COS 529 Assignment 2: Probabilistic Graphical Model Inference

Due 11:59pm 09/29/2025 ET

Collaboration policy This assignment must be completed individually. No collaboration is allowed.

AI Use Policy

- For homework assignments, help from AI is disallowed if and only if the same help is disallowed from a classmate.
  - Allowed: discussing concepts and course materials with AI/classmate
  - Disallowed: ask AI/classmate to write code or solutions for you
  - Disallowed: ask AI/classmate to review or debug your code
- For the course project, AI use is allowed for coding, with disclosure. AI use is not allowed for writing reports.
  - Allowed: vibe coding, as long as disclosed in project report. We will publish disclosure guidelines.
  - Disallowed: ask AI to write your report beyond spelling and grammar fixes.
  - *Your course project will be graded based on the human effort, not AI effort.

1 Task

In this assignment, you will be implementing the algorithms for inference and learning in undirected graphical models (*a.k.a.* Markov random fields). $G = (V, E)$ is an undirected graph without self-loop, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. We associate each vertex $v \in V$ with a random variable (also denoted by v) taking values from $\mathcal{D} = \{0, 1, \ldots, K-1\}$, where $K > 1$ is a fixed integer.

The probabilistic distribution of these variables is defined by potential functions. Each vertex v has a unary potential $\psi_v : \mathcal{D} \to \mathbb{R}^+$, which maps any assignment of v to a positive real number. Likewise, each edge (u, v) is associated with a binary potential $\psi_{u,v} : \mathcal{D} \times \mathcal{D} \to \mathbb{R}^+$. Assuming there are n vertices, their joint probabilistic distribution is:

$$p(\mathbf{v}) = \frac{1}{Z} \prod_{v \in V} \psi_v(\mathbf{v}) \prod_{(u,v) \in E} \psi_{u,v}(\mathbf{v}), \tag{1}$$

where $\mathbf{v} = (v_1, v_2, \ldots, v_n) \in \mathcal{D}^n$ is a particular assignment of all random variables. $\psi_v(\mathbf{v})$ and $\psi_{u,v}(\mathbf{v})$ are the values of the potentials under $\mathbf{v}$. Z is a normalizer to ensure the probabilities sum up to 1; we thus have:

$$Z = \sum_{v \in \mathcal{D}^n} \prod_{v \in V} \psi_v(\mathbf{v}) \prod_{(u,v) \in E} \psi_{u,v}(\mathbf{v}) \tag{2}$$

Eqn. 1 represents a family of distributions parameterized by the values of the potential functions. Each ψ_v contributes to K parameters and each $\psi_{u,v}$ contributes to $K \times K$ parameters. Given a dataset of i.i.d. samples $S = \{\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \ldots, \mathbf{v}^{(m)}\}$, we would like to fit the family of distributions to the data by maximum likelihood estimation (MLE), in which we search for the parameters that maximizes the likelihood of the data:

$$p(S \mid \psi_v, \psi_{u,v}) = \prod_{i=1}^{m} p(\mathbf{v}^{(i)}) \tag{3}$$

For numerical stability, we maximize the log-likelihood instead:

$$\mathcal{L} = \log p(S \mid \psi_v, \psi_{u,v}) = \sum_{i=1}^{m} \log p(\mathbf{v}^{(i)}) \tag{4}$$

Since the gradients $\frac{\partial \mathcal{L}}{\partial(\psi_v, \psi_{u,v})}$ can be computed efficiently, we usually maximize Eqn. 4 via first-order optimization methods such as gradient descent.

In this assignment, you will be working with trees (connected and without loops). Your task is to implement the probabilistic inference over the joint distribution specified by Eqn. 1, and to compute the gradients of the data log-likelihood with respect to the potentials.

1. Marginal distributions: For each variable v_i, compute its marginal distribution.

$$p_i(v_i) = \sum_{v_1 \in \mathcal{D}} \cdots \sum_{v_{i-1} \in \mathcal{D}} \sum_{v_{i+1} \in \mathcal{D}} \cdots \sum_{v_n \in \mathcal{D}} p(\mathbf{v}) \tag{5}$$

2. Maximum a posteriori (MAP) inference: Compute an assignment of all variables that maximizes the joint probability.

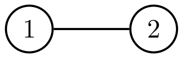
$$\mathbf{v}_{MAP} = \operatorname*{argmax}_{\mathbf{v} \in \mathcal{D}^n} \; p(\mathbf{v}) \tag{6}$$

3. Gradients of the log-likelihood. We only consider the simplified case when there is only one sample $\mathbf{v}^{(0)}$ in the dataset.

$$\frac{\partial \mathcal{L}}{\partial(\psi_v, \psi_{u,v})} = \frac{\partial \log p(\mathbf{v}^{(0)})}{\partial(\psi_v, \psi_{u,v})} \tag{7}$$

2 Example

Take G to be the graph below, with vertices taking binary values ($K = 2$).



The potential functions are:

$$\psi_1(u) = \begin{cases} 2.0 & \text{if } u = 0 \\ 2.0 & \text{if } u = 1 \end{cases} \tag{8}$$

$$\psi_2(v) = \begin{cases} 1.0 & \text{if } v = 0 \\ 5.0 & \text{if } v = 1 \end{cases} \tag{9}$$

$$\psi_{1,2}(u,v) = \begin{cases} 4.0 & \text{if } u = 0 \text{ and } v = 0 \\ 1.0 & \text{if } u = 0 \text{ and } v = 1 \\ 1.0 & \text{if } u = 1 \text{ and } v = 0 \\ 3.0 & \text{if } u = 1 \text{ and } v = 1 \end{cases} \tag{10}$$

$$\tag{11}$$

The joint probability of the two variables can be computed using Eqn. 1 and Eqn. 2.

$$Z = \sum_{u=0}^{1}\sum_{v=0}^{1} \psi_1(u)\psi_2(v)\psi_{1,2}(u,v) = 8 + 10 + 2 + 30 = 50 \tag{12}$$

$$p(0,0) = \frac{1}{Z}\psi_1(0)\psi_2(0)\psi_{1,2}(0,0) = \frac{8}{50} = 0.16 \tag{13}$$

$$p(0,1) = \frac{1}{Z}\psi_1(0)\psi_2(1)\psi_{1,2}(0,1) = \frac{10}{50} = 0.20 \tag{14}$$

$$p(1,0) = \frac{1}{Z}\psi_1(1)\psi_2(0)\psi_{1,2}(1,0) = \frac{2}{50} = 0.04 \tag{15}$$

$$p(1,1) = \frac{1}{Z}\psi_1(1)\psi_2(1)\psi_{1,2}(1,1) = \frac{30}{50} = 0.60 \tag{16}$$

The marginal distributions:

$$p_0(u) = \begin{cases} 0.36 & \text{if } u = 0 \\ 0.64 & \text{if } u = 1 \end{cases} \tag{17}$$

$$p_1(v) = \begin{cases} 0.20 & \text{if } v = 0 \\ 0.80 & \text{if } v = 1 \end{cases} \tag{18}$$

The MAP inference:

$$\mathbf{v}_{MAP} = (1, 1) \tag{19}$$

Given a dataset $\{(0, 1)\}$ consisting of one sample, we compute the gradients in Eqn. 7.

$$\frac{\partial \mathcal{L}}{\partial \psi_1} = \frac{\partial \log p(0,1)}{\partial \psi_1} = [0.32, -0.32]^T \tag{20}$$

$$\frac{\partial \mathcal{L}}{\partial \psi_2} = \frac{\partial \log p(0,1)}{\partial \psi_2} = [-0.20, 0.04]^T \tag{21}$$

$$\frac{\partial \mathcal{L}}{\partial \psi_{1,2}} = \frac{\partial \log p(0,1)}{\partial \psi_{1,2}} = \begin{bmatrix} -0.04 & 0.80 \\ -0.04 & -0.20 \end{bmatrix} \tag{22}$$

3 I/O Format

You will be working with trees in this assignment. You will implement in Python the functions for MAP inference, marginal distributions, and gradients of the log-likelihood. You will complete the `inference` and `inference_brute_force` functions in `pgm.py`. They perform the same function, but `inference` should be able to handle large input graphs.

Input The input is a graph G, the potential functions, an integer K that determines the domain of the random variables, and an assignment of all variables for computing the gradients w.r.t. potentials.

- `G` is a `Graph` object in the NetworkX library. It supports a variety of operations, including accessing nodes, edges, etc. For details, you may consult the official documentation or the starter code for this assignment.
- `G.nodes[v]['unary_potential']` is a 1-d `numpy` array of length K denoting the unary potential function for node v.
- `G.edges[u, v]['binary_potential']` is a 2-d `numpy` array of size $K \times K$ denoting the binary potential function for edge (u, v).
- K is given as an attribute of the graph, which can be accessed via `G.graph['K']`.
- The given assignment for node v is `G.nodes[v]['assignment']`.

For this assignment, G is always a tree (connected and without loops). K is greater than 1, but it not necessarily equals 2.

Output In the starter code, we have initialized buffers for the output. You only have to fill in values in them.

- `G.nodes[v]['marginal_prob']` stores the marginal distribution for node v as a 1-d `numpy` array of size K.
- `G.graph['v_map']` stores the MAP assignment as a 1-d `numpy` array of size n, where n is the number of vertices in G.
- `G.nodes[v]['gradient_unary_potential']` stores the gradient $\frac{\partial \mathcal{L}}{\partial \psi_u}$ as a 1-d `numpy` array of size K.
- `G.edges[u, v]['gradient_binary_potential']` stores the gradient $\frac{\partial \mathcal{L}}{\partial \psi_{(u,v)}}$ as a 2-d `numpy` array of size $K \times K$.

For the previous example, the correct output would be:

```
              G.nodes[0]['marginal_prob']  =  np.array([0.36, 0.64])
              G.nodes[1]['marginal_prob']  =  np.array([0.20, 0.80])
                         G.graph['v_map']  =  np.array([1, 1])
   G.nodes[0]['gradient_unary_potential']  =  np.array([0.32, -0.32])
   G.nodes[1]['gradient_unary_potential']  =  np.array([-0.20, 0.04])
G.edges[0, 1]['gradient_binary_potential'] =  np.array([[-0.04, 0.80], [-0.04, -0.20]])
```

You can run the code against this test case by "`python3 pgm.py test_graph_00.pickle`".

4 Library restrictions

In your implementation, you may not import any additional library that is not in the Python Standard Library: `https://docs.python.org/3/library/`.

5 What to submit

Please submit to Canvas the `pgm.py` file with `inference` and `inference_brute_force` completed.

6 Grading

- `inference_brute_force` (20 points): A brute-force implementation. It will be tested only on small graphs (up to 8 nodes, $K <= 5$).

- `inference` (80 points): An efficient implementation (*e.g.* using belief propagation). It will be tested on large graphs (up to 200 nodes, with $K <= 5$).

For a single test graph, you will receive credit according to the following criteria. A solution will be considered correct if *every* output value is within 0.001 of the correct value. Your final credit will be the average over 10 test graphs.

+ 40% for correct marginal_prob

+ 40% for for correct MAP

+ 10% for correct unary potential gradients

+ 10% for correct binary potential gradients

For a single input graph, both your implementations (brute force and efficient) have a time limit of 20 min on a single CPU core and a memory limit of 2GB (time/memory determined by the AWS EC2 A1 machine as reference). Implementations which exceed these limits on this machine will not receive points. Your brute force implementation will be tested on graphs of up to 8 nodes with $K <= 5$. Your efficient implementation will be tested on large graphs up to 200 nodes with $K <= 5$. Your solutions will be evaluated on graphs with unary and binary potentials in the range [1,5].

Your `brute-force` implementation does not have to be actually brute-force and can be the same as your efficient implementation. But we recommend that you implement an actual brute-force version to help debug your efficient one.

We have provided 5 small testing cases with groundtruth marginal probabilities and MAP estimation to help you debug. **note that passing the provided test cases does not in any way guarantee the correctness of your implementation, and it's your responsibility to devise your own tests and debug your code.**