Dynamic Gesture Recognition

A Succinct Application of Hidden Markov Models (HMM)

Eduardo Joaquin Castillo
Electrical and Computer Engineering
Cornell Tech at Cornell University
New York, NY 10044
ec833@cornell.edu

1 Abstract

This report documents the development of a motion prediction algorithm based on a set of purpose-built hidden Markov models (HMM).

The models receive motion data (linear and angular accelerations) collected from the inertial measurement unit (IMU) in a mobile device[1]. Using this data, the HMMs are trained to make predictions about the type of motion being performed.

Using motion data and corresponding types of motion [2], —referred to as gestures, i.e.: circle, infinity, etc.— the algorithm trains a set of HMMs to classify similar motions not present in the training set as gestures of a particular type.

2 Introduction

Hidden Markov Models (HMM) are based on an augmentation of the Markov chain. A Markov chain is a model that provides insights about the probabilities of sequences of random variables and states, each of which can take values from a predetermined set [3, 4].

Salient applications of HMMs are stock market modeling based on observed macroeconomic parameters and the use of meteorological variables (i.e.: transient barometric pressure, humidity, etc.) in weather forecasting.

3 Problem Formulation

3.1 Input Data Review

The given training dataset consists of 79340 time samples distributed across 30 training examples and 5 motion gestures.

Each time sample contains the following information:

1. Time, in units of *milliseconds.*
2. X, Y and Z accelerations in units of $^{m}/_{s^2}$.
3. Roll, Pitch and Yaw accelerations in units of $^{radians}/_{s^2}$.

As per the project specification[2], relevant statistical characteristics of the data were explored and exploited prior to algorithm development.

Most notably, the dynamic ranges of the raw data were found to show significant variability between observations within a given motion gesture class. Figure 1 illustrates this behavior.

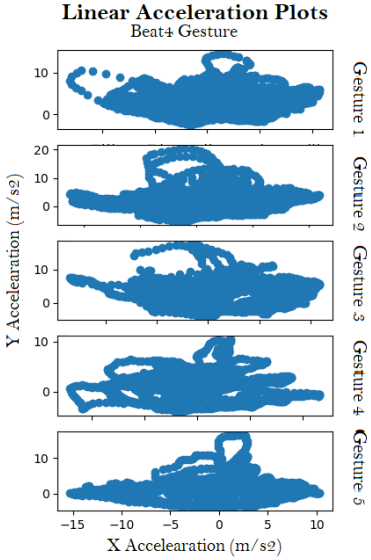

Figure 1: Raw Data Visualization

As shown in Figure 1, the dynamic range of the second observation (Gesture 2) for the Beat4 motion gesture class shows a dynamic range in excess of 25 m/s^2 for the Y acceleration, while the corresponding dynamic range for Gesture 4 is approximately 15 m/s^2.

3.2 Data Normalization

To maintain homogeneity of dynamic ranges across observations, the raw IMU training data was normalized to maintain a dynamic range of unity (1) across all dimensions and observations in the training set. Likewise, the observations in the test set are processed to exhibit the same range.

Figure 2 shows the resulting normalized (0-1 range) data, plotted on the $X - Y$, $X - Z$ and $Y - Z$ planes, respectively, for the Beat 3 gestures.

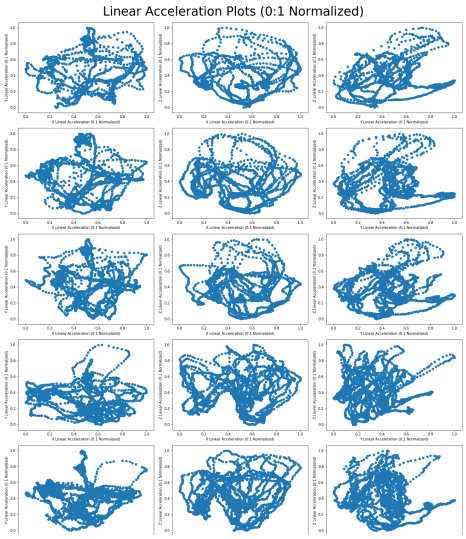

Figure 2: Normalized Data Visualization

The feature normalization was accomplished through a purpose-built *rescale Python* function, which is described in further detail in Appendix A.1.

4 Technical Approach

4.1 Model Selection

Given the prescribed HMM solution approach, [2] the main design choices revolved around the type of HMM best suited for the task (continuous, discrete, left-right, ergodic, etc). As discussed below, a vectorized, discrete left-right model was found to be most adequate for this application.

4.2 Data Vectorization

All discrete-time samples in the training dataset were found to be uniquely valued. The uniqueness of these values was an essential consideration in managing the high cardinality of the observation set.

Following rescaling, the set was vectorized to 165 centers using a purpose-built kmeans implementation, as described in Appendix A.2 and Subsection 4.7.2.

4.3 State Transition Model

Due to the sequential nature of the given motions, valid and unique state transitions under left-right HMMs —which constrain transition between states to one-to-one mappings– are best suited for this application. To minimize the complexity of the initial model, an simpler ergodic model was implemented.

The state-time transition model implements the Baum–Welch algorithm in its fully-vectorized form to iteratively estimate the pertinent **A** matrix parameter under the formulated HMMs.

The base algorithm was implemented using the built-in *Numpy Python* module. Matrix re-estimation is implemented using vector indexing and Boolean matrices, which is several times faster than a naive for-loop implementation. Section 4.7.3 provides further details on the implementation approach for the State-Transition model.

4.4 State-Observation Model

The state-observation model implements the Baum–Welch algorithm in its fully-vectorized form to iteratively estimate the pertinent **B** matrix parameter under the formulated HMMs.

The base algorithm was also implemented using the built-in *Numpy Python* module, and the corresponding computational efficiencies noted for the State Transition Model were leveraged during **B** matrix re-estimation. Section 4.7.3 provides further details on the implementation approach for the State-Observation model.

4.5 Initial State Distribution

The initial state distribution was initialized in a random fashion. While domain information was available to inform initial state assignments (i.e. maximize one of the initial states to unity), this variation was found to have a relatively small effect on training speed and predictive accuracy.

4.6 Hyperparameter Section

Over twenty runs of hyperparameter cross-validation were performed. The number of cluster centers was swept from 20 to 200, in increments of approximately 20%. Likewise, the number of hidden states was swept from 4 to 20, in increments of approximately 20%.

It was determined that 40 to 50 clusters with 5 states resulted in adequate generalization (93% and 83% training and validation set accuracy, respectively).

The parameters also produced amenable training and prediction times (approximately 1-2 seconds following data loading and processing). Figure 3 shows the cluster distribution for points in the first observation of the Beat 3 gesture class.

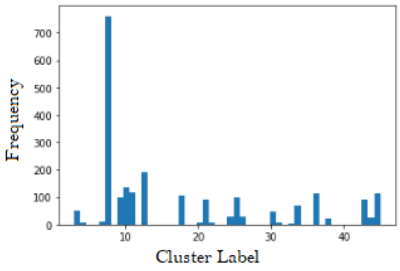

Figure 3: Cluster Histogram - Beat 3

Upon further iteration, the number of clusters was increased to 200 and the number of hidden states to 20. This, in turn, lead to predictive accuracy improving to 100% for both the training and validation sets.

This result implied that the optimal bias-variance balance lied somewhere in the range between these two parameterizations.

Upon further iteration, it was determined that HMMs with 165 centers and 15 states provided a superior compromise: 100% training accuracy was obtained and the model's tendency to over-fit the data as reduced. These parameters were the values used for the final submission.

4.7 *Python* Implementation

The high-level *Python* implementation for this project was based on a combination of purpose-built modules, functions, and classes. Below are the most notable components of the implementation.

4.7.1 The *dataloader Python* Module

The *dataloader Python* module [A.1] is a custom module built for this project. It houses several relevant functions which process various tasks such as data scaling and data segmentation. It is also used to load relevant data files into the *Python* kernel in a suitable format for further processing.

4.7.2 The *kmeans_module Python* Module

The *kmeans_module Python* module [A.2] was purpose-built for this project. Inside the module, we find the The *get_kmeans* and *find_kmeans_centers* functions. The *get_kmeans* function computes 200 iterations of the k-means algorithm, based on 165 clusters and user-specified convergence parameters.

The *find_kmeans_centers* function transforms each cluster into an unordered set and performs set intersection operations across all 200 iterations for each cluster, selecting the iteration with the highest overall rate of cluster similarity with respect to other iterations.

Due to the long runtime (several hours), the output of the kmeans runs are saved to the local drive via the *Numpy.savetxt* method.

The final submission for the project offers the user to ability to obtain center assignments and locations by either running the *get_kmeans* function or by loading files previously saved to the local drive, The *run_kmeans* variable ϵ [0,1] in the final implementation controls this functionality.

4.7.3 The HMM *Python* Module

The HMM *Python* Module is the main component of the prediction model. The custom-built module contains the *get_alpha_matrix*, *get_beta_matrix*, and *get_gamma_xi_matrix* functions, which operate in concert to return fully-trained models based on each observation.

The *get_alpha_matrix Python* Function

The *get_alpha_matrix Python* function implements the forward pass of the Baum–Welch algorithm to find the hidden α parameter of the HMM. The Baum–Welch algorithm was implemented in its fully vectorized form using the built-in *Numpy Python* module.

A recursive subroutine is used to calculate the required forward time steps, using one matrix-vector product calculation per time step. This implementation was approximately 20 times faster than the naive element-based computation.

The training run time was approximately 6 seconds/observation, based on 15 hidden states and 165 clusters. Appendix A.3 contains the *Python* implementation of the function.

The *get_beta_matrix Python* Function

The *get_beta_matrix Python* function implements the backward pass of the Baum–Welch algorithm to find the hidden β parameter of the HMM.

The Baum–Welch algorithm was also implemented in its fully vectorized form using the built-in *Numpy Python* module.

A recursive subroutine is used to calculate the required backward time steps, using one matrix-vector product calculation per time step. This implementation was also found to be approximately 20 times faster than the naive element-based computation.

The run time was approximately 6 seconds/observation, based on 15 hidden states and 165 clusters. Appendix A.3 contains the *Python* implementation of the function.

The *get_gamma_xi_matrix Python* Function

The *get_gamma_xi_matrix Python* Function implements the γ and ξ steps of the Baum–Welch algorithm to find the hidden γ and ξ parameters of the HMM.

Given the relatively high indexing complexity of the γ tensor, it was computed using element-wise multiplication followed by vector-matrix product computations to obtain the ξ matrix. Appendix A.3 contains the *Python* implementation of the function.

The *Lambda Python* Class

In order to efficiently package the learned models into suitable *Python* objects, the *Lambda Python* class was created. The constructor for the class enables instances to hold all pertinent information associated with its HMM.

Namely, the A, B, and π are stored in the class. In addition, the quantized (cluster assignments) and ground-truth labels for each instance are contained in the class.

To enhance the value derived from the class, a *ged_prediction* method was implemented within the class. This method loads appropriate model parameters into instances (which are obtained from previously trained A, B and π instances in drive storage) and returns the appropriate probability based on such parameters. This implementation routine proved its value during cross-validation. By storing all 30 objects in a list, a *Python* list iterator object could be used to invoke the *ged_prediction* method for all 30 trained HMMs, and return the predicted class via a simple $argmax$ function call.

This resulted in prediction times in the order of fractions of second in the **train_add** set provided for validation purposes. Appendix A.4 contains the *Python* implementation of the *Lambda Python* class.

5 Results and Discussion

5.1 Computation Efficiency

The model had an average training time below 1.5 seconds (after data intake and vectorization. This result could be improved further by increasing the efficiency of the recursion process for the α and β parameters as well as further vectorization of the γ and ξ parameters.

Below is an excerpt of the computation time for training observations.

Processing Observation 1 Process Terminated at 2 of 100 iterations. - 1.48 sec Execution Time -

5.2 Training Prediction Accuracy

The model was able to predict 36 out of 36 training examples (including 6 sequestered validation observations) correctly under a 165-cluster, 15-state HMM, yielding a training accuracy of 100%.

When compared to the initial 40 cluster-5 state HMM architecture, this HMM configuration yielded a predictive improvement of approximately 7% and 16% for the training and sequestered validation sets, respectively.

5.3 Test Set Predictions

An additional 8-observation set of unlabeled observations was provided for the purpose of instructor evaluations of algorithm performance. As such, the ground-truth labels for these observations remain sequestered.

Table 5.3 summarizes the top-three predictions made for the 8-observation test set provided for this project.

Test Set Predictions			
Observation ID	**Prediction Rank**		
	Top	**2nd**	**3rd**
1	wave	eight	inf
2	beat4	beat3	inf
3	inf	eight	beat3
4	beat3	beat4	circle
5	circle	beat4	beat3
6	inf	beat4	beat3
7	eight	inf	wave
8	beat4	beat3	circle

Table 5.3: Test Set Predictions

5.4 Test Set Log Probabilities

The likelihood ratio of the top final prediction probability to the second highest prediction probability was used as a metric to evaluate confidence in the accuracy of each prediction. These were computed based on the ratio of the highest predicted log-probability to sum of the probabilities for all the predictions. The results are shown in table 5.4.

Test Set Likelihoods	
Observation ID	Likelihood Ratio
1	14.4
2	1.6
3	3.5
4	2.2
5	27.7
6	2.7
7	6.9
8	1.7

Table 5.4: Test Likelihood Ratios

As shown, the likelihood ratios for test observations 1 and 5 indicate the highest prediction confidence, while observations 2 and 8 indicate the lowest prediction confidence.

5.5 Discussion and Future Work

As discussed above, the model had an average training time below 1.5 seconds (after data intake and vectorization). The result could be improved by increasing the efficiency of the recursion process for the α and β parameters as well as further vectorization of the γ and ξ parameters.

In addition, the kmeans vectorizing function required a relatively long computing time —in excess of 10 minutes— for 50 clusters when a 0.1% average relative center shift (between iterations) was used as the convergence criterion.

The computing time could be significantly reduced by introducing a more efficient cross-validation routine for kmeans runs (i.e.: random pooling for cluster similarity instead of the current exhaustive cluster similarity search).

Lastly, the model's predictive performance could be enhanced by implemented a fully constrained left-right model instead of the current ergodic HMM. These model improvements will be implemented throughout the semester on a time-permitting basis.

References

[1] Sensor Event - Android Developers. `https://developer.android.com/reference/android/hardware/SensorEvent.html`. (Accessed on 03/04/2020).

[2] Cornell Tech Electrical and Computer Engineering Department. ECE - 5242 Project 2 - Gesture Recognition Project Specification, 03 2020.

[3] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *PROCEEDINGS OF THE IEEE*, pages 257–286, 1989.

[4] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, USA, 1st edition, 2000.

A Relevant *Python* Functions

A.1 The *dataloader Python* Module

The *dataloader Python* module was purpose-built for this project. It houses several relevant functions which process various tasks such as data scaling, data segmentation. It is also used to load relevant data files into the *Python* kernel.

```
import os
import numpy as np
list_par = []
cluster_list_par= []
os.chdir(os.path.dirname(os.path.realpath(__file__)))

def load_data(folder_par):
    for count,filename in enumerate(os.listdir(os.path.join(folder_par))):
        label  = filename[:-6].replace("_","")
        cluster_list_par.append((np.loadtxt(os.path.join(folder_par,filename))))
        list_par.append((np.asarray([label,len([i for i in list_par if i[0][0]==label]),]*(cluster_list_par[count].shape[0]))))
        list_par[count] = np.reshape(list_par[count], (int(len(list_par[count])/2),2)   )
        cluster_list_par[count]= np.hstack((list_par[count],cluster_list_par[count]))

    return np.vstack(cluster_list_par)

def get_coded_observations(clustering_array_par):

    # extract unique text labels
    class_text = np.unique(clustering_array_par[:,0])

    # deep copy input array
    raw_data_par = np.copy(clustering_array_par)
    raw_data_text = np.copy(clustering_array_par)

    # transform motion class into 0-4 codeword
    indexer = []
    for i,j in enumerate(class_text):
        indexer.append(np.where(raw_data_par[:,0] == j))

        raw_data_par[:,0][indexer[-1]] = int(i)
        raw_data_text[:,0][indexer[-1]] = j
    return raw_data_par,raw_data_text


def rescale_data(data_array_par):

    # create empty lists
    ob_split = []
    ob_split_scaled = []

    # for all coded data
    for i in np.unique(data_array_par[:,0]):

        # extract elements for each code (i.e.: beat3 = 0 and so on)
        j = data_array_par[data_array_par[:,0]==i]

        # for all the datapoints that belong to a given observation (unique elements of column 1)
        for j1 in np.unique(j[:,1]):

```

```
            # get the time samples in float format
            k = j[j[:,1]==j1].astype(float)

            # append single list element containing all time samples
            ob_split.append(k)

    for cnt, observation in enumerate(ob_split):

        # copy entire array, reset set type to float
        deep_ob_copy = np.copy(observation).astype(float)

        # extract min and max feature values
        dim_max = np.amax(deep_ob_copy[:,3:],axis=0)
        dim_min = np.amin(deep_ob_copy[:,3:],axis=0)

        # rescale features
        deep_ob_copy[:,3:] = (deep_ob_copy[:,3:]-dim_min)/(dim_max-
    dim_min)

        # append to scaled list
        ob_split_scaled.append(deep_ob_copy)

        # if first iteration, put the first stack of observations in
    array
        if cnt == 0:
            ob_rescaled = deep_ob_copy

        # else stack subsequent sets of observations in array
        else:
            ob_rescaled = np.vstack((ob_rescaled,deep_ob_copy))

    # return relevant variables
    return ob_rescaled, ob_split,ob_split_scaled
```

A.2 The *kmeans_module Python* Module

The *kmeans_module Python* module was purpose-built for this project. Inside the module, we find the The *get_kmeans* and *find_kmeans_centers* functions. The *get_kmeans* function computes 200 iterations of the k-means algorithm, based on 165 clusters and user-specified convergence parameters.

The *find_kmeans_centers* function transforms each cluster into an unordered sets and performs set intersection operations across all 200 iterations for each cluster, selecting the iteration with the highest overall rate of cluster similarity with respect to other iterations. Below is the *Python* code for the *kmeans_module Python* module.

```
def find_kmeans_center(indices_list_par,k_means_runs_par):

    similarity_array= np.zeros([k_means_runs_par,k_means_runs_par,
    cluster_cnt])

    for run_id1,indices_set1 in enumerate(indices_list_par):
        old_time = time.time()
        for run_id2,indices_set2 in enumerate(indices_list_par):
            if run_id1 != run_id2:
                for current_indices_list1 , i in enumerate(
    indices_set1):
                    list_comp = []
                    for current_indices_list2 , j in enumerate(
    indices_set2):
                        list_comp.append(len(set(i[0][0])&set(j[0][0])
    ))
                    similarity_array[run_id1,run_id2,
    current_indices_list1]=max(list_comp)
```

```
        print("Centers for Run", run_id1, "completed in",  time.time()-old_time,"seconds.")
    similarity_count_final = np.sum(similarity_array,axis=2)
    print(similarity_count_final)

    print(np.argmax(similarity_count_final) % (similarity_count_final.shape[1]))
    return np.argmax(similarity_count_final) % (similarity_count_final.shape[1])

def get_kmeans(input_data, convergence_crit):


    k_means_runs = 50
    centers_list = []
    center_assignment_list = []
    indices_list =[]
#     centers_init = np.random.rand(cluster_cnt, 6)*.10 +.45
    for run_count in range(k_means_runs):
        old_time = time.time()
        centers_init = np.random.rand(cluster_cnt, 6)*.10 +.45
        centers_init=centers_init.astype(np.float)
        input_data_par = input_data.astype(np.float)
        center_assignments = np.zeros(len(input_data_par[:,1]),dtype="int")
        kmeans_centers = np.copy(centers_init)
        oldcenters = np.copy(centers_init)

        centers_shift = 10 #arbitrary initial value greater than convergence_crit
        iterations    = 0  #initialize iteration counter

        #point_dists of shape len(input_data_par) x kmeans_centers
        #cluter_assignments of shape len(input_data_par)
        #

        while centers_shift > convergence_crit and iterations < 100:
            indices = []

#             print("Computing Point to Center Distances")

            # create row indexer
            data_indexer    = np.arange(input_data_par.shape[0])[:,np.newaxis] #generate array of row indices
            # create column indexer
            center_indexer = np.transpose(np.arange(centers_init.shape[0])[:,np.newaxis]) #generate array of column indices
            # find distances to each center
            point_dists = np.linalg.norm(input_data_par[data_indexer] - kmeans_centers[center_indexer],axis = 2)
            # iteration counter
            iterations = iterations + 1

            #Update cluster assignments
#             print("Update Clusters")
            for k in range(len(input_data_par[:,1])):
                center_assignments[k] = np.argmin(point_dists[k])

            #Update kmeans_centers
            for m in range(len(init_centers)):
                indices.append([np.where(center_assignments == m)])

            for index,n in enumerate(indices):
                if len(n[0][0]) > 0:
```

```
                    kmeans_centers[index] = (np.mean(np.squeeze(
    input_data_par[n]),axis = 0))

            #Check for convergence
            update_array = np.sqrt(np.sum((kmeans_centers-oldcenters)
    **2, axis=1))
            centers_shift = np.mean(update_array)

            #if average of distance kmeans_centers moved is less than
    some tolerance, terminate
            if (centers_shift > convergence_crit):
                oldcenters  =  np.copy(kmeans_centers)
            else:
                print("The K-means Algorithm has converged for run",
    run_count+1,"of",k_means_runs," planned runs after", time.time()-
    old_time,"seconds and",\
                      iterations,"iterations.")
                center_assignment_list.append(center_assignments)
                centers_list.append(kmeans_centers)
                indices_list.append(indices)


                # if first iteration, put the first stack of
    observations in array
                new_count_stack = run_count*np.ones(kmeans_centers.
    shape[0])
                if run_count == 0:

                    center_stack = np.sort(kmeans_centers,axis = 1)
                    iteration_count_stack = new_count_stack

                # else stack subsequent sets of observations in array
                else:
                    center_stack =           np.vstack((center_stack,
    np.sort(kmeans_centers,axis = 1)))
                    iteration_count_stack =  np.hstack((
    iteration_count_stack,new_count_stack))
                break

    correct_centers_iteration = find_kmeans_center(indices_list,
    k_means_runs)

    return centers_list[correct_centers_iteration],
    center_assignment_list[correct_centers_iteration]
```

A.3 The HMM *Python* Module

The HMM *Python* Module is the main component of the prediction model. The custom-built module contains the *get_alpha_matrix*, *get_beta_matrix*, and *get_gamma_xi_matrix* functions, which work together to return fully-trained models based on each observation. Below is the *Python* code for the *HMM Python* module.

```
import numpy as np
import time

def recursive_alpha(time_steps_cnt_par, state_cnt_par,
    alpha_matrix_par,\
                A_matrix_of_ob_par, B_matrix_of_ob_par,
    time_recursion_shifter_par,c_populate_matrix_par,single_ob_par,
    rec_par):

    rec = rec_par + 1
    recursion_shifter_par = time_recursion_shifter_par

```

```
    # create deep copy of matrix_par
    alpha_recursion_mat = np.copy(alpha_matrix_par)

    # create c_populate matrix of size T-1 to populate values 1:T of
    the c_t matrix
    c_populate_matrix = np.copy(c_populate_matrix_par)

    if recursion_shifter_par <= time_steps_cnt_par - 1:

        # extract the applicable column of b, which is found by seeing
     the value of the sequence at time t
        # this will be one of the possible values, in this case, a
    cluster center label
        current_b_matrix_column = single_ob_par[recursion_shifter_par]

        alpha_recursion_mat[recursion_shifter_par,:] =  np.dot(
    alpha_matrix_par[recursion_shifter_par-1,:],A_matrix_of_ob_par)
        alpha_recursion_mat[recursion_shifter_par,:] =  np.multiply(
    alpha_recursion_mat[recursion_shifter_par,:],

    B_matrix_of_ob_par[:,current_b_matrix_column])
#         print(c_populate_matrix)
        # polulate one element [recursion_shifter_par] of
    c_populate_matrix
        c_populate_matrix  [recursion_shifter_par] =  np.sum(
    alpha_recursion_mat[recursion_shifter_par,:])

        # do one over c_populate for scale
        c_populate_matrix[recursion_shifter_par] = 1/ (
    c_populate_matrix[recursion_shifter_par])


        # rescale alpha_recursion_mat[recursion_shifter_par,:]
        # pass to recursion!!
        alpha_recursion_mat[recursion_shifter_par,:] =
    c_populate_matrix [recursion_shifter_par] * alpha_recursion_mat[
    recursion_shifter_par,:]

        # add one to the downwards shifter so we index the row above
    on next recursion
        # pass to recursion!!
        recursion_shifter_new = recursion_shifter_par + 1

        alpha_mat_temp,c_populate_matrix_temp = recursive_alpha(
    time_steps_cnt_par, state_cnt_par, alpha_recursion_mat,\
                                       A_matrix_of_ob_par,
    B_matrix_of_ob_par, recursion_shifter_new,c_populate_matrix,\

    single_ob_par,rec)

        c_populate_matrix   = c_populate_matrix_temp
        alpha_recursion_mat = alpha_mat_temp

    return alpha_recursion_mat,c_populate_matrix

def get_alpha_matrix(single_ob_par, A_matrix_of_ob_par,
    B_matrix_of_ob_par, N, pi_par):

    # extract number of time steps
    time_steps_cnt_par = single_ob_par.shape[0]

    # extract number of states
    state_cnt_par = B_matrix_of_ob_par.shape[0]
#     print("state_cnt_par", state_cnt_par)
    # create c (scaling factor matrix)
```

```
    c_scaling_factor = np.zeros([time_steps_cnt_par])

    # perform dimension test. if failed, break, print and exit if wrong.
    if state_cnt_par != A_matrix_of_ob_par.shape[0] or state_cnt_par != A_matrix_of_ob_par.shape[1]:
        return print("Dimensions of A and B are incorrect - CODE NOT EXECUTED")

    else:
        alpha_matrix_par = np.zeros([time_steps_cnt_par,state_cnt_par])

        # extract the first value in the current sequence
        obsservation_zero = single_ob_par[0]

        # compute time-zero alphas (first Nx1 row of TxN matrix)
        alpha_matrix_par[0,:] = np.multiply(pi_par, B_matrix_of_ob_par[:,obsservation_zero])

        # compute c{t=0}
        c_scaling_factor[0]   = np.sum(alpha_matrix_par[0,:])

        # recompute c{t=0} as its inverse
        c_scaling_factor[0] = 1/c_scaling_factor[0]

        # rescale alphas at time zero
        alpha_matrix_par[0,:] = c_scaling_factor[0] * alpha_matrix_par[0,:]

        # this value determines the first time index determined during recursion
        time_recursion_shifter_par = 1

        alpha_matrix_par_temp,c_matrix = recursive_alpha(time_steps_cnt_par,state_cnt_par,alpha_matrix_par,\
                                                A_matrix_of_ob_par, B_matrix_of_ob_par,time_recursion_shifter_par,\

    c_scaling_factor,single_ob_par,0)

    return alpha_matrix_par_temp , c_matrix

def beta_recursion(matrix_par,A_matrix_of_ob_par, B_matrix_of_ob_par, time_steps_cnt_par,recursion_shifter_par,c_par,\
                   single_ob_par):

    # create deep copy of matrix_par
    recursive_indexer = time_steps_cnt_par - recursion_shifter_par
    beta_recursion_mat = np.copy(matrix_par)

    if recursive_indexer >= 0:

        current_b_matrix_column = single_ob_par[recursive_indexer + 1]

        #perfom dot product from ****t+1**** parameters
        beta_recursion_mat[recursive_indexer,:] =  np.dot(beta_recursion_mat[recursive_indexer + 1,:],A_matrix_of_ob_par)
        beta_recursion_mat[recursive_indexer,:] =  c_par[recursive_indexer] * np.multiply(beta_recursion_mat[recursive_indexer,:],\

    B_matrix_of_ob_par[:,current_b_matrix_column])

```

```
        # add one to the downwards shifter so we index the row above on next recursion
        recursion_shifter_new = recursion_shifter_par + 1

        # print(np.around(beta_recursion_mat, decimals = 2))
        temp = beta_recursion(  beta_recursion_mat,A_matrix_of_ob_par, B_matrix_of_ob_par,\
                                time_steps_cnt_par, recursion_shifter_new,c_par,single_ob_par)

        beta_recursion_mat = temp

    return beta_recursion_mat

def get_beta_matrix(single_ob_par, A_matrix_of_ob_par, B_matrix_of_ob_par, N_size_par, c_par):

    # extract number of time steps
    time_steps_cnt_par = single_ob_par.shape[0]

    # create beta matrix
    beta_matrix = np.zeros([time_steps_cnt_par,N_size_par])

    # assign values from C_T_minus_1_par to the last column of every matrix in the beta matrix
    beta_matrix[-1,:] = c_par[-1]
#     print("Processing time step",time_steps_cnt_par-1,"of", time_steps_cnt_par-1,"(Backward Pass)")

    # extract not the last element's index (shape-1) but the second to last (shape-2)
    recursion_shifter = 2

    # build matrix from recursive function
    beta_matrix = beta_recursion(beta_matrix,A_matrix_of_ob_par, B_matrix_of_ob_par,\
                                 time_steps_cnt_par,recursion_shifter, c_par,single_ob_par)
    return beta_matrix

def get_gamma_xi_matrix(single_ob_par, A_matrix_of_ob_par, B_matrix_of_ob_par, N_size_par,alpha_par,beta_par):

    # extract number of time steps
    time_steps_cnt_par = single_ob_par.shape[0]

    # create gamma_xi matrix
    xi_matrix = np.zeros([time_steps_cnt_par,N_size_par,N_size_par])
    gamma_matrix    = np.zeros([time_steps_cnt_par,N_size_par])

    # special rule for final time step of gamma matrix
    gamma_matrix[-1,:] = alpha_par[-1,:]

    for time_steps in range(alpha_par.shape[0]-1):
        current_b_matrix_column = single_ob_par[time_steps + 1]

        for i_state in range(A_matrix_of_ob_par.shape[0]):
            for j_state in range(A_matrix_of_ob_par.shape[0]):
                xi_matrix[time_steps,i_state,j_state] = alpha_par[time_steps,i_state] *\

    A_matrix_of_ob_par[i_state,j_state]*\

    B_matrix_of_ob_par[j_state,current_b_matrix_column]*\
```

```
                                                                    beta_par[
    time_steps+1,j_state]

    gamma_matrix[:-1] =   np.sum(xi_matrix[:-1],axis = 2)

    return xi_matrix, gamma_matrix

def get_pi(N_par):
    pi_par = 1/np.ones([N_par]) +.02*(np.random.randn(N_par))
    return pi_par
```

A.4 The *Lambda Python* Class

The *Lambda Python* Class In order to efficiently package the learned models into suitable *Python* objects, the *Lambda Python* class was created. The constructor for the class enables instances to hold all pertinent information associated with its HMM. Namely, the A, B, and π are stored in the class. In addition, the quantized (cluster assignments) and ground-truth labels for each instance are contained in the class.

```
class Lambda:
    def __init__(self,A,B,PI,motion_class,observation_number,
    observation,N_states,custom_ob):
        self.A = A
        self.B = B
        self.PI = PI
        self.motion_class = motion_class
        self.observation_number = observation_number
        self.observation = observation
        self.N_states = N_states
        self.custom_ob = custom_ob

    def get_prediction(self):

        num = get_alpha_matrix(self.custom_ob, self.A, self.B, self.
    N_states, self.PI)[0][-1]
        den = get_alpha_matrix(self.custom_ob, self.A, self.B, self.
    N_states, self.PI)[1][-1]
        rat = num/den
        return int(np.sum(rat)*100)
```