Log-linear models
and conditional random fields

Charles Elkan
elkan@cs.ucsd.edu

February 23, 2010

The general log-linear model is a far-reaching extension of logistic regression. Conditional random fields (CRFs) are a special case of log-linear models. Section 1 below explains what a log-linear model is, and then Section 2 gives more explanation of a crucial representational idea, the generalization from features to feature-functions.

1 The general log-linear model

Let $x$ be an example, and let $y$ be a possible label for it. A log-linear model assumes that

$$p(y|x; w) = \frac{\exp \sum_{j=1}^{J} w_j F_j(x, y)}{Z(x, w)},$$

(1)

Each expression $F_j(x, y)$ is called a feature-function. Intuitively, $F_j(x, y)$ is a measure of the compatibility of the example $x$ and the label $y$. The corresponding weight $w_j$ measures the importance of this feature function. Feature-functions are defined in advance by a human, while weights are learned by a training algorithm.

The denominator in Equation 1 is a normalizing factor that is often called a partition function; it is constant given $x$ and $w$. Concretely,

$$Z(x, w) = \sum_{y'} \exp \sum_{j=1}^{J} w_j F_j(x, y').$$
For the purpose of predicting the most likely label $\hat{y}$ of a test example $x$, the partition function and the exponential operator can be ignored. Therefore, given a test example $x$, the label predicted by the model is

$$\hat{y} = \arg\max_y p(y|x; w) = \arg\max_y \sum_{j=1}^J w_j F_j(x, y).$$

Mathematically, log-linear models are very simple: there is one real-valued weight for each feature-function, no more and no fewer. There are several justifications for the form of the expression (1); these are similar to the justifications for logistic regression. First, a linear combination $\sum_{j=1}^J w_j F_j(x, y)$ can take any positive or negative real value; the exponential makes it positive, like a valid probability. Second, the division makes the results between 0 and 1, i.e. makes them be valid probabilities. Third, the ranking of the probabilities will be the same as the ranking of the linear values.

In general, a function of the form

$$b_k = \frac{\exp a_k}{\sum_{k'} \exp a_{k'}}$$

is called a softmax function because it is a differentiable analog of the maximum function, which is not smooth. In a softmax function, the exponentials enlarge the bigger $a_k$ values compared to the smaller $a_k$ values. Other functions also have the property of being similar to the maximum function, but differentiable. Softmax is widely used now, perhaps because its derivative is especially simple; see Section 6 below.

## 2 Feature functions

In general, a feature-function can be any real-valued function of the data space $X$ to which examples $x$ belong and the label space $Y$. Formally, a feature-function is any mapping $F_j : X \times Y \rightarrow \mathbb{R}$.

Suppose that $x \in \mathbb{R}^d = X$ and $y \in \{1, 2, \ldots, C\} = Y$. Then we can define feature-functions indexed from $j = 1$ to $j = dC$ as

$$F_j(x, y) = x_i \cdot I(y = c)$$

where $j = i + (c - 1)d$, for $i = 1$ to $i = d$ and $c = 1$ to $c = C$. Each of these feature-functions is zero except for one candidate value $c$ of the label $y$. 
For that value, the feature-function equals the the \( i \)th component \( x_i \) of \( x \). The corresponding weight \( w_j \) captures the degree to which \( x_i \) is predictive of the label being \( c \). Notice that the mapping \( j = i + (c - 1)d \) is arbitrary; we just need some concrete way of converting each pair \( \langle i, c \rangle \) into a single index \( j \).

The previous paragraph is only an example of how feature-functions can be constructed. It corresponds to the multiclass version of logistic regression, which is often called multinomial logistic regression. It has an important property that other feature-functions often have also: for any particular \( x \) and \( y \) most feature-functions are zero.

Often, feature-functions are presence/absence indicators, so the value of the feature-function is either 0 or 1. With log-linear models, anything and the kitchen sink can be used to define a feature-function. We can have many candidate label values, and many attributes or components of examples. Also, we can define feature-functions that pay attention to different attributes of examples for different candidate label values.

Feature-functions can overlap in arbitrary ways. For example, if \( x \) is a word then different feature-functions can use properties of \( x \) such as “starts with a capital letter,” “starts with G;” is “Graham,” and “is six letters long.” Generally we can encode suffixes, prefixes, facts from a lexicon, preceding/following punctuation, and more, in feature-functions.

## 3 Conditional random fields

Now that we understand log-linear models, we can explain conditional random fields (CRFs), specifically so-called linear-chain CRFs. First, we present linear-chain CRFs through an example application. Next, Section 4 generalizes the example, and Section 5 explains the special algorithms that make inference tractable for linear-chain CRFs. Section 6 gives a general derivation of the gradient of a log-linear model; this is the foundation of all log-linear training algorithms.

To begin, consider an example of a learning task for which a CRF is useful. Given a sentence, the task is to tag each word as noun, verb, adjective, preposition, etc. There is a fixed known set of these part-of-speech (POS) tags. Each sentence is a separate training or test example. The label of a sentence is a sequence of tags.

We represent a sentence by feature-functions based on its words. Feature-functions can be very varied:

- Some feature-functions can be position-specific, e.g. to the beginning or
to the end of a sentence, while others can be sums over all positions in a sentence.

- Some feature-functions can look just at one word, e.g. at its prefixes or suffixes.

- Some feature-functions can also use the words one to the left, one to the right, two to the left etc., up to the whole sentence.

The highest-accuracy POS taggers currently use over 100,000 feature-functions. An important restriction (that will be explained and justified below) is that a feature-function cannot depend on the entire label. Instead, it can depend on at most two tags, which must be neighboring.

POS tagging is an example of what is called a structured prediction task. The goal is to predict a complex label (a sequence of POS tags) for a complex input (an entire sentence). This task is difficult, and significantly different from a standard classifier learning task. There are at least three important sources of difficulty. First, too much information would be lost by learning just a per-word classifier. Influences between neighboring tags must be taken into account. Second, different sentences have different lengths, so it is not obvious how to represent all sentences by vectors of the same fixed length. Third, the set of all possible sequences of tags constitutes an exponentially large set of labels.

A linear conditional random field is a way to apply a log-linear model to this type of task. Use the bar notation for sequences, so $\bar{x}$ means a sequence of variable length. Specifically, let $\bar{x}$ be a sequence of $n$ words and let $\bar{y}$ be a corresponding sequence of $n$ tags. It is vital to understand the terminology we are using: $\bar{x}$ is an example, $\bar{y}$ is a label, and a component $y_i$ of $\bar{y}$ is a tag. Tags and labels should never be confused.

The standard log-linear model is

$$p(\bar{y}|\bar{x}; w) = \frac{1}{Z(\bar{x}, w)} \exp \sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}).$$

In order to specialize this for the task of predicting the label of an input sentence, assume that each feature-function $F_j$ is actually a sum along the sentence, for $i = 1$ to $i = n$ where $n$ is the length of $\bar{x}$:

$$F_j(\bar{x}, \bar{y}) = \sum_{i=1}^{n} f_j(y_{i-1}, y_i, \bar{x}, i).$$
This notation means that each low-level feature-function \( f_j \) can depend on the whole sentence \( \bar{x} \), the current tag \( y_i \) and the previous tag \( y_{i-1} \), and the current position \( i \) within the sentence. Notice that when \( i = 1 \) a low-level feature-function can refer to the tag \( y_0 \). We assume that \( y_0 = \text{START} \) where \( \text{START} \) is a special fixed tag value. Similarly, if necessary we assume that \( y_{n+1} = \text{STOP} \).

A low-level feature-function \( f_j \) may depend on only a subset of the four possible arguments. Examples of legal low-level feature-functions are “the current tag is \( \text{NOUN} \) and the current word is capitalized,” “the first word of the sentence is Mr. and the second tag is \( \text{PROPER NOUN} \),” and “the previous tag is \( \text{SALUTATION} \) and the current tag is \( \text{PROPER NOUN} \).”

Summing each \( f_j \) over all positions \( i \) means that we can have a fixed set of feature-functions \( F_j \), even though the training examples are not of fixed length.

4 Linear-chain CRFs in general

Training a CRF means finding the weight vector \( w \) that gives the best possible prediction

\[
\bar{y}^* = \arg\max_{\bar{y}} p(\bar{y}|\bar{x}; w)
\]

for each training example \( \bar{x} \). However, before we can talk about training there are two major inference problems to solve. First, how can we do the \( \arg\max \) computation in Equation 2 efficiently, for any \( \bar{x} \) and any weights \( w \)? This computation is difficult since the number of alternative tag sequences \( \bar{y} \) is exponential.

Second, given any \( \bar{x} \) and \( \bar{y} \) we want to evaluate

\[
p(\bar{y}|\bar{x}; w) = \frac{1}{Z(\bar{x}, w)} \exp \sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}).
\]

The difficulty here is that the denominator again ranges over all tag sequences \( \bar{y} \), because \( Z(\bar{x}, w) = \sum_{\bar{y}'} \exp \sum_{j} w_j F_j(\bar{x}, \bar{y}') \). For both these tasks, we will need tricks in order to take into account all possible \( \bar{y} \) efficiently, without enumerating all possible \( \bar{y} \). The fact that feature-functions can depend on at most two tags, which must be adjacent, makes these tricks exist.

The next section explains how to solve the two inference problems just described, and then the following section explains to do training via gradient following.
An issue that is the topic of considerable research is the question of which objective function to maximize during training. Often, the objective function used for training is not exactly the function that we really want to maximize on test data. Traditionally we maximize conditional log likelihood (CLL), maybe with regularization, on the training data. However, instead of maximizing CLL we could maximize yes/no accuracy of the entire predicted $\bar{y}$, or we could minimize mean-squared error if tags are numerical, or we could optimize some other measure of distance between true and predicted tags.

A fundamental question is whether we want to maximize a pointwise objective. For a long sequence, we may have a vanishing chance of predicting the entire tag sequence correctly. The single sequence with highest probability may be very different from the most probable tag at each position.

5 Inference algorithms for linear-chain CRFs

Let us solve the first problem above efficiently. First remember that we can ignore the denominator, and also the exponential inside the numerator. We want to compute

$$\bar{y}^* = \arg\max_\bar{y} p(\bar{y} | \bar{x}; w) = \arg\max_\bar{y} \sum_{j=1}^J w_j F_j(\bar{x}, \bar{y}).$$

Use the definition of $F_j$ as a sum over the sequence to get

$$\bar{y}^* = \arg\max_\bar{y} \sum_{j=1}^J w_j \sum_{i=1}^n f_j(y_{i-1}, y_i, \bar{x}, i) = \arg\max_\bar{y} \sum_{i=1}^n g_i(y_{i-1}, y_i)$$

where we define

$$g_i(y_{i-1}, y_i) = \sum_{j=1}^J w_j f_j(y_{i-1}, y_i, \bar{x}, i).$$

Note that the $\bar{x}$ argument of $f_j$ has been dropped in the definition of $g_i$, since we are considering only a single fixed input $\bar{x}$. The argument $i$ of $f_j$ is written as a subscript on $g$. For each $i$, $g_i$ is a different function. The arguments of each $g_i$ are just two tag values, because everything else is fixed.

Remember that each entry of the $\bar{y}$ vector is one of a finite set of tags. Given $\bar{x}$, $w$, and $i$ the function $g_i$ can be represented as an $m$ by $m$ matrix where $m$ is the cardinality of the set of tags. Computing this matrix requires $O(m^2 J)$ time, assuming that each low-level feature-function can be evaluated in constant time.
Let \( v \) range over the set of tags. Define \( U(k, v) \) to be the score of the best sequence of tags from position 1 to position \( k \), where tag number \( k \) is required to equal \( v \). This is a maximization over \( k - 1 \) tags because tag number \( k \) is fixed to have value \( v \). Formally,

\[
U(k, v) = \max_{y_1, \ldots, y_{k-1}} \left[ \sum_{i=1}^{k-1} g_i(y_{i-1}, y_i) + g_k(y_{k-1}, v) \right].
\]

Now we can write down a recurrence that lets us compute \( U(k, v) \) efficiently:

\[
U(k, v) = \max_u [U(k-1, u) + g_k(u, v)]
\]

With this recurrence we can compute \( U(k, v) \) for a single \( v \) in \( O(m) \) time, given knowledge of the matrix \( g_k \) and of \( U(k-1, u) \) for every \( u \), where \( m \) is the number of possible tags. Therefore, we can compute \( U(k, v) \) for every \( v \) in \( O(m^2) \) time.

In total, we can compute the optimal \( \bar{y} \) for any \( \bar{x} \) in \( O(m^2nJ + m^2n) \) time, where \( n \) is the length of \( \bar{x} \). Because most feature-functions are usually zero, in practice the factor \( J \) can be made much smaller.

The algorithm just explained is a variation of the Viterbi algorithm for computing the highest-probability path through a hidden Markov model. The base case of the recurrence is an exercise for the reader.

The second fundamental computational problem is to compute the denominator of the probability formula. This denominator is called the partition function:

\[
Z(\bar{x}, w) = \sum_{\bar{y}} \exp \sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}).
\]

Remember that

\[
\sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}) = \sum_{i=1}^{n} g_i(y_{i-1}, y_i)
\]

where \( i \) ranges over all positions of the input sequence \( \bar{x} \), so we can write

\[
Z(\bar{x}, w) = \sum_{\bar{y}} \exp \sum_{i=1}^{n} g_i(y_{i-1}, y_i) = \sum_{\bar{y}} \prod_{i} \exp g_i(y_{i-1}, y_i).
\]

We can compute the expression above efficiently by matrix multiplication. For \( t = 1 \) to \( t = n + 1 \) define \( M_t \) to be a square \( m \) by \( m \) matrix such that \( M_t(u, v) = \)
exp \( g_i(u, v) \) for any two tag values \( u \) and \( v \). Note that \( M_2 \) to \( M_n \) are fully defined, while \( M_1(u, v) \) is defined only for \( u = \text{START} \) and \( M_{n+1}(u, v) \) is defined only for \( v = \text{STOP} \).

Consider multiplying \( M_1 \) and \( M_2 \). We have

\[
M_{12}(\text{START}, w) = \sum_v M_1(\text{START}, v)M_2(v, w) = \sum_v \text{[exp} g_1(\text{START}, v)] [\text{exp} g_2(v, w)].
\]

Similarly,

\[
M_{123}(\text{START}, x) = \sum_w M_{12}(\text{START}, w)M_3(w, x)
\]

\[
= \sum_w \left( \sum_v M_1(\text{START}, v)M_2(v, w) \right)M_3(w, x)
\]

\[
= \sum_{v, w} M_1(\text{START}, v)M_2(v, w)M_3(w, x)
\]

which we can rewrite as

\[
M_{123}(\text{START}, x) = \sum_{y_1, y_2} M_1(\text{START}, y_1)M_2(y_1, y_2)M_3(y_2, x).
\]

We can continue in a similar fashion until we get the product \( M_{123...n+1} \) of all matrices. The \( \langle \text{START}, \text{STOP} \rangle \) entry of this product is

\[
M_{123...n+1}(\text{START}, \text{STOP}) = \sum_{y_1, y_2, \ldots, y_n} M_1(\text{START}, y_1)M_2(y_1, y_2) \cdots M_{n+1}(y_n, \text{STOP}).
\]

Call this value \( T \). We have

\[
T = \sum_{\text{all } y} \text{exp}[g_1(\text{START}, y_1)] \text{exp}[g_2(y_1, y_2)] \cdots \text{exp}[g_{n+1}(y_n, \text{STOP})]
\]

\[
= \sum_{\text{all } y} \prod_i \text{exp}[g_i(y_{i-1}, y_i)]
\]

which is exactly what we need.

Computational complexity: Each matrix is \( m \) by \( m \) where \( m \) is the cardinality of the tag set. Each matrix multiplication requires \( O(m^3) \) time, so the total time

\footnote{Note on notation: \( u, v, w, \) and \( x \) here are all single tags; \( w \) is not a weight and \( x \) is not a component of \( x \).}
needed appears to be $O(nm^3)$, assuming the matrices have already been computed and stored. However, when computing $M_{12}$ one only needs to multiply $M_2$ by one row of $M_1$, namely the $M_1(\text{START}, v)$ row, which takes $O(m^2)$ time. This yields the row $M_{12}(\text{START}, v)$, which then gets multiplied by $M_3$, and so on. The total time required is hence $O(nm^2)$, which is the same as the the time needed by the Viterbi algorithm.

The matrix multiplication method for computing the partition function is called a forward-backward algorithm. A similar algorithm can be used to compute any function of the form $\sum_{\bar{y}} h_i(y_{i-1}, y_i)$. For details see the survey paper by Rahul Gupta cited below.

Some extensions to the basic linear-chain CRF are not difficult. The output $\bar{y}$ must be a sequence, but the input $\bar{x}$ is treated as a unit, so it does not have to be a sequence. It could be two-dimensional, like an image for example. It could also be an unordered collection of items.

In general, what is fundamental for making a log-linear model tractable is that the set of possible labels $\bar{y}$ should either be small, or have some structure. In order to have structure, $\bar{y}$ should be made up of parts (e.g. tags) such that only small subsets of parts interact directly with each other. Here, every interacting subset of tags is a pair. Often, the real-world reason interacting subsets are small is that parts of a label only interact if they are close together according to some real-world notion of distance.

6 Gradients for log-linear models

The learning task for a log-linear model is to choose values for the weights (also called parameters). Given a set of training examples, we assume now that the goal is to choose parameter values $w_j$ that maximize the conditional probability of the training examples. In other words, the objective function for training is the conditional log-likelihood (CLL) of the set of training examples. Since we want to maximize CLL, we do gradient ascent as opposed to descent.

For online gradient ascent (also called stochastic gradient ascent) we update parameters based on single training examples. Therefore, we evaluate the partial derivative of CLL for a single training example, for each $w_j$. (There is one weight for each feature-function, so we use $j$ to range over weights.) The partial derivative of the logarithm of the CLL is

$$\frac{\partial}{\partial w_j} \log p(y|x; w) = F_j(x, y) - \frac{\partial}{\partial w_j} \log Z(x, w)$$
\[ F_j(x, y) - \frac{1}{Z(x, w)} \sum_{y'} \frac{\partial}{\partial w_j} Z(x, w) \]

Above, \( y \) is the known true label of the training example \( x \), and \( j \) is the index of the parameter for which we are computing the partial derivative. The sum over \( y' \) is a sum over all candidate labels. Expanding \( Z(x, w) \) gives

\[
\frac{\partial}{\partial w_j} Z(x, w) = \frac{\partial}{\partial w_j} \sum_{y'} \exp \left[ \sum_{j'} w_{j'} F_{j'}(x, y') \right]
\]

\[
= \sum_{y'} \frac{\partial}{\partial w_j} \left[ \exp \sum_{j'} w_{j'} F_{j'}(x, y') \right]
\]

\[
= \sum_{y'} \left[ \exp \sum_{j'} w_{j'} F_{j'}(x, y') \right] \frac{\partial}{\partial w_j} \left[ \sum_{j'} w_{j'} F_{j'}(x, y') \right]
\]

\[
= \sum_{y'} \left[ \exp \sum_{j'} w_{j'} F_{j'}(x, y') \right] F_j(x, y') .
\]

So,

\[
\frac{\partial}{\partial w_j} \log p(y|x; w) = F_j(x, y) - \frac{1}{Z(x, w)} \sum_{y'} F_j(x, y') \left[ \exp \sum_{j'} w_{j'} F_{j'}(x, y') \right]
\]

\[
= F_j(x, y) - \sum_{y'} F_j(x, y') \frac{\exp \sum_{j'} w_{j'} F_{j'}(x, y')}{Z(x, w)}
\]

\[
= F_j(x, y) - \sum_{y'} F_j(x, y') \frac{\exp \sum_{j'} w_{j'} F_{j'}(x, y') \sum_{y''} \exp \sum_{j''} w_{j''} F_{j''}(x, y'')}{Z(x, w)}
\]

\[
= F_j(x, y) - \sum_{y'} F_j(x, y') p(y'|x; w)
\]

\[
= F_j(x, y) - E_{y' \sim p(y'|x; w)} [F_j(x, y')].
\]

In words, the partial derivative with respect to weight number \( j \) for training example \( \langle x, y \rangle \) is the value of feature-function \( j \) for \( x \) and \( y \), minus the average value of the feature-function for \( x \) and all possible labels \( y' \).

The gradient of the CLL given the entire training set \( T \) is the sum of the gradients for each training example. At the global maximum this entire gradient is zero, so we have

\[
\sum_{\langle x, y \rangle \in T} F_j(x, y) = \sum_{\langle x, \cdot \rangle \in T} E_{y \sim p(y|x; w)} [F_j(x, y)].
\]
where $T$ is the training set and the notation $\langle x, \cdot \rangle$ means that the true training labels are not relevant on the righthand side of the equation. This equality is true only for the whole training set, not for training examples individually.

The left side above is the total value (mass) of feature-function $j$ on the whole training set. The right side is the total value of feature-function $j$ predicted by the model. For each feature-function, the trained model will spread out over all labels of all examples as much mass as the training data has for this feature-function.

For any particular application of log-linear modeling, we have to write code to evaluate numerically the symbolic derivatives. Then we can invoke an optimization routine to find the optimal parameter values. There are two ways that we can verify correctness. First, before running the optimization algorithm, check that each partial derivative is correct by comparing it numerically to the value obtained by finite differencing of the CLL objective function. Second, after doing the optimization, check for each feature-function $F_j$ that

$$
\sum_{(x,y) \in T} F_j(x, y) = \sum_{(x, \cdot) \in T} \sum_{y'} p(y' | x; w) F_j(x, y').
$$

7 Stochastic gradient ascent

When maximizing the conditional log-likelihood by online gradient ascent, the update to weight $w_j$ is

$$
w_j := w_j + \alpha (F_j(x, y) - E[F_j(x, y') | y' \sim p(y' | x; w)])
$$

where $\alpha$ is a learning rate parameter. If the log-linear model is a CRF, then the expectation in Equation 3 is computed by an algorithm similar to the matrix algorithm above.

Suppose that every feature-function $F_j$ is the product of an attribute value $a_j(x)$ that is a function of $x$ only, and a label function $b_j(y)$ that is a function of $y$ only, i.e. $F_j(x, y) = a_j(x)b_j(y)$. Then $\frac{\partial}{\partial a_j(x)} \log p(y | x; w) = 0$ if $a_j(x) = 0$, regardless of $y$. This implies that given example $x$ with online gradient ascent, the weight for a feature-function must be updated only for feature-functions for which the corresponding attribute $a_j(x)$ is non-zero, which can be a great saving of computational effort. In other words, the entire gradient with respect to a single training example is typically a sparse vector, just like the vector of all $F_j(x, y)$ values is sparse for a single training example.

A similar savings is possible when computing the gradient with respect to the whole training set. Note that the gradient with respect to the whole training set is
a single vector that is the sum of one vector for each training example. Typically
these vectors being summed are sparse, but their sum is not.

8 Alternative log-linear training methods

The following sections explain three special training algorithms for log-linear
models. One is a variant of the perceptron method, the second uses Gibbs sam-
ppling, and the third is a heuristic called contrastive divergence. All are useful for
CRFs.

As explained above, the partial derivative for stochastic gradient training of a
log-linear model is

$$\frac{\partial}{\partial w_j} \log p(y|x; w) = F_j(x, y) - E_{y' \sim p(y'|x; w)}[F_j(x, y')]$$.

The first term $F_j(x, y)$ is fast to compute because $x$ and its training label $y$ are
fixed. However, if the set of alternative labels $y'$ is large, then it is computa-
tionally expensive to evaluate the expectation $E_{y' \sim p(y'|x; w)}[F_j(x, y')]$. We can find
approximations to this expectation by finding approximations to the distribution
$p(y|x; w)$. Each section below describes a method based on a different approxi-
mation.

9 The Collins perceptron

Suppose we place all the probability mass on the most likely $y$ value. Thus means
that we use the approximation

$$\hat{p}(y|x; w) = I(y = \hat{y})$$

where $\hat{y} = \arg\max_y p(y|x; w)$.

Then the stochastic gradient update rule (3) simplifies to the following rule:

$$w_j := w_j + \alpha F_j(x, y)$$

$$w_j := w_j - \alpha F_j(x, \hat{y})$$.

This rule is applied for every weight $w_j$, for a given training example $\langle x, y \rangle$. Given
a training example $x$, the label $\hat{y}$ can be thought of as an “impostor” compared to
the genuine label $y$. 12
The simple update rule above is called the Collins perceptron because it was first investigated by Michael Collins, who pointed out that it is a version of the standard perceptron method. The goal is to learn a binary function of vectors in $\mathbb{R}^J$ whose components are feature-function values $\langle F_1(x, y), \ldots, F_J(x, y) \rangle$. Vectors that correspond to training examples $\langle x, y \rangle$ are positive examples for the perceptron. Vectors that correspond to incorrect labels such as $\hat{y}$ are negative examples. Hence, the two updates above are perceptron updates: the first for a positive example and the second for a negative example.

The perceptron method causes a net increase in $w_j$ for features $F_j$ whose value is higher for $y$ than for $\hat{y}$. It thus modifies the weights to directly increase the probability of $y$ compared to the probability of $\hat{y}$. In the special case where $\hat{y}$ is predicted correctly, that is $\hat{y} = y$, there is no change in the weight vector.

10 Gibbs sampling

Computing the most likely label $\hat{y}$ does not require computing the partition function $Z(x, w)$. Nevertheless, sometimes identifying $\hat{y}$ is still too difficult. In this case one option for training is to estimate $E_{y \sim p(y|x; w)}[F_j(x, y)]$ approximately by sampling $y$ values from the distribution $p(y|x; w)$.

A method known as Gibbs sampling can be used to find the needed samples of $y$. Gibbs sampling is the following algorithm. Suppose the entire label $y$ can be written as a set of parts $y = \{y_1, \ldots, y_n\}$. For example, if $y$ is the part-of-speech sequence that is the label of an input sentence $x$, then each $y_i$ can be the tag of one word in the sentence. Suppose the marginal distribution $p(y_i|x, y_1, y_{i-1}, \ldots, y_{i+1}, y_n; w)$ can be evaluated numerically in an efficient way for every $i$. Then we can get a stream of samples by the following process:

1. Select an arbitrary initial guess $\langle y_1, \ldots, y_n \rangle$.

2. Draw $y'_1$ according to $p(y_1|x, y_2, \ldots, y_n; w)$;
   - draw $y'_2$ according to $p(y_2|x, y'_1, y_3, \ldots, y_n; w)$;
   - draw $y'_3$ according to $p(y_3|x, y'_1, y'_2, y_4, \ldots, y_n; w)$;
   - and so on until $y'_n$. 

13
(3) Set \(\{y_1, \ldots, y_n\} := \{y'_1, \ldots, y'_n\}\) and repeat from (2).

It can be proved that if Step (2) is repeated an infinite number of times, then the distribution of \(y = \{y'_1, \ldots, y'_n\}\) converges to the true distribution \(p(y|x; w)\) regardless of the starting point. In practice, we do Step (2) some number of times (say 1000) to come close to convergence, and then take several samples \(y = \{y'_1, \ldots, y'_n\}\). Between each sample we repeat Step (2) a smaller number of times (say 100) to make the samples almost independent of each other.

Using Gibbs sampling to estimate the expectation \(E_{y \sim p(y|x; w)}[F_j(x, y)]\) is computationally intensive because the accuracy of the estimate only increases very slowly as the number \(s\) of samples increases. Specifically, the variance decreases proportional to \(1/s\).

Gibbs sampling relies on drawing samples efficiently from marginal distributions. Let \(y_{-i}\) be an abbreviation for the set \(\{y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n\}\). We need to draw values according to the distribution \(p(y_i|x, y_{-i}; w)\). The straightforward way to do this is to evaluate \(p(v|x, y_{-i}; w)\) numerically for each possible value \(v\) of \(y_i\). In typical applications the number of alternative values \(v\) is small, so this approach is feasible, if \(p(v|x, y_{-i}; w)\) can be computed.

Suppose the entire conditional distribution is a Markov random field

\[
p(y|x; w) \propto \prod_{m=1}^{M} \phi_m(y^m|x; w)
\]

where each \(\phi_m\) is a potential function that depends on just a subset \(y^m\) of parts of \(y\). Linear-chain conditional random fields are a special case of Equation (4). In this case

\[
p(y_i|x, y_{-i}; w) \propto \prod_{m \in C} \phi_m(y^m|x; w)
\]

where \(C\) indexes those potential functions \(\phi^m\) that depend on part \(y_i\). To compute \(p(y_i|x, y_{-i}; w)\) we evaluate the product (5) for all values of \(y_i\), with the given fixed values of \(y_{-i} = \{y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n\}\). We then normalize using

\[
Z(x, y_{-i}; w) = \sum_{v} \prod_{m \in C} \phi_m(y^m|x; w)
\]

where \(v\) ranges over the possible values of \(y_i\).
11 Contrastive divergence

A third training option is to choose a single $y^*$ value that is somehow similar to the training label $y$, but also has high probability according to $p(y|x; w)$. Compared to the impostor $\hat{y}$, the “evil twin” $y^*$ will have lower probability, but will be more similar to $y$.

The idea of contrastive divergence is to obtain a single value $y^* = \langle y^*_1, \ldots, y^*_n \rangle$ by doing only a few iterations of Gibbs sampling (often only one), but starting at the training label $y$ instead of at a random guess.

12 Tutorials and selected papers

The following are four tutorials that are available on the web.


All four surveys above are very good. The report by Memisevic places CRFs in the context of other methods for learning to predict complex outputs, especially SVM-inspired large-margin methods. Sutton’s survey is a longer discussion, with many helpful comments and explanations. The tutorial by Wallach is easy to follow and provides high-level intuition. One difference between the two tutorials is that Wallach represents CRFs as undirected graphical models, whereas Sutton uses undirected factor graphs. Sutton also does parallel comparisons of naive Bayes (NB) and logistic regression, and of hidden Markov models (HMMs) and linear-chain CRFs. This gives readers a useful starting point if they have experience with NB classifiers or HMMs. Gupta’s paper gives a detailed derivation of the important equations for CRFs.
Bibliographies on CRFs have been compiled by Rahul Gupta and Hanna Wallach. The following papers may be particularly interesting or useful. They are listed in approximate chronological order. Note that several are on topics related to CRFs, not on CRFs directly.


The basic equation for a log-linear model is

\[ p(y|x; w) = \frac{\exp \sum_{j=1}^{J} w_j F_j(x, y)}{\sum_{y'} \exp \sum_{j=1}^{J} w_j F_j(x, y')} \, . \]

We saw in class that it is sensible for a feature function \( F_j \) to depend on both the example \( x \) and the candidate label \( y \), or on just \( y \).

[3 points] Explain briefly why it is not sensible for a feature function to depend on just \( x \). That is, explain why a feature-function of the form \( F_j(x, y) = g(x) \) would be useless.
CSE 250B project assignment 4, due in class on March 11

For this final assignment you may work in a team of either two or three students. The joint report for your team must be submitted in hard copy at the start of the last CSE 250B lecture, which is on Thursday, March 11, 2010.

The objective of this project is to understand the basic algorithms for conditional random fields (CRFs) thoroughly by implementing them yourself. The goal is to learn a CRF model that can place hyphens in novel English words correctly. The training set consists of 66,001 English words with correct hyphens indicated. It is available at http://www.cs.ucsd.edu/users/elkan/hyphenation/.

You need to define and code your own set of feature-functions that identify characteristics of words that are potentially relevant. You can try feature-functions that measure word length, prefixes and suffixes, vowel/consonant patterns, and more. However, feature-functions that identify windows of three, four, five, etc. consecutive specific letters may be most useful. Start with a small set of features for debugging purposes.

You must design and implement a method for generating feature-functions systematically from high-level specifications such as “all consecutive sequences of four specific letters.” Make sure that each feature-function has value zero most of the time, and that this common case is handled efficiently.

The CRF algorithms to implement include the matrix multiplication method for computing the partition function and the Viterbi algorithm for computing the most probable labeling of a sequence. You may use a random subset of words initially, while you are optimizing the speed of your Matlab code.

You should implement and do experiments with two different training methods, namely Collins’ perceptron algorithm and one of the following:

1. a general-purpose nonlinear optimization package,
2. contrastive divergence, or
3. stochastic gradient following.

You should implement each method yourself. Note that the perceptron training approach needs only the Viterbi algorithm to be working. Contrastive divergence does not need the gradient, but does need you to understand and implement Gibbs sampling. The other two methods require computing the gradient.
The general-purpose nonlinear optimization software that I have found most useful is Mark Schmidt’s minFunc available at http://people.cs.ubc.ca/~schmidtm/Software/minFunc.html.

It is vital to be confident that your code is correct. In your report you need to convince the reader that this is true. For methods that use gradients, use the checkgrad.m function available at http://www.kyb.tuebingen.mpg.de/bs/people/carl/code/minimize/ to verify that your derivatives are correct. Modify this function to check a random subset of partial derivatives, to make it fast.

After you are sure that your implementations are correct, tune the settings of the learning algorithms using cross-validation. Use two performance metrics: word-level accuracy and letter-level accuracy. The former is the fraction of words that a method gets exactly right. The latter is the fraction of letters for which a method predicts correctly whether or not a hyphen is legal after this letter.

Do experiments with two different methods of coding the hyphen labels. The first coding method uses the tag 1 to indicate that a hyphen is legal after a letter, and the tag 0 to indicate that a hyphen is not legal. The second coding method uses the tag 1 for letters that are immediately after a legal hyphen, the tag 2 for letters that are in second place after a legal hyphen, and so on. For both coding methods, make sensible decisions about what tags to use for the first and last letters of a word, and for positions immediately before and after a word.