Homework 4
Probabilistic Graphical Models and Inference

CMU 15-780: Graduate AI (Spring 2016)
OUT: Feb 26, 2016
DUE: March 4, 2016 11:59pm

Instructions

Collaboration Policy

You may discuss assignments with other students as you work through them, but writeups must be done alone. No downloading or copying of code or other answers is allowed. If you use a string of at least 5 words from some source, you must cite the source.

Submission

Please create a tar archive of your answers and submit to Homework 4 on autolab. You should have two files in your archive: a completed `factor_graph.py` for the programming portion, and a PDF for your answers to the written component. Your completed functions will be autograded by running through several test cases and their return values will be compared to the reference implementation.

You have 8 late days for homeworks over the semester, and can use at most 3 for one homework.

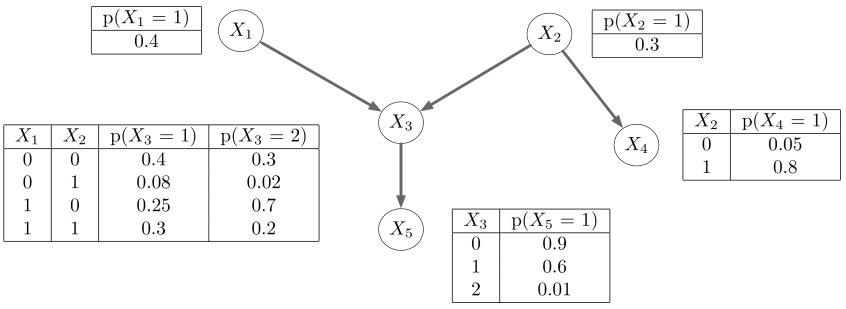
TAs

If you need help, the names beside the questions are the names of the TAs who came up with them, and are more likely to be familiar with the topics.

1 Written [Wennie/Guillermo]

1.1 15 pts

Bayesian Networks Consider a Bayesian network given by the following student example: A student's grade depends not only on his intelligence but also on the difficulty of the course, represented by a random variable X_1 whose domain is $\texttt{Val}(D) = \{\text{easy, hard}\}$. The student asks his professor for a recommendation letter. The professor is absentminded and never remembers the names of his students. He can only look at the student's grade and writes the letter for him based on that information alone. The quality of the letter is a random variable L, whose domain is $\texttt{Val}(L) = \{\text{strong, weak}\}$. The quality of the letter depends stochastically on the grade. (It can vary depending on how stressed the professor is and the quality of the coffee he had that morning). Therefore, we have five random variables in this domain: the student's intelligence X_2, the course difficulty X_1, the grade X_3, the student's SAT score X_4 and the quality of the recommendation letter X_5. All the variables except X_3 are binary-valued, and X_3 is ternary-valued.



(a) [7pts] Write this Bayesian network as a factor graph. What independences are captured by the Bayesian network form which are not implied by the factor graph structure itself?

(b) [8pts] The Bayes net as provided can be thought of as providing an ordering to the variables: difficulty, intelligence, grade, SAT, letter. Suppose instead we wanted to order the variables: letter, SAT, grade, difficulty, intelligence. Write a Bayesian network over these same variables that is consistent with this ordering (i.e., so that the parents of each node must come before the node itself) and preserves the same joint probability distribution as the original. Do not worry about explicitly writing out the tables. Aim to get correct shape of the bayesian network and get rid of any unnecessary links.

For example, consider the network $X_1 \rightarrow X_3 \leftarrow X_2$. We want to reorder the network with list X_3, X_2, X_1. The resulting reordered network is $X_1 \leftarrow X_3 \rightarrow X_2$ and $X_2 \rightarrow X_1$. We know that the product of the probability tables is the joint distribution via the chain rule. We can't simplify any of these tables using conditional independencies in the original bayesian network so we don't remove any links. Notice that we don't just reverse the edges.

Hint: Use the chain rule.

2 Programming [Wennie/Guillermo]

2.1 30 points

Variable elimination in factor graphs For this problem you will implement the sum product variable elimination algorithm to perform inference in a factor graph. In particular, you will implement the following function, in the included file `factor_graph.py`:

```
def marginal_inference(factors, variables, elim_order=None):
```

This function takes as input a list of factors, described via the `Factor` class in the `factor_graph.py` (more details on this shortly), a list of variable names for which we want to produce the marginal distribution, and (optionally) a variable ordering which consists of the order which to eliminate all the remaining variables.

The basis for the factor graph representation we will use is the `Factor` class included in the `factor_graph.py` file. You won't have to edit this class at all, or even necessarily understand all the code in this class, but you will need to understand how to use the class. Each `Factor` object contains 1) a Python dictionary containing all the variables for that Factor, along with their possible values, and 2) a set of factor values for all possible assignments to these variables. Let's look at a simple example: if we initialize `Factor` by the following:

```
f = Factor({"x1":[0,1], "x2":[0,1], "x3":[0,1,2]})
```

this will create a factor of three variables, "x1", "x2", and "x3," where the first two can take on values 0 or 1, and the third can take on values 0, 1, or 2 (note that in this problem, unlike the lecture notes, variables can take on more than two values, but this really introduces no added complexity). Note that there is no requirement that the list of possible values for each factor be numbers: indeed, for the real-world Bayes net example, these will typically be lists of strings, but this should not change your code at all. If you want to access this list of variables at any point, use the class member `f.variables`.

To access (either get or set) the particular factor value corresponding to some assignment of its variables, we can use the call

```
f[{"x1":0, "x2":1, "x3":1}]        # gets the corresponding value
f[{"x1":0, "x2":1, "x3":1}] = 0.1  # sets the corresponding value
```

(this will be the factor value when x1 equals 0, x2 equals 1 and x3 equals 1). For convenience (if you use this properly correctly, it can make your factor operations a bit more compact), if you use the get or set notation above, but with any additional variables that are not in the factor, the function just ignores these extra variables

```
# same as f[{"x1":0, "x2":1, "x3":1}]
f[{"x1":0, "x2":1, "x3":1, "x4":1}]
```

This does not work if you try to get or set the factor with fewer than all the variables specified (this will return an error).

If you want to get a list of all possible assignments to the variables, use `f.inputs()`; for example, in the above setting

```
for e in f.inputs():
    print str(e) + " = " + str(f[e])
```

will output:

```
{'x2': 0, 'x3': 0, 'x1': 0} = 0
{'x2': 1, 'x3': 2, 'x1': 0} = 0
{'x2': 0, 'x3': 1, 'x1': 1} = 0
{'x2': 0, 'x3': 2, 'x1': 0} = 0
{'x2': 1, 'x3': 0, 'x1': 0} = 0
{'x2': 1, 'x3': 1, 'x1': 1} = 0
{'x2': 1, 'x3': 1, 'x1': 0} = 0
{'x2': 1, 'x3': 2, 'x1': 1} = 0
{'x2': 0, 'x3': 1, 'x1': 0} = 0
{'x2': 1, 'x3': 0, 'x1': 1} = 0
{'x2': 0, 'x3': 2, 'x1': 1} = 0
{'x2': 0, 'x3': 0, 'x1': 1} = 0
```

since factors are initialized to have all zero values by default (or you can specify the default value as a second input to the `Factor` initialization). Don't worry about the ordering of the assignments here, a side effect of the dictionary representation is that the factors aren't stored in any particular order. You can also access a list of all the values themselves (in the same order as `f.inputs()`) using `f.values()`.

Using this representation, a factor graph (and hence a probability distribution over all the variables) can be represented using just a list of factors. For example,

```
f1 = Factor({"x1":[0,1], "x2":[0,1]})
f2 = Factor({"x2":[0,1], "x3":[0,1,2]})
f3 = Factor({"x3":[0,1,2], "x4":[1,2]})
fg = [f1,f2,f3]
```

Then `fg` implicitly represents a factor graph. This is possible (i.e., defining this distribution without ever explicitly defining a graph structure or the correspond edges) because each factor itself contains a list of all its variables, so we could immediately construct the graph given any such list of factors.

Using this representation, there are two parts to this problem:

1. Implement the aforementioned

   `def marginal_inference(factors, variables, elim_order=None):`

   assuming that it will be called with a valid elimination ordering provided in `elim_order`. Using the terminology described so far, we can define the inputs and outputs of this function more concretely:

   - `factors` will contain a list of `Factor` objects, representing the factor graph for the distribution. For example, this would be the `fg` variable from the code above.
   - `variables` contains a list of variables that we want to find the marginal distribution of. For example, this could be `["x1","x4"]`, indicating that we want to use inference to compute the marginal distribution $p(x_1, x_4)$ (i.e., summing out x2 and x3).
   - `elim_order` contains a list of variables equal to the set differenced of all the variables in `factors` minus all the variables in `variables`, ordered by the order we want to eliminate them in using the sum product algorithm. For example, given the two settings above, `elim_order` could be equal to `["x2","x3"]`, indicating that we want to eliminate x2 first and then x3
   - As its return value, this function should return a *single* factor, representing the marginal probability over the variables in `variables`. For example, for all the cases above it should return a factor of the form `Factor("x1":[0,1],"x4":[1,2])` where the elements of this factor correspond to the probabilities for each assignment of these variables (since they are probabilities, the sum of all the factors must be one).

   To do this, you will want to implement the sum product algorithm described on page 37 of the probabilistic inference notes. Your implementation will be a lot easier if you also implement the functions

   `def factor_product(f1,f2):`

   and

   `def factor_sum(f1,v):`

   given in the code skeleton. The functions respectively would form the product of two factors (which you can use to then compute the product of several factors) and would sum (marginalize) out a variable from a particular factor.

Some test cases for this code, including a test case on a Bayesian network used for patient alarm systems in a hospital, are included with the problem code.

2. Implement the

   `def marginal_inference(factors, variables, elim_order=None):`

   using all the same notation as before, but now in the case where `None` is based as `elim_order`. In this case, you will need to determine the ordering yourself, which will consist of some order for eliminating all the variables in the factor graph except for those contained in `variables`. However, since computing the optimal variable elimination ordering in a factor graph is NP hard, you'll need to use a heuristic.

   You are welcome to try whatever heuristics for ordering that you want, but a method that will work for all the test cases we will give you is the "minimum neighbors" heuristic. Two variables x and y are defined to be neighbors if there is some factor in the factor graph that contains them both. Note that is not quite the same as the typical notion of neighbors in a graph, since all the direct neighbors of a variable in the factor graph would be factors; this is actually equivalent to a "two step" neighbors in the factor graph itself. At each step of variable elimination, choose to eliminate the variable (not in `variables`) that has the fewest number of neighbors. If you do this incorrectly, variable elimination should still eventually give you the right answer, but it will most likely take too long for the test cases we give.