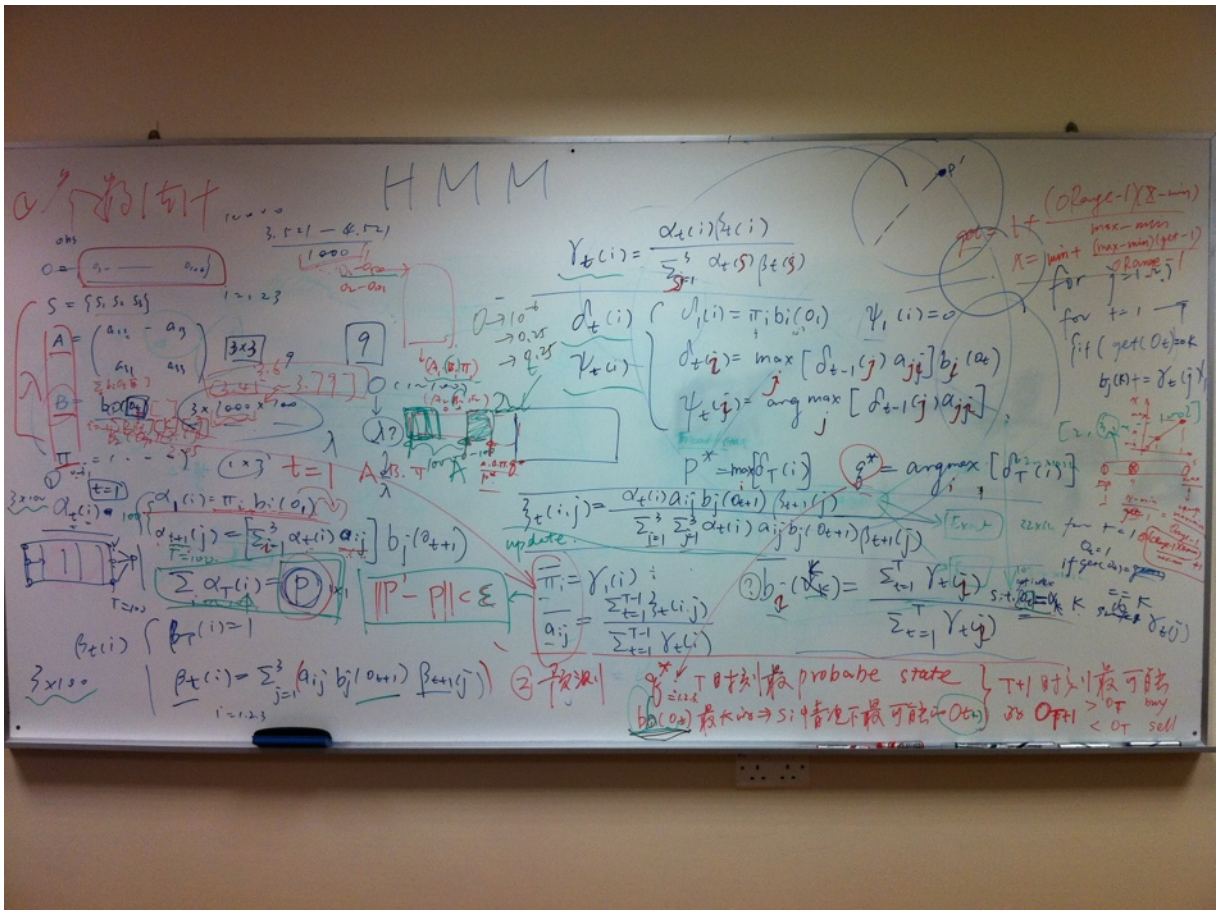
HOW TO IMPLEMENT HIDDEN MARKOV CHAIN

A Framework and C++ Code



LIM PAO SEUN, Qiao Anna & Zhang Zexuan

2010-04

To our dear Prof. Li



CONTENT

Section A Introduction

Section B Methodology

Section C Details of implementing our idea

Section D Code Structure

Section E Results

Section F Limitations and Todos

Appendix Code in C++

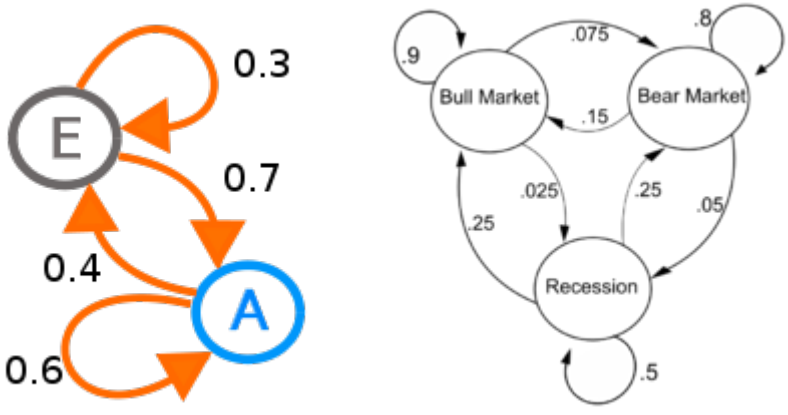
HOW TO IMPLEMENT HIDDEN MARKOV CHAIN

A framework and C++ Code

A0076574U LIM PAO SEUN
A0076561B Zhang Zexuan
A0075463B Qiao Anna
2011.04

Section A Introduction

Before we go into what is Hidden Markov Model, Let's start by introducing you to what is Markov Chain. A **Markov chain**, named for Andrey Markov, is a mathematical system that undergoes transitions from one state to another (from a finite or countable number of possible states) in a chainlike manner. It is a random process endowed with the Markov property: the next state depends only on the current state and not on the past. Formally it is a discrete random process but time can also be seen as a continuous value.



Markov Chain Models

A "discrete-time" random process means a system which is in a certain state at each "step", with the state changing randomly between steps. The steps are often thought of as time, but

they can equally well refer to physical distance or any other discrete measurement; formally, the steps are just the integers or natural numbers, and the random process is a mapping of these to states. The Markov property states that the conditional probability distribution for the system at the next step (and in fact at all future steps) *given* its current state depends only on the current state of the system, and not additionally on the state of the system at previous steps. The changes of state of the system are called transitions, and the probabilities associated with various state-changes are called transition probabilities. The set of all states and transition probabilities completely characterizes a Markov chain. By convention, we assume all possible states and transitions have been included in the definition of the processes, so there is always a next-state and the process goes on forever.

A famous Markov chain is the so-called "drunkard's walk", a random walk on the number line where, at each step, the position may change by +1 or -1 with equal probability. From any position there are two possible transitions, to the next or previous integer. The transition probabilities depend only on the current position, not on the way the position was reached. For example, the transition probabilities from 5 to 4 and 5 to 6 are both 0.5, and all other transition probabilities from 5 are 0. These probabilities are independent of whether the system was previously in 4 or 6.

Example: Imagine that the sequence of rainy and sunny days is such that each day's weather depends only on the previous day's, and the transition probabilities are given by the following table.

	Day t+1	
Day t	**Rainy**	**Sunny**
Rainy	0.9	0.1
Sunny	0.6	0.4

That is: if today is rainy, the probability that tomorrow will be rainy is 0.9; if today is sunny, that probability is 0.6. The weather is then a two-state homogeneous Markov chain, with t.p.m. $\mathbf{\Gamma}$ given by

$$\mathbf{\Gamma} = \begin{matrix} 0.9 & 0.1 \\ 0.6 & 0.4 \end{matrix}$$

Now suppose that today (time 1) is a sunny day. This means that the distribution of today's weather is
$\mathbf{u}(1) = \Pr(C1 = 1), \Pr(C1 = 2)) = (0 , 1)$

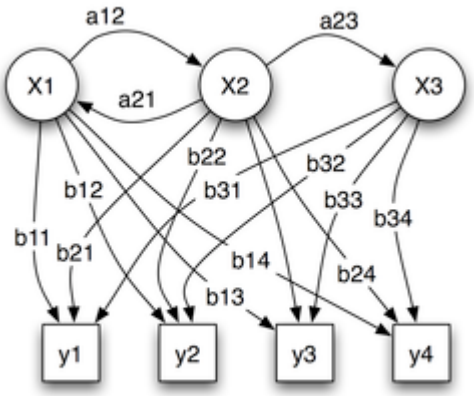
The distribution of the weather of tomorrow, the day after tomorrow, and so on, can be calculated by repeatedly postmultiplying $\mathbf{u}(1)$ by $\mathbf{\Gamma}$, the t.p.m.:
$\mathbf{u}(2) = (\Pr(C2 = 1), \Pr(C2 = 2)) = \mathbf{u}(1)\mathbf{\Gamma} = (0.6, 0.4)$,
$\mathbf{u}(3) = (\Pr(C3 = 1), \Pr(C3 = 2)) = \mathbf{u}(2)\mathbf{\Gamma} = (0.78, 0.22)$, etc.

What is Hidden Markov Model?

A *hidden Markov model* (abbreviated HMM) is, loosely speaking, a Markov chain observed in noise. Indeed, the model comprises a Markov chain, which we will denote by $\{X_k\}_{k\geq 0}$, where k is an integer index. This Markov chain is often assumed to take values in a finite set, but we will not make this restriction in general, thus allowing for a quite arbitrary state space. Now, the ***Markov chain is hidden***, that is, states are not observable. Recall In a regular Markov model, the state is directly visible to the observer, and therefore the state transition probabilities are the only parameters. In a *hidden* Markov model, the state is not directly visible, but output, dependent on the state, is visible. Each state has a probability distribution over the possible output tokens. Therefore the sequence of tokens generated by an HMM gives some information about the sequence of states. Note that the adjective 'hidden' refers to the state sequence through which the model passes, not to the parameters of the model; even if the model parameters are known exactly, the model is still 'hidden'.
What is available to the observer is another stochastic process $\{Y_k\}_{k\geq 0}$, linked to the Markov chain in that X_k governs the distribution of the corresponding Y_k. For instance, Y_k may have a normal distribution, the mean and variance of which is determined by X_k, or Y_k may have a Poisson distribution whose mean is determined by X_k. The underlying Markov chain $\{X_k\}$ is sometimes called the *regime*, or *state*. All statistical inference, even on the Markov chain itself, has to be done in terms of $\{Y_k\}$ only, as $\{X_k\}$ is not observed. There is also a further assumption on the relation between the Markov chain and the observable process, saying that X_k must be the only variable of the Markov chain that affects the distribution of Y_k. This is expressed more precisely in the following formal definition.

Example: The urn

In its discrete form, a hidden Markov process can be visualized as a generalization of the familiar Urn problem. For instance, from Rabiner 1989: A genie is in a room that is not visible to the researcher. It is drawing balls labeled y1, y2, y3, ... from the urns X1, X2, X3, ... in that room and putting the balls on a conveyor belt, where the researcher can observe the sequence of the balls but not the sequence of urns from which they were chosen. The genie has some procedure to choose urns; the choice of the urn for the n-th ball depends upon only a random number and the choice of the urn for the $(n - 1)$-th ball. Because the choice of urn does not directly depend on the urns further previous, this is called a Markov process. It can be described by the upper part of the diagram at the top of this article.



Probabilistic parameters of a hidden Markov model
x — states
y — possible observations
a — state transition probabilities
b — output probabilities

Because the Markov process itself cannot be observed, and only the sequence of labeled balls can be observed, this arrangement is called a "hidden Markov process". This is illustrated by the lower part of the diagram above, where one can see that balls y1, y2, y3, y4 can be drawn at each state. Even if the researcher knows the composition of the urns and has just observed a sequence of three balls, *e.g.* y1, y1 and y1 on the conveyor belt, the researcher still cannot be sure from which urn (*i.e.*, at which state) the genie has drawn the third ball. However, the researcher can work out other details, such as the identity of the urn the genie is most likely to have drawn the third ball from.

What is HMM used for?

Hidden Markov models have been used for at least three decades in signal-processing applications, especially in the context of automatic speech recognition, but interest in their theory and application has expanded to other fields,

e.g.:
• all kinds of recognition: face, gesture, handwriting, signature;
• bioinformatics: biological sequence analysis;
• environment: wind direction, rainfall, earthquakes;
• finance: series of daily returns;
• biophysics: ion channel modeling.

Section B Methodology

Main Idea

The main purpose of this part is to introduce the methodology we use in this project. In order to apply the Hidden Markov Chain, we divide the total historical data of 1400 data points into two parts. The first part consisted of the first 1000 numbers and is used to construct a well enough model while the second part of the rest 400 numbers is used to implement our trading strategy to see how well it works.

Constructing the model

In this part, we will show how to construct a model step by step and to forecast the foreign exchange rate with the adoption of Hidden Markov Chain. For this purpose, the knowledge on Markov Chain is essential.

Markov chain

A Markov chain, sometimes also referred to as an observed Markov Model, can be seen as

a weighted finite state automaton, which is defined by a set of states and a set of transitions between the states based on the observed input. In the case of the Markov chain the weights on the arcs going between the different states can be seen as probabilities of how likely it is that a particular path is chosen. The probabilities on the arcs leaving a node (state) must all sum up to 1. In figure 2.1 there is a simple example of how this could work.

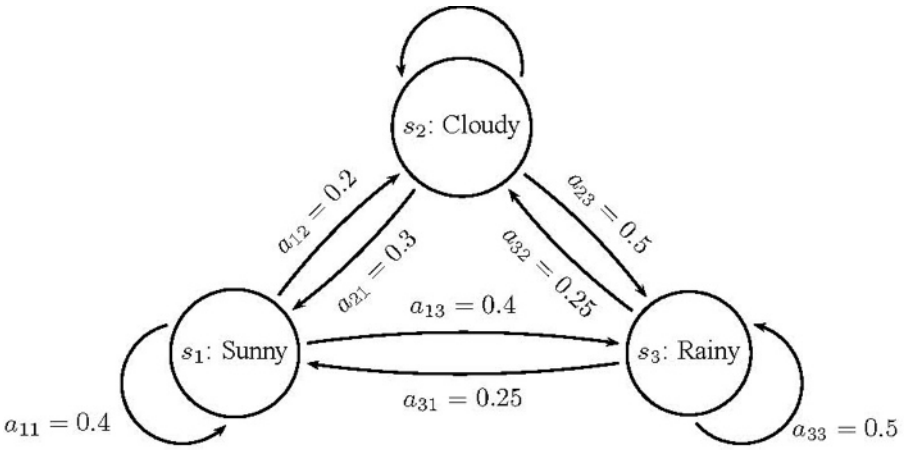


Figure 2.1: A simple example of a Markov chain explaining the weather. a_{ij}, found on the arcs going between the nodes, represents the probability of going from state s_i to state s_j.

In finance, stock prices are often assumed to follow a Markov process. This means that the present value is all that is needed for predicting the future, and that the past history and the taken path to today's value is irrelevant. Considering that equity and currency are both financial assets, traded under more or less the same conditions, it would not seem farfetched assuming that currency prices also follow a Markov process.

Hidden Markov Chain

Given the markov chain, we can extend our knowledge to hidden markov chain. In reality, more often than not, we cannot get the right or accurate transition probability, or the initial probability. What's more, it is also difficult to decide the interval and the corresponding state. Thus, we have hidden markov chain at hand.

Some useful parameters and notations:

$S = \{s_1, s_2, ..., s_N\}$	a set of N *hidden states*,
$Q = \{q_1, q_2, ..., q_T\}$	a *state sequence* of length T taking values from S,
$O = \{o_1, o_2, ..., o_T\}$	an *observation sequence* consisting of T observations, taking values from the discrete alphabet $V = \{v_1, v_2, ..., v_M\}$,
$A = \{a_{11}, a_{12}, ..., a_{NN}\}$	a *transition probability matrix* A, where each a_{ij} represents the probability of moving from state s_i to state s_j, with $\sum_{j=1}^{N} a_{ij}, \forall i$,
$B = b_i(o_t)$	a sequence of observation likelihoods, also called *emission probabilities*, expressing the probability of an observation o_t being generated from a state s_i at time t,
$\Pi = \{\pi_1, \pi_2, ..., \pi_N\}$	an *initial probability distribution*, where π_i indicates the probability of starting in state s_i. Also, $\sum_{i=1}^{N} \pi_i = 1$.

To fully describe a model within HMM, we only need the parameter set combined by A, B and Π. Then, we need to use the historical data to compute the parameter set to construct our model.

Initial model

At this stage, we first need to pick up the first 100 data points (the window length is 100). There are still three more steps to be done: Computing likelihood (P(O|λ)), Decoding and Learning.

Computing likelihood:

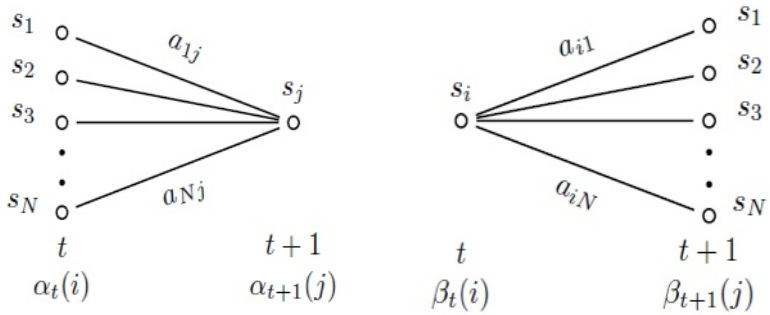
We implement the forward-backward procedure here to compute P(O|λ). The detailed methods are shown below.

Forward: αt(i) = P(o1, o2, . . . , ot, qt = si|λ).
In other words, the probability of the partial observation sequence, o 1, o2, . . . , ot until time t and given state si at time t. One can solve foαt(i) inductively as follows:

(i). Initialization:
α1(i) = πibi(o1), 1 ≤ i ≤ N.

(ii). Induction:
, 1≤t≤T, 1≤j≤N

(iii).Termination
P(O|λ)



Backward: = P(ot+1, ot+2, . . . , oT |qt = si,λ)
is the probability of the partial observation sequence from t + 1 to the last time,T, given the state si at time t and the HMM λ. By using induction, is found as follows:
(i). Initialization:
= 1, 1 ≤ i ≤ N.
(ii). Induction:
, t=T-1, T-2...,1. 1 ≤ i ≤ N.

DECODING:

In this the second problem we try to find the"correct" hidden path, i.e. trying to uncover the hidden path. This is often used when one wants to learn about the structure of the model or to get optimal state sequences. For this purpose we can refer to the Viterbi Algorithm which can be realized from the following procedures.

To find the best state sequence Q = {q1, q2, . . . , qT } for a given observation sequence O = {o1, o2, . . . , oT }, one need to define the quantity
δt(i) = max{P[q1, q2, . . . , qt = si, o1, o2, . . . , ot|λ]
which means the highest probability along a single path, at time t, which accounts for the first t observations and ends in state si.

- Initialization
  - $\delta_1(i) = \pi_i b_i(\omega_1)$
  - $\psi_1(i) = 0$
- Recursion
  - $\delta_t(i) = \max[\delta_{t-1}(i)a_{ij}]b_j(\omega_t)$
  - $\psi_t(j) = \text{argmax}[\delta_{t-1}(i)a_{ij}]$, the state chosen at t
- Termination
  - $P^* = \max[\delta_T(i)]$
  - $q^* = \text{argmax}[\delta_T(i)]$

LEARNING:

After computing the sequence, we need to re-estimate the model to improve the level of P(O|λ). We need to introduce two parameters here for re-estimation using the Baum-Welch

method.

- $\xi t(i, j) = P(qt = si, qt+1 = sj | O, \lambda)$
  which means the probability of being in state si at time t, and
  state sj at time t + 1. And it can be derived from the following way.

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{P(O|\lambda)} =$$

$$= \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}$$

- $\gamma t(i) = P(qt = si | O, \lambda)$
  which gives the probability of being in state si at time t given the observation sequence, O, and the model, λ. Also, it can be computed by the following way.

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N}\alpha_t(i)\beta_t(i)}.$$

Thus, every time a new data come in, we can re-estimate our model as below.

- By summing up over time,
- $\gamma_t(i)$ ~ the number of times s_i is visited
- $\xi_t(i,j)$ ~ the number of times the system goes from state s_i to state s_j
- Thus, the parameters λ are:
  - $\overline{\pi_i} = \gamma_1(i)$, initial state probabilities
  - $\overline{a_{ij}} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\gamma_t(i)}$, transition probabilities
  - $\overline{b_j(v_k)} = \frac{\sum_{t=1,\omega_t=v_k}^{T-1}\gamma_t(j)}{\sum_{t=1}^{T-1}\gamma_t(i)}$, conditional probabilities

One should see that the first equation can be interpreted as the frequency in state s i at time t = 1. The second equation should be interpreted as the expected number of transitions from state si to sj divided by the number of transitions from state si. And the final equation can be seen as the expected number of times in state s j and observing the symbol vk, divided by the expected number of times in state sj .

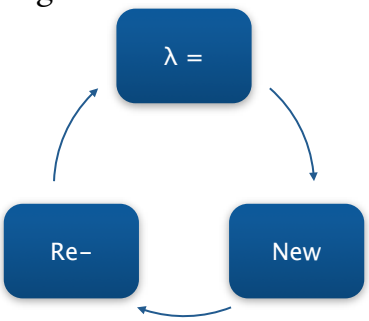
Final Model

To get the final model, we need to repeat the iteration process at each time point once a new data come in. Thus, within the first 1000 data points, we choose the parameters below according to empirical results as below:

- Window length=100;
- N=3;
- M=5000;

Then, random the initial parameter set and gets the first 100 state sequences using the Viterbi

Algorithm. The final model can be obtained step by step as shown below.



We first have the initial λ from random process. After knowing the first 100 state sequences, we move the window from time 2 to time 101, and re-estimate the model until it converges. By repeating the model with the window length of 100, a final model can be got using from time 901 to time 1000.

Applying HMM (Discrete)

After the final model constructed, we can apply the model to test how well this model works using the rest 400 data points. One of the most important steps in a trading strategy is to create a correct trading signal. Without a sophisticated signal, even a good parameter estimation can result in poor performance.
For the discrete case we have at hand, only one signal will be tested. Here, one simply determine the most probable state for tomorrow, q (t+1) = s(r), according:
r = argmaxj{aij} where i indicates the most probable state today according to the Viterbi path, i.e. q(t)=s(i), and r = 1, . . .,N where N is the number of states in the model.

Having done this, one wants to know the most probable observation, o(t+1) = os(t+1). But since B is not available for the time t + 1, one have to use the last known B, which is available for the time t, relaying on that this will give good enough estimates. This gives: s = argmaxk{br(okt)}

On the basis of the forecasted observation of o(t+1), one can have his own trading strategy on comparing it with o(t), if o(t+1)> o(t), go long, otherwise go short.

In the next part, we will see in detail how to get the model using Visual C++ and how it works to earn money using the trading strategy we have.

Section C Details of implementing our ideas

Parameter Estimation

1 The choice of B(emission matrix)

We choose discrete distribution, which means that the distribution is not smooth. We also choose time homogenous B to avoid high memory cost N * M * T(3 * 5000 * 100 for our data) in one iteration step.

Besides, the author of paper did not give the method to how to update matrix B in 'studying' section if B is a 3-dimensional matrix. So we choose time homogenous one.

Moreover, if we choose continuous distribution, we can do this because there is less parameters to estimate. Besides, how to update time varying B is given in the paper.

2 How to compute updated B?

Matrix B, as mentioned before, is a time homogenous one. So it is a N * M matrix. The difficulty here is that we need to consider all of the numbers which elements of B may take value. We compute the maximum and minimum of observation sequence and cut them into several numbers, then put them in order in matrix B. Moreover, we write a function of get_index and its inverse function to transfer between the value and the order of it.

3 Convergence condition

When we run an iteration, we get a P. Then if P converges, we stop the iteration and get the updated parameters.

We take parameters from last window as initialization of new window, when the window moves from t to t + 1. Why not only generate a random parameter in the last window and do iteration to get updated parameters in last window?

Because the maximum likelihood is local maximum, not global maximum. Accordingly, the optimal parameters are local optimal, not global optimal. If we choose wrong initialization, we may never get the correct parameters. So we do iteration time by time.

There is an assumption that parameters changes little w.r.t. time t, say, continuous.

Prediction

- Find the q from last window -> the most likely state(t + 1)
- Find the B from last window -> the most likely obs given state
- Combine them to get most likely o(t + 1)

Here is a problem that we cannot get the B(t + 1). What we can do is assume previous B(t) is same to B(t + 1), then use B(t) and q to get new observation o(t + 1).

Trading strategy and backtesting

We use the simplest strategy that if predicted o(t + 1) > o(t), we buy. Otherwise sell.

When we do backtesting, we set an indicator function I(buy or sell), which value is +1 when buy, -1 when sell. Then daily P&L = I * (true value(t + 1) - true value(t)) for valuation. Finally, we sum daily P&L to get accumulate P&L, then plot it.

Section D Code Structure

double update data {

input = window length data

output = a, b, pie in one step iteration, by using HMM}

int main {

generate a b pie(window length, state numbers) in uniformly distribution

update data(1~ window length) compute p, q

update until p converges

get best a, b, pie @ time 1 ~ window length

use a, b, pie @ time period 1 as initialization, run update data iteration until p converges

iteration until end of data

prediction:

q at last window

b at last window

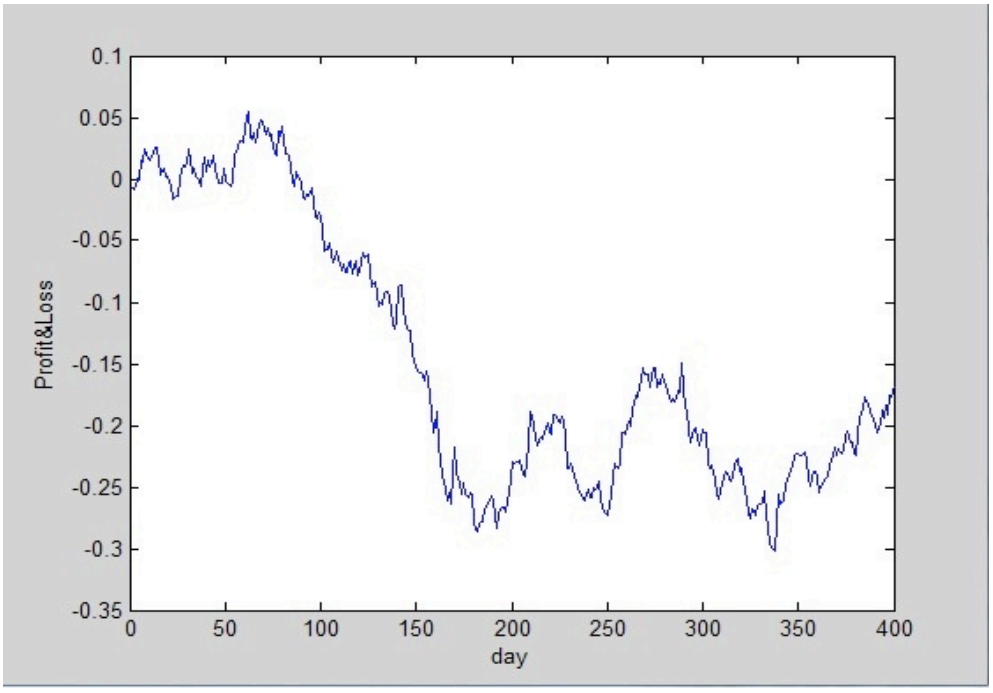
to get most likely price @ time t + 1, say o(t + 1)

our strategy

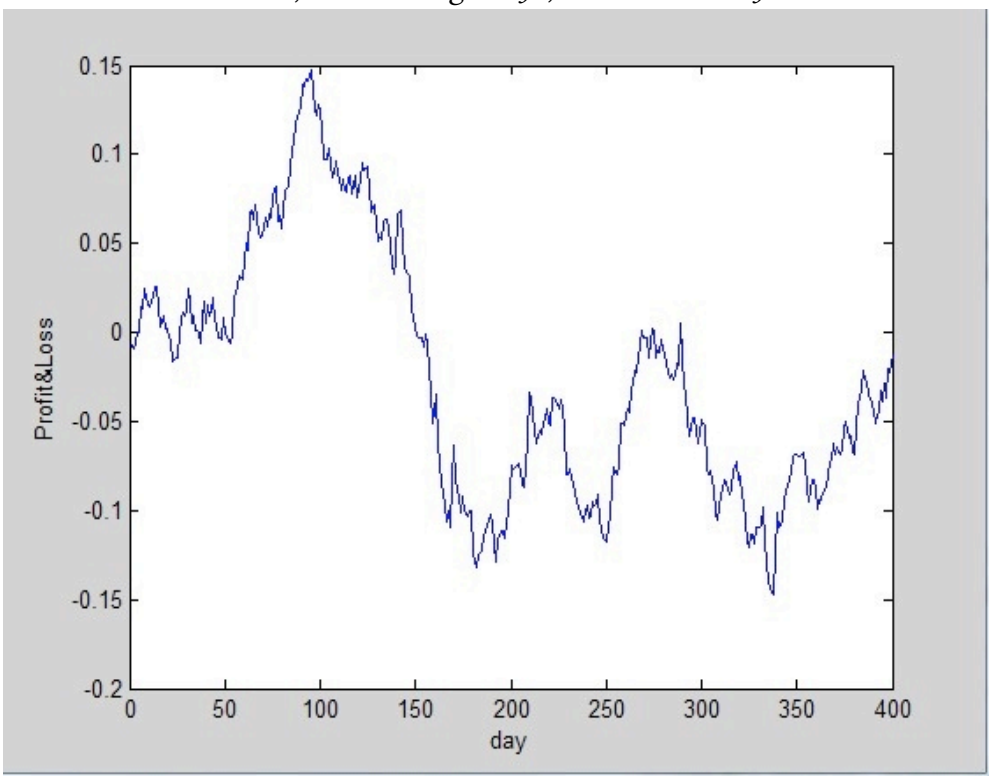
backtesting and draw p & l (in Matlab for convenience)

}

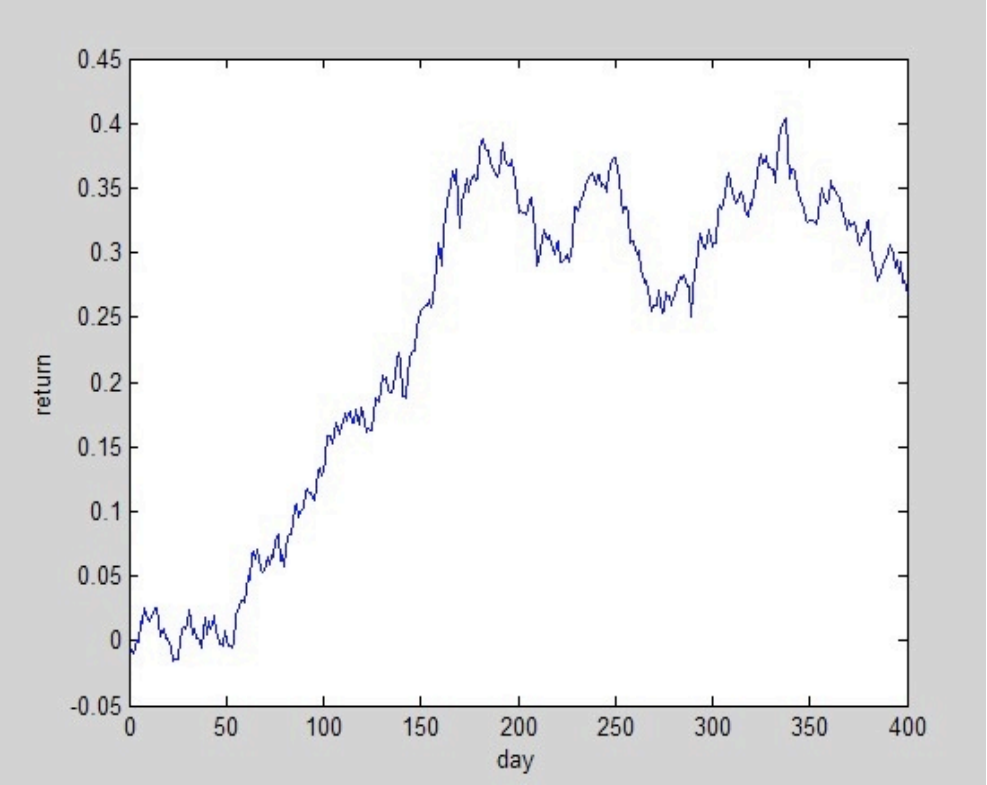
Section E Results



P&L, window length = 30, state number = 3



P&L, window length = 50, state number = 3



P&L, window length = 100, state number = 3

From previous figures we can get the same conclusion as the paper. It gets the best result when window length = 100.

We assume we can only buy or sell 1 unit of FX each time. When we earn money, we can get about 0.001 - 0.01 profit. We get 0.3 profit at last. So we win about 60 times more than lose. It's great. So our return rate is about 0.3 per 400 trading days.

I have attached my input and output datas. Please check it.

Section F Limitations and todos

Drawbacks:

a. Iteration: Why we can use paras from last window to initialize paras of this window?

b. Prediction: Why can use previous B to get next observation?

c. Matrix B: time homogenous, does not make sense

d. Discrete Distribution: Not smooth, not continuous.

e. How many shares we will buy/sell? We assume +1/-1.

TODOs

a. Continuous Distribution and GMM

b. We can create more trading strategy:

Say, computing expectation price rather than the most likely one. Because their are too many probabilities in a vector. Besides, their differences are slight; this may cause more errors.

c. Transaction cost can be easily considered. When o(t + 1) - o(t) >(<) c, buy/sell.

Appendix: Code

How to use my code?

You can import data from input.txt and input2.txt. Input.txt denotes the historical data to get model and model parameters. Input2.txt denotes the data for backtesting.

You can download data from any website as you wish, then put them into input.txt and input2.txt in a column. Then you have to put how many data you use in the first line in input.txt and input2.txt. For example, if you have 5 data, your format should be like this,

5

1.1

1.2

1.2

1.1

1.2

These two documents should be put in folder 'data'. Then modify the path in C++ code. Modify the parameters in code and run it, and a document output.txt is automatically generated and the result are in it. The results denote the daily profit and loss.

Then modify your path of output.txt and run Matlab code plot_dailypl.m to get a figure of accumulated p&l. That's the result.

C++

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <cmath>
//double atof(const char* str)

using namespace std;

#define MAXT 3000
#define MAXSTATE 4
#define MAXRANGE 5005
#define maxString 52
#define maxResult 405

double alpha[MAXT][MAXSTATE];
double beta[MAXT][MAXSTATE];
double gamma[MAXT][MAXSTATE];
double delta[MAXT][MAXSTATE];
double psi[MAXT][MAXSTATE];//Ψ
double xi[MAXT][MAXSTATE][MAXSTATE];

inline int getIndex(const double& value,const double& min,const double&
max,const int& k)
{
	int ret;
	//ret = int((value - min)*((max-min)/k)); // [possible error 1]
	ret = (k - 1)*(value - min) / (max-min);
	return ret;
}



// all the matrix start from 1 to max
// oMin is the minimal value of O
double updateModel(int& q,int tWindow, int oRange, double oMin, double oMax, int
stateNum, double _o[MAXT],double _A[MAXSTATE][MAXSTATE],double _B[MAX-
STATE][MAXRANGE],double _Pi[MAXSTATE])
{
	double p;

	/*    calculate lambda */
	// alpha
	for(int s=1;s<=stateNum;s++)
```

```
            alpha[1][s] = _Pi[s]*_B[s][getIndex(_o[1], oMin, oMax, oRange)];

      for(int t=2;t<=tWindow;t++)
      {
            for(int s=1;s<=stateNum;s++)
            {
                  alpha[t][s] = 0;
                  for(int j=1;j<=stateNum;j++)
                        alpha[t][s] += alpha[t-1][j] * _A[j][s] * _B[s][getIn-
dex(_o[t], oMin, oMax, oRange)];
            }
      }

      // p
      p = 0;
      for(int i=1;i<=stateNum;i++)
            p+=alpha[tWindow][i];

      //beta
      for(int s = 1; s <= stateNum; s++)
            beta[tWindow][s] = 1;

      for(int t = tWindow - 1; t >= 1; t--)
      {
            for(int s = 1; s <= stateNum; s++)
            {
                  beta[t][s] = 0;
                  for(int j=1;j<=stateNum;j++)
                        beta[t][s] += beta[t + 1][j] * _A[j][s] * _B[s][getIn-
dex(_o[t + 1], oMin, oMax, oRange)];
            }
      }

      //gamma
      for (int t = 1; t <= tWindow; t ++){
            for (int i = 1; i <= stateNum; i ++){
                  gamma[t][i] = 0;
                  for (int s = 1; s <= stateNum; s ++){
                        gamma[t][i] += alpha[t][s] * beta[t][s];
                  }
                  gamma[t][i] = alpha[t][i] * beta[t][i] / gamma[t][i];
            }
      }

      //delta, psi
      for (int i = 1; i <= stateNum; i ++){
            delta[1][i] = _Pi[i] * _B[i][getIndex(_o[1], oMin, oMax, oRange)];
            psi[1][i] = 0;
      }

      for (int t = 2; t <= tWindow; t ++){
            for (int i = 1; i <= stateNum; i ++){
```

```
                  int k = 1;
                  delta[t][1] = delta[t - 1][1] * _A[1][i] * _B[i][getIn-
dex(_o[t], oMin, oMax, oRange)];

                  for (int j = 2; j <= stateNum; j ++)
                  {
                        if ((delta[t - 1][j] * _A[j][i]) > (delta[t - 1][k] *
_A[k][i]) )
                        {
                              delta[t][i] = delta[t - 1][j] * _A[j][i] *
_B[i][getIndex(_o[t], oMin, oMax, oRange)];
                              k = j;
                        }
                  }
                  psi[t][i] = k;
            }
      }

      int k = 1;
      double p_star = delta[tWindow][1];
      for (int i = 1; i <= stateNum - 1; i ++)
      {
            if (delta[tWindow][i + 1] > delta[tWindow][k])
            {
                  p_star = delta[tWindow][i + 1];
                  k = i + 1;
            }
      }
      int q_star = k;

      //xi
      for (int t = 1; t <= tWindow - 1; t ++)
      {
            for (int i = 1; i <= stateNum; i ++)
            {
                  for (int j = 1; j <= stateNum; j ++)
                  {
                        xi[t][i][j] = 0;
                        for (int s1 = 1; s1 <= stateNum; s1 ++)
                        {
                              for (int s2 = 1; s2 <= stateNum; s2 ++)
                              {
                                    xi[t][i][j] = xi[t][i][j] + beta[t + 1][s2]
* _B[s2][getIndex(_o[t + 1], oMin, oMax, oRange)] *  _A[s1][s2] * alpha [t][s1];
                              }
                        }
                        xi[t][i][j] = beta[t + 1][j] * _B[j][getIndex(_o[t + 1],
oMin, oMax, oRange)] *  _A[i][j] * alpha [t][i] / xi[t][i][j];
                  }
            }
      }

      //update
```

```
	for (int i = 1; i <= stateNum; i ++)
	{
		_Pi[i] = gamma[1][i];
		for (int j = 1; j <= stateNum; j ++)
		{
			double numerator = 0;
			double denominator = 0;
			for (int t = 1; t <= tWindow - 1; t ++)
			{
				numerator += xi[t][i][j];
				denominator += gamma[t][i];
			}
			_A[i][j] = numerator / denominator;
		}

		double tmp,detmp;

		for(int k=1; k<=oRange; k++)
		{
			tmp = 0;
			detmp = 0;
			for(int t=1; t<=tWindow; t++)
			{
				if(getIndex(_o[t], oMin, oMax, oRange) == k )   tmp+=ga-
mma[t][i];
				detmp+=gamma[t][i];
			}
			_B[i][k] = tmp/detmp;
		}
	}
	q = q_star;
	return p;
}

//double _A[maxState][maxState],double _B[maxState][MAXRANGE],double _Pi[max-
State]

void converge(int& q, double previousP,double threshold, int tWindow, int
maxRange, double oMin, double oMax, int stateNum, double _o[MAXT],double _A[MAX-
STATE][MAXSTATE],double _B[MAXSTATE][MAXRANGE],double _Pi[MAXSTATE])
{
	double currentP = updateModel(q, tWindow,maxRange,oMin,oMax,stateNum, _o,
_A,_B,_Pi);
	while(fabs(currentP-previousP)>threshold)
	{
		previousP = currentP;
		currentP = updateModel(q, tWindow,maxRange,oMin,oMax,stateNum, _o,
_A,_B,_Pi);
	}
}

int main()
{
```

```
ifstream fin1("..\\data\\input.txt");
ifstream fin2("..\\data\\input2.txt");
ofstream fout("..\\data\\output.txt");

double result[maxResult];
double _o[MAXT];
double _A[MAXSTATE][MAXSTATE];
double _B[MAXSTATE][MAXRANGE];
double _Pi[MAXSTATE];

int oRange;
int nState;
double oMin;
double oMax;
int tWindow;

/*
########################################################################
Begin- Input data
*/
string tnum;
char tmps[maxString];
double t;
int cnt1, cnt2;
int cnttmp;

/*    Get the num of input1 and input2 */
if(!fin1.eof())
{
      getline(fin1,tnum);
      strcpy(tmps,tnum.c_str());
      t = atof(tmps);
      cnt1 = int(t);
}
if(!fin2.eof())
{
      getline(fin2,tnum);
      strcpy(tmps,tnum.c_str());
      t = atof(tmps);
      cnt2 = int(t);
}

/*    Get the real data of input1 and input2 */
cnttmp = 1;
oMin = oMax = 0;
while(!fin1.eof())
{
      getline(fin1,tnum);
      strcpy(tmps,tnum.c_str());
      t = atof(tmps);
      _o[cnttmp++] = t;

      if(oMin > t)      oMin = t;
```

```
            if(oMax < t)       oMax = t;
//          printf("1: %lf\n",t);
      }

      //printf("oMin = %lf, oMax = %lf\n",oMin, oMax);

      while(!fin2.eof())
      {
            getline(fin2,tnum);
            strcpy(tmps,tnum.c_str());
            t = atof(tmps);
            _o[cnttmp++] = t;
            //printf("2: %lf\n",t);
      }

      /*
      End- Input data
      ########################################################################
      */

      /*

      Parameters to set:
      int oRange;
      int tWindow;

      */
      int maxRange = 5000;
      tWindow = 70;
      nState = 3;

      double previousP = 0;
      double threshold = 1e-8;

      // [To do]
      for(int i=1;i<=nState;i++)
            for(int j=1;j<=nState;j++)
                  _A[i][j] = (1.0)/ nState;

      for(int i=1;i<=nState;i++)
            for(int j=1;j<=maxRange;j++)
                  _B[i][j] = (1.0)/maxRange;

      for(int i=1;i<=nState;i++)
            _Pi[i] = (1.0)/nState;

      /*
      ########################################################################
      Begin- Process data
      */
      int q_star;

      converge(q_star,previousP,threshold, tWindow, maxRange, oMin, oMax, 3,
```

```
_o,_A,_B,_Pi);

	int bestIndex = 1;	// the index of O(T+1)
	int tmp;
	int choice;
	double predictValue,currentValue;
	double bestValue;

	for(int k=1;k<=cnt2;k++)	// cnt2 Real Data
	{
		currentValue = _o[cnt1+k-1];

		bestValue = 0;
		for(int i=1;i<=maxRange;i++)
		{
			//tmp = getIndex(_o[cnt1+k], oMin, oMax, maxRange);

			if(_B[q_star][i] > bestValue)
			{
				bestValue = _B[q_star][i];
				bestIndex = i;
			}
		}
		predictValue = oMin + (oMax - oMin) * (bestIndex-1) /(maxRange-1);
//index --> value

		converge(q_star,previousP,threshold, tWindow, maxRange, oMin, oMax,
3, _o+k,_A,_B,_Pi);


		if(predictValue > currentValue) choice = 1;
		else choice = -1;

		result[k] = choice * (_o[cnt1+k] - _o[cnt1+k-1]);
	}

	/*
	End- Process data
	#########################################################################
	*/


	/*
	#########################################################################
	Begin- Output data
	*/

	for(int i=1;i<=cnt2;i++)
		fout <<	result[i] << endl;

	/*
	End- Output data
	#########################################################################
```

```
        */
        fin1.close();
        fin2.close();
        fout.close();

        return 0;

}
```

PLOT IN MATLAB

```
log_return=textread('F:\HMM\data\output.txt');
x = [1 : 400]';
p_l = [1 : 400]';
for i = 1:400
    p_l(i) = 0;
    for j = 1: i
        p_l(i) = p_l (i) + log_return(j);
    end
end
plot(x,p_l)
xlabel('day')
ylabel('Profit&Loss')
```