HMM-BASED ONLINE CURVE RECOGNITION
by Marcelo Cicconet
at Georgia Tech Center for Music Technology (Atlanta, GA)
in Dec 09 2011

This report has 5 main purposes:
1. Clarify the algorithm described in [2] (which was presented first in [1]).
2. Implement the algorithm described in [2], using the GHMM library [3].
3. Better understand the GHMM library (which is poorly documented).
4. Provide details of my implementation, so others can better understand and use it.
5. Better understand Hidden Markov Models.

The mentioned implementation corresponds to a slightly different problem than the one described in [1].

Problem Definition

Suppose you have a curve in space, say $C_0=\{(x_1,y_1),(x_2,y_2),\ldots,(x_N,y_N)\}$, where N is finite. And suppose you have a point, (x_τ,y_τ) , whose position is space changes with time, τ , for $\tau=1,2,\ldots$. Given a particular τ , we'd like to compare the *tail* $C_\tau=\{(x_{\tau-(N-1)},y_{\tau-(N-1)}),\ldots,(x_{\tau-1},y_{\tau-1}),(x_\tau,y_\tau)\}$ with C_0 and identify when they *seem to be the same.*

Observation 1: Notice that we are requiring C_0 and C_τ to have the same length, N . It doesn't need to be that way, though. For instance, if you are using a sensor with a fixed sample rate to input the curve, you could have C_0 captured with the point moving slowly, and want to identify a C_τ similar to C_0 even if the speed of the point when capturing C_τ is greater (in which case C_τ would have less sampled points than C_0).

Observation 2: Suppose you are using the computer mouse as input device and that C_0 is a small "S" in the top left corner of the screen. You could be interested in identify when an "S" is captured in any position (say, the bottom right), not only in the exact position C_0 was captured. This is *not* the present case.

Hidden Markov Models

A good introduction to HMM is Alpaydin's book [4], chapter 13, from where I'm grabbing the notation.

Let's consider a Discrete Markov Process with N *states*, $S_1,\ldots,S_N$. We denote by q_τ the state at time τ . So, for instance, $q_3=S_5$ means that the state at time $\tau=3$ is S_5 . We denote a_{ij} as the probability of going from state S_i at time τ to state S_j at time τ , that is, $a_{ij}=P(q_{\tau+1}=S_j|q_\tau=S_i)$. The initial probability if state S_i is denoted by π_i , that is, $\pi_i=P(q_1=S_i)$. Let's name Π the set $\{\pi_1,\ldots,\pi_N\}$, and A the set $\{a_{ij}|i,j=1,\ldots,N\}$. This way, once we know Π and A , the Markov Model is completely determined. Well, usually we don't know those sets, and they should be determined by observing the Markov Model *in action*. That's actually easy once we can observe the states that the process is going through. It turns out that, most of the time, those states are *hidden*, that is, all we have is a set of observations, $O=\{O_1,\ldots,O_\tau\}$, where each O_j can assume some values that give hints about the actual state. These hints can be of two types: *discrete* or *continuous*. In the first case, the observations are a discrete set, say $\{v_{1,}..,v_M\}$, and the model is said to have *discrete emissions.* The emission

probabilities are $b_j(m)=P(O_\tau=v_m|q_\tau=S_j)$. A Hidden Markov Model with *continuous emissions* is one where the observations are a continuum of numbers, and the emission probabilities are represented by *density functions* (usually gaussians or mixture of gaussians), each state S_j having its own associated density.

In our case, we'll be dealing with the second type of HMM: an HMM with continuous emissions.

Also, our HMM will be what is called a *left-right* (or *left to right*) HMM. In these models, once a state S_i is reached for time τ , and the last distinct state prior to S_i is, say, S_j , than S_j is never reached again (that is, the process will not be S_j anymore for any time greater than τ). In other words, once the process passes through S_j , it won't go back to S_j anymore in the future.

Our Hidden Markov Model

Back to our problem, we have a curve $C_0=\{(x_{1,}y_1),(x_{2,}y_2),\ldots,(x_N,y_N)\}$. We set the number of states as N , and the states will be the points in the curve C_0 , that is, $S_i=(x_i,y_i)$, for $i=1,\ldots,N$. The emission probabilities, for each state S_i , will be gaussians centered in (x_i,y_i) . That is, $b_j(O=(x,y))$ is a gaussian function, centered in (x_j,y_j) , with covariance matrix Σ_j , evaluated at the point (x,y) .

In our model the input points are normalized to fall in the unit square, $[0,1]\times[0,1]$. We define Σ_j as σI , where I is the identity matrix $2\text{x}2$ and σ is some value around 0.5 . We also suppose $a_{ii}=1/3, a_{i,i+1}=1/3, a_{i,i+2}=1/3$ and $a_{ij}=0\,\forall\, j>i+2$ (with proper modifications for i,j close to 0 and N , in which case a_{ij} is defined so that S_0 and S_N are *dead ends* of the process).

Now, given a *training curve* C_0 and a *test curve* C_τ , we want to match each point of the test curve with a point of the training curve, that is, each observation with a state of the underlying HMM, whose model, $\lambda=\{A,\Pi\}$, is set beforehand.

To achieve that, we need to compute, for all i , the probability of the state being S_i (that is, the associated point being (x_i,y_i)) given that the observation is some (x,y) , and than maximize over i , to find out what is the most probable associated point. And that should be done for each point of the observed curve C_τ .

In summary, given the model $\lambda=\{A,\Pi\}$ and the observations $C_\tau=O=\{O_1,\ldots,O_N\}$, we want to compute $\gamma_t(i)=P(q_t=S_i|O,\lambda)$, for all $t=1,\ldots,N$ and $i=1,\ldots,N$.

The computation of $\gamma_t(i)$ depends on what is known, in the literature of HMM, as the *forward* and *backward* variables, $\alpha_t(i)$ and $\beta_t(i)$. They are defined by $\alpha_t(i)=P(O_1,\ldots,O_t,q_t=S_i|\lambda)$ and $\beta_t(i)=P(O_{t+1},\ldots,O_N|q_t=S_i,\lambda)$, respectively. The equation that relates these variables is $\gamma_t(i)=\alpha_t(i)\beta_t(i)/(\sum_j \alpha_t(j)\beta_t(j))$.

More details about these equations can be found in [4]. We now proceed to the formulation of the problem in terms of the GHMM library [3].

Translating Our Problem to the GHMM Language

Here are the prototypes of the functions we will use:

```
int ghmm_cmodel_forward(ghmm_cmodel * smo,
                        double *O,
                        int T,
                        double ***b,
                        double **alpha,
                        double *scale,
                        double *log_p);
int ghmm_cmodel_backward(ghmm_cmodel * smo,
                         double *O,
                         int T,
                         double ***b,
                         double **beta,
                         const double *scale);
```

The first function calculates alpha[t][i] (i.e., $\alpha_t(i)$), scaling factors scale[t] and log(P(O|lambda)) for a given double sequence and a given model. The second calculates beta[t][i] (i.e., $\beta_t(i)$) given a double sequence and a model. We now go through the input parameters.

(a) ghmm_cmodel * smo

Structure of the model for HMM with continuous emissions.

```
typedef struct ghmm_cmodel {
    int N; // number of states, as in the above notation
    int M; // maximum number of components in the states;
           // emissions are modeled in GHMM as mixtures of gaussians,
           // so this parameter specifies the maximum number of gaussians
           // in the mixture for a particular state
    int dim; // number of dimensions of the emission components
             // in our case this is equal to 2
    int cos; // number of transition matrices (1 in our case)
    double prior; // a priori probability of the model;
                  // -1 means no prior specified (all models have
                  // equal prior probability (which is our case)
    char *name; // arbitrary name for the model (null terminated utf-8)
    int model_type; // bit flags for various model extensions;
                    // we are using GHMM_kContinuousHMM;
                    // the complete list is in ghmm.h
    ghmm_cstate *s; // all states of the model;
                    // (more details later...)
    ghmm_cmodel_class_change_context *class_change; // pointer to a
                    // ghmm_cmodel_class_change_context
                    // struct necessary for multiple transition classes
                    // used only if cos > 1 (which is not our case)
} ghmm_cmodel;
```

We'll come back to this structure later.

(b) double *O

Contains the sequence of observations. It's length is $2*N$ and, supposing the observed sequence is $(x_{1,}y_1),\ldots,(x_N,y_N)$, one should set O[0] = x_1 , O[1] = y_1 , O[2] = x_2 , and so on...

(c) int T

Length of the sequence O (that is, $2*N$).

(d) double ***b

Optionally precomputed emission probabilities. We'll use NULL instead.

(e) double **alpha

alpha[t][i] is $\alpha_t(i)$, according to our previous notation.

(f) double *scale
    Scale factors. Honestly, I'm not sure what they mean... We don't need them anyway.
(g) double *log_p
    Log likelihood of the observation given the model: $\log(P(O|\lambda))$.
(h) double **beta
    beta[t][i] is $\beta_t(i)$, according to our previous notation.
(i) const double *scale
    There you go: apparently the scale factors computed in the forward method are used here.

OK, let's go back to the details of the ghmm_cmodel structure now. What remains to explain is ghmm_cstate *s, the array of states. Here is the definition of the variables ghmm_cstate, the structure of the state for HMMs with continuous emissions:

```
typedef struct ghmm_cstate {
    int M; // number of output densities per state (1, in our case)
    double pi; // initial probability of the state ( π_i , remember?)
    int *out_id; // IDs of successor states;
                 // they are defined based on the transition probabilities
                 // ( a_ij , remember?); more on this later...
    int *in_id; // IDs of predecessor states
    double **out_a; // transition probabilities to successor states;
                    // it is a matrix in case of cos > 1 (which is not ours);
                    // are based on the transition probabilities;
                    // more on this later...
    double **in_a; // transition probabilities from predecessor states.
    int out_states; // number of  successor states
    int in_states;  // number of  predecessor states
    double *c; // weight vector for output function components;
               // since our emissions are mixture of gaussians with only
               // one component, c is an array of one element only,
               // and the value of this element is 1
    int fix; // flag for fixation of parameter;
             // if fix = 1 do not change parameters of output functions;
             // if fix = 0 do normal training; default is 0;
             // we are using 1;
    ghmm_c_emission *e; // vector of ghmm_c_emission;
                        // type and parameters of output function components;
                        // another thing we will explain later...
    char *desc; // contains a description of the state
                // null terminated utf-8);
                // we're not setting this
    int xPosition; // x coordinate position for graph representation plotting;
                   // we're not setting this
    int yPosition; // y coordinate position for graph representation plotting;
                   // we're not setting this
} ghmm_cstate;
```

To understand these parameters, let's suppose that our training curve has 5 points (in honor to the name of Atlanta's main subway station). In this case, the transition probabilities matrix is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} = \begin{bmatrix} 1/3 & 1/3 & 1/3 & 0 & 0 \\ 0 & 1/3 & 1/3 & 1/3 & 0 \\ 0 & 0 & 1/3 & 1/3 & 1/3 \\ 0 & 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Therefore, being $1,\ldots,5$ the IDs of the states, the output and input states for each state are as follows:

```
state 1, outlets: {1, 2, 3}, inlets: {1}
state 2, outlets: {2, 3, 4}, inlets: {1, 2}
state 3, outlets: {3, 4, 5}, inlets: {1, 2, 3}
state 4, outlets: {4, 5},    inlets: {2, 3, 4}
state 5, outlets: {5},       inlets: {3, 4, 5}
```

This shows how the variables out_id, in_id, out_states, and in_states of the structure ghmm_cstate should be filled.

out_a and in_a are filled according to the following:

```
state 1, output prob.: {1/3, 1/3, 1/3}, input prob.: {1/3}
state 2, output prob.: {1/3, 1/3, 1/3}, input prob.: {1/3, 1/3}
state 3, output prob.: {1/3, 1/3, 1/3}, input prob.: {1/3, 1/3, 1/3}
state 4, output prob.: {1/3, 2/3},      input prob.: {1/3, 1/3, 1/3}
state 5, output prob.: {51},            input prob.: {1/3, 2/3, 1}
```

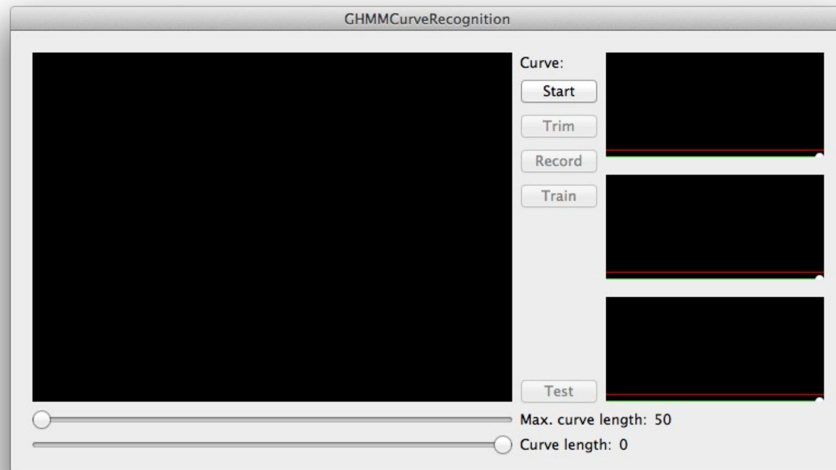
Last, we have ghmm_c_emission *e, the array of emissions. Let's take a look at the declaration of the ghmm_c_emission structure:

```
typedef struct ghmm_c_emission {
    ghmm_density_t type; // type of the density ("binormal" in our case)
    int dimension; // dimension of the multivariate normal (2 in our case)
    union {
        double val;
        double *vec;
    } mean; // mean for output functions
            // (pointer to mean vector for multivariate);
            // we set this as the coordinates of a sampled point
    union {
        double val;
        double *mat;
    } variance; // variance (or pointer to a covariance matrix
                // for multivariate normals);
                // in our case, variance.mat = {0.5, 0, 0, 0.5}, since
                // the covariance matrix of the density for each emission is
                // 0.5 0
                //   0 0.5
    double *sigmainv; // pointer to inverse of covariance matrix
                      // if multivariate normal (else NULL)
    double det; // determinant of covariance matrix for multivariate normal
    double *sigmacd; // Cholesky decomposition of covariance matrix A
    double min; // minimum of uniform distribution or left boundary
                // for right-tail gaussians;
                // we're not wetting this
    double max; // maximum of uniform distribution
                // or right boundary for left-tail gaussians;
                // we're not setting this
    int fixed; // if fixed != 0 the parameters of the density are fixed
               // we're setting this as 1
} ghmm_c_emission;
```

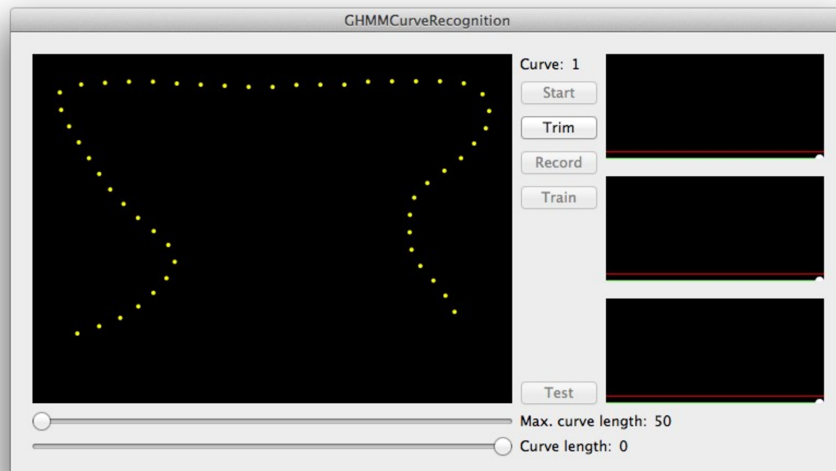
Implementation

We'll go through the interface of the implemented software, so you can understand what it (ideally) does.

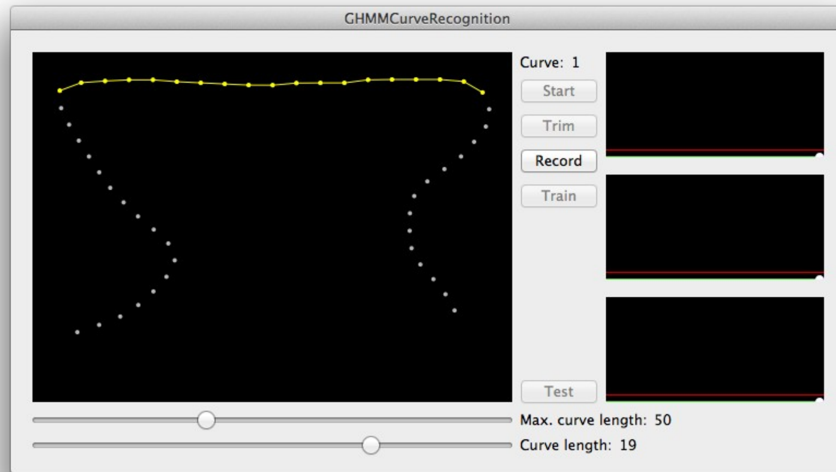
This is what it looks like right after launch:



The user start by clicking the button "Start", and then inputs a curve with the mouse on the largest black view:

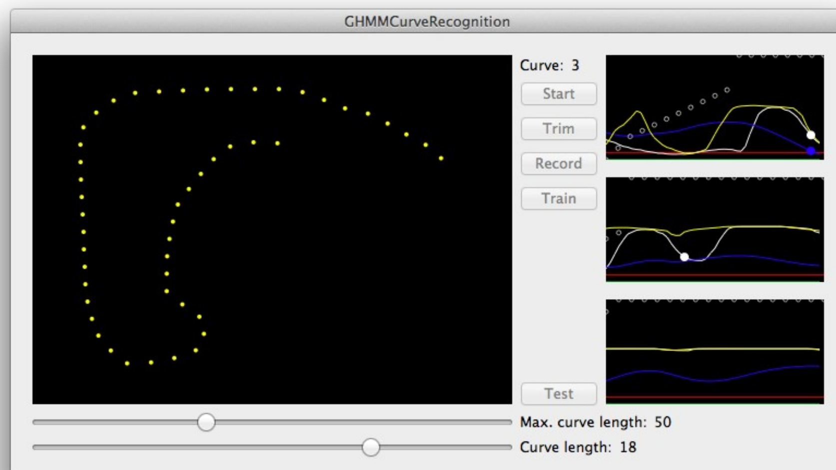


Then the button "Trim" should be pressed, so the input curve can be trimmed using the provided sliders:

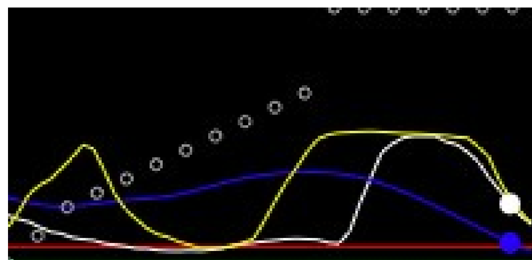


Then this training curve should be recorded by pressing “Record”. This procedure should be repeated three times (as we will have three training curves). After recording the third curve, the button “Train” is enabled (and should be clicked). After clicking in “Train”, the button “Test” is enabled (and should be clicked).

After clicking “Test”, the program is ready for comparing a test input curve with the three recorded training curves. The test curve should be input in the largest black view on the left. After the test curve has a full tail (of 50 points) some information about the matching using HMM is presented on the three black views on the right (each view, from top to bottom, corresponding to one of the training curves, from the first to the last recorded).



Let's take a closer look at one of the views on the right:



In this view, the white, yellow and blue curves represent the proximity of the test curve according to what we call the *gamma criterion*, the *alpha criterion* and the *euclidean criterion*, respectively. We'll explain these in a moment. The red line represents a lower threshold for the euclidian criterion. The filled white and blue circles show points where the test curve is close to the corresponding training curve according to the gamma and euclidian criteria, respectively. The unfilled gray circles represent the mapping between the current tail indices (imagined in the horizontal axis) and the corresponding training curve indices (imagined on the vertical axis). If the curves match perfectly, the unfilled gray circles are aligned from the bottom left to the top right of the view.

We recall that we compute $\gamma_t(i)$, for all $t=1,\ldots,N$ and $i=1,\ldots,N$, and then maximize over i to get the best match for each t . That is, the point in the training curve that is associated with the point $O_t=(x_t, y_t)$ is that which satisfies $max_i \gamma_t(i)$. This criterion is different from the one in [2], where $\alpha_t(i)$ is used instead. We think our approach makes more sense, given the definitions of these variables.

As the test and training curves have the same length, a perfect match happens when the i -th point of the training curve is matched with the i -th point of the test curve, for all i . Therefore, a possible measure of similarity between curves is "how much the set of matched indices deviates from the set corresponding to the perfect match". Let $m(i)$ be index of the point in the test curve that is matched with the point of index i in the training curve. We define the distance between the training and test curves as $(\sum_i |m(i)-i|)/N^2$. The division by N^2 is so that the distance stays in the range $[0,1]$. The distance according to the *gamma* criterion (respectively, the *alpha* criterion) is that where $m(i)$ is computed using the $\gamma_t(i)$ values (respectively, the $\alpha_t(i)$ values).

The distance according to the *euclidian* criterion is simply the euclidian distance between the training and test curves. (Actually we multiply that distance by 2 and divide by N , for a better spread in the range $[0,1]$.

In the case of the euclidian criterion, the test and training curves are identified as similar when the euclidian distance goes below the threshold shown by the red line in the picture above.

For the other criteria, similarity is detected when the tail of similarities shows a consistent decreasing behavior for a time superior to half the length of the training curve, and the accumulated amount of decreasing is above some threshold.

In all cases, once a similarity is detected, the next similarity will only be detected once a time equivalent to 70% of the length of the training curve passed.

Source Code

The software was implemented in Objective-C. The implementation contains 5 classes: InputView, Curve, ObjGHMM, ModelGHMM and RingBufferView.

InputView. Main class. It corresponds to the larger black view on the left of the software graphical interface.

Curve. Contains the points of the training and test curves.

ModelGHMM. Encapsulates the variables of the models for each training curve.

ObjGHMM. Where the matching and distance measurements take place.

RingBufferView. Handles the visualizations and the detection of proximity between training and test curves. Corresponds to the small views on the right of the software graphical interface.

That's All Folks

This is still a work in progress. Feedback is very welcome. If you have any, feel free to contact me through cicconet@gmail.com.

References

[1] Bevilacqua et al. Wireless sensor interface and gesture-follower for music pedagogy. 2007.
[2] Bevilacqua et al. Continuous Realtime Gesture Following and Recognition. 2010.
[3] http://ghmm.org/
[4] Ethem Alpaydin. Introduction to Machine Learning. 2004.