Day 2

Sequence Models

In this class, we relax the assumption that datapoints are independently and identically distributed (i.i.d.) by moving to a scenario of *structured prediction*, where the inputs are assumed to have temporal or spacial dependencies. We start by considering sequential models, which correspond to a *chain structure*: for instance, the words in a sentence. In this lecture, we will use part-of-speech tagging as our example task.

The problem setting is the following: let $\mathcal{X} = \{\bar{\mathbf{x}}^1, \ldots, \bar{\mathbf{x}}^D\}$ be a training set of independent and identically-distributed random variables. In this work $\bar{\mathbf{x}}^d$ (for notation simplicity we will drop the superscript d when considering an isolated example) corresponds to a sentence in natural language and decomposes as a sequence of observations of length N: $\bar{\mathbf{x}} = \mathbf{x}_1 \ldots \mathbf{x}_N$. Each $\mathbf{x}_n$ is a discrete random variable (a *word*), taking a value v from a finite vocabulary $\mathcal{V}$. Each $\bar{\mathbf{x}}$ has an unknown hidden structure $\bar{\mathbf{y}}$ that we want to predict. The structures are sequences $\bar{\mathbf{y}} = \mathbf{y}_1 \ldots \mathbf{y}_N$ of the same length N as the observations. Each hidden state $\mathbf{y}_n$ is a discrete random variable and can take a value y from a discrete vocabulary $\mathbf{Y}$.

We focus on the well known Hidden Markov Model (HMM) on section 2.1, where we describe how to estimate its parameters from labeled data 2.2. We will then move to how to find the most likely hidden sequence (decoding) given an observation sequence and a parameter set 2.3. This section will explain the required inference algorithms (Viterbi and Forward-Backward) for sequence models. These inference algorithms will be fundamental for the rest of this lecture, as well as for the next lecture on *discriminative* training of sequence models. Finally, section 2.4 will describe the task of part-of-speech tagging, and how HMM are suitable for this task.

Notation	
$\mathcal{X}$	training set
D	number of training examples
$\bar{\mathbf{x}} = \mathbf{x}_1 \ldots \mathbf{x}_N$	observation sequence
N	size of the observation sequence
$\mathbf{x}_i$	observation at position i in the sequence
$\mathcal{V}$	observation values set
$\lvert\mathcal{V}\rvert$	number of distinct observation types
v_i	particular observation, $i \in \lvert\mathcal{V}\rvert$
$\bar{\mathbf{y}} = \mathbf{y}_1 \ldots \mathbf{y}_N$	hidden state sequence
$\mathbf{y}_i$	hidden state at position i in the sequence
$\mathbf{Y}$	hidden states value set
$\lvert\mathbf{Y}\rvert$	number of distinct hidden value types
y_i	particular hidden value, $i \in \lvert\mathbf{Y}\rvert$

Table 2.1: General notation used in this class

2.1 Hidden Markov Models

The Hidden Markov Model (HMM) is one of the most common sequence probabilistic models, and has been applied to a wide variety of tasks. More generally, an HMM is a particular instance of a chain directed probabilistic graphical model, or a Bayesian network. In a Bayesian network, every random variable is represented as a node in a graph, and the edges in the graph are directed and represent probabilistic dependencies between the random variables. For an HMM, the random variables are divided into two sets, the *observed* variables, in our case $\mathbf{x}$, and the *hidden* variables, in our case $\mathbf{y}$. In the HMM terminology, the observed variables are called *observations*, and the hidden variables are called *states*.

As you may find out in today's lab session, implementing the inference routines of the HMM can be challenging, and debugging can become hard in large datasets. We thus start with a small and very simple (very unrealistic!) example. The idea is that you may compute the desired quantities by hand and check if your implementation yields the correct result.

Example 2.1 *Consider a person, which is only interested in four activities:*

- *walking in the park (w);*
- *shopping (s);*
- *cleaning his apartment (c);*
- *playing tennis (t).*

The choice of what to do on a given day is determined exclusively by the weather at that day. The weather can be either rainy *(r) or* sunny *(s). Now, suppose that we observe what the person did on a sequence of days; can we use that information to predict the weather each of those days? To tackle this problem, we assume that the weather behaves as a discrete Markov Chain: the weather on a given day is independent of everything else* given the weather on the previous day. *The entire system is that of a hidden Markov model (HMM).*

Assume we are asked to predict what was the weather on two different sequences of days given the following observations: "walk walk shop clean" and "clean walk tennis walk". This will be our test set.

To train our model, we will be given access to three different sequences of days, containing both the activities and the weather on those days, namely: "walk/rainy walk/sunny shop/sunny clean/sunny", "walk/rainy walk/rainy shop/rainy clean/rainy " and "shop/rainy walk/rainy shop/rainy clean/rainy". This will be our training set.

Figure 2.1 show the HMM model the first sequence of our simple example (the notation is summarized in Table 2.2).

HMM Example	
$\bar{\mathbf{x}}$	observed sentence "w w s c"
$N = 4$	observation length
i, j	positions in the sentence: $i, j \in \{1 \dots N\}$
$\mathcal{V} = \{w, s, c, t\}$	observation vocabulary
p, q	indexes into the vocabulary $p, q \in \lvert\mathcal{V}\rvert$
$\mathbf{x}_i = v_q, \mathbf{x}_j = v_p$	observation at position $\mathbf{x}_i$ ($\mathbf{x}_j$) has value v_q (v_p)
$\mathbf{Y} = \{r, s\}$	hidden values vocabulary
l, m	indexes into the hidden values vocabulary
$\mathbf{y}_i = y_l, \mathbf{y}_j = y_m$	state at position $\mathbf{y}_i$ ($\mathbf{y}_j$) has hidden value y_l (y_m)

Table 2.2: HMM notation for the running example.

A first order HMM model has the following independence assumptions over the joint distribution $p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})$:

- **Independence of previous states.** The probability of being in a given state y_l at position $\mathbf{y}_i$ only depends on the state y_m of the previous position $\mathbf{y}_{i-1}$, that is $p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m, \mathbf{y}_{i-2} \dots \mathbf{y}_1) = p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m)$, defining a first order Markov chain.[1]

- **Homogeneous transition.** The probability of making a transition from state y_l to state y_m is independent of the particular position in the sen-

[1]The order of the Markov chain depends on the number of previous positions taken into account. The remainder of the exposition can be easily extend to higher order HMM, giving the model more expressiveness, but making inference harder.

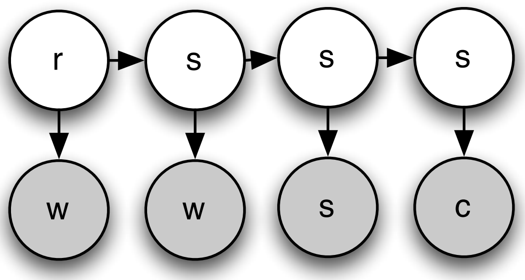

Figure 2.1: HMM structure, for the simple running example.

tence, for all $i, j \in \{1, \ldots, N\}$: $p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m) = p_\theta(\mathbf{y}_j = y_l \mid \mathbf{y}_{j-1} = y_m)$, so $p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m) = p_\theta(y_l \mid y_m)$.

- **Observation independence.** The probability of observing v_q at position i is fully determined by the state at that position, that is $p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l)$, and this probability is independent of the particular position, that is $p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l) = p_\theta(v_q \mid y_l)$.

These conditional independence assumptions are crucial to allow efficient inference, as will be described. We also need to define a *start probability*, the probability of starting at state y_l. Furthermore, when dealing with text, it is usual to break the homogeneous transition for the last position, and model the final transitions as independent parameters. The four probability distributions that define the HMM model are summarized in Table 2.3. For each one of them we will use a short notation to simplify the exposition.

HMM distributions		
Name	probability distribution	short notation
initial probability	$p_\theta(\mathbf{y}_1 = y_l)$	π_l
final probability	$p_\theta(\mathbf{y}_T = y_l \mid \mathbf{y}_{T-1} = y_m)$	$f_{m,l}$
transition probability	$p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m)$	$a_{m,l}$
observation probability	$p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l)$	$b_l(\mathbf{x}_i)$

Table 2.3: HMM probability distributions.

The joint distribution can be expressed as:

$$p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \pi_{\mathbf{y}_1} b_{\mathbf{y}_1}(\mathbf{x}_1)[\prod_{i=2}^{N-1} a_{\mathbf{y}_{i-1},\mathbf{y}_i} b_{\mathbf{y}_i}(\mathbf{x}_i)] f_{\mathbf{y}_{N-1},\mathbf{y}_N} b_{\mathbf{y}_N}(\mathbf{x}_N), \tag{2.1}$$

short notation	probability distribution	—parameters—	constraint
π_j	$p_\theta(\mathbf{y}_1 = y_j)$	$\lvert\mathbf{Y}\rvert$ - 1	$\sum_{y\in\mathbf{Y}} \pi_j = 1;$
$a_{l,m}$	$p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m)$	$(\lvert\mathbf{Y}\rvert - 1)^2$	$\sum_{y_l\in\mathbf{Y}} a_{m,l} = 1;$
$f_{l,m}$	$p_\theta(\mathbf{y}_N = y_l \mid \mathbf{y}_{N-1} = y_m)$	$(\lvert\mathbf{Y}\rvert - 1)^2$	$\sum_{y_l\in\mathbf{Y}} t_{m,l} = 1;$
$b_q(l)$	$p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l)$	$(\lvert\mathcal{V}\rvert - 1)^{\lvert\mathbf{Y}\rvert}$	$\sum_{v_q\in\mathcal{V}} b_q(l) = 1.$

Table 2.4: Multinomial parametrization of the HMM distributions.

which for the example from Figure 2.1 is:

$$p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \pi_r b_r(''w'') a_{r,s} b_s(''w'') a_{s,s} b_s(''s'') f_{s,s} b_s(''c''). \tag{2.2}$$

In the next section we turn our attention to estimating the different probabilities distributions of the model π_l, $a_{m,l}$, $f_{m,l}$ and $b_l(\mathbf{x}_i)$.

Exercise 2.1 *Load the simple sequence dataset: From the ipython command line (Note start ipython from the* code *directory), create a simple sequence object and look at the training and test set.*

```
 In[]: run readers/simple_sequence.py
 In[]: simple = SimpleSequence()
 In[]: simple.train
Out[]: [w/r w/s s/s c/s , w/r w/r s/r c/s , w/s s/s s/s c/s ]
 In[]: simple.test
Out[]: [w/r w/s s/s c/s , c/s w/s t/s w/s ]
```

2.2 Finding the Maximum Likelihood Parameters

So far we have not committed to any form for the probability distributions, π_l, $a_{m,l}$,$f_{m,l}$ and $b_l(\mathbf{x}_i)$. In both applications addressed in this class, both the observations and the hidden variables are discrete. The most common approach is to model each of these probability distributions as multinomial distributions, summarized in Table 2.4.

We will refer to the set of all parameters as θ. The HMM model is trained to maximize the Log Likelihood of the data. Given a dataset $\mathcal{D}$ the objective being optimized is:

$$\arg\max_\theta \sum_{\bar{\mathbf{x}},\bar{\mathbf{y}}\in\mathcal{D}} \log p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) \tag{2.3}$$

Multinomial distributions are attractive for several reasons: first of all they are easy to implement; secondly the estimation of the maximum likelihood parameters has a simple closed form. They are just normalized counts of events that occur in the corpus (the same as the Naïve Bayes from previous class).

Given a labeled corpus the estimation process consists in counting how many times each event occurs in the corpus and normalize the counts to form proper probability distributions. Let us define the following quantities, called sufficient statistics, that represent the counts of each event in the corpus:

$$\textbf{Initial Counts}: \quad ic(y_l) = \sum_{\bar{\mathbf{x}},\bar{\mathbf{y}} \in \mathcal{D}} \mathbf{1}(\mathbf{y}_1 = y_j); \tag{2.4}$$

$$\textbf{Final Counts}: \quad fc(y_l, y_m) = \sum_{\bar{\mathbf{x}},\bar{\mathbf{y}} \in \mathcal{D}} \mathbf{1}(\mathbf{y}_N = y_l \mid \mathbf{y}_{N-1} = y_m); \tag{2.5}$$

$$\textbf{Transition Counts}: \quad tc(y_l, y_m) = \sum_{\bar{\mathbf{x}},\bar{\mathbf{y}} \in \mathcal{D}} \sum_{i=1}^{N-1} \mathbf{1}(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m); \tag{2.6}$$

$$\textbf{State Counts}: \quad sc(v_q, y_l) = \sum_{\bar{\mathbf{x}},\bar{\mathbf{y}} \in \mathcal{D}} \sum_{i=1}^{N} \mathbf{1}(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l) \tag{2.7}$$

Note that **1** is an indicator function that has the value one when the particular event happens, and zero otherwise. In words the previous equations amount to do a pass over the training corpus and count how many times each even occurs: e.g. the word "w" appears with state "s", or state "s" follows another state "s", or state "s" begins the sentence.

After computing the counts, one can perform some sanity checks to make sure the implementation is correct. Summing over all entries of each count table we should observe the following:

- **Initial Counts** - Should sum to the number of sentences.
- **Final Counts** - Should sum to the number of sentences.
- **Transition Counts** - Should sum to the number of tokens minus 2 times the number of sentences. Note that there are N-1 edges for each sentence, and the last edge is being accounted by the final transitions. So this leaves us with N-2 edges per sentence, where N is the number of tokens in that sentence.
- **Observation Counts** - Should sum to the number of tokens.

Exercise 2.2 *Convince yourself of the sanity check described above.*

Implement a method to sanity check the counts table.

```
def sanity_check_counts(self,seq_list):
```

Collect the counts from a supervised corpus using method collect_counts_from_corpus(self,sequence_list) *and make sure your sanity check method is correct.*

```
In []: run sequences/hmm.py
In []: hmm = HMM(simple)
In []: hmm.collect_counts_from_corpus(simple)
In []: hmm.sanity_check_counts(simple)
Init Counts match
Final Counts match
Transition Counts match
Observations Counts match
```

Using the sufficient statistics (counts) the parameter estimates are:

$$\hat{\pi}_l = \frac{ic(y_l)}{\sum_{y_m \in \mathbf{Y}} ic(y_m)} \tag{2.8}$$

$$\hat{f}_{l,m} = \frac{fc(y_l, y_m)}{\sum_{y_m \in \mathbf{Y}} fc(y_l, y_m)} \tag{2.9}$$

$$\hat{a}_{l,m} = \frac{tc(y_l, y_m)}{\sum_{y_m \in \mathbf{Y}} tc(y_l, y_m)} \tag{2.10}$$

$$\hat{b}_l(v_q = o) = \frac{sc(v_q, y_l)}{\sum_{v_p \in \mathcal{V}} sc(v_p, y_l)} \tag{2.11}$$

Exercise 2.3 *Implement a function that estimates the maximum likelihood estimates for the parameters given the corpus in the class HMM. The function header is in the hmm.py file.*

```
def train_supervised(self,sequence_list):
```

Run this function given the simple dataset and compare the results with ones given (If the results are the same then you are ready to go).

```
In[]:  run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In []: hmm.init_probs
Out[]:
array([[ 0.66666667],
       [ 0.33333333]])
In []: hmm.transition_probs
Out[]:
array([[ 0.66666667,  0.        ],
       [ 0.33333333,  1.        ]])
```

```
In []: hmm.final_probs
Out[]:
array([[ 0.,  0.],
       [ 1.,  1.]])
In []: hmm.observation_probs
Out[]:
array([[ 0.75 ,  0.25 ],
       [ 0.25 ,  0.375],
       [ 0.   ,  0.375],
       [ 0.   ,  0.   ]])
```

2.3 Finding the most likely sequence - Decoding

Given the learned parameters and an observation $\bar{\mathbf{x}}$ we want to find the best hidden state sequence $\bar{\mathbf{y}}^*$. There are several ways to define what we mean by the best $\bar{\mathbf{y}}^*$, for instance, the best assignment to each hidden variable $\mathbf{y}_i$ and the best assignment to the sequence $\bar{\mathbf{y}}$ as a whole. The first way, normally called **Posterior decoding** consists in picking the highest state posterior for each position in the sequence:

$$\bar{\mathbf{y}}^* = \arg\max_{\mathbf{y}_1 \cdots \mathbf{y}_N} \gamma_i(\mathbf{y}_i), \tag{2.12}$$

where $\gamma_i(\mathbf{y}_i)$ is the posterior probability $P(\mathbf{y}_i|\bar{\mathbf{x}})$. This method does not guarantee that the sequence $\bar{\mathbf{y}}^*$ is a valid sequence of the model. For instance there might be a transition probability between two of the best node posteriors with probability zero.

The second approach called **Viterbi decoding** consists in picking the best global hidden state sequence $\bar{\mathbf{y}}$.

$$\bar{\mathbf{y}}^* = \arg\max_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}). \tag{2.13}$$

Both approaches rely on dynamic programming making use of the independence assumptions of the HMM model over an alternative representation of the HMM called a Trellis (Figure 2.2).

This representation unfolds all possible states for each position and makes explicit the independence assumption, each position only depends on the previous position. Figure 2.2 shows the trellis for the particular example in Figure 2.1.

Each column represents a position, in the sentence as is associated with its observation, and each row represents a possible state. For the algorithms described in the following section it would be useful to define a new re-parametrization

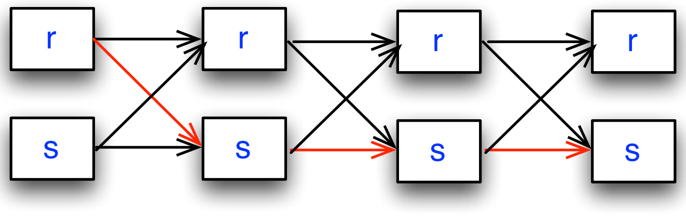

Figure 2.2: Trellis representation of the HMM in Figure 2.1, for the observation sequence "w w s c", where each hidden variable can take the values r, s.

of the model, in terms of node potentials $\phi_n(l)$ (the value of the boxes in Figure 2.2) and edge potentials $\phi_n(l, m)$ (the value of the links in Figure 2.2). There is a direct mapping from the HMM parameters to this new representation:

- *Edge Potentials* - correspond to the transition parameters, with the exception of the edges into the last position that correspond to the final transition parameters.
- *Node Potentials* - correspond to the observation parameters for a given state and the observation at that position. The first position is the exception and corresponds to the product between the observation parameters for that state and the observation in that position with the initial parameters for that state.

Although this re-parametrization simplifies the exposition, and will be used in this lectures, it is not necessarily very practical since we will be reproducing several values (for instance the transition parameters for each position). Given the node potentials and edge potentials the probability of the sequence form the example is:

$$p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \phi_1(r)\phi_1(r,s)\phi_2(s)\phi_2(s,s)\phi_3(s)\phi_3(s,s)\phi_4(s). \tag{2.14}$$

Exercise 2.4 *Convince yourself that the Equation 2.14 is equivalent to equation 2.2*

Implement a function that builds the node and edge potentials for a given sequence. The function head is in the hmm.py *file:*

```
def build_potentials(self,sequence):
```

Run this function for the first training sequence from the simple dataset and compare the results given (If the results are the same then you are ready to go).

```
In[]:  run sequences/hmm.py
```

```
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In[]: node_potentials,edge_potentials = hmm.build_potentials(
    simple.train.seq_list[0])
In []: node_potentials
Out[]:
array([[ 0.5       ,  0.75      ,  0.25      ,  0.        ],
       [ 0.08333333,  0.25      ,  0.375     ,  0.375     ]])

In []: edge_potentials
Out[]:
array([[[ 0.66666667,  0.66666667,  0.        ],
        [ 0.33333333,  0.33333333,  1.        ]],

       [[ 0.        ,  0.        ,  0.        ],
        [ 1.        ,  1.        ,  1.        ]]])
```

2.3.1 Posterior Decoding

Posterior decoding consists in picking the highest state posterior for each position in the sequence:

$$\bar{\mathbf{y}}^* = \underset{\mathbf{y}_1\cdots\mathbf{y}_N}{\arg\max}\, \gamma_i(\mathbf{y}_i). \tag{2.15}$$

For a given sequence, the **sequence posterior distribution** is the probability of a particular hidden state sequence given that we have observed a particular sentence. Moreover, we will be interested in two other posteriors distributions: the **state posterior distribution**, corresponding to the probability of being in a given state in a certain position given the observed sentence; the **transition posterior distribution**, which is the probability of making a particular transition, from position i to $i+1$ given the observed sentence.

$$\textbf{Sequence Posterior}: \quad p_\theta(\bar{\mathbf{y}} \mid \bar{\mathbf{x}}) = \frac{p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})}{p_\theta(\bar{\mathbf{x}})}; \tag{2.16}$$

$$\textbf{State Posterior}: \quad \gamma_i(y_l) = p_\theta(\mathbf{y}_i = y_l \mid \bar{\mathbf{x}}); \tag{2.17}$$

$$\textbf{Transition Posterior}: \quad \xi_i(y_l, y_m) = p_\theta(\mathbf{y}_i = y_l, \mathbf{y}_{i+1} = y_m \mid \bar{\mathbf{x}}). \tag{2.18}$$

To compute the posteriors a first step is to be able to compute the likelihood of the sequence $p_\theta(\bar{\mathbf{x}})$, which corresponds to summing the probability of all possible hidden state sequences.

$$\textbf{Likelihood}: \quad p_\theta(\bar{\mathbf{x}}) = \sum_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}). \tag{2.19}$$

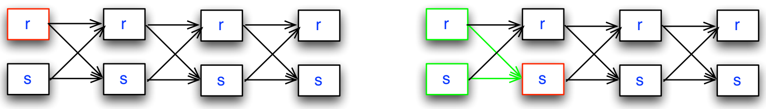

Figure 2.3: Forward trellis for the first sentence of the training data at position 1 (left) and at position 2 (right)

The number of possible hidden state sequences is exponential in the length of the sentence ($|\mathbf{Y}|^N$), which makes summing over all of them hard. In this particular small example there are $2^4 = 16$ such paths and we can actually enumerate them explicitly and calculate their probability using Equation 2.14. But this is as far as it goes, for part-of-speech induction with a small tagset of 12 tags and a medium size sentence of length 10, there are $12^{10} = 61917364224$ such paths. Yet, we must be able to compute this sum (sum over $\bar{\mathbf{y}}$) to compute the above likelihood formula, this is called the inference problem. For sequence models , there is a well know dynamic programming algorithm, the **Forward Backward** algorithm (FB), that allows the computation to be performed in linear time, by making use of the independence assumptions.

The FB algorithm relies on the independence of previous states assumption, which is illustrated in the trellis view by only having arrows between consecutive states. The FB algorithm defines two auxiliary probabilities, the forward probability and the backward probability.

$$\textbf{Forward Probability}: \quad \alpha_i(y_l) = p_\theta(\mathbf{y}_i = y_l, \mathbf{x}_1 \dots \mathbf{x}_i) \tag{2.20}$$

The forward probability represents the probability that in position i we are in state $\mathbf{y}_i = y_l$ and that we have observed $\bar{\mathbf{x}}$ up to that position. The forward probability is defined by the following recurrence (we will use the marginals parametrization of the model):

$$\alpha_1(y_l) = \phi_1(l) \tag{2.21}$$

$$\alpha_i(y_l) = \left[\sum_{y_m \in \mathbf{Y}} \phi_{(i-1)}(m,l)\alpha_{i-1}(y_m)\right] \phi_i(l) \tag{2.22}$$

At position 1, the probability of being in state "r" corresponding to being in state "r" and observing word "w" is just the node marginal for that position $\phi_1(r) = \alpha_1(r)$ (see Figure 2.3 left). At position 2 the probability of being in state "s" and observing the sequence of words "w w" corresponds to the sum of all possible ways of reaching position 2 in state "s" namely: "rs" or "ss" (see Figure 2.3 right). The probability of the first is $\phi_1(r)\phi_1(r,s)\phi_2(s)$ and the second is $\phi_1(s)\phi_1(s,s)\phi_2(s)$, so:

$$\begin{aligned}
\alpha_2(s) &= \phi_1(r)\phi_1(r,s)\phi_2(s) + \phi_1(s)\phi_1(s,s)\phi_2(s)\\
\alpha_2(s) &= [\phi_1(r)\phi_1(r,s)\phi_2(s) + \phi_1(s)\phi_1(s,s)]\,\phi_2(s)\\
\alpha_2(s) &= \sum_{\mathbf{y}\in\mathbf{Y}} [\phi_1(\mathbf{y})\phi_1(\mathbf{y},s)\phi_2(s)]\\
\alpha_2(s) &= \sum_{\mathbf{y}\in\mathbf{Y}} [\alpha_1(\mathbf{y})\phi_1(\mathbf{y},s)\phi_2(s)]
\end{aligned}$$

Using the forward trellis one can compute the likelihood by summing over all possible hidden states for the last position.

$$p_\theta(\bar{\mathbf{x}}) = \sum_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}},\bar{\mathbf{y}}) = \sum_{y\in\mathbf{Y}} \alpha_N(y). \qquad (2.23)$$

Although the forward probability is enough to calculate the likelihood of a given sequence, we will also need the backward probability to calculate the node and edge posteriors. The backward probability, represents the probability of observing $\bar{\mathbf{x}}$ from position $i+1$ up to N, given that at position i we are at state $\mathbf{y}_i = y_l$:

$$\textbf{Backward Probability}: \quad \beta_i(y_l) = p_\theta(\mathbf{x}_{i+1}...\mathbf{x}_N|\mathbf{y}_i = y_l). \qquad (2.24)$$

The backward recurrence is given by:

$$\begin{aligned}
\beta_N(y_l) &= 1 && (2.25)\\
\beta_i(y_l) &= \sum_{y_m\in\mathbf{Y}} \phi_i(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m). && (2.26)
\end{aligned}$$

The backward probability is similar to the forward probability, but operates in the inverse direction. At position N there are no more observations, and the backward probability is set to 1. At position i the probability of having observed the future and being in state y_l, is given by the sum for all possible states of the probability of having transitioned from position i in state y_l to position $i+1$ with state y_m and observed $\mathbf{x}_{i+1}$ at time $i+1$ and the future given by $\beta_{i+1}(\mathbf{y}_{i+1} = y_m)$.

With the FB probability one can compute the likelihood of a given sentence using any position in the sentence.

$$p_\theta(\bar{\mathbf{x}}) = \sum_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}},\bar{\mathbf{y}}) = \forall i \sum_{y\in\mathbf{Y}} \alpha_i(y)\beta_i(y). \qquad (2.27)$$

Note that for time N, $\beta_N(y) = 1$ and we get back to equation 2.23. Although redundant, this fact is useful when implementing an HMM as a sanity check that the computations are being performed correctly, since one can compare the

Algorithm 7 Forward Backward algorithm

```
1:  input: sentence $\bar{\mathbf{x}}$, parameters $\theta$
2:  Forward pass: Compute the forward probabilities
3:  Initialization
4:  for $y_l \in \mathbf{Y}$ do
5:      $\alpha_1(y_l) = \phi_1(y_l)$
6:  end for
7:  for $i = 2$ to $N$ do
8:      for $y_l \in \mathbf{Y}$ do
9:          $\alpha_i(y_l) = \left[\sum_{m \in \mathbf{Y}} \phi_{(i-1)}(m,l)\alpha_{i-1}(y_m)\right] \phi_i(l)$
10:     end for
11: end for
12: Backward pass: Compute the backward probabilities
13: Initialization
14: for $y_l \in \mathbf{Y}$ do
15:     $\beta_N(y_l) = 1$
16: end for
17: for $i = N-1$ to 1 do
18:     $\beta_i(y_l) = \sum_{y_m \in \mathbf{Y}} \phi_i(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m)$.
19: end for
20: output: The forward and backward probabilities $\alpha$ and $\beta$
```

likelihood at each position that should be the same. The FB algorithm may fail for long sequences since the nested multiplication of numbers smaller than 1 may easily become smaller than the machine precision. To avoid that problem, Rabiner (1989) presents a scaled version of the FB algorithm that avoids this problem.

Algorithm 7 shows the pseudo code for the forward backward algorithm.

Exercise 2.5 *Given the implementation of the forward pass of the forward backward algorithm in the file* forward_backward.py *implement the backward pass.*

```
def forward_backward(node_potentials,edge_potentials):
    H,N = node_potentials.shape
    forward = np.zeros([H,N],dtype=float)
    backward = np.zeros([H,N],dtype=float)
    forward[:,0] = node_potentials[:,0]
    ## Forward loop
    for pos in xrange(1,N):
        for current_state in xrange(H):
            for prev_state in xrange(H):
                forward_v = forward[prev_state,pos-1]
```

```
                    trans_v = edge_potentials[prev_state,
                        current_state,pos-1]
                    prob = forward_v*trans_v
                    forward[current_state,pos] += prob
                forward[current_state,pos] *= node_potentials[
                    current_state,pos]
        ## Backward loop
            ## Your code
        return forward,backward
```

Use the provided function that makes use of Equation 2.27 to make sure your implementation is correct:

```
In[]:  run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In[]: forward,backward =  hmm.forward_backward(simple.train.
    seq_list[0])
In []: sanity_check_forward_backward(forward,backward)
Out[]:
array([[ 0.03613281],
       [ 0.03613281],
       [ 0.03613281],
       [ 0.03613281]])
```

Moreover, given the forward and backward probabilities one can compute both the state and transition posteriors.

$$\textbf{State Posterior}: \quad \gamma_i(y_l) = p_\theta(\mathbf{y}_i = y_l \mid \bar{\mathbf{x}}) = \frac{\alpha_i(y_l)\beta_i(y_l)}{p_\theta(\bar{\mathbf{x}})}; \qquad (2.28)$$

$$\textbf{Transition Posterior}: \quad \xi_i(y_l, y_m) = p_\theta(\mathbf{y}_i = y_l, \mathbf{y}_{i+1} = y_m \mid \bar{\mathbf{x}}) = \frac{\alpha_i(y_l)\phi_i(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m)}{p_\theta(\bar{\mathbf{x}})}. \qquad (2.29)$$

A graphical representation of these posteriors is illustrated in figure 2.4. On the left it is shown that $\alpha_i(y_l)\beta_i(y_l)$ returns the sum of all paths that contain the state y_i, weighted by $p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})$, and on the right we can see that $\alpha_i(y_l)\phi_i(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m)$ returns the same for all paths containing the edge from y_l to y_m. Thus, these posteriors can be seen as the ratio of the number of paths that contain the given state or transition (weighted by $p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})$) and the number of possible paths in the graph $p_\theta(\bar{\mathbf{x}})$. As an practical example, given that the person perform the sequence of actions "w w s c", we want to know the probability of having been

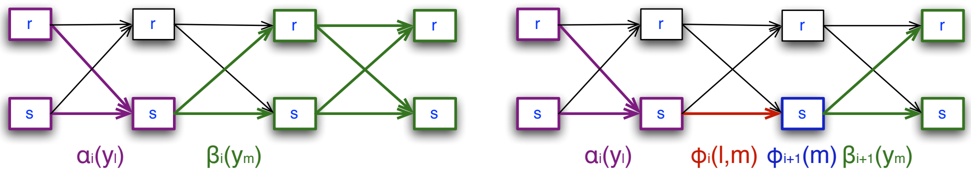

Figure 2.4: A graphical representation of the components in the state and transition posteriors.

Algorithm 8 Posterior Decoding algorithm

```
1: input: The forward and backward probabilities $\alpha$ and $\beta$.
2: Compute Likelihood: Compute the likelihood of the sentence
3: $L = 0$
4: for $y_l \in \mathbf{Y}$ do
5:   $p_\theta(\bar{\mathbf{x}}) = p_\theta(\bar{\mathbf{x}}) + \alpha_N(y_l)$
6: end for
7: $\hat{\mathbf{y}} = []$
8: for $i = 1$ to $N$ do
9:   $max = 0$
10:  for $y_l \in \mathbf{Y}$ do
11:    $\gamma_i(y_l) = \frac{\alpha_i(y_l)\beta_i(y_l)}{p_\theta(\bar{\mathbf{x}})}$
12:    if $\gamma_i(y_l) > max$ then
13:      $max = \gamma_i(y_l)$
14:      $\hat{y}_i = y_l$
15:    end if
16:  end for
17: end for
18: output: the posterior path $\hat{\mathbf{y}}$
```

raining in the second day. The state posterior probability for this event can be seen as the probability that the sequence of actions “walk walk shop clean” was generated by a sequence of weathers and where it was raining in the second day. In this case, the possible sequences would be ”r r r r”, ”s r r r”, ”r r s r”, ”r r r s”, ”s r s r”, ”s r r s”, ”r r s s”, and ”s r s s”.

Using the node posteriors we are ready to perform posterior decoding. Algorithm 8 shown the posterior decoding algorithm.

Exercise 2.6 *Given the node and edge posterior formulas 2.17,2.18 and the forward and backward formulas 2.20,2.24 convince yourself that formulas 2.28,2.29 are correct.*

Compute the node posteriors for the first training sentence, and look at the output. Can you come up with a sanity check for calculating the posteriors (Hint: Note that the node posteriors are a proper probability distribution).

```
In[]:  run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In [15]: node_posteriors = hmm.get_node_posteriors(simple.train
    .seq_list[0])

In [16]: node_posteriors
Out[16]:
array([[ 0.91891892,  0.75675676,  0.43243243,  0.        ],
       [ 0.08108108,  0.24324324,  0.56756757,  1.        ]])
```

Implement the posterior decode method using the method you tried previously.

```
def posterior_decode(self,seq):
```

Run the posterior decode on the first test sequence, and evaluate it.

```
In[]:  run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In [19]: y_pred = hmm.posterior_decode(simple.test.seq_list[0])
In [20]: w/r w/r s/s c/s
Out[20]: array([0, 0, 1, 1])
In [21]: simple.test.seq_list[0]
Out[21]: w/r w/s s/s c/s
```

Do the same for the second test sentence:

```
In [22]: y_pred = hmm.posterior_decode(simple.test.seq_list[1])
In [23]: y_pred
Out[23]: c/r w/r t/r w/r
In [24]: simple.test.seq_list[1]
Out[24]: c/s w/s t/s w/s
```

What is wrong? Observe the observations for that sentence. In fact the observation t was never seen at training time, so the probability for it will be zero (no matter what state) and make all possible state sequences have zero probability (the value 0 is just the default. As seen in the previous lecture, this is a problem with generative models, that can be corrected using smoothing (there are other options).

Change your train method to add smoothing:

```
def train_supervised(self,sequence_list, smoothing):
```

Repeat the last exercise with the new parameters. What do you observe?

Note that if you use smoothing when training you need to account for that in the counts sanity checks you did in the previous exercise.

Exercise 2.7 *Change the function you just created* def sanity_check_counts(self,seq_list): *to account for smoothing. Make sure it works properly.*

```
In []: run sequences/hmm.py
In []: hmm = HMM(posc)
In []: hmm.collect_counts_from_corpus(posc.train,smoothing=0.1)
In []: hmm.sanity_check_counts(posc.train,smoothing=0.1)
Init Counts match
Final Counts match
Transition Counts match
Observations Counts match
```

2.3.2 Viterbi Decoding

Viterbi decoding consists in picking the best global hidden state sequence $\bar{\mathbf{y}}$.

$$\bar{\mathbf{y}}^* = \arg\max_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}). \tag{2.30}$$

The viterbi algorithm is very similar to the forward procedure of the FB algorithm, making use of the same trellis structure to efficiently represent and use all the exponential number of sequences. In fact the only difference from the forward algorithm is in the recursion 2.22 where instead of summing over all possible hidden states, we take their maximum.

$$\textbf{Viterbi} \quad \delta_i(y_l) = \arg\max_{y_1 \dots y_i} p_\theta(\mathbf{y}_i = y_l, \mathbf{x}_1 \dots \mathbf{x}_i) \tag{2.31}$$

The viterbi trellis represents the path with maximum probability in position i when we are in state $\mathbf{y}_i = y_l$ and that we have observed $\bar{\mathbf{x}}$ up to that position. The viterbi algorithm is defined by the following recurrence (we will use the marginals parametrization of the model):

$$\delta_1(y_l) = \phi_1(l) \tag{2.32}$$

$$\delta_i(y_l) = \left[\max_{y_1 \dots y_i} \phi_{(i-1)}(m, l)\delta_{i-1}(y_m)\right]\phi_i(l) \tag{2.33}$$

$$\psi_i(y_l) = \left[\arg\max_{y_1 \dots y_i}\right] \tag{2.34}$$

Algorithm 9 shows the pseudo code for the Viterbi algorithm.

Algorithm 9 Viterbi algorithm

```
1: input: sentence $\bar{\mathbf{x}}$, parameters $\theta$
2: Forward pass: compute the maximum paths for every end state
3: Initialization
4: for $y_l \in \mathbf{Y}$ do
5:     $\delta_1(y_l) = \phi_1(y_l)$
6: end for
7: for $i = 2$ to $N$ do
8:     for $y_l \in \mathbf{Y}$ do
9:         $\delta_i(y_l) = \left[\max_{m \in \mathbf{Y}} \phi_{(i-1)}(m,l)\delta_{i-1}(y_m)\right] \phi_i(l)$
10:        $\psi_i(y_l) = m$
11:    end for
12: end for
13: Backward pass: Build the most likely path
14: $\hat{\mathbf{y}} = []$
15: $\hat{y}_N = \arg\max_{y_m \in |\mathbf{Y}|} \delta_N(y_l)$
16: for $i = N$ to $2$ do
17:     $\hat{y}_i = \psi_{i+1}(y_{i+1})$
18: end for
19: output: the viterbi path $\hat{\mathbf{y}}$
```

Exercise 2.8 *Implement a method for performing viterbi decoding.*

```
        def viterbi_decode(self,seq):
```

Test your method on both test sentences and compare the results with the ones given.

```
In []: y_pred = hmm.viterbi_decode(simple.test.seq_list[0])
In []: y_pred
Out[]: w/r w/r s/r c/s
In []: y_pred = hmm.viterbi_decode(simple.test.seq_list[1])
In []: y_pred
Out[]: c/r w/r t/r w/r
```

2.4 Part-of-Speech Tagging (POS)

Part of speech tagging is probably one the most common NLP tasks. The task is to assign each word with a grammatical category, i.e Noun, Verb, Adjective,

... . In English, using the Penn Treebank (ptb) (Marcus et al., 1993) the current state of the art for part of speech tagging is around the 97% for a variety of methods [2].

In the rest of this class we will use a subset of the ptb corpus, but instead of using the original 45 tags we will use a reduced tag set of 12 tags, to make the algorithms faster for the class. In this task, $\bar{\mathbf{x}}$ is a sentence and $\bar{\mathbf{y}}$ is the sequence of possible PoS tags.

The first step is to load the pos corpus, we will start by loading 1000 sentences for training and 200 sentences both for development and testing, and train the HMM model.

```
In []: run readers/pos_corpus.py
In []: posc = PostagCorpus("en",max_sent_len=15,train_sents=
    1000,dev_sents=200,test_sents=200)
In []: hmm = HMM(posc)
In []: hmm.train_supervised(posc.train)
Warning: invalid value encountered in divide
```

Note the warning "Warning: invalid value encountered in divide" again due to the lack of smoothing. This is because we are trying to estimate the probability of words that were never seen during training, so the counts are zero and we are trying to do 0/0.

Look at the transitions probabilities of the trained model (see Figure 2.5) and see if they match your intuition about the English language (e.g. adjectives tend to come before nouns).

Exercise 2.9 *Test the model using both posterior decoding and viterbi decoding on both the train and test set, using the methods in class HMM:*

```
viterbi_decode_corpus(self,seq_list)
posterior_decode_corpus(self,seq_list)
evaluate(self,seq_list,predictions)
```

What do you observe. Remake the previous exercise but now train the HMM using smoothing. Try different values and report the results on the train and development set (1,0.1,0.001,0.0001). (Use function pick_best_smoothing*).*

```
In []: hmm.pick_best_smoothing(posc.train,posc.dev,[0,0.1,0.01,
    1])
Warning: invalid value encountered in divide
Smoothing 0.000000 --  Train Set Accuracy: Posterior Decode 0.
    985, Viterbi Decode: 0.985
```

[2]See ACL state of the art wiki

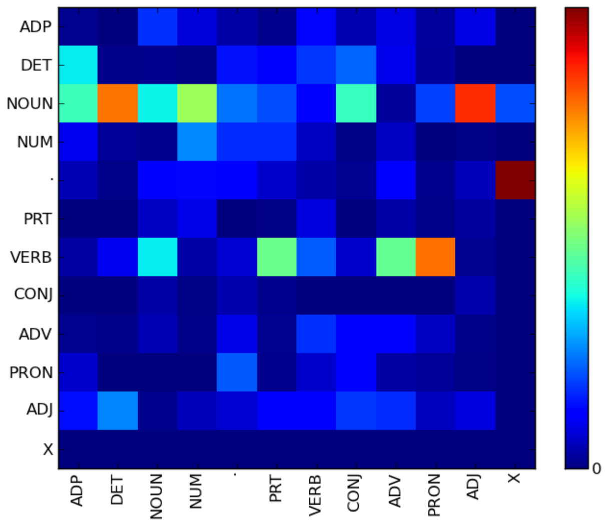

Figure 2.5: Transition probabilities of the trained model. Each column is previous state and row is current state. Note the high probability of having Noun after Adjective, as expected.

```
Smoothing 0.000000 -- Test Set Accuracy: Posterior Decode 0.177
    , Viterbi Decode: 0.381
Smoothing 0.100000 --  Train Set Accuracy: Posterior Decode 0.
    971, Viterbi Decode: 0.967
Smoothing 0.100000 -- Test Set Accuracy: Posterior Decode 0.852
    , Viterbi Decode: 0.836
Smoothing 0.010000 --  Train Set Accuracy: Posterior Decode 0.
    983, Viterbi Decode: 0.982
Smoothing 0.010000 -- Test Set Accuracy: Posterior Decode 0.816
    , Viterbi Decode: 0.809
Smoothing 1.000000 --  Train Set Accuracy: Posterior Decode 0.
    890, Viterbi Decode: 0.876
Smoothing 1.000000 -- Test Set Accuracy: Posterior Decode 0.857
    , Viterbi Decode: 0.829
```

Using the best smoothing value evaluate the accuracy on the test set.

```
In []: run readers/pos_corpus.py
In []: posc = PostagCorpus("en",max_sent_len=15,train_sents=
    1000,dev_sents=200,test_sents=200)
In []: hmm = HMM(posc)
In []: hmm.train_supervised(posc.train,smoothing=1)
In []: pred = hmm.viterbi_decode_corpus(posc.test.seq_list)
```

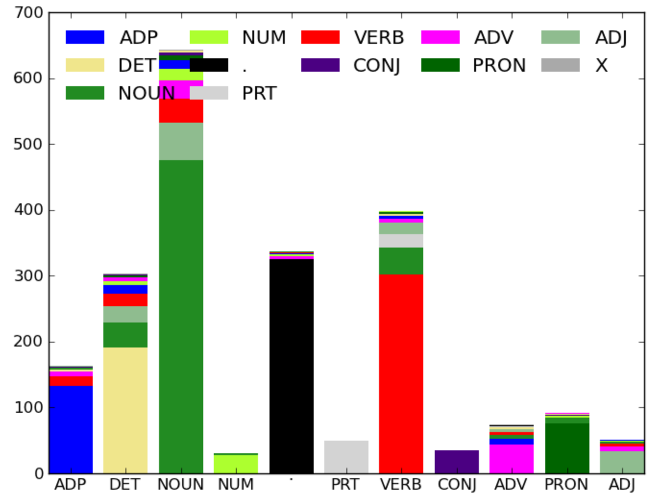

Figure 2.6: Confusion Matrix for the previous example. Predict tags are columns and the true tags corresponds to the constituents of each column.

```
In []: eval_test = hmm.evaluate_corpus(posc.test.seq_list,pred)
In []: eval_test
Out[]: 0.77711397058823528
```

Perform some error analysis to understand were the errors are coming from. You can start visualizing the confusion matrix (true tags vs predicted tags).

```
In []: run sequences/confusion_matrix.py
In []: cm = build_confusion_matrix(posc.test.seq_list,pred,len(
    posc.int_to_pos),hmm.nr_states)
```

Another option to look at is to the error for words with different number of occurrences, rare words vs commons words.

Exercise 2.10 *Implement a function that produces the accuracy for rare words vs common words. Use you own definition of rare word.*

Can you come up with other error analysis methods? Which?

Exercise 2.11 *So far we have only worked with a limited dataset of 1000 words. In this exercise you will test the accuracy of the HMM model for different training sizes (1000,5000,10000,50000). What do you observe?*