Multi-Agent Optimization of Hidden Markov Models

Magnus Erik Hvass Pedersen (971055)
Daimi, University of Aarhus, January 2004

1 Introduction

The purpose of this document is to verify attendance of the author to the *Data Mining* course at DAIMI, University of Aarhus.

Following an introduction of the Hidden Markov Model, its training is described in terms of multi-agent optimization and the related implications are solved.

Rabiner's article [1] is used throughout without explicit citation. The reader is assumed to be familiar with basic probability theory, state-machines, meta-heuristical optimization, and related topics.

2 Hidden Markov Model

A *Hidden Markov Model* (HMM) can be considered a state-based machine in which the transitions between states occur probabilistically. Then, for each state the machine is in, a value from a discrete and finite set is output also according to a probability.

The transition and output probabilities of the HMM are then *trained* to mimic a given observation sequence so that the HMM may be used to classify or synthesize similar sequences, or the HMM may be used to predict likely values for future or unseen data.

2.1 Notation

The states of the HMM are denoted $S = \{S_1, \cdots, S_N\}$, and the probability of transition from state i to state j is denoted a_{ij}. The probability of the machine being in state i upon initialization is π_i, and the state the machine is in at time t is denoted q_t. So the transition probability may be written as:

$$a_{ij} = P[q_t = S_j | q_{t-1} = S_i] \tag{1}$$

and the probability for the initial state being i can be written as:

$$\pi_i = P[q_1 = S_i]$$

For first order HMMs, the transition probabilities only depend on the current state of the machine:

$$a_{ij} = P[q_{t+1} = S_j | q_t = S_i] = P[q_{t+1} = S_j | q_t = S_i,\ q_{t-1} = S_k,\ \cdots]$$

When the socalled *output alphabet* $V = \{v_1, \cdots, v_M\}$ is discrete and finite, the probability for outputting the *character* $v_k \in V$ when the machine is in state j, is given by $b_j(k)$. The output sequence is denoted $O = O_1\, O_2 \cdots O_T$, consisting of T individual *observations*. Thus the probability of outputting character v_k may be written as:

$$b_j(k) = P[O_t = v_k | q_t = S_j] \tag{2}$$

Note that this probability is independent of the time-step t.

The model can then be summarized as $\lambda = (A, B, \pi)$, where $A = \{a_{ij}\}$, $B = \{b_j(k)\}$, and $\pi = \{\pi_i\}$. Hence, A and B can be thought of as matrices, and π as an array.

2.2 Topology and Constraints

When any state can be reached from any other state in a single step, that is: $a_{ij} > 0$, the model is said to be *ergodic*. Another possibility is *left-right* in which transitions to previously visited states are disallowed: $i > j \Rightarrow a_{ij} = 0$. Note that transitions to the same state are still allowed though, and the length of the output can therefore still be greater than the number of states, if so desired.

We always assume that the probabilities are in the range $[0, 1]$ and are mutually consistent:

- Exactly one initial state is chosen:

$$\sum_{i=1}^{N} \pi_i = 1 \tag{3}$$

- Exactly one transition will occur from each state i:

$$\sum_{j=1}^{N} a_{ij} = 1 \tag{4}$$

- Exactly one character will be output each time the machine is in any given state j:

$$\sum_{k \in V} b_j(k) = 1 \tag{5}$$

2.3 Probabilistic Synthesis

Synthesizing observation sequences with a probabilistic model based on analysis of real data, has the advantage that plausible but seemingly unique data can be produced without detailed knowledge of the particular domain.

For example, in a computer game where the player may be situated in many different worldly locations, the weather can be simulated by such a model trained

with data for each location. A manual implementation would require considerably more effort on behalf of the developer, both in regards to analysis of weather data, but also transforming this into sensible `if-then-else` statements or similar programming paradigms.

It is naturally assumed that the underlying process of the physical system is indeed probabilistic in the HMM fashion. This means that we should generally not expect HMM's to recognize or generate highly ordered (e.g. long-periodic) sequences with great accuracy. Hidden Markov Models are therefore perhaps best used in *classification* tasks where we merely need to determine if one model is better than another at recognizing the sequence to be classified - but where the probability of the HMM generating that exact sequence is actually very low.

2.4 Synthesis Algorithm

Once the model λ exists however, synthesizing an observation sequence O of length T, is done by traversing the states as one would do in any state-based machine - only the transitions aswell as the choice of output are of course probabilistic. The algorithm is as follows:

- The iterative state variable q is initialized according to the probability distribution π. The counter t is also initialized ($t := 1$), and the following is repeated until the entire sequence has been generated ($t = T$).
  - Select a character $v \in V$ according to the probabilities $b_q(k)$. This is the output for the current time-step: $O_t = v$.
  - Select a new state q' according to the probabilities a_{qj}, and update the iterative state variable: $q \leftarrow q'$.
  - Update the counter $t \leftarrow t + 1$.

Selecting one of N available states can be implemented with socalled *roulette wheel selection* [2], in which each state would be assigned a slice on a roulette wheel of size according to its probability, a random number is then drawn to choose amongst these, simulating a spin of the wheel.

3 HMM Training

We need to determine the parameters λ - and implicitly the number of states N - that maximize the likelihood of observing the training sequence O:

$$P = P[O|\lambda]$$

Again, let the sequence contain the individual observations $O = O_1 \cdots O_T$ and assume that the HMM is ergodic, then the probability P is the sum over the probabilities of outputting O through all possible state sequences of length T.

The probability of outputting O over a given state sequence $q_1\, q_2 \cdots q_T$, is simply their product because their probabilities are independent, hence:

$$P[O|\lambda] = \sum_{q_1,\cdots,q_T \in S} \pi_{q_1} b_{q_1}(O_1) \cdot a_{q_1 q_2} b_{q_2}(O_2) \cdots a_{q_{T-1} q_T} b_{q_T}(O_T)$$

Calculating this directly however, yields exponential time-complexity and a faster algorithm is thus needed.

3.1 Forward Procedure

The probability P may be computed with polynomial (quadratic) time-complexity by the following algorithm. First define the *forward variable* $\alpha_t(i)$ that is the probability for the given model λ to be in state i at time-step t, and having observed the first part of the sequence $O_1 \cdots O_t$:

$$\alpha_t(i) = P[O_1 \cdots O_t, q_t = S_i|\lambda]$$

The algorithm is then:

- **Initialization:** Initially the machine may be in any one of the N states, and may output O_1 from any one of these. Since the probability of outputting the character is independent of being in that state, the probability of both events occuring, is the product:

$$\alpha_1(i) = \pi_i \cdot b_i(O_1)$$

- **Induction:** Now calculate α_{t+1} from α_t. The machine may go to state j at time-step $t+1$ from any one of the machine's states. So we need the probability for the machine to be in state j now, while having output $O_1 \cdots O_t$ in the previous states. Since these state sequences are mutually exclusive, the probability of either one of them occuring, is the sum of their individual probabilities. This is then multiplied by the probability of outputting O_{t+1} from the current state j:

$$\alpha_{t+1}(j) = \left(\sum_{i=1}^{N} \alpha_t(i) \cdot a_{ij} \right) b_j(O_{t+1})$$

- **Termination:** Since the machine may end up in any one (and only one) of its states after T time-steps, the probability of having output the sequence O is the sum over all of the forward variables for time-step T:

$$P[O|\lambda] = \sum_{i=1}^{N} \alpha_T(i) \tag{6}$$

The time-complexity of $O(TN^2)$ follows from noticing that the inductive step is repeated $O(T)$ times, and uses $O(N)$ operations for each state - of which there are N.

3.2 Scaled Forward Procedure

For each inductive step of the forward procedure, the forward variables will become smaller as the probabilities they are multiplied with, are generally less than 1. For practical use, the forward variables quickly underflow the dynamic range of the floating point unit.

The following method preserves the relationship between the individual probabilities for each time-step, and it can be thought of as *stretching* or *dragging* the forward variables back towards 1 by multiplying all of them by a common factor.

The **inductive** step in the algorithm of section 3.1 now uses the *scaled* forward variable $\hat{\alpha}_t$ for the previous time-step t, to calculate the *temporary* forward variable $\dot{\alpha}_{t+1}$ for the current time-step $t+1$:

$$\dot{\alpha}_{t+1}(j) = \left(\sum_{i=1}^{N} \hat{\alpha}_t(i) \cdot a_{ij} \right) b_j(O_{t+1})$$

Where the scaled forward variable is defined as a scaling of the temporary forward variable:

$$\hat{\alpha}_t(i) = c_t \cdot \dot{\alpha}_t(i) \tag{7}$$

With the initial $\dot{\alpha}$ equalling that of the non-scaled procedure: $\dot{\alpha}_1(i) = \alpha_1(i)$. The scale c_t is common to all states, and defined as the reciprocal sum of all $\dot{\alpha}_t(i)$ for the given time-step:

$$c_t = \frac{1}{\sum_{i=1}^{N} \dot{\alpha}_t(i)}$$

That is, the scaling factor ensures that all the $\hat{\alpha}_t(i)$'s for the given time-step add up to 1:

$$\sum_{j=1}^{N} \hat{\alpha}_t(j) = \sum_{j=1}^{N} (c_t \cdot \alpha_t(j)) = c_t \cdot \sum_{j=1}^{N} \alpha_t(j) = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)} \cdot \sum_{j=1}^{N} \alpha_t(j) = 1$$

The scaling factor c_t can therefore be said to normalize the sum $\sum_{i=1}^{N} \alpha_t(i)$, thus ensuring that the individual $\hat{\alpha}_t(i)$'s are kept within the proper dynamic range, although some of them will naturally still approach zero.

The forward variable is scaled for each time-step, so the **termination** sum $\sum_{i=1}^{N} \hat{\alpha}_T(i)$ naturally also equals 1. However, writing out this sum, we see that it is really just the sum of Eq.(6) multiplied by all of the scaling factors:

$$\sum_{i=1}^{N} \hat{\alpha}_T(i) = \cdots = \prod_{t=1}^{T} c_t \cdot \sum_{i=1}^{N} \alpha_T(i)$$

Again from Eq.(6), we have:

$$\prod_{t=1}^{T} c_t \cdot \sum_{i=1}^{N} \alpha_T(i) = \prod_{t=1}^{T} c_t \cdot P[O|\lambda]$$

And since this equals 1 and all c_t are non-zero, it can be rewritten as:

$$P[O|\lambda] = \frac{1}{\prod_{t=1}^{T} c_t}$$

Then taking the logarithm on both sides, and applying its appropriate mathematical laws, we get:

$$\log\left(P[O|\lambda]\right) = -\sum_{t=1}^{T} \log(c_t)$$

For convenience we denote this as $\log(P)$.

So we can calculate the logarithmic probability $\log(P)$ of observing the sequence O given the model λ, when using the scaled forward procedure. Fortunately this can still be used in direct comparison of how well one model performs over another, since the logarithm preserves order:

$$\begin{array}{c} a > b > 0 \\ \Updownarrow \\ \log(a) > \log(b) \end{array}$$

3.3 Number of States

In general one does not know the number of states of the stochastic system generating the training sequence O. Therefore we either need to decide the number of states, or devise a method to find it automatically.

Imagine the HMM has only a single state. If it is trained to comply optimally with the sequence O, the probability of that state outputting a character, is simply the number of times it occurs in O divided by the total number of observations T. That is, the sequential tendencies or patterns of O are not modelled in the single state HMM at all.

The other extreme is one in which there is a single state for each observation: $N = T$. For this HMM to comply optimally with O, there would only be one possible state sequence, with each state outputting only the correct character for that time-step.

This is known as *memorizing* and is the extremity of *overfitting* the model to the training set. The single state HMM then corresponds to *underfitting* [2].

One suggestion of algorithmically finding the number of states, would be to try various values of N. First choose some boundaries N_{min} and N_{max} to avoid gross under- and over-fitting. Then for example, iteratively bisect that range, calculate P for one value in each of these half-ranges and adjust the boundaries to the half-range that had the highest P. When $N_{\text{min}} = N_{\text{max}}$ this is hopefully[1] a sensible choice of N.

Of course, when using non-exhaustive search for N, one should always remember the one that maximized P so far, and not expect it to be the last number found by the search algorithm.

[1] Meaning that no experiments are carried out to validate this, and that some regularity on the relationship between number of states and P is needed for this to work.

3.4 Training- And Test-Sets

Traditionally in the training of *data mining* models, the data-set is split into two mutually exclusive training- and test-sets. The purpose is to avoid overfitting when continuously refining the model to match the training-set, but use the model that performed best on the test-set.

Since the degree of fitting in the HMM may be controlled by the number of states, it appears there is no need for such precautions. Nevertheless if there are several sequences available for the training, as is the case with speech-recognition, in which several persons have recorded their voice to train the HMM to recognize the *words* and not the particular sonic features of a single person's voice. Then the HMM may be trained continuously on a subset, the training-set, of all these recordings - meanwhile evaluating the HMM's performance on the other recordings, and then picking the model parameters λ that maximize performance on this test-set.

3.5 Meta-Heuristical Optimization of $P[O|\lambda]$

The number of probability values that define an ergodic model λ for a discrete alphabet V of size M, is:

$$|A| + |B| + |\pi| = N^2 + NM + N = N(N + M + 1)$$

with each probability belonging to $[0, 1] \subset \mathbb{R}$. Hence, the problem of maximizing P can be formulated as an optimization problem over the real-valued search-space $[0, 1]^{N(N+M+1)}$, and is therefore applicable to optimization by *Genetic Algorithms* [2] or other meta-heuristical optimization schemes, with each agent representing a model λ_i, and the *fitness* to be maximized being P (or equivalently $\log(P)$).

To optimize the HMM parameters, this implementation uses the *acPSO* [3] which is a variation of the *Particle Swarm Optimization* (PSO) scheme.

3.5.1 Normalizing Probabilities

This way of meta-heuristically updating the probabilities, generally does not preserve the consistencies required by Eqs.(3,4,5). One solution is to normalize the individual probabilities similarly to the forward variable scaling of section 3.2.

Assuming that the sum of the probabilities is non-zero, they can be normalized as in the following example where $\hat{\pi}_i$ denotes the normalized initial state probability π_i for the machine to start in state i:

$$\hat{\pi}_i = \frac{\pi_i}{\sum_{j=1}^{N} \pi_j}$$

And similar normalization is done for the transition and character output probabilities of Eqs.(1,2).

3.5.2 Pre-Emptive Fitness Evaluation

If the exact fitness of the agent is only needed in case of improvement over its own or the entire population's previous best fitness, the fitness evaluation may be aborted pre-emptively in the training of the HMM once it becomes worse.

This works because the probability for observing longer sequences is a non-increasing function - that is:

$$P[O_1 \cdots O_t|\lambda] \geq P[O_1 \cdots O_t\, O_{t+1}|\lambda]$$

So for the acPSO where a particle only uses its own as well as the swarm's previous best $\vec{gbest}$, if for some t we find that $P[O_1 \cdots O_t|\lambda]$ (or $\log(P[O_1 \cdots O_t, |\lambda])$ is less than the particle's own previous best, we stop calculating any further as the HMM model is not any better than what the particle has previously seen, and will therefore not cause any change to the swarm's behaviour.

Although this trick was devised for this particular application of the acPSO, it is generally applicable if the optimization scheme and the fitness evaluation allow it.

4 Implementation

Implementation is done in *Microsoft Visual .NET C++* and the source-code and *Windows*-executable can be found on `http://www.daimi.au.dk/~u971055/` under the filenames `hmm*.*`. There is no explicit error-handling.

The ergodic HMM is implemented in the `HMM`-class, which is then derived and implemented for integer-valued sequences in `IntegerHMM`, supporting observations from discrete and finite alphabets V.

The `HMM`-class derives from `LVectorNDim` which is the class representing a position in the real-valued search-space, and the model parameters λ are stored using methods of this class.

Since the acPSO was readily available for minimization problems, the fitness of the particles are merely taken to be $Fitness(\lambda) = -log(P[O|\lambda]) \in [0, \text{inf})$.

The executable file has a hard-coded seed for the random generator, but the user decides most other parameters: Number of states and characters, input- and output-sequence lengths, the swarm-velocity factor (so that particles are limitied in their movements across the search-space in a single step), the number of swarm-iterations, and number of swarm runs.

Because no real-world sequences were available, the implementation is tested with an artificially constructed observation sequence:

$$O_t = (t \bmod m) \bmod M \tag{8}$$

where the user inputs the modulo-value m. The output is then synthesized with the best model λ found over all of the acPSO runs.

5 Experimental Results

As this work is of an experimental nature, the reader is also encouraged to run the program with different parameters. The following provides an example of such a run. The parameters for the acPSO are deliberately chosen to be the same as in [3], but with the velocity factor decided by the user.

The effect of pre-emptive fitness evaluation is most evident with a high number of analysis observations, so that the fitness-track is displayed slowly. Then after the swarm disperses (happens for every 40.000 swarm-steps), the fitness track is displayed more quickly until the particles start to converge on $\vec{gbest}$, and the forward procedure again needs to be calculated for almost the entire observation sequence.

Figure 1 displays the 100-character long analysis sequence generated from Eq.(8) using a 4-character alphabet (e.g a numeration of the DNA-characters: $\{A, C, T, G\}$), and the modulo-value $m = 7$.

0 1 2 3 0 1 2 0 1 2 3 0 1 2 0 1 2 3 0 1 2 0 1 2 3 0 1 2 0 1 2 3 0 1 2
0 1 2 3 0 1 2 0 1 2 3 0 1 2 0 1 2 3 0 1 2 0 1 2 3 0 1 2 0 1 2 3 0 1 2
0 1 2 3 0 1 2 0 1 2 3 0 1 2 0 1

Figure 1: The 100-character long sequence generated from Eq.(8) with $m = 7$ and $M = 4$, used in the training of a HMM using acPSO.

This is then analyzed by HMM's having 5 states, with the acPSO having a velocity factor of 0.01 and doing 1000 iterations for each of 50 runs, and then using the best fitting HMM to synthesize the 200-character long output sequence in figure 2. Its best fitness was $log(P) = -19.41$ which is obtained in about half of the runs and corresponds to a probability of $P = 3.72e - 9$.

0 1 2 0 1 2 3 0 1 2 0 1 2 0 1 2 3 0 1 2 3 0 1 2 0 1 2 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2
3 0 1 2 0 1 2 3 0 1 2 3 0 1 2 0 1 2 3 0 1 2 3 0 1 2 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
0 1 2 3 0 1 2 0 1 2 0 1 2 3 0 1 2 0 1 2 3 0 1 2 3 0 1 2 0 1 2 3 0 1 2 3 0 1 2 3 0 1
2 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 0 1 2 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2
0 1 2 0 1 2 3 0 1 2 3 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 3 0 1

Figure 2: The 200-character long sequence generated from the 5 state HMM trained with acPSO optimization to mimic the sequence of figure 1.

If all probabilities in the HMM were equal - which corresponds to a single-state HMM with all output probabilities being equal - the probability for outputting a given sequence is M^{-N}, meaning 0.25^{-100} (or the log-likelihood of this, approximately -138) for these parameters. So, the acPSO finds significantly better HMM probabilites for this, albeit artificial, example. This is also underlined by the random sampling in the acPSO initialization, which does not get better than roughly $log(P) = -130$.

Preliminary experiments indicate that the acPSO is not always capable of finding known optimal HMM parameters. For example, if the period of the

test sequence is four (set m to be a non-zero multiple of four), and the HMM has four states, then the acPSO optimization does not always find the HMM that produces this periodical sequence. It appears that training the HMM with acPSO is susceptible to too many observations.

It further seems that too many states may also degrade the results. For example, the sequence in figure 1 does not always result in a perfectly trained HMM when it has 20 states. This may be a result of the acPSO not performing well on that many dimensions (500 in this case).

Some would argue though, that the probabilities should not be allowed to be 0 nor 1, e.g. $\pi_i \in (0, 1)$ instead of the usual $\pi_i \in [0, 1]$. A probability of zero or one means the behaviour is deterministic, but the physical system is assumed to be stochastic.

6 Conclusion

Although this document primarily describes the implications of multi-agent optimization of the HMM parameters λ, it would be interesting to compare its performance on real-world data with the traditional, but also iterative, *Baum-Welch Reestimation* algorithm. Experiments with other multi-agent optimization schemes and their parameters, could also prove interesting.

Furthermore, a method for pre-emptively aborting fitness calculations in particular kinds of multi-agent optimization was outlined and then adopted in the optimization of the probability $log(P[O|\lambda])$ of observing the sequence O given the Hidden Markov Model parameters λ.

References

[1] A Tutorial on Hidden Markov Models and
Selected Applications in Speech Recognition
Lawrence R. Rabiner
Proceedings of The IEEE, Vol. 77, No. 2, February 1989
`http://www.ai.mit.edu/~murphyk/Bayes/rabiner.pdf`

[2] Genetic Algorithms for Rule Discovery in Data Mining
Magnus Pedersen (971055)
Daimi, University of Aarhus, October 2003
`http://www.daimi.au.dk/~u971055/`

[3] Convulsive Particle Swarm Optimization Addendum
Magnus Pedersen (971055)
Daimi, University of Aarhus, September 2003
`http://www.daimi.au.dk/~u971055/`