Hidden Markov Models and Dynamic Programming

Jonathon Read

October 14, 2011

1 Last week: stochastic part-of-speech tagging

Last week we reviewed parts-of-speech, which are linguistic categories of words. These categories are defined in terms of syntactic or morphological behaviour. Parts-of-speech for English traditionally include:

Nouns are concrete or abstract entity;

Pronouns substitute for nouns;

Adjective modify nouns;

Verbs are actions or states of being;

Adverbs modify adjectives, verbs or other adverbs;

Prepositions express relations; and

Conjuctions connect expressions.

Linguists typically distinguish between many more types however. For example, the tagset you will use in the assignment exercises is The Penn Treebank Tagset, which contains 45 parts-of-speech.

Automatic part-of-speech tagging can be achieved using a large set of heuristics, but defining such heuristics is a very time-consuming task. Instead we can use a stochastic method, which, given a sequence of observed words w_1^n we want to determine the most probable corresponding sequence of classes $\hat{t}_1^n$.

$$\hat{t}_1^n = \arg\max_{t_1^n} P\left(t_1^n | w_1^n\right)$$

To make this computation tractable we can refactor using Bayes' rule, while making two assumptions:

1. the probability of a word is only dependent on its own part-of-speech tag; and
2. the probability of a tag is only dependent on the previous tag.

Which leaves us computing:

$$\hat{t}_1^t \approx \arg\max_{t_1^n} \prod_i^n P(w_i|t_i)P(t_i|t_{i-1})$$

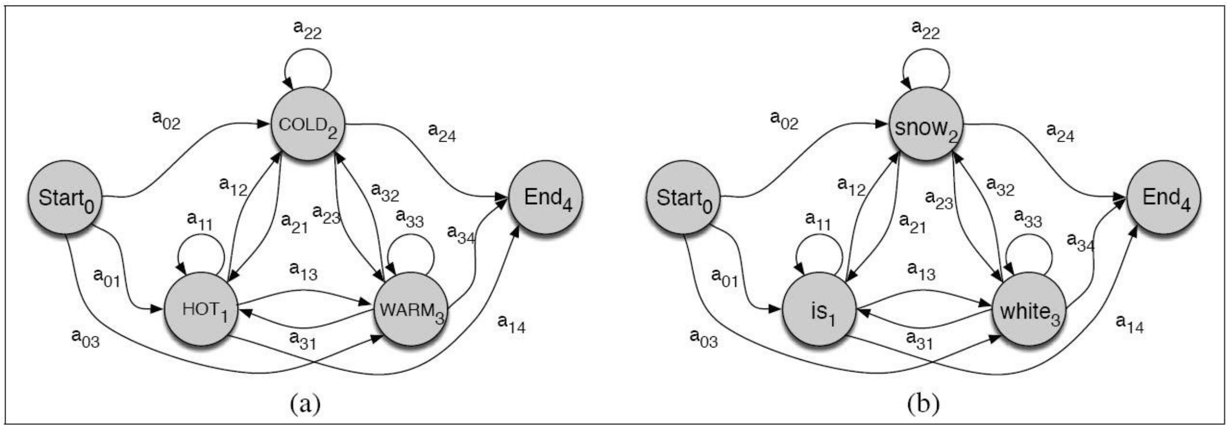
We can then estimate the probabilities $P(w_i|t_i)$ and $P(t_i|t_{i-1})$ using maximum likelihood estimates from frequency counts in a corpus.

We should note though, that the search space for a sequence of tags given a corresponding sequence of words is often very large, being exponential in terms of the length of the sequence and the ambiguity of the words.

2 Markov chains

We have a problem with tractability, but can make the computation more efficient. Each of the possible tag sequences for a given word sequence will share subsequences with others, which means many calculations will be repeated. In this week's lecture we'll be covering some Dynamic Programming algorithms. These break down the calculations such that they are only performed once. Before that, though, we need a formal specification of the approach, which is known as a hidden Markov model (HMM).

We begin with something somewhat simpler—Markov chains. Markov chains are extensions of the finite state automata we covered a few weeks ago, being defined by a set of states and a set of transitions between the states. The transitions of Markov chains are weighted, such that each transition is associated with a probability, and the probability of all transitions leaving a state sums to 1.



$Q = q_1 q_2 \ldots q_N$	a set of N **states**
q_0, q_F	special **start** and **final** states
$A = \begin{pmatrix} a_{01} & \cdots & a_{0N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix}$	a **transition probability matrix**, where a_{ij} the probability of moving from state i to state j

Markov chains are a generalisation of the bigram models we've covered previously, and embody the Markov assumption, that the probability of a particular state only depends on the previous state.

$$\textbf{Markov Assumption}: P(q_i|q_1 \ldots q_{i-1}) \approx P(q_i|q_{i-1})$$

3 Hidden Markov models

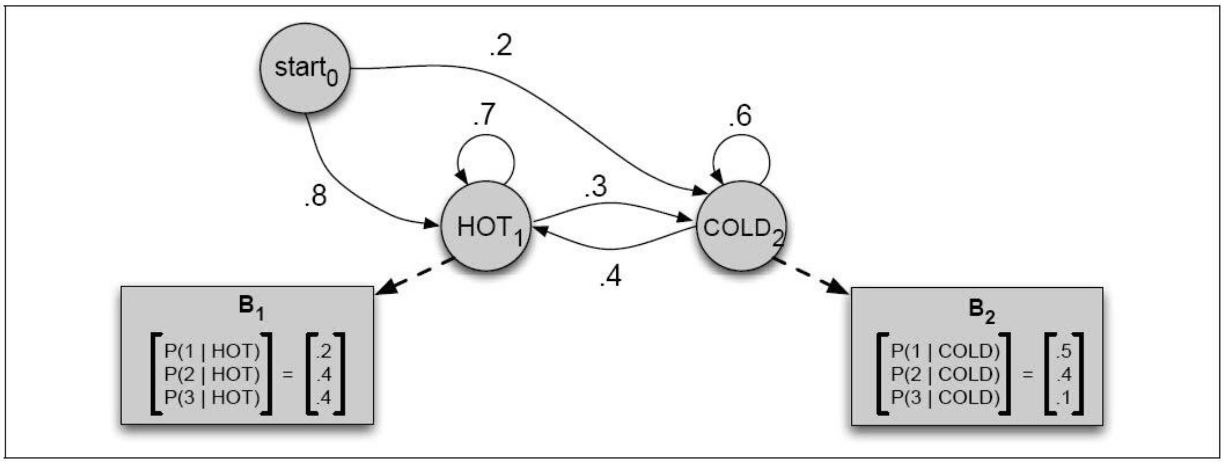
Markov chains are useful when computing the probability of observable sequences. However, in many applications in natural language processing we are interested in events that are not observable. For example in part-of-speech tagging we infer correct tags given the observed sequence of words. We call the part-of-speech tags **hidden** because they are not observed. Another example is that of speech recognition—where we observe a sequence of acoustic events and have to infer the corresponding hidden sequence of words.

A **hidden Markov Model** (HMM) lets us process both observed events (such as the words we see in input) and their associated hidden events that we think of as causal factors (such as the part-of-speech tags).

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
q_0, q_F	special **start** and **final** states
$A = \begin{pmatrix} a_{01} & \cdots & a_{0N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix}$	a **transition probability matrix**, where a_{ij} the probability of moving from state i to state j
$O = o_1 o_2 \ldots o_T$	a sequence of **observations**
$B = b_i(o_t)$	a sequence of **observation likelihoods**

To demonstrate these models we'll use Jason Eisner's example, as presented by Jurafsky and Martin. We'll imagine we are studying global warming but can not find records of the weather in Baltimore during Summer 2007. However we do have Jason Eisner's diary, which lists how many ice creams he ate every day that summer. We'll simplify the task by saying we're only interested in whether the weather was hot or cold, and use a hidden Markov model to estimate the temperature each day. So the Eisner task is as follows:

> Given a sequence of observations O, each observation an integer corresponding to the number of ice-creams eaten on a given day, figure out the correct 'hidden' sequence Q of weather states (H or C) which caused Jason to eat the ice cream.



Note that for simplicity this diagram does not depict the final state, but instead allows for both states to be final states.

4 Computing likelihoods: The Forward algorithm

How can we determine the probability of an observed sequence, for example the sequence of Jason eating 3, 1 and 3 ice creams? Or, more formally:

> Given a HMM $\lambda = (A, B)$ and an observation sequence O, determine the likelihood $P(O|\lambda)$

Let's start by assuming that we have actually observed both O and Q. The joint probability $P(O, Q)$ can be computed as:

$$P(O, Q) = P(O|Q)P(Q) = \prod_{i=1}^{T} P(o_i|q_i) \prod_{i=1}^{T} P(q_i|q_{i-1})$$

At first glance, this doesn't help us much because we don't actually know the state sequence Q. However, we can compute the sum over all possible state sequences:

$$P(O) = \sum_{Q} P(O, Q) = \sum_{Q} P(O|Q)P(Q)$$

In our ice cream problem, this becomes the sum over all eight 3-event sequences:

$$P(3\ 1\ 3) = P(3\ 1\ 3, \text{cold cold cold}) + P(3\ 1\ 3, \text{cold cold hot}) + P(3\ 1\ 3, \text{hot hot cold}) + \ldots$$

However, for non-trival problems with N hidden states and T observations, there are N^T possible state sequences. There is exponential computational complexity $O(N^T T)$.

Instead we can use the Forward algorithm, which employs dynamic programming to reduce the complexity to $O(N^2 T)$. The basic idea is to store and resuse the results of partial computations. This is achieved by filling in the cells of a trellis structure.

A trellis is a graph in which the nodes are ordered into verticle slices representing time steps. Each node is connected to at least one node in a previous time step and at least one node in a subsequent time step.

A cell $\alpha_t(j)$ in the trellis stores the probability of being in state q_j after seeing the first t observations.

$$\alpha_t(j) = P(o_1 \ldots o_t, q_t = j)$$

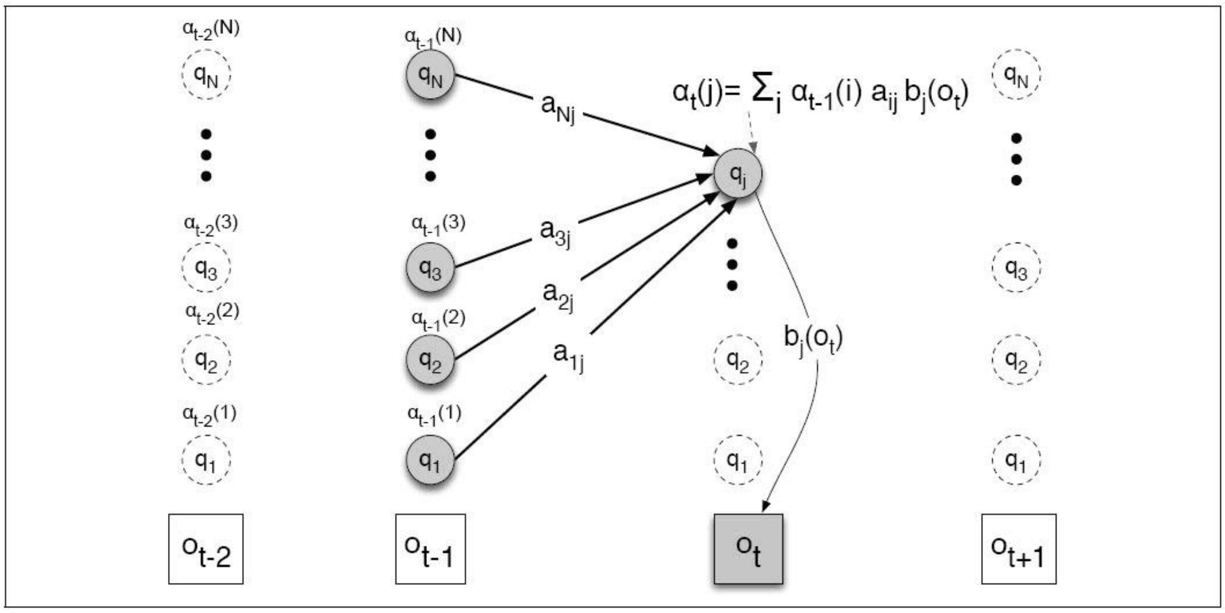
($q_t = j$ should be read as "the probability that the tth state in the sequence is state j). The value of each cell is computed by summing over the probabilities of all possible paths leading to that cell:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t)$$

Where the three factors that are multiplied are:

$\alpha_{t-1}(i)$	the **previous forward path probability**
a_{ij}	the **transition probability** from the previous state q_i to the current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation o_t given state j

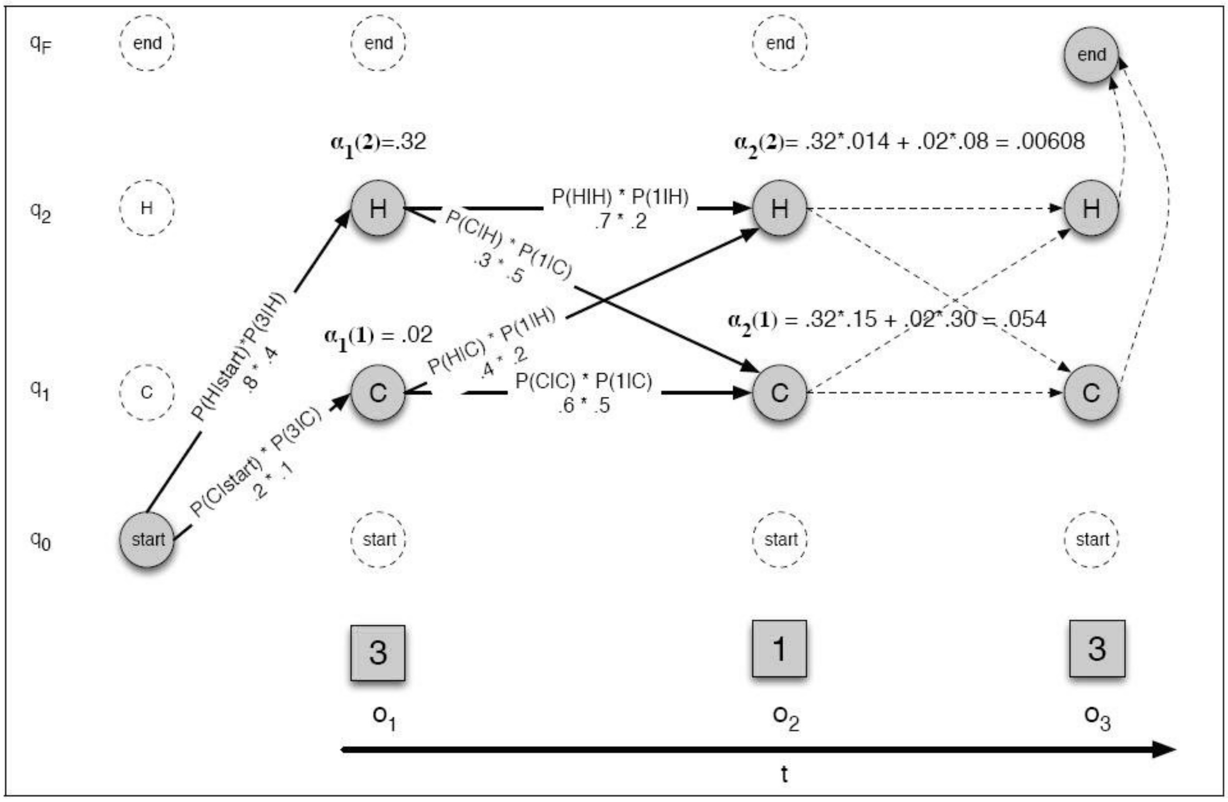
Below is a visualisation of the computation of a single element $\alpha_t(i)$ in the trellis by summing the previous values at α_{t-1} weighted by their transition probabilities from a and the observation probability $b_j(o_t)$.



Here is the pseudocode for the Forward algorithm (from Jurafsky and Martin):

```
**Input**: *observations* of length $T$, *state-graph* of length $N$
**Output**: *forward-probability*
create a probability matrix $forward[N+2,T]$
**foreach** *state s from 1 to N* **do**
    $forward[s,1] \leftarrow a_{o,s} \times b_s(o_1)$
**end**
**foreach** *time step t from 2 to T* **do**
    **foreach** *state s from 1 to N* **do**
        $forward[s,t] \leftarrow \sum_{s'=1}^{N} forward[s,t-1] \times a_{s',s} \times b_s(o_t)$
    **end**
**end**
$forward[q_F,T] \leftarrow \sum_{s=1}^{N} forward[s,T] \times a_{s,qF}$
**return** $forward[q_F,T]$
```

Here is an example forward trellis for calculating the likelihood for the ice cream events 3 1 3. Hidden states are in circles, observations in squares. The computation of each cell follows the equation above.



5 Decoding hidden states: The Viterbi algorithm

In any model that contains hidden variables (such as HMMs), the task of determining the sequence of variables is known as decoding. For example, suppose we have a sequence of ice cream events 3 1 3, and want the find the most likely weather sequence, or more formally:

> Given a HMM $\lambda = (A, B)$ and a sequence of observations $O = o_1, o_2, \dots, o_T$, find the most probable corresponding sequence of hidden states $Q = q_1, q_2, \dots, q_T$.

This entails the same procedure as last week for finding the most probable sequence of part-of-speech, given the observed words. However, just as for likelihood, this approach is not tractable due to the exponentially large number of state sequences. Instead we can employ another dynamic programming algorithm, Viterbi.

Let each cell of the Viterbi trellis $v_t(j)$ represent the probability of our HMM being in state q_j after seeing the previous sub-sequence of observations $o_1 \dots, o_t$ and passing through the most probable sequence of states $q_1, \dots, q_{t-1}$.

$$v_t(j) = \max_{(q_1,\dots,q_{t-1})} P(o_1 \dots o_t, q_1 \dots q_{t-1}, q_t = j)$$

We move forward through the trellis, updating each cell based on the values of the previously computed cells:

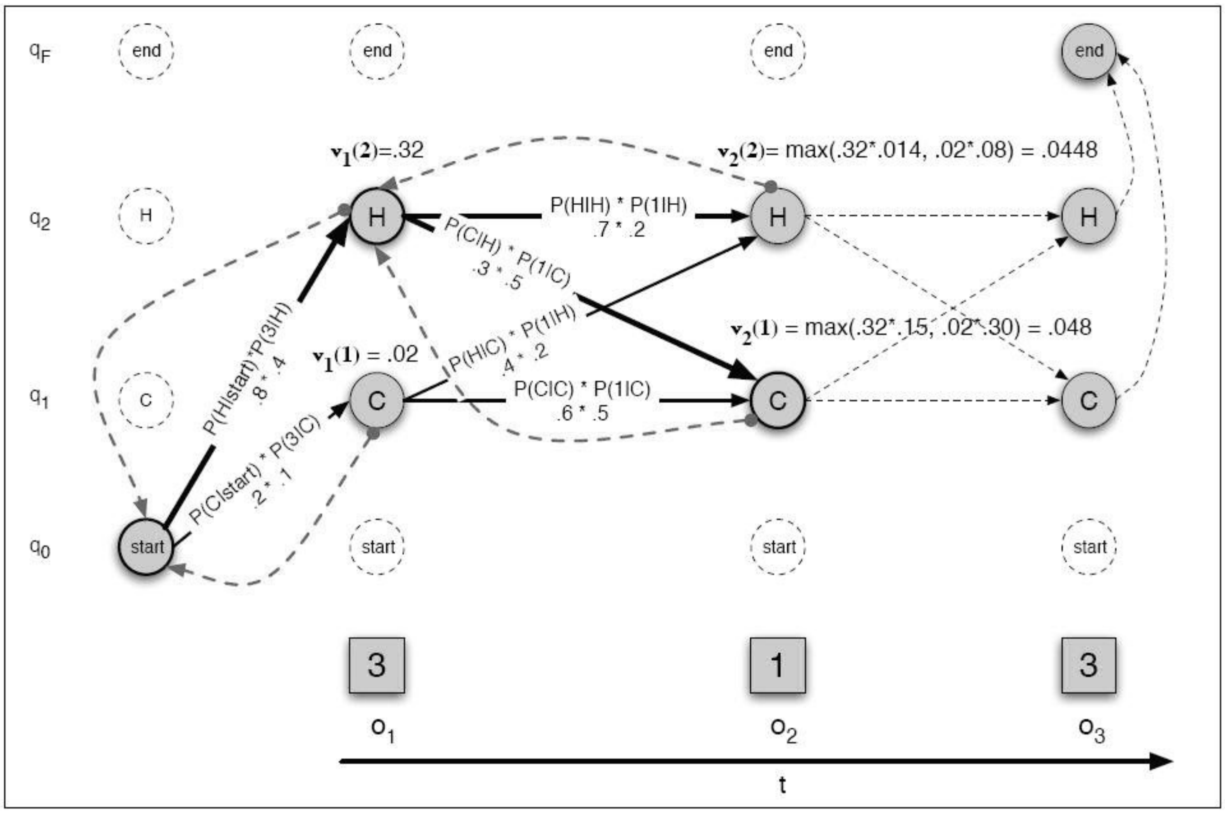
$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t)$$

Which is fairly similar to the Forward algorithm, except that each cell stores the maximum probability (instead of the sum. However, we also want to determine the state sequence that corresponds to the most probable path, so we need to keep track of the route through the trellis. To accomplish this we let $\beta_t(j)$ denote the backtrace pointer from state q_j at time t, such that it points back to the previous state q_{t-1} which resulted in the most probable path to the current state.

Here is Jurafsky and Martin's pseudocode for the Viterbi algorithm:

```
Input: observations of length $T$, state-graph of length $N$
Output: best-path
create a path probability matrix $viterbi[N+2, T]$
create a path backpointer matrix $backpointer[N+2, T]$
foreach state s from 1 to N do
    $forward[s,1] \leftarrow a_{o,s} \times b_s(o_1)$
    $backpointer[s,1] \leftarrow 0$
end
foreach time step t from 2 to T do
    foreach state s from 1 to N do
        $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s', t-1] \times a_{s',s} \times b_s(o_t)$
        $backpointer[s,t] \leftarrow \arg\max_{s'=1}^{N} viterbi[s', t-1] \times a_{s',s}$
    end
end
$viterbi[q_F, T] \leftarrow \max_{s=1}^{N} viterbi[s,T] \times a_{s,q_F}$
$backpointer[q_F, T] \leftarrow \arg\max_{s=1}^{N} viterbi[s,T] \times a_{s,q_F}$
return the backtrace path by following backpointers to states back in time from $backpointer[q_F, T]$
```

And this is the Viterbi trellis for computing the best path for the ice cream events 3 1 3. At each stage in the iteration we keep a backpointer (indicated by the dashed lines) to the best path that led to the current state.



A further note on efficiency when multiplying probabilities—recall the property of logarithms, that $\log xy = \log x + \log y$, which can be usefully employed here as sums are much faster to compute than products. For example:

$$\max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t) = \max_{i=1}^{N} \log v_{t-1}(i) + \log a_{ij} + \log b_j(o_t)$$

This also has the helpful side affect of avoiding computing the product of very small floating point numbers, which can often result in underflow errors.

6 Evaluation

Using a manually labeled test set—a **gold standard**—we can compute the **accuracy** of our model:

$$\text{accuracy} = \frac{\text{tags correct}}{\text{observations}}$$

We'll often compare this to some points of reference. An **upper-bound** can indicate the best performance expected—for example how well humans would do on the same task. A **baseline** typically represents the current best performance, for example that of some other system. In novel tasks, we might consider baselines generated from random selection, or perhaps picking tags based on the most likely tag for a given word.