

Inference and Representation, Fall 2017

Problem Set 3: Gibbs Sampling, Belief Propagation, Tree Structure Learning

You are allowed to use basic graph packages (e.g., for representing and working with undirected graphs, or for finding the maximum spanning tree), but are **not** permitted to use any machine learning, graphical models, or probabilistic inference packages.

1. Ising Model - Gibbs sampling ([1] Ex. 12.5.3(a)).

This problem considers an application of MCMC techniques to image analysis. Imagine a 2D image consisting of an $L \times L$ grid of black-or-white pixels. Let Y_j be the indicator of the j th pixel being white, for $j = 1, \dots, L^2$. Viewing the pixels as nodes in a network, the neighbors of a pixel are the pixels immediately above, below, to the left, and to the right (except for boundary cases).

Let $i \sim j$ stand for " i and j are neighbors". A commonly used model for the joint PMF of $Y = (Y_1, \dots, Y_{L^2})$ is

$$P(Y = y) \propto \exp(\beta \sum_{(i,j): i \sim j} I(y_i = y_j))$$

If β is positive, this says that neighboring pixels prefer to have the same color. The normalizing constant of this joint PMF is a sum over all 2^{L^2} possible configurations, so it may be very computationally difficult to obtain. This motivates the use of MCMC to simulate from the model.

Suppose that we wish to simulate random draws from the joint PMF of Y for a particular known value of β . Explain how we can do this using Gibbs sampling, cycling through the pixels one by one in a fixed order.

2. **Sum-product algorithm, Homework 1 in [4] adapted to Python.** We implement the sum-product variant of the belief propagation algorithm to compute marginals. To understand the details of the sum-product algorithm, we recommend Chapter 5 of Barber's "Bayesian Reasoning and Machine Learning", as well as "Factor Graphs and the Sum-Product Algorithm" by Kschischang, Frey, & Loeliger, IEEE Trans. Information Theory 47, pp. 498-519, 2001.

You must write your own novel implementation of the sum-product algorithm in Python, not copy code from other students or existing software packages.

We provided code implementing a data structure to store the graph adjacency structure, and numeric potential tables, defining any discrete factor graph. We also provided code that explicitly builds a table containing the probabilities of all joint configurations of the variables in a factor graph, and sums these probabilities to compute the marginal distribution of each variable. Such "brute force" inference code is of course inefficient, and will only be computationally tractable for small models.

We recommend (but do not require) that you use these same data structures for your own implementation of the sum-product algorithm, by implementing `run_loopy_bp_parallel` and `get_beliefs`. To gain intuition for the graph structure, examine the output of `make_debug_graph.ipynb`. Think of the code we provide as a starting point: you are welcome to create additional functions or data structures as needed.

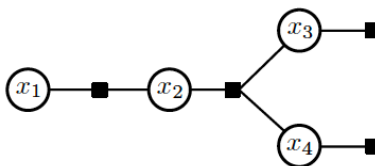


Figure 1: A tree-structured factor graph in which four factors link four random variables. Variable x_2 takes one of three discrete states, and the other three variables are binary. [4]

- (a) Implement the sum-product algorithm. Your code should support an arbitrary factor graph linking a collection of discrete random variables. Use a parallel message update schedule, in which all factor-to-variable messages are updated given the current variable-to-factor messages, and then all variable-to-factor messages given the current factor-to-variable messages. Initialize by setting the variable-to-factor messages to equal 1 for all states. Be careful to normalize messages to avoid numerical underflow.
 - (b) Consider the four-node, tree-structured factor graph illustrated in Figure 1 with binary variables. Numeric values for the potential functions are defined in `make_debug_graph.ipynb`. Run your implementation of the sum-product algorithm on this graph, and report the marginal distributions it computes. Verify that these are consistent with the results of `marg_brute_force.ipynb`.
3. **LDPC, Homework 2 in [4] adapted to Python.** We begin by designing algorithms for reliable communication in the presence of noise. We focus on error correcting codes based on highly sparse, low density parity check (LDPC) matrices, and use the sum-product variant of the loopy belief propagation (BP) algorithm to estimate partially corrupted message bits. For background information on LDPC codes, see Chap. 47 of MacKay’s Information Theory, Inference, and Learning Algorithms, which is freely available online: <http://www.inference.phy.cam.ac.uk/mackay/itila/>.

We consider rate $1/2$ error correcting codes, which encode N message bits using a $2N$ -bit codeword. LDPC codes are specified by a $N \times 2N$ binary parity check matrix H , whose columns correspond to codeword bits, and rows to parity check constraints. We define $H_{ij} = 1$ if parity check i depends on codeword bit j , and $H_{ij} = 0$ otherwise. Valid codewords are those for which the sum of the bits connected to each parity check, as indicated by H , equals zero in modulo-2 addition (i.e., the number of “active” bits must be even). Equivalently, the modulo-2 product of the parity check matrix with the $2N$ -bit codeword vector must equal a N -bit vector of zeros. As illustrated in Figure 2, we can visualize these parity check constraints via a corresponding factor graph. The parity check matrix H can then be thought of as an adjacency matrix, where rows correspond to factor (parity) nodes, columns to variable (codeword bit) nodes, and ones to edges linking actors to variables.

- (a) Implement code that, given an arbitrary parity check matrix H , constructs a corresponding factor graph. The parity check factors should evaluate to 1 if an even number of adjacent bits are active (equal 1), and 0 otherwise. Your factor graph representation should interface with your implementation of the sum-product algorithm from the previous problem. Define a small test case, and verify that your graphical model assigns zero probability to invalid codewords.

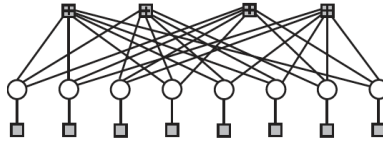


Figure 2: A factor graph representation of a LDPC code linking four factor (parity constraint) nodes to eight variable (message bit) nodes. The unary factors encode noisy observations of the message bits from the output of some communications channel. [4]

- (b) Load the $N = 128$ -bit LDPC code using the Python library `pyldpc` (for a tutorial see github.com/hichamjanati/pyldpc-tutos). To evaluate decoding performance, we assume that the all-zeros codeword is sent, which always satisfies any set of parity checks. Using the random module, simulate the output of a binary symmetric channel: each transmitted bit is flipped to its complement with error probability $\epsilon = 0.05$, and equal to the transmitted bit otherwise. Define unary factors for each variable node which equal $1 - \epsilon$ if that bit equals the "received" bit at the channel output, and ϵ otherwise. Run the sum-product algorithm for 50 iterations of a parallel message update schedule, initializing by setting all variable-to-factor messages to be constant. After the final iteration, plot the estimated posterior probability that each codeword bit equals one. If we decode by setting each bit to the maximum of its corresponding marginal, would we find the right codeword?
- (c) Repeat the experiment from part (b) for 10 random channel noise realizations with error probability $\epsilon = 0.06$. For each trial, run sum-product for 50 iterations. After each iteration, estimate the codeword by taking the maximum of each bit's marginal distribution, and evaluate the Hamming distance (number of differing bits) between the estimated and true (all-zeros) codeword. On a single plot, display 10 curves showing Hamming distance versus iteration for each Monte Carlo trial. Is BP a reliable decoding algorithm?
- (d) Repeat part (c) with two higher error probabilities, $\epsilon = 0.08$ and $\epsilon = 0.10$. Discuss any qualitative differences in the behavior of the loopy BP decoder.
- (e) For the LDPC codes we consider, we also define a corresponding $2N \times N$ generator matrix G . To encode an N -bit message vector we would like to transmit, we take the modulo-2 matrix product of the generator matrix with the message. The generator matrix has been constructed (via linear algebra over the finite field $\text{GF}(2)$) such that this product always produces a valid $2N$ -bit codeword. Geometrically, its columns are chosen to span the null space of H . We use a systematic encoding, in which the first N codeword bits are simply copies of the message bits. The problems below use precomputed $(G;H)$ pairs using the relevant functions in `pyldpc`. Generate the $N = 1600$ -bit LDPC code. Using this, we will replicate the visual decoding demonstration from MacKay's Fig. 47.5. Start by converting a 40×40 binary image to a 1600-bit message vector; you may use the logo image we provide, or create your own. Encode the message using the generator matrix G , and add noise with error probability $\epsilon = 0.06$. For this input, plot images showing the output of the sum-product decoder after 0, 1, 2, 3, 5, 10, 20, and 30 iterations. The `%` operator may be useful for computing modulo-2 sums. You can use the numpy reshape function to easily convert between images and rasterized message vectors.
- (f) Repeat the previous part with a higher error probability of $\epsilon = 0.10$, and discuss

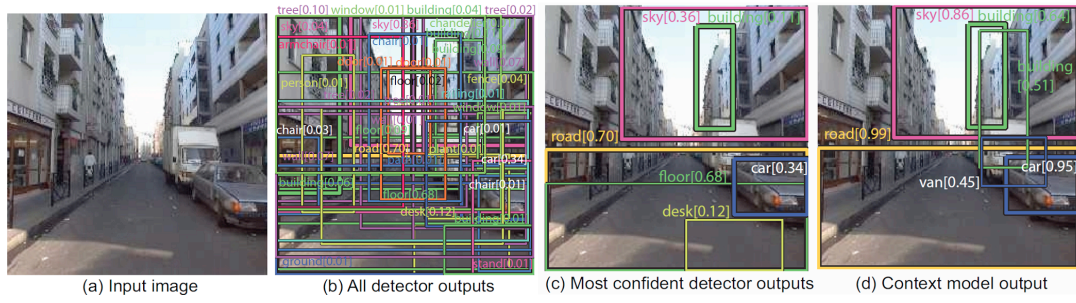


Figure 3: Using context within object detection for computer vision. [2]

differences.

4. **Chow-Liu algorithm.** When trying to do object detection from computer images, *context* can be very helpful. For example, if "car" and "road" are present in an image, then it is likely that "building" and "sky" are present as well (see Figure 3). In recent work, a tree-structured Markov random field (see Figure 4) was shown to be particularly useful for modeling the prior distribution of what objects are present in images and using this to improve object detection [2].

You will replicate some of the results from [2] (it is not necessary to read this paper to complete this assignment). Specifically, you will implement the Chow-Liu algorithm (1968) for maximum likelihood learning of tree-structured Markov random fields [3]. See also Murphy's book Section 26.3 for a brief overview (the Murphy book is available online for free for NYU students; see course website).

The goal of learning is to find the tree-structured distribution $p_T(\mathbf{x})$ that maximizes the log-likelihood of the training data $\mathcal{D} = \{\mathbf{x}\}$:

$$\max_T \max_{\theta_T} \sum_{\mathbf{x} \in \mathcal{D}} \log p_T(\mathbf{x}; \theta_T).$$

We will show in Lecture 9 that for a fixed structure T , the maximum likelihood parameters for a MRF have a property called *moment matching*, meaning that the learned distribution will have marginals $p_T(x_i, x_j)$ equal to the empirical marginals $\hat{p}(x_i, x_j)$ computed from the data \mathcal{D} , i.e. $\hat{p}(x_i, x_j) = \text{count}(x_i, x_j) / |\mathcal{D}|$ where $\text{count}(x_i, x_j)$ is the number of data points in \mathcal{D} with $X_i = x_i$ and $X_j = x_j$. Thus, using the factorization from Eq. (2) of question 4 of PS2, the learning task is reduced to solving

$$\max_T \sum_{\mathbf{x} \in \mathcal{D}} \log \left[\prod_{(i,j) \in T} \frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i) \hat{p}(x_j)} \prod_{j \in V} \hat{p}(x_j) \right].$$

We can simplify the quantity being maximized over T as follows (let $N = |\mathcal{D}|$):

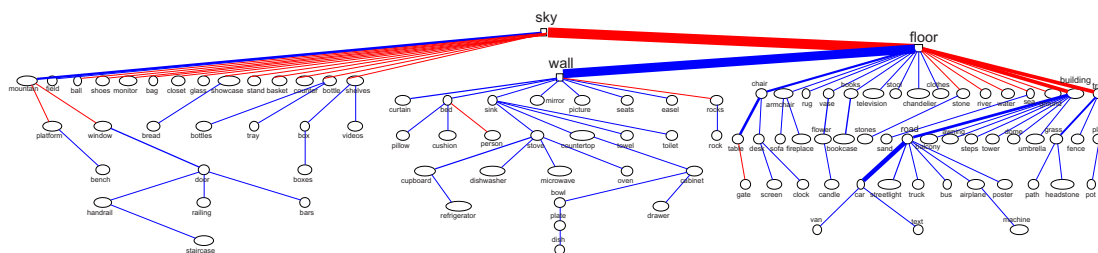


Figure 4: Pairwise MRF of object class presences in images [2]. Red edges denote negative correlations between classes. The thickness of each edge represents the strength of the link. You will be learning this MRF in question 3.

$$\begin{aligned}
&= \sum_{\mathbf{x} \in \mathcal{D}} \left(\sum_{(i,j) \in T} \log \left[\frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i) \hat{p}(x_j)} \right] + \sum_{j \in V} \log [\hat{p}(x_j)] \right) \\
&= \sum_{(i,j) \in T} \sum_{\mathbf{x} \in \mathcal{D}} \log \left[\frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i) \hat{p}(x_j)} \right] + \sum_{j \in V} \sum_{\mathbf{x} \in \mathcal{D}} \log [\hat{p}(x_j)] \\
&= \sum_{(i,j) \in T} \sum_{x_i, x_j} N \hat{p}(x_i, x_j) \log \left[\frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i) \hat{p}(x_j)} \right] + \sum_{j \in V} \sum_{x_i} N \hat{p}(x_i) \log [\hat{p}(x_j)] \\
&= N \left(\sum_{(i,j) \in T} I_{\hat{p}}(X_i, X_j) - \sum_{j \in V} H_{\hat{p}}(X_j) \right),
\end{aligned}$$

where $I_{\hat{p}}(X_i, X_j) = \sum_{x_i, x_j} \hat{p}(x_i, x_j) \log \frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i)\hat{p}(x_j)}$ is the empirical *mutual information* of variables X_i and X_j , and $H_{\hat{p}}(X_i)$ is the empirical *entropy* of variable X_i . Since the entropy terms are not a function of T , these can be ignored for the purpose of finding the maximum likelihood tree structure. **We conclude that the maximum likelihood tree can be obtained by finding the maximum-weight spanning tree in a complete graph with edge weights $I_{\hat{p}}(X_i, X_j)$ for each edge (i, j) .**

The Chow-Liu algorithm then consists of the following two steps:

- (a) Compute each edge weight based on the empirical mutual information.
- (b) Find a maximum spanning tree (MST) via Kruskal or Prim's Algorithm.
- (c) Output a pairwise MRF with edge potentials $\phi_{ij}(x_i, x_j) = \frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i)\hat{p}(x_j)}$ for each $(i, j) \in T$ and node potentials $\phi_i(x_i) = \hat{p}(x_i)$.

We have one random variable $X_i \in \{0, 1\}$ for each object type (e.g., “car” or “road”) specifying whether this object is present in a given image. For this problem, you are provided with a matrix of dimension $N \times M$ where $N = 4367$ is the number of images in the training set and $M = 111$ is the number of object types. This data is in the file “chowliu-input.txt”, and the file “names.txt” specifies the object names corresponding to each column.

Implement the Chow-Liu algorithm described above to learn the maximum likelihood tree-structured MRF from the data provided. Your code should output the MRF in the standard UAI format described here:

<http://www.hlt.utdallas.edu/~vgogate/uai14-competition/modelformat.html>

References

- [1] J.K. Blitzstein and J. Hwang. *Introduction to Probability*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press, 2014.
- [2] Myung Jin Choi, Joseph J. Lim, Antonio Torralba, and Alan S. Willsky. Exploiting hierarchical context on a large database of object categories. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.
- [3] C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14:462–467, 1968.
- [4] Brown University. Cs 242 course, probabilistic graphical models. *Available online at <http://cs.brown.edu/courses/cs242/>*, Fall 2016.