CS 228, Winter 2010-2011
Programming Assignment #1—Knowledge Engineering in Bayesian and Markov Networks

This assignment is due at 11:59 PM on January 27.

---

1 Introduction

In this programming assignment, you will examine several design choices in both a Bayesian network and a Markov network, and see what effect they have on network behavior. In the first part, you will use a graphical Bayes Net software package to view a Credit Score network and gain intuition into how CPDs in a network affect its behavior. The second part of the assignment involves coding energy factors for an image segmentation network and analyzing how a variety of factors impact the results of inference in the Markov Network.

2 [15 points] Bayesian Network Knowledge Engineering

In this portion of the assignment, you will experiment with a Bayesian network model using the software package SAMIAM.

2.1 Using SAMIAM

SAMIAM was developed by the Automated Reasoning Group at UCLA to provide a graphical interface for manipulating probabilistic networks on Windows, Linux, or Mac OS. It has extensive functionality for learning and inference in probabilistic networks; however, we will only use it to examine and manipulate a pre-existing network, and to view the marginal posterior distributions. While we provide the program as part of the assignment, you can also find the program and more information at this website:

`http://reasoning.cs.ucla.edu/samiam/index.php`

The SAMIAM program can be found in the **Bayes Net** folder within PA1. There are separate folders for each operating system (Linux, Windows, and Mac) so use whichever is appropriate for the machine you are on. Use the executable **runsamiam** to start the program. The network we will use is an example credit score evaluation network based on a network created as an example for the GeNIe graphical probabilistic models program. You can find this program and more examples at `http://genie.sis.pitt.edu/`. To load the credit network into SAMIAM, go to File/Open, navigate to the **network_samples** folder and open **Credit_net.net**.

Initially, we will be using **Query Mode** to monitor the marginals of nodes in this network. Go to the **Mode** menu and select **Query Mode**. On the left you should have a list of nodes. Clicking on one will reveal the values that node can take on. Clicking one of these values will assign it to have that observed value. Clicking again will free the node to be unobserved. To view a node's marginal, right click and select **Monitor**. You can also display all marginals by going to the **Query** menu, **Show monitors**, and selecting **Show All**.

For the last few questions, we will also use **Edit Mode** to update some marginals and add nodes. To engage **Edit Mode**, again go to the **Mode** menu and simply select **Edit Mode**. To add a node, go to **Edit**, **Add Node** and click in the network pane to add that node. To add an edge, again go to **Edit**, **Add Edge** click on whichever node you want to be the parent, then click on the node you want to be the child. To change the node's properties, double click on it. In the Properties tab you can change the name/identifier of a node by clicking on the appropriate name or identifier field. You can edit the names of the states your node can be in by double clicking on the table in the bottom half of the properties tab. You can also add/remove states by clicking **Insert** and **Remove** in the **State Actions** box. You can edit the CPD for the selected node by click on the **Probabilities** tab, double clicking on any of the values, and inputting your own value. Remember that you should edit the CPD for the child node when you add an edge.

2.2 Credit Network Details

This network has 12 nodes with the root being **Credit Worthiness**. The goal of the network is to gather evidence regarding someone applying for credit (say, in the form of a loan) and make an informed decision about whether to give the applicant a loan or not via inference in this probabilistic network. Here is a listing of the nodes, their values, and purposes.

- **Credit Worthiness:** Takes the values Positive and Negative. Positive indicates that the person should be given a loan, Negative indicates no credit should be offered.
- **Age:** Encodes the age of the subject. The values, of the form a$\#_1$_$\#_2$, indicate that the subject has age between $\#_1$ and $\#_2$. We might expect younger subjects would be less responsible and less able to pay back a loan.
- **Assets:** This node encodes how much wealth a person has. The node influences **Worth** as we might expect someone who is more wealthy would have higher net worth.
- **Debit:** Encodes how much debt the subject has. a$\#_1$_$\#_2$ indicates the amount of debt is between $\#_1$ and $\#_2$.
- **Income:** Encodes the subject's income. Values of the form s$\#_1$_$\#_2$ indicate a salary/income between $\#_1$ and $\#_2$. This influences worth as someone with more income is likely to have higher economic worth.
- **Payment History:** Encodes how well the subject has paid back loans or credit bills in the past. Someone with a better payment history can be expected to be better at paying back loans in the future as well, hence the influence on reliability.
- **Profession:** This node denotes how much money someone in the subject's profession is expected to make. As profession will likely be stable, this can help predict future income.
- **Work History:** This node encodes either a stable or unstable work history, or a lack of work history with justification or with no justification. An example of no work history, but with justification, would be that the subject has just finished high school and has never had a chance to work before. In this network we take this stability as an indicator of reliability.

- **Future Income:** This node would not be observed, as we cannot see the future, but is used to make the influences in the model clearer. It is binary valued as Promising or not. Promising indicates that we expect the person's future income to be good enough to pay back any credit we extend them.
- **Ratio of Debts to Income:** Another binary valued node, indicating whether the subject's income and debt have a favorable or unfavorable ratio. We might expect someone with a better income to debt ratio to be more likely to pay back any loan or credit we extend them.
- **Reliability:** An intermediate node which indicates whether we can trust the subject to follow through with payment of a debt. This node would probably not be observed, but we can gain information about it from its parents.
- **Worth:** A node indicate the subject's economic worth. This is a parent of **Future Income** because we might expect someone with higher worth to have higher future income. As the saying goes, 'it takes money to make money.'

2.3 [15 points] Questions

Some of the questions are qualitative in nature, and their purpose is to get you to think about the properties of these algorithms. A few sentences are sufficient for each question, and be specific about any results you cite (i.e., specific values for marginals).

You must include the answers to these questions in an ASCII text file called README when you submit your code

1. [**1 point**] Select work history and observe it to be 'Stable.' What is the marginal for Credit Worthiness? Now change the variable to 'Unstable.' How does the marginal for Credit Worthiness change? How does this change affect the marginal for Payment History? Is this what you expected? Why?
2. [**1 point**] Now observe 'Reliability' to be reliable. How does this affect the marginal for credit worthiness?
3. [**1 point**] Leaving 'Reliability' set to 'Reliable,' change the observation for work history from 'Unstable' back to 'Stable.' Does the marginal for Credit Worthiness change? Is this what you expected? Why or why not?
4. [**1 point**] Repeat the above experiment, but this time monitor the marginal for Payment History. Does the marginal change? Is this what you expected? Why or why not?
5. [**3 points**] A coworker comes to you and tells you a group of hackers has compromised your credit worthiness predictor by tampering with one of the CPDs. She tells you that it now seems people with more assets seem to be having worse credit worthiness predictions – the opposite of what you would expect. Which CPD do you think the hackers tampered with? Why? (Hint: try setting observed values at various nodes and seeing how Credit Worthiness is affected)
6. [**2 points**] Having identified the problem, you want to fix the network. Change the broken CPD to something you think is reasonable. How did you change the CPD? How did the behavior of the network change?

7. [**2 points**] You decide that the network is too simple and should include other variables. You decide that knowing a person's highest education achieved will give you some information about their profession and could improve your model. Add a node **Education**, with possible values **High School**, **College**, and **Graduate School**, and determine a reasonable set of parents and a CPD for it. What decisions did you make in define the model? How does this addition affect the network's marginals?

8. [**4 points**] Still unsatisfied with your model, you decide to add at least one more variable. Add a new node and connect it to the network. Explain what variable you added, why you added it, what other nodes you connected it to, and why you chose those connections. How does your addition effect the network's marginals in **Query Mode**?

3 [45 points] Scene Segmentation – Markov Nets

In this portion of the assignment you will create a Markov Network which we will use to label an image such that each pixel is assigned to one of eight classes. The eight classes we use are sky, tree, road, grass, water building, mountain, and foreground. The foreground class represents objects of interest

3.1 The Model

We break the image into a set of superpixels, either as a 2x2 pixel based model or larger patches which are uniform in appearance. Our goal is to label each superpixel with one of our eight specified classes. Below you can see an image having been broken into similarity-based superpixels and the correct labeling for that image.

(a) Base Image

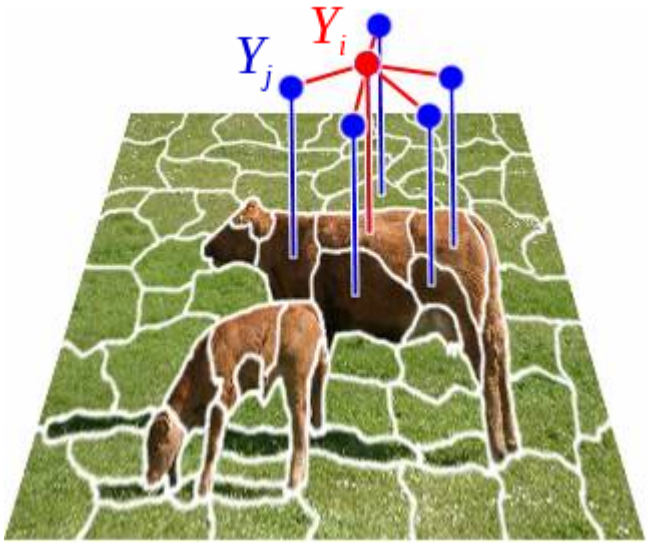
(b) Superpixels

(c) Ground Truth

(d) sky (e) tree (f) road (g) grass (h) water (i) bldn (j) mntn (k) fore

For the purpose of this assignment, we treat each superpixel as a node in our Markov network and ask you to define factors between adjacent superpixels in hopes of improving our segmentation of the image. Thus, there are two types of energy factors to consider, the singleton

terms giving an energy value to each class label for each superpixel, and pairwise terms giving an energy value to any combination of class labels for adjacent superpixels. You can see this approach in the figure below which shows the portion of the network surrounding superpixel i. Y_i is a variable representing the label for superpixel i and is connected in the Markov Net to the variables Y_j of all adjacent superpixels.



We also provide you with the results of a human and car object detector on these pictures. Since the detector is far from being 100% reliable, these detections may or may not correspond to valid instances of these object classes. These candidate detections also act as nodes and we will define edges between detections and superpixels to complete the network and hopefully benefit both segmentation and object detection.

3.2 Code Details

All code for this assignment should be copied from

/afs/ir/class/cs228/ProgrammingAssignments/PA1/

You should be able to access this by logging into the Stanford Unix machines (e.g., cardinal.stanford.edu) using your SUNet ID. Note that in this assignment, we will use a MEX-file (Matlab EXecutable) compiled from the Stanford Vision Library (SVL) (see more details in the Detail section), and it requires a Unix/Linux environment. You can use a copy of Matlab on any of the machines that allow CPU-intensive jobs (e.g., myth, corn, pod). If you want to remotely login to Stanford unix machines, for Linux/Mac, open the terminal and use the command `ssh -X <yourSUID>@myth.stanford.edu` (you can change myth to other machine names, e.g., corn). For Windows, you can use a third-party software to do ssh, e.g. SecureCRT (`http://www.stanford.edu/services/ess/pc/securecrt.html`).

Your job is to fill in the missing portions of code — marked with YOUR CODE HERE — in the files described below. The script **TestPA1.m** is the main routine that sets up the experiments to be run and also evaluates whether your implementation is correct. Initially, running this script without making any code changes should result in all tests failing (though they should run without error).

The experiments run in **TestPA1.m** and the function that must be implemented for each one are enumerated below. For each experiment, sample inputs and outputs are loaded, and the test passes if the output produced by your implementation matches the correct outputs that are loaded in. Note that when you submit, your final implementation it will be tested with different inputs and outputs.

Once the tests confirm that your function implementations are giving correct outputs, you will run **PA1_eval.m** allowing you to see the performance of your code.

3.2.1 Existing Code

Your main interface for working with the code will be **PA1_eval.m**. This function take parameters you specify, builds a Markov Network, performs inference, and shows you the results using different types of factors. The images also display the segmentation accuracy in their titles and returns a confusion matrix for the segmentation given by the full model you specified. The options you will specify in this function are as follows:

- **imgNum:** Set from 1 to 6. For the questions in this assignment, use 4, but you can experiment with other values.
- **SPNum:** What type of superpixel model to use (values 0 to 4). Setting to 0 indicates that a pixel-based model should be used, while 1 through 4 indicate different precomputed superpixels with 1 being the smallest and 4 being the largest superpixels.
- **potts_lambda:** The Potts energy weight.
- **contrast_lambda:** The λ to be used in the contrast exponential weight. Set to 0 to disable contrast weighting.
- **full_lambda:** The weight given to your full user-defined energy matrix. Set to 0 to disable.
- **object_weight:** The weight given to potentials between object detections and superpixels. This can be disabled by setting the value to 0.
- **new_object_weight:** The weight for the new object potentials that you create in **GetObjectSPFactors.m**.

In creating your own potentials, you may want to use bounding boxes for object detections, the objects themselves, and/or the superpixel labeling of the image. If so, you will want to use the following data structures:

- **SPs:** This is the matrix giving the superpixel labeling of the image. **SPs**(i, j) gives the number of superpixel to which pixel (i, j) belongs.
- **objects:** This is an array of the object detections in an image with the following fields:
  - **p:** Probability of the detection
  - **type:** The type of this detection. Either car or person.
  - **x:** X coordinate of the upper left corner of the detection's bounding box.
  - **y:** Y coordinate of the upper left corner of the detection's bounding box.
  - **h:** Height of the detection's bounding box.
  - **w:** Width of the detection's bounding box.

- **bb:** An array specifying the bounding box for an object detection. The first value and third values are the x and y coordinates of the upper-left corner of the bounding box respectively, while the second and fourth values give the x and y coordinates of the bottom-right corner.

The data structure **segm_params** is used throughout this assignment and has the following fields:

- **img_dim:** The dimensions of the current image
- **LK** The number of labels for our segments
- **labels:** The name associated with each numeric label (ie labels{1} = sky means the numeric label 1 is for sky).
- **LC:** The RGB color for each label. LC(1,:) is the RGB value for label 1.

You will also build an array of factors called **F_new** which will be used for inference. Each entry in this array should have the following fields:

- **vars:** A 1-by-N vector, where N is the number of variables in the current factor. Variable names must be consistent across factors and should start from 1.
- **cards:** A 1-by-N vector giving the cardinality of each variable's labeling. **cards**(i) should be the number of possible assignments to the variable **vars**(i).
- **data:** The energy matrix with size consistent with **vars** and **cards**. Specifically, the entries with label $l_1, \ldots, l_N$ should represent the energy for the assignment where each variable **vars**(i) takes the value l_i.

We also provide three functions that determine relationships between superpixels and between a superpixel and an object for you to use. They are as follows:

- **GetSuperpixelAdjacencies.m:** Takes the superpixel matrix **SPs** and returns a matrix A such that $A(i, j) = 1$ if superpixels i and j are adjacent and $A(i, j) = 0$ if not.
- **GetSuperpixelRelation.m:** Takes the superpixel matrix **SPs** and the labels of the two superpixels you want a relation for. Returns the following:
  - **Adjacent:** An indicator of whether the two superpixels are adjacent (0 if not, 1 if so).
  - **Distance:** The distance between the centroids of the two superpixels.
  - **Angle:** The angle (in radians) from the first superpixel to the second. Angle is specified such that if the first is directly right of the second, our angle would be π and if it were directly below the second, the angle would be $\frac{\pi}{2}$.
- **GetObjectSPRelation.m:** Takes the superpixel matrix **SPs**, an object bounding box, and a superpixel label and returns the following:
  - **Coverage:** The fraction of the superpixel covered by the bounding box (0 if no coverage).
  - **Distance:** The distance between the centroids of the superpixel and bounding box.
  - **Angle:** The angle (in radians) from the superpixel to the bounding box. Angle is specified such that if the superpixel is directly right of the box, our angle would be π and if it were directly below the box, the angle would be $\frac{\pi}{2}$.

3.3 [45 points] Code and Questions

Some of the questions are qualitative in nature, and their purpose it to get you to think about the properties of these algorithms. A few sentences are sufficient for each question, but be specic about any results you cite. For these questions, use **PA1_eval.m** to view the results of inference and use **TestPA1.m** to verify that your code is working as intended. **PA1_eval.m** will create a set of plots showing the results using each type of model. The title of the model gives the accuracy of segmentation as well. Use the default values specified in the file unless told otherwise.

You must include the answers to these questions in an ASCII text file called README when you submit your code

3.3.1 [28 points] Creating Segmentation Factors

Our first coding task is to create the pairwise factors between adjacent superpixels. The general motivation for these energies is that adjacent superpixels are likely to share the same label. We want to enforce some consistency between adjacent pixels, but not to such an extent that we force pixels to share labels across the actual segmentation boundaries. The energy model we build on is known as the Potts Model, which is discussed along with other Metric MRFs on page 127 in the textbook.

1. **[2 points] GetAdjacencyList.m:** In order to create these pairwise terms, we must figure out which superpixels are adjacent. This function takes matrix superpixel matrix **SPs** and calculates which superpixels are adjacent. The function returns a cell array where each pair of indices i, j for which superpixels i and j are adjacent appear as a pair in one cell of this cell array. Note that it does not matter if i, j or j, i is listed, but only one of these two pairs should appear. Your task is to populate this cell array from the given adjacency matrix. (Hint: we suggest you use the function **GetSuperpixelAdjacencies.m** in implementing this function)

2. **[2 points] GetPottsFactor.m:** This function will create a Potts energy matrix E that will be used by each pairwise factor to enforce consistency between adjacent labels. The energy matrix should be 8 by 8 (as we have 8 labels) with $E(l_1, l_2)$ giving the energy for one of the adjacent superpixels taking label l_1 and the other taking label l_2. The energy for labeling l_1, l_2 for the two superpixels is $w\mathrm{I}(l_1 \neq l_2)$, where w is the given weight for the Potts factor energy and I is the indicator function which evalutes to 1 when true and 0 otherwise.

   (a) **[4 points]** In **PA1_eval.m**, use just the pixel based model with the Potts model (set the Potts weight to be nonzero, but leave all other weights as 0). Try increasing Potts weights. How does the inclusion of this factor impact performance? How do different Potts weights affect performance?

   (b) **[4 points]** Repeat the above experiment, but instead of using the pixel based model, try using the various sizes of superpixels. Note that you may have to significantly reduce the Potts weight. How does the Potts energy impact performance in the superpixel model? How does performance differ as we use larger superpixels?

3. **[3 points] GetContrastFactor.m:** As you may have seen in the previous question, a simple Potts factor may not always help segmentation. Another bit of intuition we can use to further improve our model is the idea that similar superpixels should probably have

the same label, while dissimilar superpixels are more likely to have differing labels. The energy matrix E should have the same structure as in **GetPottsFactor.m**, however, we will multiply our energies by a contrast weight. In this case, the energy between two adjacent superpixels should have the form $w\mathrm{I}(l_1 \neq l_2)\exp(-\lambda c)$. In this equation w is the weight as in our Potts factor, λ is the contrast factor, and c is the contrast between the two superpixels. You should use the function **GetPairwiseContrast.m** to calculate the contrast between two superpixels.

(a) **[4 points]** Again, use the pixel-based model, but this time with both a non-zero Potts weight and contrast weight. Vary the contrast and Potts weights. Does adding the contrast weighting improve performance? How do various values of the contrast weight affect segmentation?

(b) **[4 points]** In **PA1_eval.m**, start by using a Potts weight of .1 and a contrast weight of .0001 with the second smallest superpixels (set SPNum=2). How does adding the contrast weight affect segmentation as compared to the original labeling and Potts model? Vary the contrast weight and report how this affects segmentation. Does the contrast model improve segmentation?

4. **[5 points] GetFullSPEnergy.m:** In this file, you will define individual energies based on the labelings of adjacent pixels to go beyond simply enforcing consistency. One reason for this is that some co-occurrences, such as building above a street, might be common and should not be penalized, while others — such as building below grass — might be unlikely. We determine the spatial relationship of the two input superpixels, so all you need to do is fill the energy matrix. You should fill it such that $E(l_1, l_2)$ is the energy for a superpixel with label l_1 being above one with label l_2. Explain 3 energies you added to the matrix. Why do these make sense? Did they improve segmentation?

3.3.2 [17 points] Object-superpixel interactions

In this next part, we want to add in our potential object (car or person) detections. We will add a node for each candidate detection, which takes the value 1 if the detection corresponds to a real object (of the appropriate class) and 0 otherwise. This node will have a singleton energy derived from a probability output by an object detector we have run for you. We want to improve on the detections by using the knowledge that objects should appear as foreground in our image. In order to enforce this, we will add potentials between object detections and the superpixels beneath them.

1. **[3 points] CreateObjectSPEnergy.m:** Now you will define the energy for a factor between an object and a superpixel. The energy matrix E will be 2 by 8 as the object node will either have an object detected or not (2 values) and the superpixel must take one of 8 labels. The first row corresponds to energies for a false detection (value 0), while the second row has energies for a true detection (value 1). The energy should be 0 whenever an object is not detected, or the superpixel is foreground. In the other case, where the object is detected but the superpixel label is not foreground, the energy should be wc, where w is the weight for object-superpixel factors passed to the function and c is the coverage fraction returned by **CalculateObjectCoverage**.

(a) **[4 points]** In **PA1_eval.m**, define nonzero Potts and contrast weights and set the object weight to .25. How does this affect the detections and segmentation? Vary the object weight, how does the detection and segmentation vary as you change this weight? Why do you think this is?

2. [**10 points**] **GetObjectSPFactors.m**: This function creates the factors between the superpixels and potential object detections for our Markov network. The model we have defined for object/superpixel interactions is rather simple. For this question you will define your own type of factor between objects and superpixels that is not already included. One idea would be to define factors between superpixels that are above and/or below objects and the objects themselves. The preexisting code in this function should serve as a guide for how to create the factors. Note that the variable label for object i is actually i plus an offset denoted as $objStart$. Explain what sort of factors you added and what decisions you made in creating the energy function. Did you get performance improvement for detection? For segmentation? Are there particular entries in the segmentation confusion matrix that got better or worse as a consequence of your new energy?