CS 228, Winter 2009-2010 Programming Assignment #3—Learning in Graphical Models

This assignment is due at 11:59 PM on March 7.

---

1 Learning in Bayes Networks (Human Pose Model)

In this programming assignment you will explore learning in probabilistic graphical models. We use a directed model (Bayes Nets) for modeling and clustering human poses. You will tackle problems including learning CPDs for continuous variables, learning a tree-structured graph among body parts, and the EM algorithm to deal with a hidden class variable.

All code and data for this programming assignment are located at

/afs/ir/class/cs228/ProgrammingAssignments/PA3/code

Please test your code on corn clusters, where we will be grading your code.

1.1 Basic Settings

[Representation of human pose] We model the human body by ten parts as shown in Figure 1. The configuration of each body part is determined by 3 real numbers: (y, x, α), where (y, x) is the position of the anchor point of the body part (shown as a filled black square) and α is the orientation of the body part (shown as an arrow). The y value increases from top to bottom; x value increases from left to right, α increases clockwise and has the origin (zero value) pointing upright.

Given this setting, a pose of the human body can be specified by a 10×3 matrix, where the 10 rows correspond to body parts (in the order shown in Figure 1), the 3 columns are (in order) y, x, and α. We will provide you with a large number of example poses represented in this format for learning the model.

Suppose there are 3 underlying classes in the dataset, each represents a different "style" of pose variation. Our goal is to learn a model that captures the variation of human pose in each of the 3 classes. As the class labels are not provided to you in the training set, the problem is a typical example of what we have learned in the video chunk "Bayesian clustering using EM".

[Variables and their state spaces] Specifically, let us consider a Bayesian Net, which has one variable (node) for each of the body parts, and one variable (node) for the class label. The class variable is never observed, so we call it a *hidden variable*. We denote the body part variables as $\{\mathbf{O}_i\}_{i=1}^{10}$, each of which take on continuous values in $\mathbb{R}^3$: $\mathbf{O}_i = (y_i, x_i, \alpha_i)$[1]. And we denote the class variable as $\mathbf{C}$, which takes on discrete values $\{1, 2, 3\}$.

[1] You may have noticed that the domain of α is actually a circle rather than $\mathbb{R}$. Cutting the circle at any point (e.g., enforcing that $\alpha \in [-\pi, \pi]$) might introduce artifacts in modeling. However, for simplicity, in this programming assignment we will just assume that α can take any real value ignoring the fact that α and $\alpha + 2\pi$ are the same orientation.

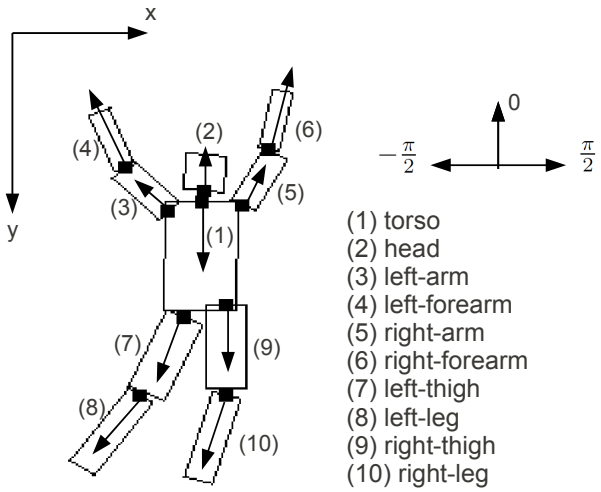

Figure 1: Representation of the human body. Note that the boxes for the body parts are just for visualization purpose, they are not part of the model specification, i.e., we don't actually know (nor try to model) the "length" or "width" of the body parts.

As we will be implementing the learning procedure for different choices of graph structure and local CPDs in a unified interface, we first go through all the theoretical discussions to get an idea of what to expect, then we introduce the data structure and functions, and you will code them up step-by-step.

1.2 Naive Bayes Pose Classification Model

As a first attempt, we adopt the naive Bayes assumption: all variables $\{\mathbf{O}_i\}_{i=1}^{10}$ are independent given the class label $\mathbf{C}$. This assumption gives rise to the graph structure shown in Figure 2 (a). Recall that this is the plate representation we have encountered in the first part of the course.

In the naive Bayes model, each body part only has the class label variable as its parent. The local CPDs can be parametrized as follows:

$$\mathbf{P}(\mathbf{C} = k) = c_k, \tag{1}$$

$$y_i|\mathbf{C} = k \quad \sim \mathcal{N}(\mu_{ik}^y, {\sigma_{ik}^y}^2), \tag{2}$$

$$x_i|\mathbf{C} = k \quad \sim \mathcal{N}(\mu_{ik}^x, {\sigma_{ik}^x}^2), \tag{3}$$

$$\alpha_i|\mathbf{C} = k \quad \sim \mathcal{N}(\mu_{ik}^\alpha, {\sigma_{ik}^\alpha}^2), \tag{4}$$

where $i = 1, 2, \cdots, 10$ indexes the body parts, and $k = 1, 2, 3$ indexes the classes. Given the class label, each continuous variable is modeled by a Gaussian distribution. Therefore each variable would have 3 sets of Gaussian parameters corresponds to the 3 classes. And we want to learn all these parameters from the training data.

Since the class label $\mathbf{C}$ is never observed, we use the EM algorithm. As a reminder of what we have learned in class, in the E-step, for each training example, we infer the conditional

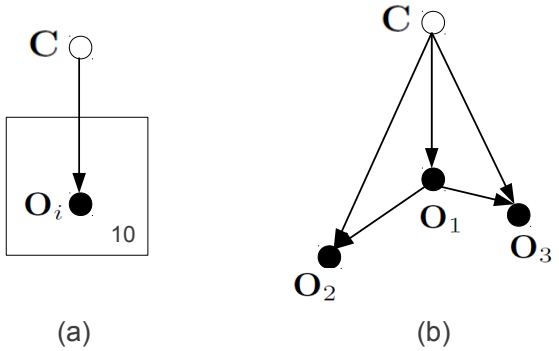

Figure 2: Graph structures. (a) The naive Bayes model. (b) Introducing edges among body part variables. Note that only 3 body part variables are shown here for clarity.

probabilities of its class label using current model parameters, and collect the expected sufficient statistics over the dataset. In the M-step, we re-estimate the model parameters using the expected sufficient statistics.

1.3 Learning with Known Skeleton Structure

The naive Bayes assumption is over-simplified: the body parts are connected by joints, clearly not independent to each other given the class label. In this subsection, we try to exploit the kinematic structure of the human body by adding edges among the body part variables.

Specifically, we take *torso* as the root variable, add *head*, *left* and *right arm, left* and *right thigh* as its children; and we add *left-forearm* as a child of *left-arm*, *left-leg* and a child of *left-thigh*, *right-forearm* as a child of *right-arm*, *right-leg* as a child of *right-thigh*. Without the class variable $\mathbf{C}$, the body parts form a tree-structured model (recall that a tree-structured model is a Bayes Net where each variable has at most one parent). With the class variable $\mathbf{C}$ being a parent of all the body part variables, we have the graph structure as shown (simplified) in Figure 2 (b), where we only drew $\mathbf{O}_1$, $\mathbf{O}_2$, $\mathbf{O}_3$ for clarity.

The next problem is how do we parametrize the local CPDs. There are two cases, the root body part (torso) only has the class variable $\mathbf{C}$ as its parent, so its parametrization is exactly the same as in the naive Bayes model. However, all other body parts have two parents: the class variable $\mathbf{C}$ and a parent body part. How do we parametrize that?

Recall that in the first part of the course we introduced the linear Gaussian model to parametrize local CPDs for continuous variable with continuous parents. Now with both continuous parents and a discrete parent $\mathbf{C}$, we use the **conditional linear Gaussian (CLG)** model, which is a straightforward generalization of the linear Gaussian model. Intuitively, for each possible assignment to the discrete parent, we have a different linear Gaussian model. In our case we have 3 possible classes, which give rise to 3 sets of linear Gaussian parameters for each continuous variable we want to model.

Specifically, the configuration of each child body part $\mathbf{O}_i = (x_i, y_i, \alpha_i)$ is modeled as a linear Gaussian in the configurations of its parent body part $\mathbf{O}_{p(i)} = (x_{p(i)}, y_{p(i)}, \alpha_{p(i)})$ conditioned on each possible assignment of the class label:

$$y_i | y_{p(i)}, \mathbf{C} = k \quad \sim \mathcal{N}(\theta_{ik}^{(1)} + \theta_{ik}^{(2)} y_{p(i)} + \theta_{ik}^{(3)} x_{p(i)} + \theta_{ik}^{(4)} \alpha_{p(i)}, \, {\sigma_{ik}^{y}}^2), \tag{5}$$

$$x_i|x_{p(i)}, \mathbf{C} = k \quad \sim \mathcal{N}(\theta_{ik}^{(5)} + \theta_{ik}^{(6)} y_{p(i)} + \theta_{ik}^{(7)} x_{p(i)} + \theta_{ik}^{(8)} \alpha_{p(i)},\ {\sigma_{ik}^{x}}^2), \tag{6}$$

$$\alpha_i|\alpha_{p(i)}, \mathbf{C} = k \quad \sim \mathcal{N}(\theta_{ik}^{(9)} + \theta_{ik}^{(10)} y_{p(i)} + \theta_{ik}^{(11)} x_{p(i)} + \theta_{ik}^{(12)} \alpha_{p(i)},\ {\sigma_{ik}^{\alpha}}^2), \tag{7}$$

where $i = 2, \cdots, 10; \quad k = 1, 2, 3$. Note that the CPDs (1) for $\mathbf{C}$ still applies here.

Now consider the problem of learning the parameters in this CLG model. The high-level structure of EM algorithm we discussed in the previous subsection still applies here. The differences are: in the E-step, we have to respect the new graph structure and parametrization in inferring the conditional probabilities of the class label; in the M-step, we need to fit a linear Gaussian instead of a Gaussian (except that we will still be fitting a Gaussian for the root body part). Details on how to fit the linear Gaussian parameters will be given when we reach the implementations part.

1.4 Learning with Domain-Specific Parametrization

As you will find out, the model you learned in the previous subsection, although it uses the "correct" graph structure, does not capture the variation of human poses very well (because you will be visualizing unrealistic samples from the learned model).

In this subsection, we will try to fix it by re-parametrizing the model (using the same graph structure). You might have realized that the position of a body part should not have a linear dependence with the orientation angle of its parent body part, but rather with the sine and cosine values. Specifically we introduce the following parametrization:

$$y_i|y_{p(i)}, \mathbf{C} = k \quad \sim \mathcal{N}(\gamma_{ik}^{(1)} + \gamma_{ik}^{(2)} y_{p(i)} + \gamma_{ik}^{(3)} x_{p(i)} + \gamma_{ik}^{(4)} \sin\alpha_{p(i)} + \gamma_{ik}^{(5)} \cos\alpha_{p(i)},\ {\sigma_{ik}^{y}}^2), \tag{8}$$

$$x_i|x_{p(i)}, \mathbf{C} = k \quad \sim \mathcal{N}(\gamma_{ik}^{(6)} + \gamma_{ik}^{(7)} y_{p(i)} + \gamma_{ik}^{(8)} x_{p(i)} + \gamma_{ik}^{(9)} \sin\alpha_{p(i)} + \gamma_{ik}^{(10)} \cos\alpha_{p(i)},\ {\sigma_{ik}^{x}}^2), \tag{9}$$

$$\alpha_i|\alpha_{p(i)}, \mathbf{C} = k \quad \sim \mathcal{N}(\gamma_{ik}^{(11)} + \gamma_{ik}^{(12)} y_{p(i)} + \gamma_{ik}^{(13)} x_{p(i)} + \gamma_{ik}^{(14)} \alpha_{p(i)},\ {\sigma_{ik}^{\alpha}}^2), \tag{10}$$

where $i = 2, \cdots, 10; \quad k = 1, 2, 3$. Note that the CPDs (1) for $\mathbf{C}$ still applies here, and (10) is still parametrized in angle.

The sine and cosine functions introduce nonlinear dependencies between the variables. However, these do not make the problem any harder as far as fitting the linear Gaussian parameters is concerned, because we can simply apply the sine and cosine functions to the orientation angle of the parent body part, and fit the linear Gaussian parameters using exactly the same method we used in the previous subsection.

1.5 Learning the Skeleton Structure from Data

Now let us assume that we don't know the skeleton structure of the human body (for whatever reasons you could imagine), and try to learn it from data.

We still assume that the class label variable $\mathbf{C}$ is a parent of all the body part nodes $\{\mathbf{O}_i\}_{i=1}^{10}$, and we want to learn a tree-structured model among the body parts. Recall that in class we have learned how to learn a tree-structured model. Given a score function that satisfies *score decomposability* and *score equivalence*, we compute the family score (weight) between all pairs of variables, and find the maximum spanning tree. Using the likelihood score, we have:

$$w_{i \to j} = \text{FamScore}_L(\mathbf{O}_i|\mathbf{O}_j : \mathcal{D}) - \text{FamScore}_L(\mathbf{O}_i : \mathcal{D}) = M \cdot \mathbf{I}_{\hat{P}}(\mathbf{O}_i, \mathbf{O}_j) \tag{11}$$

We have learned in class how to compute and use the mutual information between discrete variables. It turns out that the same concept applies to continuous variables as well.

In general, the mutual information between two multi-dimensional Gaussian variables $\mathbf{X}$ and $\mathbf{Y}$ (their dimensionalities can be different) can be computed by:

$$I(\mathbf{X}, \mathbf{Y}) = \frac{1}{2} \log \left(\frac{|\mathbf{\Sigma_{XX}}| \cdot |\mathbf{\Sigma_{YY}}|}{|\mathbf{\Sigma}|} \right) \tag{12}$$

where

$$\mathbf{\Sigma} = \begin{pmatrix} \mathbf{\Sigma_{XX}} & \mathbf{\Sigma_{XY}} \\ \mathbf{\Sigma_{XY}} & \mathbf{\Sigma_{YY}} \end{pmatrix}, \tag{13}$$

where $I(\mathbf{X}, \mathbf{Y})$ is the mutual information between $\mathbf{X}$ and $\mathbf{Y}$. And $|\bullet|$ denotes the determinant of matrices. The $\mathbf{\Sigma}$'s are covariance matrices. The log function in (12), and all other log functions in this programming assignment, refer to the logarithm with base 2.

1.6 Data Structure and Code

Now you will code up all the learning procedures discussed above step-by-step in a bottom-up manner. We will give you very specific guidelines in each step. We start with introducing the data structure.

In order to simplify the interface of the program, we decide to use a uniform data structure to represent the parameters and graph structures we encountered in all previous subsections. Specifically, the parameters are encoded in a structural array P. The 3×1 vector $P.c$ encodes the class label probabilities in (1). The graph parametrization choice is encoded in a 10×2 matrix G, each row of G corresponds to a body part (in the order as shown in Figure 1), the first column of G represents the graph parametrization choice:

- $G(i, 1) = 0$ indicates that it only has the class variable as its parent (in this case $G(i, 2)$ can be an arbitrary value). Its parametrization follows (2,3,4). The parameters can be found in 3×1 vetors: $P.clg(i).mu\_x$, $P.clg(i).mu\_y$, $P.clg(i).mu\_angle$, $P.clg(i).sigma\_x$, $P.clg(i).sigma\_y$, $P.clg(i).sigma\_angle$. For example, $P.clg(2).sigma\_y(3)$ is the standard deviation of the y-position of the head given that the class label is 3.

- $G(i, 1) = 1$ indicates that it also has a parent $G(i, 2)$, and the CPDs are parametrized following (5,6,7). The parameters can be found in the 3×12 matrix $P.clg(i).theta$ and 3×1 vectors $P.clg(i).sigma\_x$, $P.clg(i).sigma\_y$, $P.clg(i).sigma\_angle$. For example, $P.clg(i).theta(k, 9)$ corresponds to $\theta_{ik}^{(9)}$ in equation (7).

- $G(i, 1) = 2$ indicates that it also has a parent $G(i, 2)$, and the CPDs are parametrized following (8,9,10). The parameters can be found in the 3×14 matrix $P.clg(i).gamma$ and 3×1 vectors $P.clg(i).sigma\_x$, $P.clg(i).sigma\_y$, $P.clg(i).sigma\_angle$. For example, $P.clg(i).gamma(k, 3)$ corresponds to $\gamma_{ik}^{(3)}$ in equation (8).

In the last question of this assignment, we will be dealing with multiple graph structures (one for each of the K classes, such that we can learn it in the EM algorithm). And we will be using a $10 \times 2 \times K$ matrix G, where $G(:, :, k)$ specifies the graph structure and parametrization for the k-th class (in the same way as described above).

Now we will first go through all functions (".m" files) that are provided to you, which you can invoke in your program without modifying.

FILES THAT DO NOT NEED MODIFYING

- **data.mat**: All the training data, graph structure, initial states that you need in the experiments.
- **ShowPose.m**: This function outputs an bitmap image of the human figure given a pose configuration specified by a 10×3 matrix.
- **SamplePose.m**: This function takes as input the graph structure G and the parameter structural array P in the format specified above, samples from the distribution and outputs a pose configuration represented by a 10×3 matrix. You can also fix the class label and sampled from the conditional distribution. It calls **SampleGaussian.m** and **SampleMultinomial.m**, which are as simple as their names suggest.
- **VisualizeModels.m**: This function give you an intuitive assessment of the quality of your learned model. It calls **SamplePose.m** and **ShowPose.m**, and shows a window with 3 sub-windows, each of which showing an animation of samples from the model given a fixed class label. So you know that you have done a terrific job when you see 3 different styles of realistic pose variations.
- **VisualizeDataset.m**: This function shows each example pose in a dataset one-by-one in an animation.
- **MaxSpanningTree.m**: This function finds the maximum spanning tree given the symmetric weight matrix. Output of this function is represented as an (non-symmetric) adjacency matrix A (where one of $A(i,j)$ and $A(j,i)$ is 1 if i and j are connected by an edge, both are 0 otherwise).
- **DrawGraph.m**: Draw the graph structure among body part nodes. This function is invoked in **LearnGraphStructure.m** to visualize the result.
- **ConvertAtoG.m**: Convert the adjacent matrix representation into our representation G of the tree structured model. In this function we always use the first body part (torso) as root node (such that the conversion is unique), and we always use the parametrization in (8,9,10) (by setting $G(i,j) = 2$ for $i \neq 1$).

Now let us code up all the learning procedures step by step. You will be implementing the functions (or part of it) in the following order:

FILES YOU NEED TO WORK ON

- (5 pts) **FitGaussianParameters.m**: In this function you will implement the algorithm for fitting maximum likelihood parameters $\mu = \mathbf{E}_{\hat{P}}[X]$ and $\sigma = \sqrt{\mathbf{Var}_{\hat{P}}[X]}$ of the Gaussian distribution given samples and weights. Note that the Matlab functions *mean* and *var* does not take into account the weights on samples, thus cannot be used in the EM algorithm. Instead, you should compute the variance as $\mathbf{Var}_{\hat{P}}[X] = \mathbf{E}_{\hat{P}}[X^2] - \mathbf{E}_{\hat{P}}[X]^2$, where $\mathbf{E}_{\hat{P}}$ is the expectation with respect to the empirical distribution $\hat{P}$, which is specified by the dataset $\mathcal{D}$ and the weights of examples (also given as a input parameter of the function. Incorporating the weights allows us to use this function in the EM algorithm. Also note that this function needs to return the standard deviation σ instead of the variance σ^2.
- (15 pts) **FitLinearGaussianParameters.m**: In this function you will implement the algorithm for fitting linear Gaussian parameters in the general form (such that it can be used in different scenarios):

$$X|U \sim \mathcal{N}(\beta_1 U_1 + \beta_2 U_2 + \cdots + \beta_k U_k + \beta_{k+1}, \sigma^2) \tag{14}$$

Note that we use β_{k+1} in place of the usual notation β_0 (in the textbook) because Matlab does not allow 0 as an index of array, and you should keep this in mind when you invoke this function in subsequent tasks. By taking derivatives of the log-likelihood function[2], we can see that the parameters $(\beta_1, \cdots, \beta_{k+1})$ satisfy the following system of linear equations:

$$\mathbf{E}_{\hat{P}}[X] = \beta_1 \mathbf{E}_{\hat{P}}[U_1] + \beta_2 \mathbf{E}_{\hat{P}}[U_2] + \cdots + \beta_k \mathbf{E}_{\hat{P}}[U_k] + \beta_{k+1} \tag{15}$$

$$\mathbf{E}_{\hat{P}}[X \cdot U_i] = \beta_1 \mathbf{E}_{\hat{P}}[U_1 \cdot U_i] + \beta_2 \mathbf{E}_{\hat{P}}[U_2 \cdot U_i] + \cdots + \beta_k \mathbf{E}_{\hat{P}}[U_k \cdot U_i] + \beta_{k+1} \mathbf{E}_{\hat{P}}[U_i] \tag{16}$$

where $i = 1, 2, \cdots, k$.

Now we have $k+1$ variables and $k+1$ equations, so we can use the Matlab function *linsolve* to solve them. Having obtained the parameters $(\beta_1, \cdots, \beta_{k+1})$, the standard deviation σ can be computed by:

$$\sigma = \sqrt{\mathbf{Cov}_{\hat{P}}[X; X] - \sum_{i=1}^{k} \sum_{j=1}^{k} \beta_i \beta_j \mathbf{Cov}_{\hat{P}}[U_i; U_j]} \tag{17}$$

Note that you cannot use the Matlab function *cov* to compute the covariance. (Why?)[3]. Instead, you should compute covariance $\mathbf{Cov}_{\hat{P}}[X; Y] = \mathbf{E}_{\hat{P}}[X \cdot Y] - \mathbf{E}_{\hat{P}}[X] \cdot \mathbf{E}_{\hat{P}}[Y]$ by reusing some statistics computed in the previous step.

- (15 pts) **ComputeLogJointProb.m**: In this function you will compute the joint probabilites of each possible class label and the (observed) pose configuration $\{\mathbf{O}_i\}_{i=1}^{10}$. The result of this function can be used to compute (a) the conditional probabilities of class labels given the pose (which is used to "re-assign" (in a soft manner) the examples to the classes given the current estimation of parameters), and (b) the log likelihood of the learned parameters over the dataset (which is used to assess how the model "fits" the observed dataset.)

  Specifically, we have:

$$P(\mathbf{C} = k, \mathbf{O}_1, \cdots, \mathbf{O}_{10}) = P(\mathbf{C} = k) \prod_{i=1}^{10} P(\mathbf{O}_i | \mathbf{C} = k, \mathbf{O}_{p(i)}). \tag{18}$$

  Recall that $\mathbf{O}_{p(i)}$ is the parent of $\mathbf{O}_i$, and $p(i)$ can be find in $G(i, 2)$. You can use the Matlab function *normpdf* to evaluate the Gaussian density function. However, be careful with numerical issues introduced by extremely small numbers returned by *normpdf*. (Hint: instead of multiplying probabilities, try adding up log-probabilities. Actually this function should return log-probabilities instead of probabilities). Please follow the instructions given as comments in this file to complete the implementation. You may refer to the usage of this function in **LearnCPDsGivenGraph.m**.

- (5 pts) **ComputeConditionalClassProb.m** This function computes the probabilites of the class label given the pose (which is used to "re-assign" (in a soft manner) the examples to the classes):

$$P(\mathbf{C} = k | \mathbf{O}_1, \cdots, \mathbf{O}_{10}) \tag{19}$$

[2] If you are interested in the details of derivation, please refer to **17.2.4 Gaussian Bayesian Networks** of the text book. This is optional and not necessary for completing the programming assignment.

[3] As we have mentioned, because it ignores the weights in $\hat{P}$.

You should be able to compute this using the log joint probabilities computed in the last function you just implemented. (You don't explicitly invoke **ComputeLogJointProb.m** here. Instead, its output is passed as input of this function. And we will invoke these two functions one after another in the learning procedure.) You may refer to the usage of this function in **LearnCPDsGivenGraph.m**.

- (10 pts) **ComputeLogLikelihood.m** This function compute the log likelihood of the model over the dataset (which measures how the model "fits" the observed data).

$$\sum_{i=1}^{N} \log P(\mathbf{O}_1 = \mathbf{o}_1^{(i)}, \cdots, \mathbf{O}_{10} = \mathbf{o}_{10}^{(i)}) \tag{20}$$

  Again, you should be able to compute this using the log joint probabilities computed in the function **ComputeLogJointProb.m**, whose output is passed as input of this function. You may refer to the usage of this function in **LearnCPDsGivenGraph.m**.

- (10 pts) **LearnCPDsGivenGraph.m**: This is the function where we actually run the EM procedure to learn the parameters. This function takes the dataset and a known graph structure G as input, looks into G to decide which parametrization is used for each of the body parts. So this function handles all the 3 cases we discussed in a unified interface. We have already written some infrastructure code in this function for you, and all you need to do is to invoke the functions you just implemented by feeding them with the correct parameters, and write the output in the correct format into the parameter structural array P. Please follow the instructions given as comments in this file to complete the implementation.

- (5 pts) **GaussianMutualInformation.m**: This function computes the mutual information between two multi-dimensional Gaussian variables (they can have different dimensionality). You can implement it by following the equations (12) and (13). The covariance matrix can be computed using the Matlab function *cov*.

- (10 pts) **LearnGraphStructure.m**: Given the last function you just implemented and the function **MaxSpanningTree.m**, learning the tree-structured graph should be quite straightforward. However, as the position of a body part has a linear relationship with the cosine and sine of the orientation angle of its parent body part. We should feed $[y_i, x_i, \alpha_i, \sin(\alpha_i), \cos(\alpha_i)]$[4] (instead of $[y_i, x_i, \alpha_i]$) into the function we just implemented to compute mutual information. Please follow the instructions given as comments in this file to complete the implementation.

1.7 Experiments and Questions

1.7.1 Experiments

Before we move on, you should check to make sure that all your implementations are correct. We provide you with an easy way to do this. Just run the Matlab script *TestPA3*.

Now you have implemented (and verified) all the learning procedures we have discussed in this section. Play around with it for a while and then answer the short questions in the end of the section.

[4]It does not follow the 5 dimensional Gaussian distribution as we have nonlinear deterministic relations among the variables. However, empirically this gives more robust results in estimating the graph structure.

First of all, you can load the dataset, graph structures and initial states by:
load data;

And then you can learn the parameters by:
[P1 likelihood1] = LearnCPDsGivenGraph(DatasetTrain, G1, InitialClassProb1. 20);
[P2 likelihood2] = LearnCPDsGivenGraph(DatasetTrain, G2, InitialClassProb1. 20);
[P3 likelihood3] = LearnCPDsGivenGraph(DatasetTrain, G3, InitialClassProb1, 20);
Note that we use the same initial state in learning all the three models. Compare the likelihood scores you obtained in these 3 runs. You can see (quantitatively) how more complicated model better "fits" the data.

Now let us compare them qualitatively. Your learned models can be visualized by:
VisualizeModels(P1, G1);
VisualizeModels(P2, G2);
VisualizeModels(P3, G3);
Does the model learned by the naive Bayes assumption capture the variation of human poses? How about the second and third model? (You don't need to submit the answers to these questions.) Note that the samples generated by the model reflect the model's "understanding" of how human poses vary.

You can learn the graph structure and visualize the results by
LearnGraphStructure(DatasetTrain, 1);

1.7.2 Questions

[IMPORTANT] You must include the answers to questions 1, 2 and 3 in **SEPARATE ASCII FILES FOR EACH QUESTION**. They should be named **1.txt**, **2.txt** and **3.txt**. For questions 1 and 2, these txt files should look like: **"Yes, ...", or "No, ... "**. (First answer the question with "Yes" or "No", then briefly explain.) **Do NOT include other information** (such as your name) before your answer. Question 4 only requires code submission. Any submission violating this format will not get credit for that question.

1. (5 pts) **[Different Initial States]** Try to learn the model for *G3* from two different initial states: *InitialClassProb1* and *InitialClassProb2*:

   [P3 loglikelihood1] = LearnCPDsGivenGraph(DatasetTrain, G3, InitialClassProb1, 20);

   [P3 loglikelihood2] = LearnCPDsGivenGraph(DatasetTrain, G3, InitialClassProb2, 20);

   Compare the log-likelihoods you obtained. (Try to plot them in the same figure.) You can see that *InitialClassProb1* is a better initial state (the log-likelihood in first iteration is higher). And after the EM-algorithm converged, the first run reached a higher log-likelihood than the second run. Now suppose we have another new initial state, whose log-likelihood in the first iteration is even higher than that of *InitialClassProb1*. Is running from this new initial state guaranteed to reach a better (or at least the same) final log-likelihood value than running from *InitialClassProb1*? Briefly explain.

2. (5 pts) **[Model Complexity]** Now try learning with 10 (instead of 3) class labels. You can try this by:

   [P3 loglikelihood3] = LearnCPDsGivenGraph(DatasetTrain, G3, InitialClassProb3, 10);

Because *InitialClassProb3* has 10 columns, the algorithm will automatically figure out that we are learning with 10 class labels. (This will be significantly slower than learning with 3 class labels, so we will only run 10 EM iterations.) Compare the log-likelihood with the ones you obtained in the previous question. We have already seen from the previous experiments that the model with higher log-likelihood score better captures the pose variation. Now the new model with 10 class labels has even higher log-likelihood value. Is this a better model? Specifically, suppose we apply the new model (with 10 class labels) and the old one (with 3 class labels) to a new dataset that is different from the training set but sampled from the same underlying distribution. Is the new model guaranteed to have higher log-likelihood value on the new dataset than the old model? Briefly explain.

3. (5 pts) **[Model Selection]** Now suppose we have a dataset (we don't know how many underlying classes are there in the dataset, and we don't have another dataset from the same distribution to test our model). How do you determine the number of class labels to use in training the model? (There are several different correct answers to this question, you can get full credit by giving one of them.)

4. (10 pts) **[Learning Graph Structure in EM]** We have another dataset with two underlying classes. One of them is human poses, the other is poses of some kind of alien (they may have a different body structure). Now we try to learn a model to cluster them and capture the pose variation of these two spieces respectively. First we try to learn it using a fixed graph structure (say, $G3$) and visualize the results:

   [P loglh1] = LearnCPDsGivenGraph(DatasetTrain_aliens, G3, InitialClassProb_aliens, 20);

   VisualizeModels(P, G3);

   You can see that the model captures the variation of one class (the human class) very well. But the pose variation of the other class is not successfully learned. (You will see samples with body parts detached from each other, and the body structure of the aliens is not clear.) This is because the graph structure we are using is not the correct graph structure for the aliens.

   In order to learn a better model in this scenario, we have to learn the graph structure within the EM algorithm. That is, for each EM iteration, we not only re-estimate the parameters for each class, but also re-estimate the graph structure for each class. Please complete the implementation of **LearnGraphAndCPDs.m** by following the instructions given as comments. (Hint: You can copy most of the code directly from **LearnCPDsGivenGraph.m**.) Note that we use hard-assignment EM[5] here (which is already implemented for you) because the function for learning graph structure we have implemented does not incorporate weights. When you are done, try:

   [P G loglh2] = LearnGraphAndCPDs(DatasetTrain_aliens, InitialClassProb_aliens, 20);

   VisualizeModels(P, G);

   If your implementation is correct, you can clearly see the body structure of the aliens from the samples of the model. And you will also see that the new algorithm achieves higher log likelihood on the dataset. For this question you only have to submit the code (which will be tested on a different dataset).

[5] The hard assignment EM is a variant of the EM algorithm, where we assign each example to one class (with the highest conditional probability) without weighting. If you are interested in details, please refer to **19.2.2.6 Hard-Assignment EM** in the textbook.