Lab 2: Inference in Graphical Models

Machine Learning: Principles and Methods, November 2013

- The lab exercises should be made in groups of three people, or at least two people.
- The deadline is Wednesday, Dec 4, 23:59.
- Assignment should be sent to T.S.Cohen at uva dot nl (Taco Cohen). The subject line of your email should be "[MLPM2013] lab#_lastname1_lastname2_lastname3".
- Put your and your teammates' names in the body of the email
- Attach the .IPYNB (IPython Notebook) file containing your code and answers. Naming of the file follows the same rule as the subject line. For example, if the subject line is "[MLPM2013] lab01_Kingma_Hu", the attached file should be "lab01_Kingma_Hu.ipynb". Only use underscores ("_") to connect names, otherwise the files cannot be parsed.

Notes on implementation:

- You should write your code and answers in an IPython Notebook: http://ipython.org/notebook.html. If you have problems, please contact us.
- Among the first lines of your notebook should be "%pylab inline". This imports all required modules, and your plots will appear inline.
- NOTE: test your code and make sure we can run your notebook / scripts!

Introduction

In this assignment, we will implement the sum-product and max-sum algorithms for factor graphs over discrete variables. The relevant theory is covered in chapter 8 of Bishop's PRML book, in particular section 8.4. Read this chapter carefuly before continuing!

We will first implement sum-product and max-sum and apply it to a simple poly-tree structured factor graph for medical diagnosis. Then, we will implement a loopy version of the algorithms and use it for image denoising.

For this assignment we recommended you stick to numpy ndarrays (constructed with np.array, np.zeros, np.ones, etc.) as opposed to numpy matrices, because arrays can store n-dimensional arrays whereas matrices only work for 2d arrays. We need n-dimensional arrays in order to store conditional distributions with more than 1 conditioning variable. If you want to perform matrix multiplication on arrays, use the np.dot function; all infix operators including *, +, -, work element-wise on arrays.

Part 1: The sum-product algorithm

We will implement a datastructure to store a factor graph and to facilitate computations on this graph. Recall that a factor graph consists of two types of nodes, factors and variables. Below you will find some classes for these node types to get you started. Carefully inspect this code and make sure you understand what it does; you will have to build on it later.

In [104]:
```
%pylab inline
class Node(object):
    """
    Base-class for Nodes in a factor graph. Only instantiate sub-classes of Node.
    """
    def __init__(self, name):
        # A name for this Node, for printing purposes
        self.name = name

        # Neighbours in the graph, identified with their index in this list.
        # i.e. self.neighbours contains neighbour 0 through len(self.neighbours) - 1
        self.neighbours = []

        # Reset the node-state (not the graph topology)
        self.reset()

    def reset(self):
        # Incomming messages; a dictionary mapping neighbours to messages.
        # That is, it maps  Node -> np.ndarray.
        self.in_msgs = {}

        # A set of neighbours for which this node has pending messages.
        # We use a python set object so we don't have to worry about duplicates.
        self.pending = set([])

    def add_neighbour(self, nb):
        self.neighbours.append(nb)

    def send_sp_msg(self, other):
        # To be implemented in subclass.
        raise Exception('Method send_sp_msg not implemented in base-class Node')

    def send_ms_msg(self, other):
        # To be implemented in subclass.
        raise Exception('Method send_ms_msg not implemented in base-class Node')

    def receive_msg(self, other, msg):
        # Store the incomming message, replacing previous messages from the same noc
        self.in_msgs[other] = msg

        # TODO: add pending messages
        # self.pending.update(...)

    def __str__(self):
        # This is printed when using 'print node_instance'
        return self.name


class Variable(Node):
    def __init__(self, name, num_states):
        """
        Variable node constructor.
        Args:
            name: a name string for this node. Used for printing.
            num_states: the number of states this variable can take.
            Allowable states run from 0 through (num_states - 1).
            For example, for a binary variable num_states=2,
            and the allowable states are 0, 1.
        """
        self.num_states = num_states
```

1.1 Instantiate network (10 points)

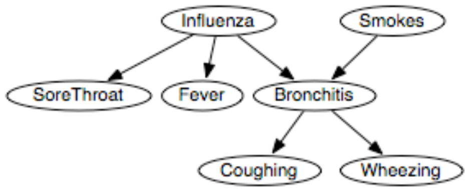
Convert the directed graphical model ("Bayesian Network") shown below to a factor graph. Instantiate this graph by creating Variable and Factor instances and linking them according to the graph structure. To instantiate the factor graph, first create the Variable nodes and then create Factor nodes, passing a list of neighbour Variables to each Factor. Use the following prior and conditional probabilities.

$p(\texttt{Influenza}) = 0.05$
$p(\texttt{Smokes}) = 0.2$

$p(\texttt{SoreThroat} = 1|\texttt{Influenza} = 1) = 0.3$
$p(\texttt{SoreThroat} = 1|\texttt{Influenza} = 0) = 0.001$
$p(\texttt{Fever} = 1|\texttt{Influenza} = 1) = 0.9$
$p(\texttt{Fever} = 1|\texttt{Influenza} = 0)0.05$
$p(\texttt{Bronchitis} = 1|\texttt{Influenza} = 1, \texttt{Smokes} = 1) = 0.99$
$p(\texttt{Bronchitis} = 1|\texttt{Influenza} = 1, \texttt{Smokes} = 0) = 0.9$
$p(\texttt{Bronchitis} = 1|\texttt{Influenza} = 0, \texttt{Smokes} = 1) = 0.7$
$p(\texttt{Bronchitis} = 1|\texttt{Influenza} = 0, \texttt{Smokes} = 0) = 0.0001$
$p(\texttt{Coughing} = 1|\texttt{Bronchitis} = 1) = 0.8$
$p(\texttt{Coughing} = 1|\texttt{Bronchitis} = 0) = 0.07$
$p(\texttt{Wheezing} = 1|\texttt{Bronchitis} = 1) = 0.6$
$p(\texttt{Wheezing} = 1|\texttt{Bronchitis} = 0) = 0.001$

In [105]:
```python
from IPython.core.display import Image
Image(filename='bn.png')
```

Out[105]:



1.2 Factor to variable messages (20 points)

Write a method `send_sp_msg(self, other)` for the Factor class, that checks if all the information required to pass a message to Variable `other` is present, computes the message and sends it to `other`. "Sending" here simply means calling the `receive_msg` function of the receiving node (we will implement this later). The message itself should be represented as a numpy array (np.array) whose length is equal to the number of states of the variable.

An elegant and efficient solution can be obtained using the n-way outer product of vectors. This product takes n vectors $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$ and computes a n-dimensional tensor (ndarray) whose element $i_0, i_1, \dots, i_n$ is given by $\prod_j \mathbf{x}^{(j)}_{i_j}$. In python, this is realized as `np.multiply.reduce(np.ix_(*vectors))` for a python list `vectors` of 1D numpy arrays. Try to figure out how this statement works -- it contains some useful functional programming techniques. Another function that you may find useful in computing the message is `np.tensordot`.

1.3 Variable to factor messages (10 points)

Write a method `send_sp_message(self, other)` for the Variable class, that checks if all the information required to pass a message to Variable var is present, computes the message and sends it to factor.

1.4 Compute marginal (10 points)

Later in this assignment, we will implement message passing schemes to do inference. Once the message passing has completed, we will want to compute local marginals for each variable. Write the method `marginal` for the Variable class, that computes a marginal distribution over that node.

1.5 Receiving messages (10 points)

In order to implement the loopy and non-loopy message passing algorithms, we need some way to determine which nodes are ready to send messages to which neighbours. To do this in a way that works for both loopy and non-loopy algorithms, we make use of the concept of "pending messages", which is explained in Bishop (8.4.7): "we will say that a (variable or factor) node a has a message pending on its link to a node b if node a has received any message on any of its other links since the last time it send (sic) a message to b. Thus, when a node receives a message on one of its links, this creates pending messages on all of its other links."

Keep in mind that for the non-loopy algorithm, nodes may not have received any messages on some or all of their links. Therefore, before we say node a has a pending message for node b, we must check that node a has received all messages needed to compute the message that is to be sent to b.

Modify the function `receive_msg`, so that it updates the self.pending variable as described above. The member self.pending is a set that is to be filled with Nodes to which self has pending messages. Modify the `send_msg` functions to remove pending messages as they are sent.

1.6 Inference Engine (10 points)

Write a function `sum_product(node_list)` that runs the sum-product message passing algorithm on a tree-structured factor graph with given nodes. The input parameter `node_list` is a list of all Node instances in the graph, which is assumed to be ordered correctly. That is, the list starts with a leaf node, which can always send a message. Subsequent nodes in `node_list` should be capable of sending a message when the pending messages of preceding nodes in the list have been sent. The sum-product algorithm then proceeds by passing over the list from beginning to end, sending all pending messages at the nodes it encounters. Then, in reverse order, the algorithm traverses the list again and again sends all pending messages at each node as it is encountered. For this to work, you must initialize pending messages for all the leaf nodes, e.g. `influenza_prior.pending.add(influenza)`, where `influenza_prior` is a Factor node corresponding the the prior, `influenza` is a Variable node and the only connection of `influenza_prior` goes to `influenza`.

1.6 Observed variables and probabilistic queries (15 points)

We will now use the inference engine to answer probabilistic queries. That is, we will set certain variables to observed values, and obtain the marginals over latent variables. We have already provided functions `set_observed` and `set_latent` that manage a member of Variable called `observed_state`. Modify the `Variable.send_msg` and `Variable.marginal` routines that you wrote before, to use `observed_state` so as to get the required marginals when some nodes are observed.

1.7 Sum-product and MAP states (5 points)

A maximum a posteriori state (MAP-state) is an assignment of all latent variables that maximizes the probability of latent variables given observed variables:

$\mathbf{x}_{\mathtt{MAP}} = \arg\max_{\mathbf{x}} p(\mathbf{x}|\mathbf{y})$

Could we use the sum-product algorithm to obtain a MAP state? If yes, how? If no, why not?

Part 2: The max-sum algorithm

Next, we implement the max-sum algorithm as described in section 8.4.5 of Bishop.

2.1 Factor to variable messages (10 points)

Implement the function `Factor.send_ms_msg` that sends Factor -> Variable messages for the max-sum algorithm. It is analogous to the `Factor.send_sp_msg` function you implemented before.

2.2 Variable to factor messages (10 points)

Implement the `Variable.send_ms_msg` function that sends Variable -> Factor messages for the max-sum algorithm.

2.3 Find a MAP state (10 points)

Using the same message passing schedule we used for sum-product, implement the max-sum algorithm. For simplicity, we will ignore issues relating to non-unique maxima. So there is no need to implement backtracking; the MAP state is obtained by a per-node maximization (eq. 8.98 in Bishop). Make sure your algorithm works with both latent and observed variables.

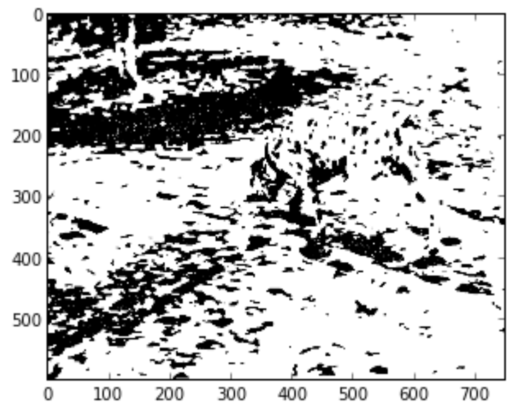
Part 3: Image Denoising and Loopy BP

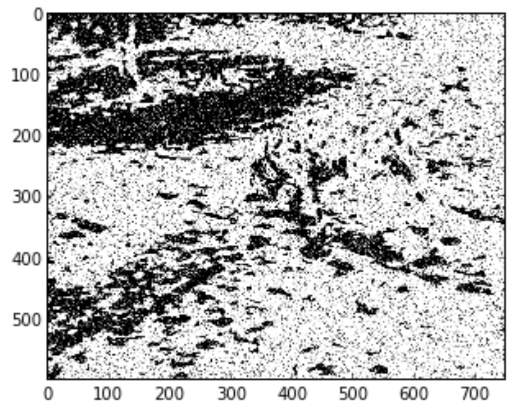
Next, we will use a loopy version of max-sum to perform denoising on a binary image. The model itself is discussed in Bishop 8.3.3, but we will use loopy max-sum instead of Iterative Conditional Modes as Bishop does.

The following code creates some toy data. `im` is a quite large binary image, `test_im` is a smaller synthetic binary image. Noisy versions are also provided.

In [130]:
```
from pylab import imread, gray
# Load the image and binarize
im = np.mean(imread('dalmatian1.png'), axis=2) > 0.5
imshow(im)
gray()

# Add some noise
noise = np.random.rand(*im.shape) > 0.9
noise_im = np.logical_xor(noise, im)
figure()
imshow(noise_im)

test_im = np.zeros((10,10))
#test_im[5:8, 3:8] = 1.0
#test_im[5,5] = 1.0
figure()
imshow(test_im)

# Add some noise
noise = np.random.rand(*test_im.shape) > 0.9
noise_test_im = np.logical_xor(noise, test_im)
figure()
imshow(noise_test_im)
```

Out[130]:
```
<matplotlib.image.AxesImage at 0x238c5e10>
```





3.1 Construct factor graph (10 points)

Convert the Markov Random Field (Bishop, fig. 8.31) to a factor graph and instantiate it.

3.2 Loopy max-sum (10 points)

Implement the loopy max-sum algorithm, by passing messages from randomly chosen nodes iteratively until no more pending messages are created or a maximum number of iterations is reached.

Think of a good way to initialize the messages in the graph.