MCMC method, we initialize a Markov chain with some initial assignment x_0. At each iteration t, a new state, x_{t+1} is sampled from the transition model. The chain is run for some number of iterations, over which a subset of samples is collected (remember that each sample is an assignment to all variables in the distribution we are trying to estimate). The collected samples are then used to estimate the posterior marginals of variables.

Using Gibbs and Metrpolis-Hastings ensures that the stationary distribution of the chain is equivalent to the actual posterior distribution of the network, but two other critical issues affect the utility of a Markov chain: its susceptibility to local optima and the rate at which its samples "forget" the initial state and converge to its stationary distribution (colloquially, we refer to this convergence rate as the "mixing time"). For example, if the stationary distribution has two highly peaked modes that are very different and the MCMC transition probability only allows local moves in the state space, it is likely that the Markov chain will get stuck near one of the modes for a long time. Or, if samples are collected before a chain has mixed, then the distribution from which they are drawn will be biased toward the initial state and will not be a good approximation of the stationary distribution.

In the starter code, we provide you with a visualization function, **VisualizeMCMCMarginals.m**, that allows you to analyze a Markov chain to estimate properties such as mixing time and whether or not it is getting stuck in a local optimum. See the description of the code below for more details on this function.

3.1 Gibbs

Recall that the Gibbs chain is a Markov chain where the transition probability $T(x \rightarrow x')$ is defined as follows. We iterate over the variables in some fixed order, say $X_1, ..., X_n$. For the variable X_i, we sample a new value from $P(X_i|x_{-i})$ (which is just $P(X_i|\text{MarkovBlanket}(X_i))$), and update its new value. Note that the terms on the right-hand-side of the conditioning bar use the currently sampled assignment to the variables $X_1, ..., X_{i-1}$. Once we sample a new value for each of $X_1, ..., X_n$, the result is our new sample x'. Our first task is thus to implement a function that computes and samples from $P(X_i|\text{MarkovBlanket}(X_i))$. To do this, we'll implement a helper function to produce sampling distributions. You will then use this function as a transition probability for MCMC sampling. (Recall that a Markov Blanket of a node X is the set of nodes Y such that X_i is independent of all other nodes given Y; it thus consists of X's parents, children, and children's parents.)

- **BlockLogDistribution.m: (5 points)** – This function will produce the sampling distribution used in Gibbs sampling (and one of the versions of Metropolis-Hastings). It takes as input a set of variables $\mathbf{X}_I$ and an assignment $\mathbf{x}$ to all variables in the network, and returns the distribution associated with sampling $\mathbf{X}_I$ as a monolithic block (i.e., the variables are constrained to take on the same value) given the joint assignment to all other variables in the network. That is, for each value l, we compute the (unnormalized) probability $\tilde{P}(\mathbf{X}_I = l|\mathbf{x}_{-I})$ where $\mathbf{X}_I = l$ is shorthand for the statement"$X_i = l$ for all $X_i \in \mathbf{X}_I$" and $\mathbf{x}_{-I}$ is the assignment to all other variables in the network. Note: In this function, your solution should only contain one for-loop; otherwise your function will probably be very slow and hang the submit script for a long time. You can try using multiple for-loops initially and then figure out how to replace for-loops with matrix operations. Also note

that the distribution should be returned in log-space to avoid numerical underflow issues.

- **GibbsTrans.m (5 points)** – This function defines the transition process in the Gibbs chain as described above (i.e., we iteratively resample X_i from $P(X_i|\text{MarkovBlanket}(X_i))$ for each i) . It should call BlockLogDistribution to sample a value for each variable in the network.

3.1.1 Running Gibbs Sampling and Questions

Now that our first transition process has been defined, we need to implement the top-level function for running different variants of MCMC.

- **MCMCInference.m PART 1 (3 points)**– This function defines the general framework for conducting MCMC inference. It takes as input a probabilistic graphical model (e.g., as returned by ConstructToyNetwork.m), a set of factors, a list of evidence, the name of the MCMC transition to use, and other MCMC parameters such as the target burn-in time and number of samples to collect. As a first step, you only need to implement the logic that transitions the Markov chain to its next state and records the sample.

With this complete, we can now conduct Gibbs sampling on a probabilistic network. You should now answer quiz question 4, which deals with the behavior of Gibbs sampling.

3.2 Metropolis-Hastings

Metropolis-Hastings is an MCMC framework that defines the Markov chain transition in terms of a proposal distribution $Q(x \to x')$ and an acceptance probability $A(x \to x')$. Proposed moves in the state space are drawn from the proposal distribution and those proposals are then accepted based on the acceptance probability (if a move is rejected, the state remains unchanged). The proposal distribution and acceptance probability must satisfy the detailed balance equation in order to generate the correct stationary distribution. Given a proposal distribution, it turns out that a satisfying acceptance probability is given as follows (where π is the stationary distribution):

$$A(x \to x') = \min\left[1, \frac{\pi(x')Q(x' \to x)}{\pi(x)Q(x \to x')}\right]$$

Note that this equation means that we can pick an arbitrary proposal distribution and construct the acceptance distribution that yields a Markov chain with the correct stationary distribution. However, if the acceptance probabilities are extremely low, this implies the chain will take longer to mix. Thus, using a good proposal distribution is crucial.

In this section of the assignment, you will implement a general Metropolis-Hastings framework that is capable of utilizing different proposal distributions, specifically the uniform distribution and the Swendsen-Wang distribution (described later). We will provide you with the implementations of these proposal distributions and you will need to compute the correct acceptance probability. Furthermore, you will study the relative merits of each proposal type. To start, let's implement the uniform proposal distribution:

- **MHUniformTrans.m (5 points)**– This function defines the transition process associated with the uniform proposal distribution in Metropolis-Hastings. You should fill in the code to compute the correct acceptance probability.

Now that we have that as a baseline, let's move onto Swendsen-Wang. Swendsen-Wang was designed to propose more global moves in the MCMC state space for pairwise Markov networks such as Ising models where adjacent variables are likely to share the same value. Fundamentally, it is a graph node clustering algorithm. Given a pairwise Markov network and a current joint assignment x to all variables, it generates clusters as follows (see the figure):

1. First it eliminates all edges in the Markov network between variables that have different values in x.
2. For each remaining edge $\{i, j\}$, it "activates" the edge with some probability $q_{i,j}$ (which can depend on the variables i and j but not on their values in x).
3. It then computes the connected components of the graph over the activated edges.
4. It selects one connected component, $\mathbf{Y}$, uniformly at random from all connected components. Note that all nodes in $\mathbf{Y}$ will have the same label l.
5. We then (randomly) choose a new value l' that will be taken by all nodes in this connected component. These variables are then updated in the joint assignment to produce the new assignment x'. In other words, the new assignment x' is the same as x, except that the variables in $\mathbf{Y}$ are all labeled l' instead of l. Note that this proposed move can flip a large number of variables at the same time and thus can take much larger steps in the state space than Gibbs sampling.

Let $q(Y|x)$ be the probability that a set $\mathbf{Y}$ is selected to be updated using this procedure. It is possible to show that

$$\frac{q(\mathbf{Y}|x')}{q(\mathbf{Y}|x)} = \frac{\prod_{(i,j)\in\mathcal{E}(\mathbf{Y},(\mathbf{X}'_{\mathbf{l}'}-\mathbf{Y}))}(1-q_{i,j})}{\prod_{(i,j)\in\mathcal{E}(\mathbf{Y},(\mathbf{X}_{\mathbf{l}}-\mathbf{Y}))}(1-q_{i,j})} \tag{1}$$

where: $\mathbf{X_l}$ is the set of vertices with label l in x, $\mathbf{X}'_{\mathbf{l}'}$ is the set of vertices with label l' in $\mathbf{x}'$; and where $\mathcal{E}(\mathbf{Y}, \mathbf{Z})$ (between two disjoint sets $\mathbf{Y}, \mathbf{Z}$) is the set of edges connecting nodes in $\mathbf{Y}$ to nodes in $\mathbf{Z}$. (NOTE: The log of the quotient in equation 1 is called log_QY_ratio in the code.) Then we have that

$$\frac{\mathcal{T}^Q(x' \to x)}{\mathcal{T}^Q(x \to x')} = \frac{q(\mathbf{Y}|x')}{q(\mathbf{Y}|x)} \frac{R(\mathbf{Y} = l|x'_{-\mathbf{Y}})}{R(\mathbf{Y} = l'|x_{-\mathbf{Y}})} \tag{2}$$

where: $\mathcal{T}$ is the proposal distribution and $R(\mathbf{Y} = l'|x_{-\mathbf{Y}})$ is a distribution specified by you for choosing the label l' for $\mathbf{Y}$ given $x_{-\mathbf{Y}}$ (the assignment to all variables outside of $\mathbf{Y}$). Note that $x_{-\mathbf{Y}} = x'_{-\mathbf{Y}}$.

In this assignment, the code for generating a Swendsen-Wang proposal is given to you, but you will have to compute the acceptance probability (hint: use the equations above) and use that to define the sampling process for the Markov chain. You will implement 2 variants that experiment with different parameters for the proposal distribution. In particular, you will change the value of the $q_{i,j}$'s and $R(\mathbf{Y} = l|x_{-\mathbf{Y}})$. The two variants are as follows:

1. Set the $q_{i,j}$'s to be uniformly 0.5, and set the distribution R to be uniform.
2. Set R to be the block-sampling distribution (as defined in BlockLogDistribution.m) for sampling a new label and make $q_{i,j}$ dependent on the pairwise factor $F_{i,j}$ between i and j.

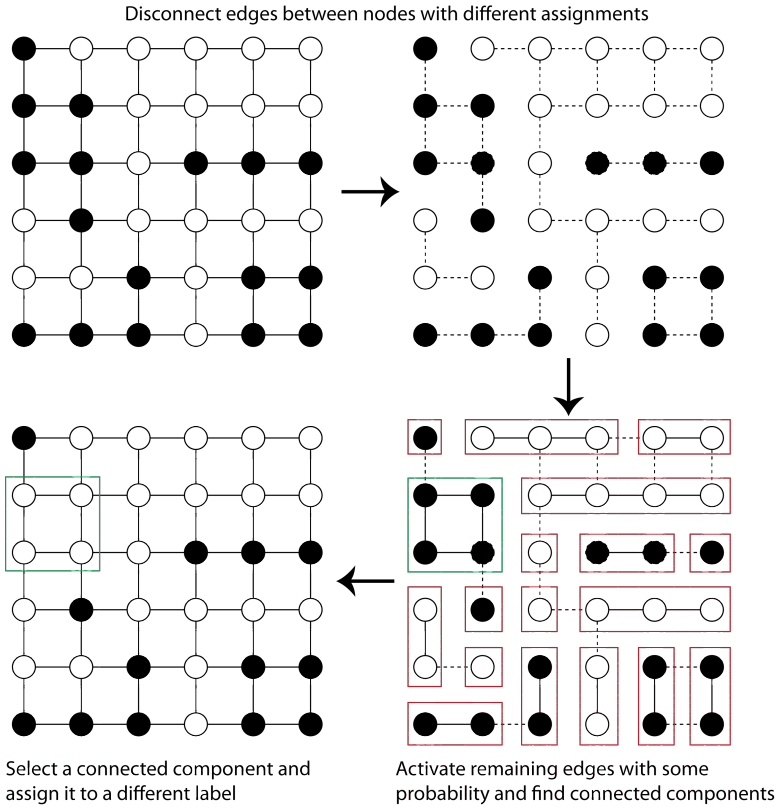

Figure 1: Visualization of Swendsen-Wang Procedure

In particular, set

$$q_{i,j} := \frac{\sum_u F_{i,j}(u,u)}{\sum_{u,v} F_{i,j}(u,v)}$$

- **MHSWTrans.m (Variant 1) (3 points)** – This function defines the transition process associated with the Swendsen-Wang proposal distribution in Metropolis-Hastings. You should fill in the code to compute the proposal distribution values and then use these to compute the acceptance probability. Implement the first variant for this test.
- **MHSWTrans.m (Variant 2) (3 points)** – Now implement the second variant of SW. Note: the first variant should still function after the second variant has been implemented.

With Swendsen-Wang, we will need to compute the values of our $q_{i,j}$'s, so we must update our inference function:

- **MCMCInference.m PART 2 (4 points)**– Fill in this function to run our Swendsen-Wang variants in addition to Gibbs. Your task here is to implement the calculations of the $q_{i,j}$'s for both variants of Swendsen-Wang in this function. (The reason that this is done here and not in MHSWTrans.m is to improve efficiency.)

Now that we've finished implementing all of these functions, compare these inference algorithms by answering the remaining questions in the quiz.

4 Concluding comments

You've now implemented a full suite of algorithms for exact and approximate inference. Given a representation of a probabilistic graphical model, you can use the tools you've built to answer queries about variables in the network given evidence. In the next assignment, we'll use these algorithms in a real-world setting. Later on in the class, we'll look at how to learn a probabilistic graphical model itself from data.

5 Starter Code Reference

A few methods you may find useful:

1. **exampleIOPA5.mat:** Mat-file containing example input and output corresponding to the 13 preliminary tests for PA5. For argument j of the function call in part i, you should use **exampleINPUT.t**$\#_i$**a**$\#_j$ (replacing the $\#_i$ with i). If there are multiple function calls in one test (for example, we iterate over multiple inputs) then for iteration k you should reference **exampleINPUT.t**$\#_i$**a**$\#_j$.$\{\#_k\}$. For output, look at **exampleOUTPUT.t**$\#_i$ for the output to part i. If there are multiple outputs or iterations, the functionality is the same as for the input example.

2. **ConstructToyNetwork.m:** Function that constructs a toy pairwise Markov Network that you will use in some of the questions. This function accepts two arguments, an "on-diagonal" weight and an "off-diagonal" weight. These refer to the weights in our image network's pairwise factors, where on-diagonal refers to the weight associated with adjacent nodes agreeing and off-diagonal corresponds to having different assignments. The output network will be a 4 x 4 grid.

3. **ConstructRandNetwork.m:** Function that constructs a randomized pairwise Markov Network that you will use in some of the questions. The functionality is essentially the same as **ConstructToyNetwork.m**.

4. **VisualizeMCMCMarginals.m:** This displays two things. First, it displays a plot of the log-likelihood of each sample over time. Recall that in quickly mixing chains, this value should increase until it roughly converges to some constant log-likelihood, where it should remain (with some occasional jumps down). This function also visualizes the estimate of the marginal distributions of specified variables in the network as estimated by a string of samples obtained from MCMC over time. In particular, it takes a fixed-window subset of the samples around a given iteration t and uses these to compute a sliding-window average of the estimated marginal(s). It then plots the sliding-window average of each value in the marginal as its estimate progresses over time. The function also can accept samples from more than one MCMC run, in which case the marginal values that correspond to one another are plotted in the same color, allowing you to determine whether the different MCMC runs are converging to the same result. This is particularly helpful if you are trying to identify whether the chain is susceptible to local optima (in which case, different runs will converge to different marginals) or whether the chain has mixed by a given iteration.

5. **TestToy.m:** This function constructs a toy image network where each variable is a binary pixel that can take on a value of 1 or 2. This network is a pairwise Markov net structured as a 4 x 4 grid. The parameterization for this network can be found in TestToy.m and you will tune the parameters of the pairwise factors to study the corresponding behavior of

different inference techniques. You can visualize the marginal strengths of this toy image by calling the function VisualizeToyImageMarginals.m, which will display the marginals as a gray-scale image, where the intensity of each pixel represents the probability that that pixel has the label 2.

6. **VisualizeToyImageMarginals.m:** Visualizes the marginals of the variables in the toy network on a 4x4 grid. We have provided a lot of the infrastructure code for you so that you can concentrate on the details of the inference algorithms.