Hidden Markov Models

Phil Blunsom pcbl@cs.mu.oz.au

August 19, 2004

Abstract

The Hidden Markov Model (HMM) is a popular statistical tool for modelling a wide range of time series data. In the context of natural language processing(NLP), HMMs have been applied with great success to problems such as part-of-speech tagging and noun-phrase chunking.

1 Introduction

The Hidden Markov Model(HMM) is a powerful statistical tool for modeling generative sequences that can be characterised by an underlying process generating an observable sequence. HMMs have found application in many areas interested in signal processing, and in particular speech processing, but have also been applied with success to low level NLP tasks such as part-of-speech tagging, phrase chunking, and extracting target information from documents. Andrei Markov gave his name to the mathematical theory of Markov processes in the early twentieth century[3], but it was Baum and his colleagues that developed the theory of HMMs in the 1960s[2].

Markov Processes Diagram 1 depicts an example of a Markov process. The model presented describes a simple model for a stock market index. The model has three states, *Bull*, *Bear* and *Even*, and three index observations *up*, *down*, *unchanged*. The model is a finite state automaton, with probabilistic transitions between states. Given a sequence of observations, example: *up-down-down* we can easily verify that the state sequence that produced those observations was: *Bull-Bear-Bear*, and the probability of the sequence is simply the product of the transitions, in this case $0.2 \times 0.3 \times 0.3$.

Hidden Markov Models Diagram 2 shows an example of how the previous model can be extended into a HMM. The new model now allows all observation symbols to be emitted from each state with a finite probability. This change makes the model much more expressive

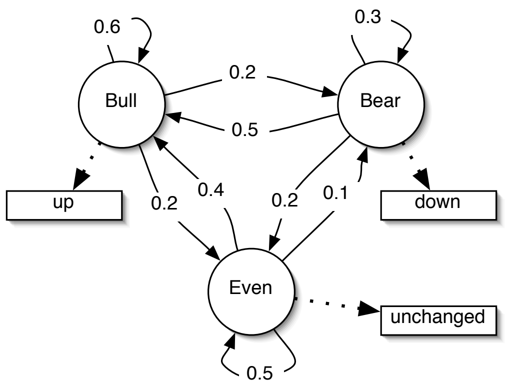

Figure 1: Markov process example[1]

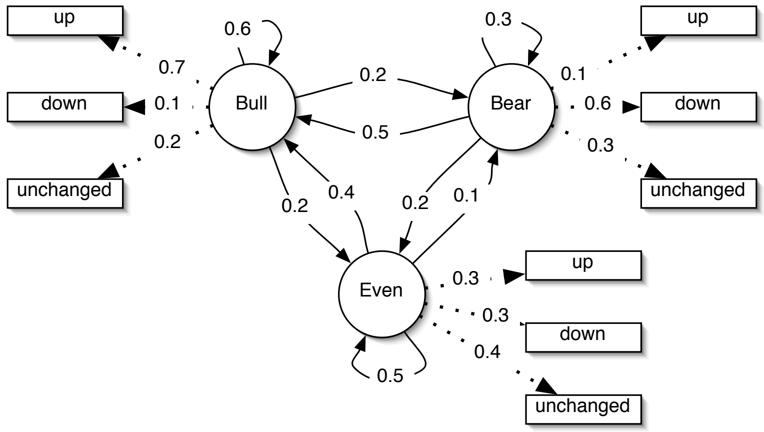

Figure 2: Hidden Markov model example[1]

and able to better represent our intuition, in this case, that a bull market would have both good days and bad days, but there would be more good ones. The key difference is that now if we have the observation sequence *up-down-down* then we cannot say exactly what state sequence produced these observations and thus the state sequence is 'hidden'. We can however calculate the probability that the model produced the sequence, as well as which state sequence was most likely to have produced the observations. The next three sections describe the common calculations that we would like to be able to perform on a HMM.

The formal definition of a HMM is as follows:

$$\lambda = (A, B, \pi) \tag{1}$$

S is our state alphabet set, and V is the observation alphabet set:

$$S = (s_1, s_2, \cdots, s_N) \tag{2}$$

$$V = (v_1, v_2, \cdots, v_M) \tag{3}$$

We define Q to be a fixed state sequence of length T, and corresponding observations O:

$$Q = q_1, q_2, \cdots, q_T \tag{4}$$

$$O = o_1, o_2, \cdots, o_T \tag{5}$$

A is a transition array, storing the probability of state j following state i. Note the state transition probabilities are independent of time:

$$A = [a_{ij}], a_{ij} = P(q_t = s_j | q_{t-1} = s_i). \tag{6}$$

B is the observation array, storing the probability of observation k being produced from the state j, independent of t:

$$B = [b_i(k)], b_i(k) = P(x_t = v_k | q_t = s_i). \tag{7}$$

π is the initial probability array:

$$\pi = [\pi_i], \pi_i = P(q_1 = s_i). \tag{8}$$

Two assumptions are made by the model. The first, called the Markov assumption, states that the current state is dependent only on the previous state, this represents the memory of the model:

$$P(q_t | q_1^{t-1}) = P(q_t | q_{t-1}) \tag{9}$$

The independence assumption states that the output observation at time t is dependent only on the current state, it is independent of previous observations and states:

$$P(o_t | o_1^{t-1}, q_1^t) = P(o_t | q_t) \tag{10}$$

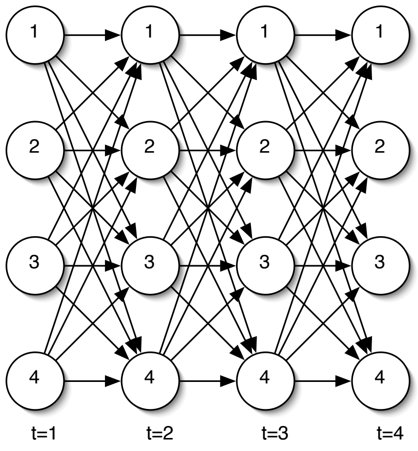

Figure 3: A trellis algorithm

2 Evaluation

Given a HMM, and a sequence of observations, we'd like to be able to compute $P(O|\lambda)$, the probability of the observation sequence given a model. This problem could be viewed as one of evaluating how well a model predicts a given observation sequence, and thus allow us to choose the most appropriate model from a set.

The probability of the observations O for a specific state sequence Q is:

$$P(O|Q,\lambda) = \prod_{t=1}^{T} P(o_t|q_t,\lambda) = b_{q_1}(o_1) \times b_{q_2}(o_2) \cdots b_{q_T}(o_T) \tag{11}$$

and the probability of the state sequence is:

$$P(Q|\lambda) = \pi_{q_1} a_{q_1 q_2} a_{q_2 q_3} \cdots a_{q_{T-1} q_T} \tag{12}$$

so we can calculate the probability of the observations given the model as:

$$P(O|\lambda) = \sum_{Q} P(O|Q,\lambda) P(Q|\lambda) = \sum_{q_1 \cdots q_T} \pi_{q_1} b_{q_1}(o_1) a_{q_1 q_2} b_{q_2}(o_2) \cdots a_{q_{T-1} q_T} b_{q_T}(o_T) \tag{13}$$

This result allows the evaluation of the probability of O, but to evaluate it directly would be exponential in T.

A better approach is to recognise that many redundant calculations would be made by directly evaluating equation 13, and therefore caching calculations can lead to reduced complexity. We implement the cache as a trellis of states at each time step, calculating the cached valued (called α) for each state as a sum over all states at the previous time step. α is the probability of the partial observation sequence $o_1, o_2 \cdots o_t$ and state s_i at time t. This can be visualised as in figure 3. We define the forward probability variable:

$$\alpha_t(i) = P(o_1 o_2 \cdots o_t, q_t = s_i|\lambda) \tag{14}$$

so if we work through the trellis filling in the values of α the sum of the final column of the trellis will equal the probability of the observation sequence. The algorithm for this process is called the forward algorithm and is as follows:

1. Initialisation:

$$\alpha_1(i) = \pi_i b_i(o_1), 1 \leq i \leq N. \tag{15}$$

2. Induction:

$$\alpha_{t+1}(j) = [\sum_{i=1}^{N} \alpha_t(i) a_{ij}] b_j(o_{t+1}), 1 \leq t \leq T-1, 1 \leq j \leq N. \tag{16}$$

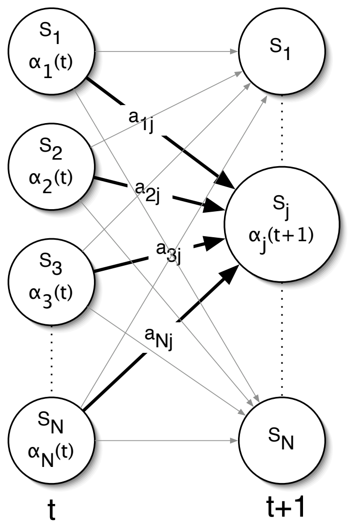

Figure 4: The induction step of the forward algorithm

3. Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \tag{17}$$

The induction step is the key to the forward algorithm and is depicted in figure 4. For each state s_j, $\alpha_j(t)$ stores the probability of arriving in that state having observed the observation sequence up until time t.

It is apparent that by caching α values the forward algorithm reduces the complexity of calculations involved to N^2T rather than $2TN^T$. We can also define an analogous backwards algorithm which is the exact reverse of the forwards algorithm with the backwards variable:

$$\beta_t(i) = P(o_{t+1}o_{t+2}\cdots o_T|q_t = s_i, \lambda) \tag{18}$$

as the probability of the partial observation sequence from $t+1$ to T, starting in state s_i.

3 Decoding

The aim of decoding is to discover the hidden state sequence that was most likely to have produced a given observation sequence. One solution to this problem is to use the Viterbi algorithm to find the single best state sequence for an observation sequence. The Viterbi algorithm is another trellis algorithm which is very similar to the forward algorithm, except that the transition probabilities are maximised at each step, instead of summed. First we define:

$$\delta_t(i) = \max_{q_1,q_2,\cdots,q_{t-1}} P(q_1q_2\cdots q_t = s_i, o_1, o_2\cdots o_t|\lambda) \tag{19}$$

as the probability of the most probable state path for the partial observation sequence.

The Viterbi algorithm and is as follows:

1. Initialisation:

$$\delta_1(i) = \pi_i b_i(o_1), 1 \leq i \leq N, \psi_1(i) = 0. \tag{20}$$

2. Recursion:

$$\delta_t(j) = \max_{1\leq i\leq N}[\delta_{t-1}(i)a_{ij}]b_j(o_t), 2 \leq t \leq T, 1 \leq j \leq N, \tag{21}$$

$$\psi_t(j) = \arg\max_{1\leq i\leq N}[\delta_{t-1}(i)a_{ij}], 2 \leq t \leq T, 1 \leq j \leq N. \tag{22}$$

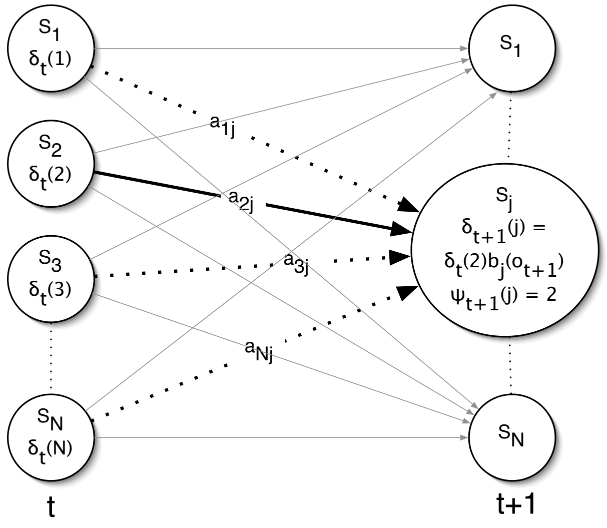

Figure 5: The recursion step of the viterbi algorithm

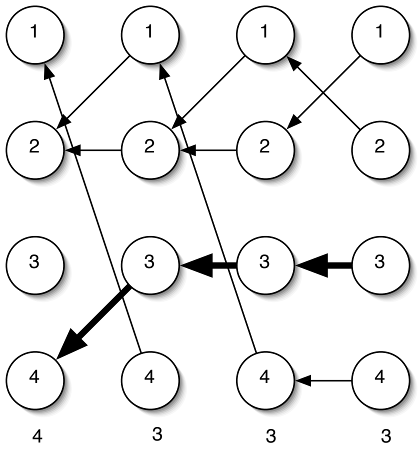

Figure 6: The backtracing step of the viterbi algorithm

3. Termination:

$$P^* = \max_{1 \leq i \leq N} [\delta_T(i)] \tag{23}$$

$$q_T^* = \arg \max_{1 \leq i \leq N} [\delta_T(i)]. \tag{24}$$

4. Optimal state sequence backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), t = T-1, T-2, \cdots, 1. \tag{25}$$

The recursion step is illustrated in figure 5. The main difference with the forward algorithm in the recursions step is that we are maximising, rather than summing, and storing the state that was chosen as the maximum for use as a backpointer. The backtracking step is shown in 6. The backtracking allows the best state sequence to be found from the back pointers stored in the recursion step, but it should be noted that there is no easy way to find the second best state sequence.

4 Learning

Given a set of examples from a process, we would like to be able to estimate the model parameters $\lambda = (A, B, \pi)$ that best describe that process. There are two standard approaches to this task, dependent on the form of the examples, which will be referred to here as supervised and unsupervised training. If the training examples contain both the inputs and outputs of a process, we can perform supervised training by equating inputs to observations, and outputs to states, but if only the inputs are provided in the training data then we must used unsupervised training to guess a model that may have produced those observations. In this section we will discuss the supervised approach to training, for a discussion of the Baum-Welch algorithm for unsupervised training see [5].

The easiest solution for creating a model λ is to have a large corpus of training examples, each annotated with the correct classification. The classic example for this approach is PoS tagging. We define two sets:

- $t_1 \cdots t_N$ is the set of tags, which we equate to the HMM state set $s_1 \cdots s_N$
- $w_1 \cdots w_M$ is the set of words, which we equate to the HMM observation set $v_1 \cdots v_M$

so with this model we frame part-of-speech tagging as decoding the most probable hidden state sequence of PoS tags given an observation sequence of words. To determine the model parameters λ, we can use maximum likelihood estimates(MLE) from a corpus containing sentences tagged with their correct PoS tags. For the transition matrix we use:

$$a_{ij} = P(t_i|t_j) = \frac{Count(t_i, t_j)}{Count(t_i)} \tag{26}$$

where $Count(t_i, t_j)$ is the number of times t_j followed t_i in the training data. For the observation matrix:

$$b_j(k) = P(w_k|t_j) = \frac{Count(w_k, t_j)}{Count(t_j)} \tag{27}$$

where $Count(w_k, t_j)$ is the number of times w_k was tagged t_j in the training data. And lastly the initial probability distribution:

$$\pi_i = P(q_1 = t_i) = \frac{Count(q_1 = t_i)}{Count(q_1)} \tag{28}$$

In practice when estimating a HMM from counts it is normally necessary to apply smoothing in order to avoid zero counts and improve the performance of the model on data not appearing in the training set.

5 Multi-Dimensional Feature Space

A limitation of the model described is that observations are assumed to be single dimensional features, but many tasks are most naturally modelled using a multi-dimensional feature space. One solution to this problem is to use a multinomial model that assumes the features of the observations are independent [4]:

$$v_k = (f_1, \cdots, f_N) \tag{29}$$

$$P(v_k|s_j) = \prod_{j=1}^{N} P(f_j|s_j) \tag{30}$$

This model is easy to implement and computationally simple, but obviously many features one might want to use are not independent. For many NLP systems it has been found that flawed Baysian independence assumptions can still be very effective.

6 Implementing HMMs

When implementing a HMM, floating-point underflow is a significant problem. It is apparent that when applying the Viterbi or forward algorithms to long sequences the extremely small probability values that would result could underflow on most machines. We solve this problem differently for each algorithm:

Viterbi underflow As the Viterbi algorithms only multiplies probabilities, a simple solution to underflow is to log all the probability values and then add values instead of multiply. In fact if all the values in the model matrices (A, B, π) are stored logged, then at runtime only addition operations are needed.

forward algorithm underflow The forward algorithm sums probability values, so it is not a viable solution to log the values in order to avoid underflow. The most common solution to this problem is to use scaling coefficients that keep the probability values in the dynamic range of the machine, and that are dependent only on t. The coefficient c_t is defined as:

$$c_t = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)} \tag{31}$$

and thus the new scaled value for α becomes:

$$\hat{\alpha}_t(i) = c_t \times \alpha_t(i) = \frac{\alpha_t(i)}{\sum_{i=1}^{N} \alpha_t(i)} \tag{32}$$

a similar coefficient can be computed for $\hat{\beta}_t(i)$.

References

[1] Huang et. al. *Spoken Language Processing.* Prentice Hall PTR.

[2] L. Baum et. al. A maximization technique occuring in the statistical analysis of probablistic functions of markov chains. *Annals of Mathematical Statistics*, 41:164–171, 1970.

[3] A. Markov. An example of statistical investigation in the text of eugene onyegin, illustrating coupling of tests in chains. Proceedings of the Academy of Sciences of St. Petersburg, 1913.

[4] A. McCallum and K. Nigram. A comparison of event models for naive bayes classification. In AAAI-98 Workshop on Learning for Text Categorization, 1998.

[5] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of IEEE, 1989.