PGM Programming Assignment: Learning with Incomplete Data

1 Introduction

In the previous programming assignment, you learned tree-structured Bayesian networks to represent body poses. In this assignment, you will be using the network you learned for human poses (in the previous assignment) to perform action recognition on real-world Kinect^{TM}data. The Kinect^{TM}data was collected by performing actions and extracting the human pose data associated with each action. We have collected data for the following three actions: "clap", "high_kick", and "low_kick". Your goal in this assignment is to build a model that is able to identify the action performed.

The approach that we will take in this assignment is to model an action as a sequence of poses. However, unlike in the previous programming assignment, we do not have the class labels for these poses. More specifically, in that assignment we knew whether a pose belonged to a human or an alien, but in the Kinect^{TM}data, we do not know in advance what poses comprise an action. In fact, we do not even know what the possible poses that comprise the various actions are. As such, we will have to build and learn models that incorporate a latent variable for pose classes, which makes this a more challenging task than the classification problem you previously solved. You will make extensive use of the Expectation-Maximization (EM) algorithm to learn these models with latent variables.

2 Data Representation

We have processed the action data into a sequence of constituent poses. Each action has a different number of poses in the sequence depending on the length of the action. The constituent poses are in the same format used in the previous assignment: that is, each pose has the 10 body parts each with the three (y, x, α) components representing the pose. Refer to Section 2 and Figure 1 from the previous assignment for the representation and notation of each of the parts.

More formally, you are given a set of actions $a_1, \ldots, a_n$. Each action a_i is a sequence of poses $p_1^i, \ldots, p_m^i$, where the number of poses m is the length of the action. A pose p_j^i can be described by its body parts $\{\mathbf{O}_k\}_{k=1}^{10}$, with $\mathbf{O}_k = (y_k, x_k, \alpha_k)$. A pose is stored as a 10 x 3 matrix.

We will also use two of the data structures you saw earlier, so take a moment to re-familiarize yourself with them:

- **P** — This structure holds the model parameters, and is the same as the one used before. Slight changes are made to this structure for the section on HMM's, which we describe when they come up.
- **G** — This is the same graph structure used before for defining the tree structure dependencies between parts of a pose. Remember to handle all possible parameterizations of **G** in your code.

3 Bayesian Clustering of the Poses

You will be experimenting with two latent variable models. The first is a Bayesian clustering model, which you will use to cluster poses from different actions: we will combine the constituent poses from all the action data and cluster them into groups of similar poses. Ideally, we will automatically discover meaningful pose clusters where the majority of poses in the each cluster are part of a single action.

The structure of the model is exactly as shown in Figure 1. This is similar to the model from before, with a key difference that *the class variables C are now hidden variables*; these variables were observed in the previous assignment and given as part of the training data. As such, we no longer have a completely labeled dataset to learn the model from, and we will use the EM algorithm to learn the model from the partially labeled data.

In the clustering model, each cluster, or class, is associated with its own set of conditional linear Gaussian (CLG) parameters for the tree-structured human pose network. If we have K classes, then we will have K sets of CLG parameters. Our goal in the EM procedure is to learn the CLG parameters for each class without knowledge of the true class labels for each pose. This is similar to what you did before, except that now we are not given the class label for each of the poses. We describe the EM algorithm for learning the model in the following section.

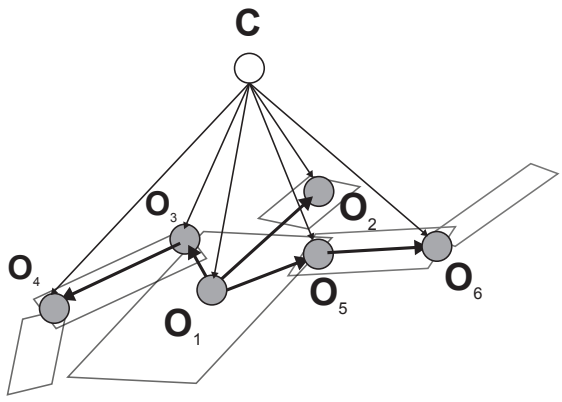

Figure 1: Graphical model for Bayesian clustering of poses. The class variable is hidden here, and we've only shown the first 6 body parts for clarity.

3.1 The EM algorithm for Bayesian Clustering

The general concept of EM is to generate a set of soft assignments for your hidden variables (E-step) and use these assignments to update your estimates of the model parameters (M-step). Each of these steps is done to maximize the model's likelihood taking either parameters or unknown data as given, and we iterate back and forth between these steps until we have reached a local maxima and can no longer improve the likelihood. We'll take a look at the details of these steps below:

E-step: In the E-step our goal is to do soft assignment of poses to classes. Thus, for each pose, we infer the conditional probabilities for each of the K class labels using the current model

parameters. This can be done by first computing the joint probability of the class assignment and pose (body parts), which decomposes as follows:

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

where $pa(i)$ is the index of part i's parent. Each of the probabilities in the product can be computed as described in the previous assignment, and vary depending on the tree structure used. Once we have computed the joint probability of the class assignment and pose, we can then condition on the pose to obtain the conditional class probability (which are our expected sufficient statistics):

$$P(\mathbf{C} = k | \mathbf{O}_1, \cdots, \mathbf{O}_{10}) \tag{2}$$

Note that this is the conditional class probability for a single pose, and you will need to compute this probability for all poses in the dataset.

M-step: In M-step, we seek to find the maximum likelihood CLG parameters given our soft assignments of poses to classes (conditional class probabilities) from the E-step. Thus, for each class k, we fit the CLG parameters with each pose weighted by its conditional probability as given in Equation 2. The weight of a particular pose for class k is equivalent to the current estimated probability that the pose belongs to the class. This step is identical to the parameter fitting step in the previous assignment, with the only difference that each pose now makes a weighted contribution to the estimation of the CLG parameters for all classes (as opposed to only for the given class).

3.2 Data Structures and Code

Now let's get to the implementation. We have provided you with the general structure for the EM algorithm and the expected input and output parameters. It's up to you to implement the rest!
Note: if you need to normalize probabilities, do this in the log probability space before converting to probabilities. This is required to avoid numerical issues.

- `EM_cluster.m` **(35 points)** — This is the function where we run the overall EM procedure for clustering. This function takes as input the dataset of poses, a known graph structure G, a set of initial probabilities, and the maximum number of iterations to be run. Descriptions of the input and output parameters are given as comments in the code. Follow the comments and structure given in the code to complete this function. Note that we start with the M-step first.

Note on decreasing likelihood: You may notice that the log likelihood can decrease sometimes. This is because we are only printing out the log likelihood of the data. Since we smooth the variances of our Gaussians to avoid numerical issues, the log likelihood of this smoothing prior can affect things. If we calculated the log likelihood of the data together with the prior, that number should always increase. The same holds in the following sections as well.

You may find the following functions that we've provided useful in your implementation:

- `FitG.m` — This is similar to the function you wrote before, while allowing for weighted training examples. The weights control the influence of each example on the parameters learned.

- `FitLG.m` — Similar to the function you wrote before, with the same addition of weights as in `FitG.m`.
- `lognormpdf.m` — Computes the natural logarithm of the normal probability density function.
- `logsumexp.m` — Computes the log(sum(exp(X))) of the input X. This function's implementation avoids numerical issues when we must add probabilities that are currently in log space.

Relevant Data Structures:

- **poseData** — N x 10 x 3 matrix that contains the pose data, where N is the number of poses, and each pose is associated with a 10 x 3 matrix that describes its parts, similar to before. The poseData is created by randomly sampling poses from all of the actions.
- **ClassProb** — N x K matrix that contains the conditional class probabilities of the N examples to the K classes. ClassProb(i, j) is the conditional probability that example i belongs to class j.

Sample Input/Output: To test your code, a seperate file **PA9SampleCases.mat** contains the input and correct output for each of the sample cases that we test for in the submit script (Similar to the previous assignment). For argument j of the function call in part i, you should use `exampleINPUT.t#`$_i$`a#`$_j$ (replacing the $\#_j$ with j). For output, look at `exampleOUTPUT.t#`$_i$`o#`$_j$ for argument j of the output to part i.

3.3 Evaluating the Clusters

With the implementation complete for the clustering algorithm, we can proceed to see how well the clustering algorithm works in practice. We have given you a pre-extracted pose dataset, a graph structure, and a set of initial probabilities in the file **PA9Data.mat**. You can also experiment on your own with different graph structures and initial probabilities.

Try running the following command, which clusters the poses into 3 clusters:

```
[P loglikelihood ClassProb] = EM_cluster(poseData1, G, InitialClassProb1, 20)
```

With the posterior class probabilities **ClassProb**, we can obtain the cluster that each pose associates most strongly with, and visualize the poses in each of these clusters. You can visualize a set of poses using the function `VisualizeDataset.m`. These two steps can be done with the following commands to visualize poses in cluster i:

```
[maxprob, assignments] = max(ClassProb, [], 2)
VisualizeDataset(poseData1(assignments == i, :, :))
```

We can see from the visualized poses that the clusters are not perfect, as the Kinect^{TM}data is not as well-behaved as the synthetic data used before. In addition, we can look at the ground truth action labels in the **labels1** vector, which gives the action class that each pose was taken from. This can be done for poses in cluster i as follows:

```
labels1(assignments == i)'
```

We can see that the clusters typically do not consist of poses from a single action. This makes intuitive sense, as actions may share similar poses. For example, in the "high_kick" and "low_kick" actions, it's possible that there are poses that both have the leg raised to the waist. We've also provided a set of initial probabilities for clustering into 6 clusters using the following command:

```
[P loglikelihood ClassProb] = EM_cluster(poseData1, G, InitialClassProb2, 20)
```

What happens when we have twice the number of clusters? Are the clusters more concentrated with a single action class? Do the visualizations look better?

4 A Bayesian Classifier for Action Recognition

In the action recognition problem, we are given a set of N training examples $\mathcal{D} = \{(a_i, c_i)\}_{i=1}^N$ consisting of actions a_i and their associated classes c_i; there are 3 different types of actions in the training set, so $c_i \in \{1, 2, 3\}$, where 1 is "clap", 2 is "high_kick" and 3 is "low_kick". We wish to build a model to classify unseen actions. To do so, we will use the general framework of a Bayesian classifier: we model the joint distribution over an action A and its class C as

$$P(A, C) = P(A|C)P(C) \tag{3}$$

The intuition behind this model is that each action has a set of distinct characteristics that we can capture using the class conditional action model $P(A|C)$.

We can classify an action a by picking the class assignment c^* with the highest posterior probability $P(c^*|a)$ given the action:

$$c^* = \arg\max_c P(c|a) = \arg\max_c \frac{P(a|c)P(c)}{P(a)} = \arg\max_c P(a|c) \tag{4}$$

The last equality follows because we have a uniform class distribution in the training set, and the denominator $P(a)$ is the same for each of the classes. Thus, given an unseen action, we can classify it by simply computing $P(a|c)$ over each class, and picking the class whose model yields the highest likelihood.

We have yet to specify a key component of our classifier: the class conditional action model $P(A|C)$. What model should we use? One possibility would be to fit a Bayesian clustering model (a mixture of poses model) for each action to model the types of poses that appear in each action class. However, we can guess that this will probably not work, as suggested by the clustering results in the previous section where many of these actions shared poses that look similar. Thus, it is likely that the class conditional action models will be similar and the classifier will not perform well.

A key observation about actions that will help us build a better action model is that though the poses comprising an action may look similar, or even be the same, the sequence in which these poses occur defines the action. For example, in the "low_kick" action, we would expect a sequence of poses in which the foot is lifted, kicked in a direction, and then returned back to the original position. Thus, we should try to leverage the *temporal* nature of action poses in our model. The mixture of poses model does not account for this temporal nature of actions, so we will turn to a different model that allows us to leverage this information.

4.1 HMM action models

Using a Hidden Markov Model (HMM) for the action model will allow us to capture the sequential nature of the data. Given an action a_i consisting of a sequence of m poses $p_1^i, \ldots, p_m^i$, we

can construct a HMM of length m with hidden state variables $S_1, \ldots, S_m$ for each pose that correspond to the unknown pose classes. The HMM action model defines a joint distribution over the hidden state variables and the poses comprising an action of length m:

$$P(S_1, \ldots, S_m, P_1, \ldots, P_m) = P(S_1)P(P_1|S_1)\prod_{i=2}^{m} P(S_i|S_{i-1})P(P_i|S_i)$$

Since the HMM is a template model, it is parameterized by 3 CPDs – the prior distribution over the initial states $P(S_1)$, the transition model $P(S'|S)$, and the emission model $P(P|S)$. The first two CPDs $P(S_1)$ and $P(S'|S)$ are table CPDs, while the emission model for pose P_j with parts $\{\mathbf{O}_k\}_{k=1}^{10}$ is

$$P(P_j|S) = \prod_{i=1}^{10} P(\mathbf{O}_i|S, \mathbf{O}_{pa(i)})$$

where $pa(i)$ denotes the parent of node i, is actually the pose model that you worked with in Section 2.4 of the previous assignment, with the skeletal structure for humans that you learned in the assignment. This equation is very similar to Equation 1, with the only difference here being that we aren't accounting for prior class probabilities anymore. Figure 2 shows an example HMM for an action consisting of a sequence of 3 poses, where we have explicitly shown the first 6 body parts that comprise a pose.

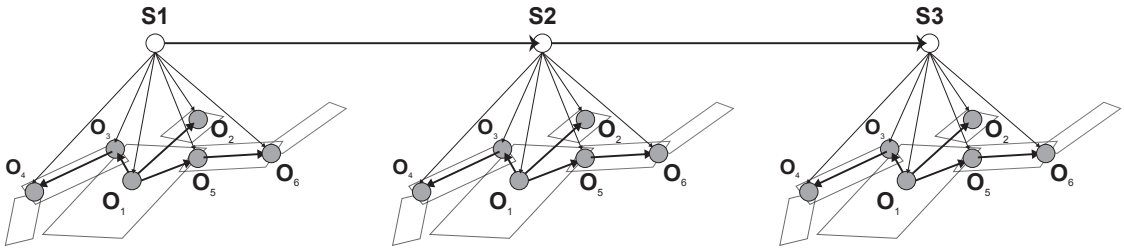

Figure 2: Graphical model for the HMM. In our HMM, each state variable represents the class of the underlying pose. The emission probabilities for each pose are computed using the learned parameters.

4.2 Learning the HMM action model using EM

Since the state variables are hidden, we will use the EM algorithm to learn the parameters of the HMM. The general EM framework (iterating between estimating assignments and parameters) does not change, but the specifics of these steps can be tricky so we will now describe each of the steps in detail.

E-step: In the E-step, we would like to compute the expected sufficient statistics needed to estimate our parameters. The two sets of expected sufficient statistics needed are the marginals over each individual state variable S_i, as well as the pairwise marginals over consecutive state variables S_i and S_{i-1}.
Though computing these marginals may seem daunting, the E-step of the algorithm can be implemented using clique tree inference, which you wrote in PA4 and used in PA7. We include

optimized solution code for the functions from PA4 to help with inference in your HMM model, which you can call using **ComputeExactMarginalsHMM.m**. In particular, we have modified it to work in log space to avoid numerical issues.

As described in Section 4.1, the HMM consists of 3 types of factors (CPDs) that you will need to include in your clique-tree for performing inference in the E-step. Note that for a single action a_i with a sequence of m observed poses $p_1^i, \ldots, p_m^i$, you will be able to reduce the factors $\phi(P_j, S_j) = P(P_j|S_j)$ by the observed poses p_j^i to obtain singleton factors over S_j alone, $\phi'(S_j) = \phi(p_j^i, S_j)$. With these factors, we can run inference on the HMM for our action a_i.

After running inference, we can extract the expected sufficient statistics $M_j[s_j] = P(S_j = s_j|a_i)$ and $M[s', s] = \sum_{j=2}^{m} P(S_j = s', S_{j-1} = s|a_i)$ from the calibrated clique tree for estimating the initial state prior and the transition CPD, as explained below. Note that these statistics are described for a single action, and you will need to aggregate these statistics across the entire dataset of actions. We also extract the conditional class probabilities $P(S_j = s_j|a_i)$ for each state to estimate the emission CPD, as explained below. Also, as in PA7, we can compute the likelihood of an action by marginalizing out the probabilities of any of the cliques.

M-step:
Using the aggregated expected sufficient statistics, we can re-estimate the parameters in the model:

- **Initial state prior:** The prior $P(S_1)$ can be re-estimated as:

$$P(S_1 = s_1) = \frac{M_1[s_1]}{\sum_{k=1}^{K} M_1[k]}$$

- **Transition CPD:** We can re-estimate these using the expected sufficient statistics as:

$$P(S' = s'|S = s) = \frac{M[s', s]}{\sum_{k=1}^{K} M[k, s]}$$

  Note that the provided code incorporates a Dirichlet prior to avoid zero probabilities. This is effectively equivalent to adding a small constant to both the numerator and each summand in the denominator of the above expression. This constant is already computed for you in the code.

- **Emission CPD:** The parameters for the emission probabilities are the CLG parameters used in pose clustering, and can be fit in the same way as the previous section, where we treat each pose as having a weight equal to its conditional class probability, which reduces to the M-step from pose clustering. Essentially, in this step you are doing pose clustering over all the poses in all the actions.

4.3 Data Structures and Code

In practice, there are many numerical issues that arise when implementing the HMM. Thus, for many of these functions, we would like to keep our probabilities in log space whenever possible, and to always normalize in log space to avoid numerical issues. Before you start writing this function, be sure to go over the details of the useful data structures and functions in Sections 5 and 6.

- `EM_HMM.m` **(35 points)** — This is the function where you will implement the EM procedure to learn the parameters of the action HMM. Many parts of this function should be the same as portions of `EM_cluster.m`. This function takes as input the dataset of actions, the dataset of poses, a known graph structure G, a set of initial probabilities, and the number of iterations to be run. Descriptions for the input and output parameters are given as comments in the code. Follow the comments and structure given in the code to complete this function. Note that we start with the M-step first.

4.4 Recognizing Actions

Now that you've implemented the EM algorithm to learn an HMM, we can now move on to our end goal of action recognition. As discussed in Section 4, we will be using a Bayesian classifier, which means that we will train a separate HMM for each action type, then classify actions based on which HMM gives the highest likelihood for the action.

- `RecognizeActions.m` **(10 points)** — In this function, you should train an HMM for each action class using the training set **datasetTrain**. Then, classify each of the instances in the test set **datasetTest** and compute the classification accuracy. Details on the **datasetTrain** and **datasetTest** data structures can be found in Section 6.

At this point you have successfully implemented an action recognition system. Let's see how well this performs in practice. Run the command:

```
load PA9Data;
[accuracy, predicted_labels] = RecognizeActions(datasetTrain1, datasetTest1, G, 10)
```

If everything is implemented correctly, you should obtain an accuracy of 82%, which is excellent accuracy on the recognition task; in comparison, random guessing will yield an accuracy of 33%. You can use the `VisualizeDataset.m` function to visualize the actions. Try visualizing some of the actions used to train the HMMs as well as the unknown actions that were classified using the HMMs. Also, we have set the cardinality of the hidden state variables to be 3 for faster training.

4.4.1 Effect of EM Initialization on Recognition Accuracy

In the lectures, we learnt that the initialization of the EM algorithm can significantly affect the quality of the model that it returns. You will now see this effect for yourself. Execute the following command, which uses the same datasets for training and testing, but uses a different set of initialializations:

```
[accuracy, predicted_labels] = RecognizeActions(datasetTrain2, datasetTest2, G, 10)
```

You should see that the accuracy drops to 73%, which is 9% less than that obtained with the previous initialization! Both of the initial model parameters were randomly sampled from a uniform distribution, but the difference in accuracy is significant. This occurs because of the multiple local maxima present in the likelihood function: the latter initialization caused the algorithm to converge to a different local maximum. At this particular local maxima, the learned models find it harder to distinguish between the different actions. This example illustrates the importance of initialization in learning HMMs, and more generally speaking, latent variable models. Typically, we will want to construct an initial set of model parameters using knowledge about the latent variables in our model, rather than picking these parameters randomly.

4.5 Extending the Model

Your final task is to refine and extend the code you've written for recognizing actions. We have provided you with the variables **datasetTrain3** and **datasetTest3** in the **PA9Data.mat** file. Your task is to train models on the **datasetTrain3** data to recognize the actions in **datasetTest3**. We've given you a random set of initial probabilities in **datasetTrain3** that don't perform very well. Note that we do not give you the labels in **datasetTest3**.

- `RecognizeUnknownActions.m` **(20 points)** — In this function, you should train a model for each action class using the training set **datasetTrain3**. Then, classify each of the instances in the test set **datasetTest3** and save the classifier predictions by calling `SavePrediction.m`. This function is left empty for you to be creative in your approach (we give some suggestions below). When you are done, write a short description of your method in `YourMethod.txt`, which is submitted along with your predictions in the submit script. Make sure you execute this function before running the submit script so that the saved predictions are ready for submission. Your score for this part will be determined by the percentage of unknown action instances you successfully recognize. If you obtain an accuracy of $x\%$, you will receive $0.2x$ points for this part.

To help you get started, here are some suggestions. Try to be creative in your approach, leveraging the ideas you have learned over the entire class. Good luck! Note that you are not allowed to manually hand label the test set. Try your best, but do not worry if you cannot get close to full marks on this part – real-world machine learning algorithms are not fully accurate either!

- **Better initializations** — Can you devise a better method for initializing the model's parameters that will help EM to find a good local maxima? For example, you can try initializing the probabilities by clustering the poses together using a simple method like K-means or Hierarchical Agglomerative Clustering (HAC). To evaluate your initalizations, you may want to split your dataset into a training set and a validation set, so you can check your performance on the validation set.
- **Hidden state variables** — In the sample data we had you run, we fixed the hidden state variables S_i to have cardinality of 3. However, it's possible that there are more than 3 underlying pose classes. Is there an optimal cardinality for the hidden state variables? Again, a validation set might be useful to evaluate your performance.
- **Extending the HMM** — Is the HMM the best model for our action data? You might try implementing a more complicated model, such as a duration HMM. Another possibility is to place restrictions on the states and their transitions. For example, we might be certain that the first state is always the same pose, and that the transitions typically occur in a fixed order. How can we encode this in the initial state probabilities and transition matrix?
- **Generating more data** — The amount of data you are given to learn the HMMs is not large. Given that we are learning a large number of parameters, more data could be very helpful. Can you think of ways of warping the existing instances you have to obtain more training data?

5 Useful Functions for HMM

List of functions we have written for you that you may find useful:

- `ComputeExactMarginalsHMM.m` — Similar to the **ComputeExactMarginalsBP.m** function you wrote in PA4. This function takes as input a set of factors in log space for an HMM, and runs clique tree inference in log space to avoid numerical issues. The function returns marginals (normalized) for each variable, as well as the calibrated clique tree (unnormalized). Note that the messages are unnormalized, as this may be helpful in computing the log likelihood.
- `SavePrediction.m` — Call this function to save your predictions for the unknown actions. This will prepare the predictions in a mat file so that they can be submitted in the submit script.
- `VisualizeDataset.m` — Helps to visualize a dataset of N poses of size N x 10 x 3.

6 Useful Data Structures for HMM

- **P** — This structure holds the model parameters. There are 2 changes made to this structure for HMM's that make this different from before and the pose clustering section. First, there is now a field P.transMatrix, which stores the K x K transition matrix, where K is the number of possible classes. Second, **the field P.c now contains the initial state prior probabilities, instead of the prior class probability for all states.**
- **actionData** — This is an array of structs that contains the actions. Each element of the array is an instance of an action, with the fields **action, marg_ind**, and **pair_ind** defined as follows:
  - **action** — a string with the name of this action.
  - **marg_ind** — a set of indices that index the first dimensions in the associated **poseData** and **ClassProb** matrices that correspond to the sequence of poses that make up this action. Thus, if **marg_ind**= $[10, 11, \ldots, 19]$ then poseData(10, :, :), poseData(11, :, :), ..., poseData(19, :, :) give the coordinates of this action's 19-10+1=10 component poses. Additionally, rows 10 through 19 of **ClassProb** give the corresponding class assignment probabilities of each of these individual poses.
  - **pair_ind** — a set of indices giving the rows in the **PairProb** data structure that correspond to the pairwise transition probabilities for each pair of consecutive states S_i and S_{i-1} in the action. Thus, if **pair_ind**=$[1, 2, \ldots 9]$ then the pairwise transition probability between the first and second poses in this action will be in row 1 of **PairProb**, the transition between the 2^{nd} and 3^{rd} will be in row 2, and so forth.

  For an illustration, see Figure 3.
- **ClassProb** — N x K matrix that contains the conditional class probabilities of the N poses to the K classes. Note that N here is the total number of poses over all the actions. ClassProb(i, j) is the conditional probability that pose i belongs to state j. Rows of this matrix are indexed by the **actionData** structure. Using this matrix, you should be able to estimate parameters in the M-step for the initial state prior and the emission CPDs. For an illustration, see Figure 3. The ClassProb matrix is indexed in the same way as poseData, and thus they have the same number of rows.
- **PairProb** — V x K^2 matrix that contains the pairwise transition probabilities (expected sufficient statistics for the transition CPD, $M[s', s]$) for every pair of consecutive states S_i

and S_{i-1} in all actions. Thus, every pair of consecutive states connected by an edge has an entry in this matrix. V is the number of these edges. Rows of this matrix are indexed by the **actionData** structure. Each row is a column-by-column concatenation of a K x K matrix representing the transition probabilities. For an illustration, see Figure 3.

- **datasetTrain** — This is an array of structs that contains actions used for training. Each element of the array is a struct for a different action, and contains the fields actionData, poseData, InitialClassProb, and InitialPairProb, which are described above. The first element of the array is "clap", the second is "high_kick", and the third is "low_kick".
- **datasetTest** — This is the struct that contains actions used for testing. There are 3 fields: actionData, poseData, and labels. The actionData and poseData are described above, and the labels field is an N x 1 vector with the labels for each of the N action instances in the datasetTest (with "clap" = 1, "high_kick" = 2, "low_kick" = 3). Note that for the unknown test data, the labels field is not provided.

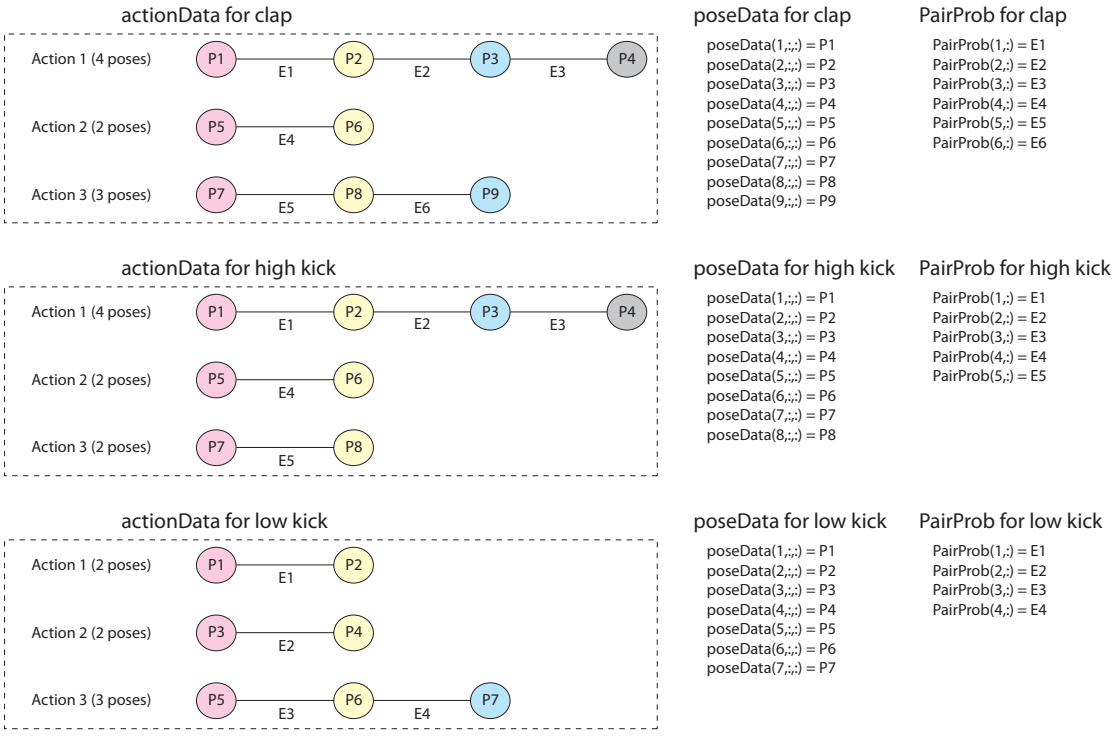

Figure 3: Illustration of the data structures used for this part. Note that each actionData/poseData/PairProb structure will only contain data for a single action class. The actionData structure uses marg_ind to index where to find the poses in poseData and pair_ind to index where to find the edges in PairProb. For example, the first pose in Action 2 of the "clap" action can be found in the 5th row of poseData, and the edge between the first and second poses of Action 2 can be found in the 4th row of PairProb.

7 Conclusion

Congratulations! You've completed the final programming assignment for Probabilistic Graphical Models! In this programming assignment, you've put together many of the ideas you've learned in class, ranging from the basic factor data structures to advanced concepts like the Expectation-Maximization algorithm, to create a system that is able to recognize human actions in real-world data. We hope that you've enjoyed the assignment, and good luck for the final!