Log-Linear Models, MEMMs, and CRFs

Michael Collins

1 Notation

Throughout this note I'll use *underline* to denote vectors. For example, $\underline{w} \in \mathbb{R}^d$ will be a vector with components $w_1, w_2, \ldots w_d$. We use $\exp(x)$ for the exponential function, i.e., $\exp(x) = e^x$.

2 Log-linear models

We have sets $\mathcal{X}$ and $\mathcal{Y}$: we will assume that $\mathcal{Y}$ is a finite set. Our goal is to build a model that estimates the conditional probability $p(y|x)$ of a label $y \in \mathcal{Y}$ given an input $x \in \mathcal{X}$. For example, x might be a word, and y might be a candidate part-of-speech (noun, verb, preposition etc.) for that word. We have a feature-vector definition $\underline{\phi} : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^d$. We also assume a parameter vector $\underline{w} \in \mathbb{R}^d$. Given these definitions, log-linear models take the following form:

$$p(y|x; \underline{w}) = \frac{\exp\left(\underline{w} \cdot \underline{\phi}(x, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\underline{w} \cdot \underline{\phi}(x, y')\right)}$$

This is the conditional probability of y given x, under parameters $\underline{w}$.

Some motivation for this expression is as follows. The inner product

$$\underline{w} \cdot \underline{\phi}(x, y)$$

can take any value (positive or negative), and can be interpreted as being a measure of the plausibility of label y given input x. For a given input x, we can calculate this inner product for each possible label $y \in \mathcal{Y}$. We'd like to transform these quantities into a well-formed distribution $p(y|x)$. If we exponentiate the inner product,

$$\exp\left(\underline{w} \cdot \underline{\phi}(x, y)\right)$$

we have a strictly positive quantity—i.e., a value that is greater than 0. Finally, by dividing by the normalization constant

$$\sum_{y' \in \mathcal{Y}} \exp\left(\underline{w} \cdot \underline{\phi}(x, y')\right)$$

we ensure that $\sum_{y \in \mathcal{Y}} p(y|x; w) = 1$. Hence we have gone from inner products $\underline{w} \cdot \underline{\phi}(x, y)$, which can take either positive or negative values, to a probability distribution.

An important question is how the parameters $\underline{w}$ can be estimated from data. We turn to this question next.

The Log-Likelihood Function. To estimate the parameters, we assume that we have a set of n labeled examples, $\{(x_i, y_i)\}_{i=1}^n$. The *log-likelihood function* is

$$L(\underline{w}) = \sum_{i=1}^{n} \log p(y_i|x_i; \underline{w})$$

We can think of $L(\underline{w})$ as being a function that for a given $\underline{w}$ measures how well $\underline{w}$ explains the labeled examples. A "good" value for $\underline{w}$ will give a high value for $p(y_i|x_i; \underline{w})$ for all $i = 1 \ldots n$, and thus will have a high value for $L(\underline{w})$.

The *maximum-likelihood estimates* are

$$\underline{w}^* = \arg\max_{\underline{w} \in \mathbb{R}^d} \sum_{i=1}^{n} \log p(y_i|x_i; \underline{w})$$

The maximum-likelihood estimates are thus the parameters that best fit the training set, under the criterion $L(\underline{w})$.[1]

Finding the maximum-likelihood estimates. So given a training set $\{(x_i, y_i)\}_{i=1}^n$, how do we find the maximum-likelihood parameter estimates $\underline{w}^*$? Unfortunately, an analytical solution does not in general exist. Instead, people generally use gradient-based methods to optimize $L(w)$. The simplest method, "vanilla" gradient ascent, takes roughly the following form:

1. Set $\underline{w}^0$ to some initial value, for example set $w_j^0 = 0$ for $j = 1 \ldots d$
2. For $t = 1 \ldots T$:

[1] In some cases this maximum will not be well-defined—intuitively, some parameter values may diverge to $+\infty$ or $-\infty$—but for now we'll assume that the maximum exists, and that all parameters take finite values at the maximum.

- For $j = 1 \ldots d$, set

$$w_j^t = w_j^{t-1} + \alpha_t \times \frac{\partial}{\partial w_j} L(\underline{w}^{t-1})$$

where $\alpha_t > 0$ is some stepsize, and $\frac{\partial}{\partial w_j} L(\underline{w}^{t-1})$ is the derivative of L with respect to w_j.

3. Return the final parameters $\underline{w}^T$.

Thus at each iteration we calculate the gradient at the current point $\underline{w}^{t-1}$, and move some distance in the direction of the gradient.

In practice, more sophisticated optimization methods are used: one common to choice is to use L-BFGS, a quasi-newton method. We won't go into the details of these optimization methods in the course: the good news is that good software packages are available for methods such as L-BFGS. Implementations of L-BFGS will generally require us to calculate the value of the objective function $L(\underline{w})$, and the value of the partial derivatives, $\frac{\partial}{\partial w_j} L(\underline{w})$, at any point $\underline{w}$. Fortunately, this will be easy to do.

So what form do the partial derivatives take? A little bit of calculus gives

$$\frac{\partial}{\partial w_j} L(\underline{w}) = \sum_i \phi_j(x_i, y_i) - \sum_i \sum_y p(y|x_i; \underline{w}) \phi_j(x_i, y)$$

The first sum in the expression, $\sum_i \phi_j(x_i, y_i)$, is the sum of the j'th feature value $\phi_j(x_i, y_i)$ across the labeled examples $\{(x_i, y_i)\}_{i=1}^n$. The second sum again involves a sum over the training examples, but for each training example we calculate the *expected* feature value, $\sum_y p(y|x_i; \underline{w}) \phi_j(x_i, y)$. Note that this expectation is taken with respect to the distribution $p(y|x_i; \underline{w})$ under the current parameter values $\underline{w}$.

Regularized log-likelihood. In many applications, it has been shown to be highly beneficial to modify the log-likelihood function to include an additional *regularization term*. The modified criterion is then

$$L(\underline{w}) = \sum_{i=1}^n \log p(y_i|x_i; \underline{w}) - \frac{\lambda}{2} ||\underline{w}||^2$$

where $||\underline{w}||^2 = \sum_j w_j^2$, and $\lambda > 0$ is parameter dictating the strength of the regularization term. We will again choose our parameter values to be

$$\underline{w}^* = \arg\max_{\underline{w} \in \mathbb{R}^d} L(\underline{w})$$

Note that we now have a trade-off when estimating the parameters: we will try to make the $\log p(y_i|x_i;\underline{w})$ terms as high as possible, but at the same time we'll try to keep the norm $||\underline{w}||^2$ small (the larger the value of λ, the smaller we will require the norm to be). The regularization term penalizes large parameter values.

Intuitively, we can think of the $||\underline{w}||^2$ term as being a penalty on "complexity" of the model, where the larger the parameters are, the more complex the model is. We'd like to find a model that fits the data well, but that also has low complexity.[2]

In practice, the regularization term has been found to be very useful in building log-linear models, in particular in cases where the number of parameters, d, is large. This scenario is very common in natural language processing applications. It is not uncommon for the number of parameters d to be far larger than the number of training examples n, and in this case we can often still achieve good generalization performance, as long as a regularizer is used to penalize large values of $||\underline{w}||^2$. (There are close connections to support vector machines, where linear models are learned in very high dimensional spaces, with good generalization guarantees hold as long as the *margins* on training examples are large. Margins are closely related to norms of parameter vectors.)

Finding the optimal parameters $\underline{w}^* = \arg\max_{\underline{w}} L(\underline{w})$ can again be achieved using gradient-based methods (e.g., LBFGS). The partial derivatives are again easy to compute, and are slightly modified from before:

$$\frac{\partial}{\partial w_j} L(\underline{w}) = \sum_i \phi_j(x_i, y_i) - \sum_i \sum_y p(y|x_i;\underline{w})\phi_j(x_i, y) - \lambda w_j$$

3 MEMMs

We'll now return to sequence labeling tasks, and describe *maximum-entropy Markov models* (MEMMs), which make direct use of log-linear models. In the previous lecture we introduced HMMs as a model for sequence labeling problems. MEMMs will be a useful alternative to HMMs.

Our goal will be to model the conditional distribution

$$p(s_1, s_2 \ldots s_m | x_1 \ldots x_m)$$

where each x_j for $j = 1 \ldots m$ is the j'th input symbol (for example the j'th word in a sentence), and each s_j for $j = 1 \ldots m$ is the j'th state. We'll use $\mathcal{S}$ to denote the set of possible states; we assume that $\mathcal{S}$ is a finite set.

[2] More formally, from a Bayesian standpoint the regularization term can be viewed as $\log p(\underline{w})$ where $p(\underline{w})$ is a prior (specifically, $p(\underline{w})$ is a Gaussian prior): the parameter estimates $\underline{w}^*$ are then MAP estimates. From a frequentist standpoint there have been a number of important results showing that finding parameters with a low norm leads to better generalization guarantees (i.e., better guarantees of generalization to new, test examples).

For example, in part-of-speech tagging of English, $\mathcal{S}$ would be the set of all possible parts of speech in English (noun, verb, determiner, preposition, etc.). Given a sequence of words $x_1 \dots x_m$, there are k^m possible part-of-speech sequences $s_1 \dots s_m$, where $k = |\mathcal{S}|$ is the number of possible parts of speech. We'd like to estimate a distribution over these k^m possible sequences.

In a first step, MEMMs use the following decomposition:

$$p(s_1, s_2 \dots s_m | x_1 \dots x_m) = \prod_{i=1}^{m} p(s_i | s_1 \dots s_{i-1}, x_1 \dots x_m) \quad (1)$$

$$= \prod_{i=1}^{m} p(s_i | s_{i-1}, x_1 \dots x_m) \quad (2)$$

The first equality is exact (it follows by the chain rule of conditional probabilities). The second equality follows from an *independence assumption*, namely that for all i,

$$p(s_i | s_1 \dots s_{i-1}, x_1 \dots x_m) = p(s_i | s_{i-1}, x_1 \dots x_m)$$

Hence we are making an assumption here that is similar to the Markov assumption in HMMs, i.e., that the state in the i'th position depends only on the state in the $(i-1)$'th position.

Having made these independence assumptions, we then model each term using a log-linear model:

$$p(s_i | s_{i-1}, x_1 \dots x_m) = \frac{\exp\left(\underline{w} \cdot \underline{\phi}(x_1 \dots x_m, i, s_{i-1}, s_i)\right)}{\sum_{s' \in \mathcal{S}} \exp\left(\underline{w} \cdot \underline{\phi}(x_1 \dots x_m, i, s_{i-1}, s')\right)}$$

Here $\underline{\phi}(x_1 \dots x_m, i, s, s')$ is a feature vector where:

- $x_1 \dots x_m$ is the entire sentence being tagged
- i is the position to be tagged (can take any value from 1 to m)
- s is the previous state value (can take any value in $\mathcal{S}$)
- s' is the new state value (can take any value in $\mathcal{S}$)

See the lecture slides on log-linear models (from Lecture 1) to see examples of features used in applications such as part-of-speech tagging.

Once we've defined the feature vectors $\underline{\phi}$, we can train the parameters $\underline{w}$ of the model in the usual way for log-linear models. The training examples will consist of

sentences $x_1 \dots x_m$ annotated with state sequences $s_1 \dots s_m$. Once we've trained the parameters we will have a model of

$$p(s_i | s_{i-1}, x_1 \dots x_m)$$

and hence a model of

$$p(s_1 \dots s_m | x_1 \dots x_m)$$

The next question will be how to *decode* with the model.

Decoding with MEMMs. The decoding problem is as follows. We're given a new test sequence $x_1 \dots x_m$. Our goal is to compute the most likely state sequence for this test sequence,

$$\arg \max_{s_1 \dots s_m} p(s_1 \dots s_m | x_1 \dots x_m)$$

There are k^m possible state sequences, so for any reasonably large sentence length m brute-force search through all the possibilities will not be possible.

Fortunately, we will be able to again make use of the Viterbi algorithm: it will take a very similar form to the Viterbi algorithm for HMMs. The basic data structure in the algorithm will be a dynamic programming table π with entries

$$\pi[j, s]$$

for $j = 1 \dots m$, and $s \in \mathcal{S}$. $\pi[j, s]$ will store the maximum probability for any state sequence ending in state s at position j. More formally, our algorithm will compute

$$\pi[j, s] = \max_{s_1 \dots s_{j-1}} \left(p(s | s_{j-1}, x_1 \dots x_m) \prod_{k=1}^{j-1} p(s_k | s_{k-1}, x_1 \dots x_m) \right)$$

for all $j = 1 \dots m$, and for all $s \in \mathcal{S}$.

The algorithm is as follows:

- Initialization: for $s \in \mathcal{S}$

  $$\pi[1, s] = p(s | s_0, x_1 \dots x_m)$$

  where s_0 is a special "initial" state.

- For $j = 2 \dots m$, $s = 1 \dots k$:

  $$\pi[j, s] = \max_{s' \in \mathcal{S}} \left[\pi[j-1, s'] \times p(s | s', x_1 \dots x_m) \right]$$

Finally, having filled in the $\pi[j, s]$ values for all j, s, we can calculate

$$\max_{s_1 \ldots s_m} p(s_1 \ldots s_m | x_1 \ldots x_m) = \max_s \pi[m, s]$$

The algorithm runs in $O(mk^2)$ time (i.e., linear in the sequence length m, and quadratic in the number of states k). As in the Viterbi algorithm for HMMs, we can compute the highest-scoring sequence using backpointers in the dynamic programming algorithm (see the HMM slides from lecture 1).

Comparison between MEMMs and HMMs So what is the motivation for using MEMMs instead of HMMs? Note that the Viterbi decoding algorithms for the two models are very similar. In MEMMs, the probability associated with each state transition s_{i-1} to s_i is

$$p(s_i | s_{i-1}, x_1 \ldots x_m) = \frac{\exp\left(\underline{w} \cdot \underline{\phi}(x_1 \ldots x_m, i, s_{i-1}, s_i)\right)}{\sum_{s' \in \mathcal{S}} \exp\left(\underline{w} \cdot \underline{\phi}(x_1 \ldots x_m, i, s_{i-1}, s')\right)}$$

In HMMs, the probability associated with each transition is

$$p(s_i | s_{i-1}) p(x_i | s_i)$$

The key advantage of MEMMs is that the use of feature vectors $\underline{\phi}$ allows much richer representations than those used in HMMs. For example, the transition probability can be sensitive to *any* word in the input sequence $x_1 \ldots x_m$. In addition, it is very easy to introduce features that are sensitive to spelling features (e.g., prefixes or suffixes) of the current word x_i, or of the surrounding words. These features are useful in many NLP applications, and are difficult to incorporate within HMMs in a clean way.

4 CRFs

We now turn to conditional random fields (CRFs).

One brief note on notation: for convenience, we'll use $\underline{x}$ to refer to an input sequence $x_1 \ldots x_m$, and $\underline{s}$ to refer to a sequence of states $s_1 \ldots s_m$. The set of all possible states is again $\mathcal{S}$; the set of all possible state sequences is $\mathcal{S}^m$. In conditional random fields we'll again build a model of

$$p(s_1 \ldots s_m | x_1 \ldots x_m) = p(\underline{s} | \underline{x})$$

A first key idea in CRFs will be to define a feature vector

$$\underline{\Phi}(\underline{x}, \underline{s}) \in \mathbb{R}^d$$

that maps an *entire input sequence* $\underline{x}$ *paired with an entire state sequence* $\underline{s}$ to some d-dimensional feature vector. We'll soon give a concrete definition for $\underline{\Phi}$, but for now just assume that some definition exists. We will often refer to $\underline{\Phi}$ as being a "global" feature vector (it is global in the sense that it takes the entire state sequence into account).

We then build a *giant* log-linear model,

$$p(\underline{s}|\underline{x};\underline{w}) = \frac{\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s})\right)}{\sum_{\underline{s}'\in\mathcal{S}^m}\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s}')\right)}$$

This is "just" another log-linear model, but it is is "giant" in the sense that: 1) the space of possible values for $\underline{s}$, i.e., $\mathcal{S}^m$, is huge. 2) The normalization constant (denominator in the above expression) involves a sum over the set $\mathcal{S}^m$. At first glance, these issues might seem to cause severe computational problems, but we'll soon see that under appropriate assumptions we can train and decode efficiently with this type of model.

The next question is how to define $\underline{\Phi}(\underline{x},\underline{s})$? Our answer will be

$$\underline{\Phi}(\underline{x},\underline{s}) = \sum_{j=1}^{m}\underline{\phi}(\underline{x},j,s_{j-1},s_j)$$

where $\underline{\phi}(\underline{x},j,s_{j-1},s_j)$ are the same as the feature vectors used in MEMMs. Or put another way, we're assuming that for $k = 1\ldots d$, the k'th global feature is

$$\Phi_k(\underline{x},\underline{s}) = \sum_{j=1}^{m}\phi_k(\underline{x},j,s_{j-1},s_j)$$

Thus Φ_k is calculated by summing the "local" feature vector ϕ_k over the m different state transitions in $s_1\ldots s_m$.

We now turn to two critical practical issues in CRFs: first, *decoding*, and second, *parameter estimation*.

Decoding with CRFs The decoding problem in CRFs is as follows: for a given input sequence $\underline{x} = x_1, x_2, \ldots x_m$, we would like to find the most likely underlying state sequence under the model, that is,

$$\arg\max_{\underline{s}\in\mathcal{S}^m} p(\underline{s}|\underline{x};\underline{w})$$

We simplify this expression as follows:

$$\arg\max_{\underline{s}\in\mathcal{S}^m} p(\underline{s}|\underline{x};\underline{w}) \quad = \quad \arg\max_{\underline{s}\in\mathcal{S}^m} \quad \frac{\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s})\right)}{\sum_{\underline{s}'\in\mathcal{S}^m}\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s}')\right)}$$

$$
\begin{aligned}
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \exp\left(\underline{w} \cdot \underline{\Phi}(\underline{x}, \underline{s})\right) \\
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \underline{w} \cdot \underline{\Phi}(\underline{x}, \underline{s}) \\
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \underline{w} \cdot \sum_{j=1}^{m} \underline{\phi}(\underline{x}, j, s_{j-1}, s_j) \\
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j)
\end{aligned}
$$

So we have shown that finding the most likely sequence under the model is equivalent to finding the sequence that maximizes

$$\arg\max_{\underline{s} \in \mathcal{S}^m} \quad \sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j)$$

This problem has a clear intuition. Each transition from state s_{j-1} to state s_j has an associated score

$$\underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j)$$

This score could be positive or negative. Intuitively, this score will be relatively high if the state transition is plausible, relatively low if this transition is implausible. The decoding problem is to find an entire *sequence* of states such that the sum of transition scores is maximized.

We can again solve this problem using a variant of the Viterbi algorithm, in a very similar way to the decoding algorithm for HMMs or MEMMs:

- Initialization: for $s \in \mathcal{S}$

  $$\pi[1, s] = \underline{w} \cdot \underline{\phi}(\underline{x}, 1, s_0, s)$$

  where s_0 is a special "initial" state.

- For $j = 2 \ldots m$, $s = 1 \ldots k$:

  $$\pi[j, s] = \max_{s' \in \mathcal{S}} \left[\pi[j-1, s'] + \underline{w} \cdot \underline{\phi}(\underline{x}, j, s', s)\right]$$

We then have

$$\max_{s_1 \ldots s_m} \sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j) = \max_s \pi[m, s]$$

As before, backpointers can be used to allow us to recover the highest scoring state sequence. The algorithm again runs in $O(mk^2)$ time. Hence we have shown that decoding in CRFs is efficient.

Parameter Estimation in CRFs. For parameter estimation, we assume we have a set of n labeled examples, $\{(\underline{x}^i, \underline{s}^i)\}_{i=1}^n$. Each $\underline{x}^i$ is an input sequence $x_1^i \dots x_m^i$, each $\underline{s}^i$ is a state sequence $s_1^i \dots s_m^i$. We then proceed in exactly the same way as for regular log-linear models. The *regularized log-likelihood function* is

$$L(\underline{w}) = \sum_{i=1}^{n} \log p(\underline{s}^i | \underline{x}^i; \underline{w}) - \frac{\lambda}{2} ||\underline{w}||^2$$

Our parameter estimates are then

$$\underline{w}^* = \arg\max_{\underline{w} \in \mathbb{R}^d} \sum_{i=1}^{n} \log p(\underline{s}^i | \underline{x}^i; \underline{w}) - \frac{\lambda}{2} ||\underline{w}||^2$$

We'll again use gradient-based optimization methods to find $\underline{w}^*$. As before, the partial derivatives are

$$\frac{\partial}{\partial w_k} L(\underline{w}) = \sum_i \Phi_k(\underline{x}^i, \underline{s}^i) - \sum_i \sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \Phi_k(\underline{x}^i, \underline{s}) - \lambda w_k$$

The first term is easily computed, because

$$\sum_i \Phi_k(\underline{x}^i, \underline{s}^i) = \sum_i \sum_{j=1}^{m} \phi_k(\underline{x}^i, j, s_{j-1}^i, s_j^i)$$

Hence all we have to do is to sum over all training examples $i = 1 \dots n$, and for each example sum over all positions $j = 1 \dots m$.

The second term is more difficult to deal with, because it involves a sum over $\mathcal{S}^m$, a very large set. However, we will see that this term can be computed efficiently using dynamic programming. The derivation is as follows:

$$\sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \Phi_k(\underline{x}^i, \underline{s}) \tag{3}$$

$$= \sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \sum_{j=1}^{m} \phi_k(\underline{x}^i, j, s_{j-1}, s_j) \tag{4}$$

$$= \sum_{j=1}^{m} \sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \phi_k(\underline{x}^i, j, s_{j-1}, s_j) \tag{5}$$

$$= \sum_{j=1}^{m} \sum_{a \in \mathcal{S}, b \in \mathcal{S}} \sum_{\substack{\underline{s} \in \mathcal{S}^m : \\ s_{j-1} = a, s_j = b}} p(\underline{s} | \underline{x}^i; \underline{w}) \phi_k(\underline{x}^i, j, s_{j-1}, s_j) \tag{6}$$

$$= \sum_{j=1}^{m} \sum_{a \in \mathcal{S}, b \in \mathcal{S}} \phi_k(\underline{x}^i, j, a, b) \sum_{\substack{\underline{s} \in \mathcal{S}^m: \\ s_{j-1}=a, s_j=b}} p(\underline{s}|\underline{x}^i; \underline{w}) \tag{7}$$

$$= \sum_{j=1}^{m} \sum_{a \in \mathcal{S}, b \in \mathcal{S}} q_j^i(a, b) \phi_k(\underline{x}^i, j, a, b) \tag{8}$$

where

$$q_j^i(a, b) = \sum_{\underline{s} \in \mathcal{S}^m : s_{j-1}=a, s_j=b} p(\underline{s}|\underline{x}^i; \underline{w})$$

The important thing to note is that if we can compute the $q_j^i(a, b)$ terms efficiently, we can compute the derivatives efficiently, using the expression in Eq. 8. The quantity $q_j^i(a, b)$ has a fairly intuitive interpretation: it is the probabilty of the i'th training example $\underline{x}^i$ having state a at position $j - 1$ and state b at position j, under the distribution $p(\underline{s}|\underline{x}; \underline{w})$.

A critical result is that for a given i, *all $q_j^i(a, b)$ terms can be calculated together, in $O(mk^2)$ time*. The algorithm that achieves this is the *forward-backward algorithm*. This is another dynamic programming algorithm, and is closely related to the Viterbi algorithm.