# Inference and Representation, Fall 2017

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

Zhuoru Lin
zlin@nyu.edu

---

*Disclaimer: Due to the lack of .bib file in original GitHub repository, some of the citation will show up as ? in the following session. For the citation please refer to the original pdf version of problem set. I always fight hard to ensure a smooth Grading experience for you. Codes that are less related to the problem is removed for the conciseness of the solutions. For graders, the codes for problem 2, 3 and 4 can be found in attachment in form of .ipynb. For anyone else who may concern, the original codes can be found in my GitHub repository.*

*I adhered to honor code in this assignment. I worked with Qianyu Cheng, Yiqiu Shen and Yuwei Tu on problem 2 and 3 and Qianyu Cheng on problem 4. All codes below and attached are developed independently.*

1. **Ising Model - Gibbs sampling ([?] 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 $j = 1, ..., L^2$. 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 $Y = (Y_1, ..., Y_{L^2})$ is

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

   If $\beta$ 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 $\beta$. Explain how we can do this using Gibbs sampling, cycling through the pixels one by one in a fixed order.

---

**Solutions:**

Consider the conditional probability of $Y_i = 1$ given every other $Y_j \in Y \setminus Y_i$ and $Y_j = y_j$:

$$p(Y_i = 1|Y \setminus Y_i) = \frac{p(Y_i = 1, Y \setminus Y_j)}{p(Y \setminus Y_i)} \tag{1}$$

$$= \frac{p(Y_i = 1, Y \setminus Y_j)}{\sum_{Y_i} p(Y_i, Y \setminus Y_j)} \tag{2}$$

$$= \frac{p(Y_i = 1, Y \setminus Y_j)}{p(Y_i = 1, Y \setminus Y_j) + p(Y_i = 0, Y \setminus Y_j)} \tag{3}$$

The numerator in equation (3) can be calculated by the joint PMF of Ising model:

$$p(Y_i = 1, Y \setminus Y_i) \propto \exp[\beta \sum_{i \sim j} I(y_j = 1) + \beta \sum_{k \sim j, k \neq i} I(y_k = y_j)] \tag{4}$$

Since $\beta \sum_{k \sim j, k \neq i} I(y_k = y_j)$ is a constant, let's use $\alpha$ to denote $\exp[\beta \sum_{k \sim j, k \neq i} I(y_k = y_j)]$. Now we have:

$$p(Y_i = 1, Y \setminus Y_i) \propto \alpha \exp[\beta \sum_{i \sim j} I(y_j = 1)] \tag{5}$$

Similar to equation (5), we can calculate $p(Y_i = 0, Y \setminus Y_i)$:

$$p(Y_i = 0, Y \setminus Y_i) \propto \alpha \exp[\beta \sum_{i \sim j} I(y_j = 0)] \tag{6}$$

Now we can use (3) to calculate $p(Y_i = 1 | Y \setminus Y_i)$:

$$p(Y_i = 1 | Y \setminus Y_i) = \frac{\exp[\sum_{i \sim j} I(y_j = 1)]}{\exp[\sum_{i \sim j} I(y_j = 1)] + \exp[\sum_{i \sim j} I(y_j = 0)]} \tag{7}$$

$$= \frac{1}{1 + \exp[-(\sum_{i \sim j} I(y_j = 1) - \sum_{i \sim j} I(y_j = 0))]} \tag{8}$$

$$= sigmoid(\mu) \tag{9}$$

for $\mu = \sum_{i \sim j} I(y_j = 1) - \sum_{i \sim j} I(y_j = 0)$, which is the difference between number of same-sign neighbors and opposite-sign neighbors.
Given what we derived above, we can use the following algorithm to do simulation:

**Input:** $L \times L$ pixels canvas
**Result:** Simulation results
**initialization**: Any configuration
**while** *not converged* **do**
  **for** $Y_i$ *in* $Y$ **do**
    Calculate $\mu = \sum_{i \sim j} I(y_j = y_i) - \sum_{i \sim j} I(y_j \neq y_i)$
    Randomly generate $e \in [0, 1]$ with uniform distribution.
    **if** $e \geq sigmoid(\mu)$ **then**
      Flip $Y_i$
    **else**
      Do nothing
    **end**
  **end**
**end**

**Algorithm 1:** Ising model Gibbs sampling

2. **Sum-product algorithm, Homework 1 in [?] 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.

   (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.

---

**Solutions:**
The belief propagation code is attached below. I used the .spa method defined in original fglib.nodes module for message calculating. The codes consist of two function:

i. **schedule_propagation**
This function takes two inputs: the edge visiting schedule and the fglib factor graph. The function simply update the message of each edge using sum-product algorithm.

ii. **get_belief**
This function will iterate through the graph and return the beliefs history for each nodes. This breaks down to two cases. If the input factor graph is acyclic, this function will create a efficient schedule for message update by depth first search in the graph. The marginal beliefs converge after each edge is visited twice. When the input factor graph contains cycle, the function generate a iterative update schedule that updates the factor-to-variable messages first then variable-to-factor messages.

```
1 # Update belief given a edge visiting schedule
2 def schedule_propagation(schedule, graph):
3     '''
```

```
    schedule: list of edges (in tuple form)parralel_update
    '''
    for node_origin, node_destination in schedule:
        # Get fglib edge object
        edge = graph.get_edge_data(node_origin, node_destination)['object']
        # get message using sum-product algorithm
        #print('%s --> %s'%(node_origin, node_destination))
        message = node_origin.spa(node_destination).normalize()
        # set message
        edge.set_message(node_origin,node_destination,message)
    return

def get_beliefs(fg, n_iteration=10, parallel_update=True, saving_iterations=[]
    # If acyclic use depth first search to generate a efficient schedule
    if not parallel_update:
        root_node = list(fg.get_vnodes())[0]
        root2leaf = list(nx.depth_first_search.dfs_edges(fg, root_node))
        leaf2root = [(v,u) for u,v in reversed(root2leaf)]
        schedule_propagation(leaf2root, fg)
        schedule_propagation(root2leaf, fg)

    # Otherwise, use iterative updating (Loopy propagation)
    else:
        fnodes = fg.get_fnodes()
        vnodes = fg.get_vnodes()
        nodes_sequence = fnodes + vnodes
        schedule = [(node, neighbor) for node in nodes_sequence for neighbor in
        bar = progressbar.ProgressBar()
        #print('Iterating')
        output_dict = OrderedDict([(str(vnode), []) for vnode in fg.get_vnodes
        for i in bar(range(n_iteration)):
            if i in saving_iterations:
                for vnode in vnodes:
                    output_dict[str(vnode)].append(vnode.belief().pmf)
            # Propagate
            schedule_propagation(schedule, fg)

    # Final configuration saving
    for vnode in vnodes:
                    output_dict[str(vnode)].append(vnode.belief().pmf)
    return output_dict
```

(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.

---

**Solutions:**

| | 0 | 1 |
|---|---|---|
| x1 | 0.658973 | 0.341027 |
| x2 | 0.205136 | 0.794864 |
| x3 | 0.526409 | 0.473591 |
| x4 | 0.286797 | 0.713203 |

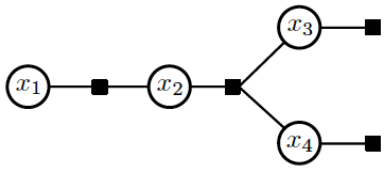


Figure 1: A tree-structured factor graph in which four factors link four random variables. Variable $x_2$ takes one of three discrete states, and the other three variables are binary. [**?**]

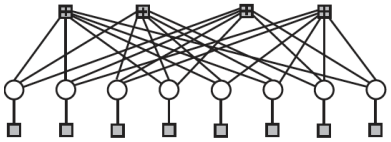

Figure 2: A factor graph representation of a LDPC code linking four factor (parity constraint) nodes to eight variable (message bit) nodes. The unary factors encode noisy observations of the message bits from the output of some communications channel. [**?**]

3. **LDPC, Homework 2 in [?] 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.

---

**Solutions:**

First I defined the factor function as below:

```
1 def construct_check_factor(check_fnode):
2     nodes_in_check = list(check_fnode.neighbors())
3     n_nodes_in_check = len(nodes_in_check)
4     pmf_shape = [2]*n_nodes_in_check
5     pmf_out = np.zeros(pmf_shape)
6     all_idx = np.where(pmf_out==0)
7     pmf_values = 1-np.stack(all_idx).sum(0)%2
8     pmf_out[all_idx] = pmf_values
9     return rv.Discrete(pmf_out, *nodes_in_check)
```

This function takes a fglib fnode as input and will return a fglib.rv random variable object. To show a couple of test cases, let's look at the pmf_out at line 8, which is the pmf of the check factor and check that it evaluates to 1 when the input x are valid codeword (Modulo sum to 0).

```
# Suppose we have a fnode that check 3 binary nodes the pmf is generated
# By following code for this factor
n_nodes_in_check = 3
pmf_shape = [2]*n_nodes_in_check
pmf_out = np.zeros(pmf_shape)
all_idx = np.where(pmf_out==0)
pmf_values = 1-np.stack(all_idx).sum(0)%2
pmf_out[all_idx] = pmf_values

# Can observe that the factor will only evaluate to 1
# when there are even number of 1 among x1,x2 and x3
for x1 in [0,1]:
    for x2 in [0,1]:
        for x3 in [0,1]:
            print('x1=%d, x2=%d, x3=%d, f(x1,x2,x3)=%d'%(x1,x2,x3,
            pmf_out[x1][x2][x3]))
```

The results shows:

x1=0, x2=0, x3=0, f(x1,x2,x3)=1
x1=0, x2=0, x3=1, f(x1,x2,x3)=0
x1=0, x2=1, x3=0, f(x1,x2,x3)=0
x1=0, x2=1, x3=1, f(x1,x2,x3)=1
x1=1, x2=0, x3=0, f(x1,x2,x3)=0
x1=1, x2=0, x3=1, f(x1,x2,x3)=1
x1=1, x2=1, x3=0, f(x1,x2,x3)=1
x1=1, x2=1, x3=1, f(x1,x2,x3)=0

This shows that the factor potential assigns zero probability to invalid codewords.

**Graph Construction** The following codes are used to constructed graph based on a parity code check matrix $H$ and received noisy codeword.

```python
# Construct factor graph
def construct_graph_with_H_and_observations(H, observations, error_rate=0.05):
    n_checks, n_bits = H.shape
    # Construct empty factor graph
    factor_graph = graphs.FactorGraph()
    # Construct bits V nodes
    bits_vnodes = [nodes.VNode('x%d'%n) for n in range(n_bits)]
    # Construct checks F nodes
    checks_fnodes = [nodes.FNode('h%d'%m) for m in range(n_checks)]
    # Construct priors F nodes
    priors_fnodes = [nodes.FNode('f%d'%m) for m in range(n_bits)]

    # Register Nodes to the graph
    factor_graph.set_nodes(priors_fnodes)
    factor_graph.set_nodes(bits_vnodes)
    factor_graph.set_nodes(checks_fnodes)

    # Construct Edges between priors and bits
    factor_graph.set_edges(zip(priors_fnodes, bits_vnodes))
    # Constrct Edges between bits and checks
    check_idx, bit_idx = np.where(H==1)
    for m, n in zip(check_idx, bit_idx):
        factor_graph.set_edge(checks_fnodes[m], bits_vnodes[n])

    # Initialize check factors
    for check_fnode in checks_fnodes:
        check_fnode.factor = construct_check_factor(check_fnode)
        # Initialize message rmn to 1 so that we can call belief() before iter
        for bit_vnode in check_fnode.neighbors():
            edge = factor_graph.get_edge_data(check_fnode, bit_vnode)['object'
            message = rv.Discrete([1,1], bit_vnode)
            edge.set_message(check_fnode, bit_vnode, message)
    # Initialize prior factors
    for prior_fnode, observed_r in zip(priors_fnodes, observations):
        prior_fnode.factor = construct_prior_factor(prior_fnode,observed_r,err

    # Initialize message hm->xn and fn->xn
    for prior_fnode,bit_vnode in zip(priors_fnodes, bits_vnodes):
        # initialize prior message so that we can call belief() before iterati
        prior_message = prior_fnode.spa(bit_vnode).normalize()
        factor_graph.get_edge_data(prior_fnode, bit_vnode)['object'].set_messa
        # Initialize qmn
        checks_for_this_node = bit_vnode.neighbors(prior_fnode)
        message = prior_fnode.factor.pmf
        message = rv.Discrete(message, bit_vnode)
        for check in checks_for_this_node:
            edge = factor_graph.get_edge_data(bit_vnode, check)['object']
            edge.set_message(bit_vnode, check, message)
    return factor_graph
```

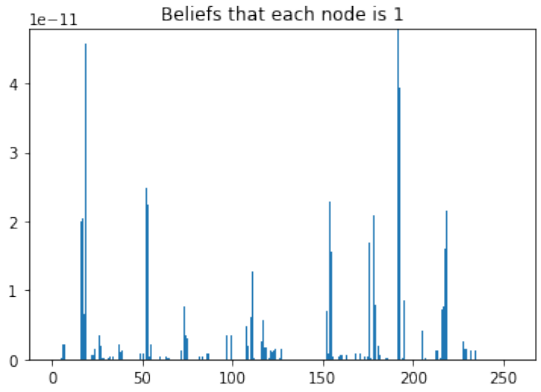


Figure 3: Posterior beliefs on each bits after 50 iterations. Note that this does not guarantee to happen every time and is depend on the received codeword quality. Problem 3-c provide a better assessment of the model performance.

(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 $\epsilon$ 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?

---

**Solutions:**
The iteration results is shown in Figure 3b. It clearly shows that posterior in each bit equals 1 is very small. In this case, we would find the right codeword. However this performance is never guaranteed, as discussed in part c). The performance of belief propagation algorithm is not stable.

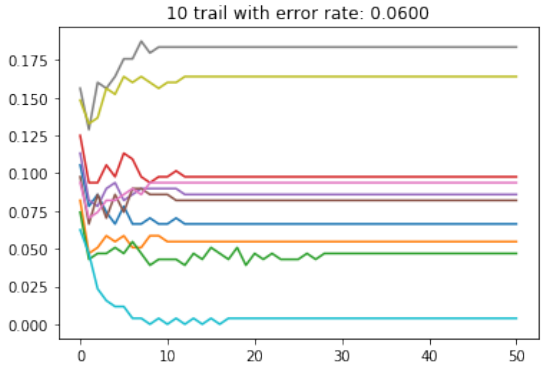


Figure 4: Hamming distances vs iteration for 10 different Monte Carlo trail.

(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?

---

**Solutions:**
The result is shown in Figure 4. We can see that the convergence of belief propagation is not very stable and therefore BP is not a very reliable decoding algorithm. .

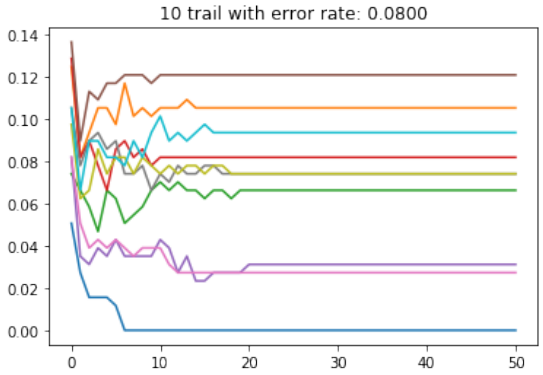


Figure 5: Hamming distances vs iteration for 10 different Monte Carlo trail at error rate 0.08.

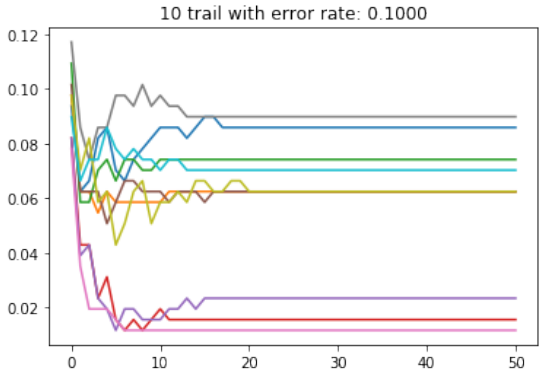


Figure 6: Hamming distances vs iteration for 10 different Monte Carlo trail at error rate 0.10.

(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.

---

**Solutions:**

. . The results are shown in Figure 5 and 6. We can observed that when the error rate is higher, the hamming distances fluctuate more and takes longer to converge.

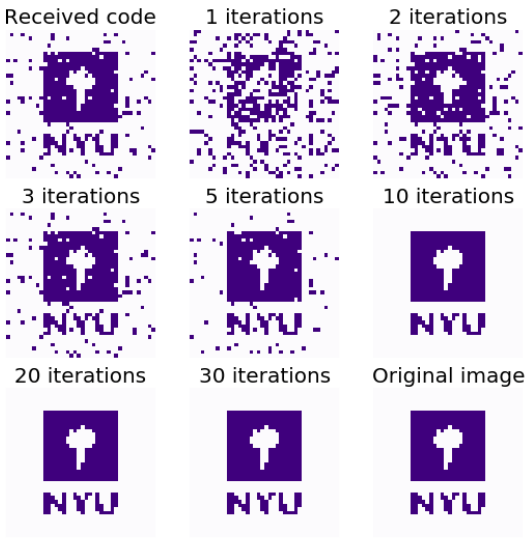


Figure 7: Image decoding at error rate 0.06

(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 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 $40 \times 40$ 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.

---

**Solutions:**
The results is shown in Figure 7 .

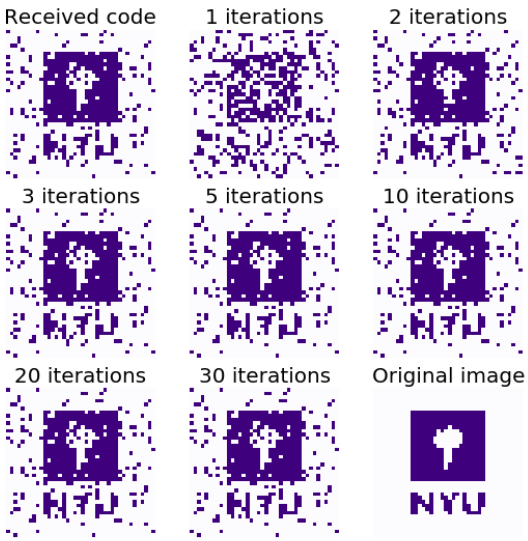


Figure 8: Repeat part e with error rate 0.10.

(f) Repeat the previous part with a higher error probability of $\epsilon = 0.10$, and discuss differences.

---

**Solutions:**
The results is shown in Figure 8. With a higher error rate, the model have only limited ability to decode and cannot fully restore the original image within 30 iterations.

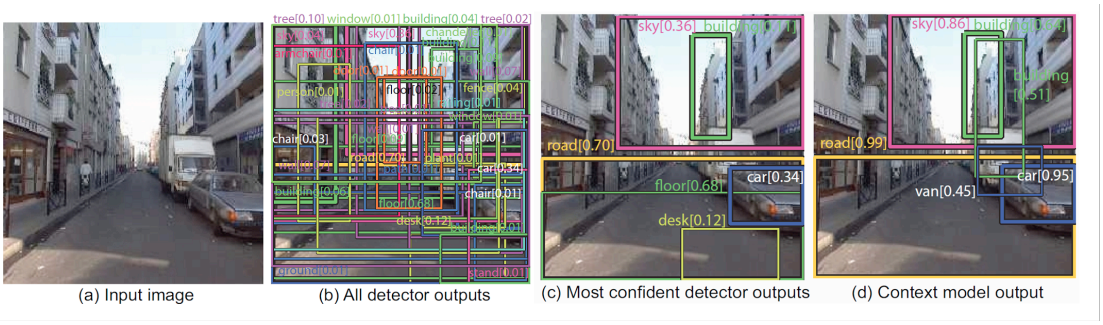


Figure 9: Using context within object detection for computer vision. [**?**]

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 9). In recent work, a tree-structured Markov random field (see Figure 10) 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 [**?**].

   You will replicate some of the results from [**?**] (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 [**?**]. 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. $\hat{p}(x_i, x_j) = count(x_i, x_j)/|\mathcal{D}|$ where $count(x_i, x_j)$ 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 $N = |\mathcal{D}|$):

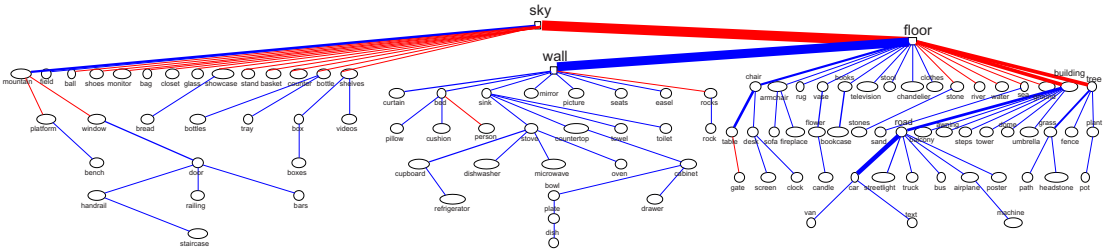


Figure 10: Pairwise MRF of object class presences in images [**?**]. Red edges denote negative correlations between classes. The thickness of each edge represents the strength of the link. You will be learning this MRF in question 3.

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

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`

---

**Solutions:**
The following codes are used to compute the empirical marginal probability of $p(x_i)$ and the empirical co-occurrence probability $p(x_i, x_j)$ and mutual information.

```
def calculate_empirical_coocurence_probability(occurence):
    n_records, n_classes = occurence.shape
    matrix_11 = np.zeros((n_classes, n_classes))
    matrix_00 = np.zeros((n_classes, n_classes))
    matrix_10 = np.zeros((n_classes, n_classes))
    matrix_01 = np.zeros((n_classes, n_classes))
    bar = ProgressBar()
    for row in bar(occurence):
        idx_1 = np.where(row==1)[0]
        idx_0 = np.where(row==0)[0]
        idx_11 = combinations(idx_1, 2)
        idx_00 = combinations(idx_0, 2)
        idx_10 = product(idx_1, idx_0)
        idx_01 = product(idx_0, idx_1)
        for idx in idx_11:
            matrix_11[idx]+=1
        for idx in idx_00:
            matrix_00[idx]+=1
        for idx in idx_10:
            matrix_10[idx]+=1
        for idx in idx_01:
            matrix_01[idx]+=1

    return matrix_11/n_records, matrix_00/n_records, matrix_10/n_records,
            matrix_01/n_records

def calculate_empirical_marginal_probability(occurence):
    return occurence.mean(0), 1-occurence.mean(0)

def calculate_mutial_information(occurence):
    n_records, n_classes = occurence.shape
    mutial_information_matrix = np.zeros((n_classes, n_classes))
    prob_11, prob_00, prob_10, prob_01 = calculate_empirical_coocurence_probabilit
    prob_1, prob_0 = calculate_empirical_marginal_probability(occurence)
    all_idx = combinations(np.arange(n_classes), 2)
    for i,j in all_idx:
        mutial_information_matrix[i,j]+=prob_11[i,j]*np.log(prob_11[i,j]/(prob_1[i
        mutial_information_matrix[i,j]+=prob_00[i,j]*np.log(prob_00[i,j]/(prob_0[i
        mutial_information_matrix[i,j]+=prob_10[i,j]*np.log(prob_10[i,j]/(prob_1[i
        mutial_information_matrix[i,j]+=prob_01[i,j]*np.log(prob_01[i,j]/(prob_0[i
    return np.nan_to_num(mutial_information_matrix)
```

**Then The following codes are used for getting the max spanning tree edges and calculate the node(unary) and edges(pairwise) potential:**

```python
def get_tree_edges_from_mutual_information(mutial_information_matrix):
    neg_mut_info_csr = csr_matrix(-mutial_information_matrix)
    mstree_adj = minimum_spanning_tree(neg_mut_info_csr).todense()
    return [edge for edge in zip(*np.where(mstree_adj<0))]

def calculate_edge_potential(edges, occurence,name=None):
    prob_11, prob_00, prob_10, prob_01 = calculate_empirical_coocurence_probabilit
    prob_1, prob_0 = calculate_empirical_marginal_probability(occurence)
    pot_dict = {}
    for edge in edges:
        if name is None:
            key = edge
        else:
            key = '--'.join(name[np.array(edge)])
        i,j = edge
        pot_11 =prob_11[i,j]/(prob_1[i]*prob_1[j])
        pot_00 =prob_00[i,j]/(prob_0[i]*prob_0[j])
        pot_10 =prob_10[i,j]/(prob_1[i]*prob_0[j])
        pot_01 =prob_01[i,j]/(prob_0[i]*prob_1[j])
        pot_dict[key] = np.array([[pot_00, pot_01], [pot_10, pot_11]])
    return pot_dict

def calculate_node_potentials(occurence):
    prob_1, prob_0 = calculate_empirical_marginal_probability(occurence)
    return np.stack([prob_0, prob_1]).T
```

**Finnally, I used the following function to output the potentials into standard UAI format:**

```python
def get_uai_str(edges, node_potentials, edge_potentials):
    num_vars = node_potentials.shape[0]
    network_type = 'MARKOV'
    num_vars_str = str(num_vars)
    var_cardinals = ' '.join(['2']*num_vars)
    num_cliques = str(len(edges))
    edge_cliques = [' '.join(['2', str(i), str(j)]) for i, j in edges]
    preamble = ([network_type, num_vars_str, var_cardinals, num_cliques]+ edge_cli
    function_tables = []
    # node potentials
    for i in range(num_vars):
        prob_str = ' '.join([' ', str(node_potentials[i][0]), str(node_potentials[
        function_tables += ['', '2', prob_str]
    # edge potentials
    for i, j in edges:
        prob00 = edge_potentials[(i,j)][0][0]
        prob01 = edge_potentials[(i,j)][0][1]
        prob_str1 = ' '.join([' ', str(prob00), str(prob01)])
        prob10 = edge_potentials[(i,j)][1][0]
        prob11 = edge_potentials[(i,j)][1][1]
```

```
21          prob_str2 = ' '.join([' ', str(prob10), str(prob11)])
22          function_tables += ['', '4', prob_str1, prob_str2]
23      return '\n'.join(preamble + function_tables + [''])
```

**The final results is stored in a file named 'p4.uai' and is enclosed in the submission.**