Log-Linear Models and Conditional Random Fields

Charles Elkan
elkan@cs.ucsd.edu

December 6, 2007

This document describes log-linear models, which are a far-reaching extension of logistic regression, and conditional random fields (CRFs), which are a special case of log-linear models.

Section 1 explains what a log-linear model is, and introduces feature functions. Section 2 then presents linear-chain CRFs as an example of log-linear models, and Section 3 explains the special algorithms that make inference tractable for these CRFs. Section 4 gives a general derivation of the gradient of a log-linear model; this is the foundation of all log-linear training algorithms. Finally Section 5 presents two special CRF training algorithms, one that is a variant of the perceptron method and another one called contrastive divergence.

1 Log-linear models

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

$$p(y|x;w) = \frac{\exp \sum_j w_j F_j(x,y)}{Z(x,w)} \tag{1}$$

where the partition function $Z(x,w) = \sum_{y'} \exp \sum_j w_j F_j(x,y')$. Therefore, given x, the label predicted by the model is

$$\hat{y} = \text{argmax}_y p(y|x;w) = \text{argmax}_y \sum_j w_j F_j(x,y).$$

Each expression $F_j(x, y)$ is called a feature-function. In general, a feature-function can be any real-valued function of both the data space X and the label space Y. Formally, a feature-function is any mapping $F_j : X \times Y -> \mathbb{R}$.

Often, a feature-function is zero for all values of y except one particular value. Given some attribute of x, we can have a different weight for this attribute and each different label. The weights for these feature-functions can then capture the affinity of this attribute-value for each label. Often, feature-functions are presence/absence indicators, so the value of the feature-function is either 0 or 1. If we have a conventional attribute $a(x)$ with k alternative values, and n classes, we can make kn different features as defined above. With log-linear models, anything and the kitchen sink can be a feature. We can have lots of classes, lots of features, and we can pay attention to different features for different classes.

Feature-functions can overlap in arbitrary ways. For example, if x is a word different feature-functions can use attributes of x such as "starts with a capital letter," "starts with G,", is "Graham," "is six letters long." Generally we can encode suffixes, prefixes, facts from a lexicon, preceding/following punctuation, etc., as features.

Mathematically, log-linear models are very simple: there is one real-valued weight for each feature, no more no fewer. There are several possible justifications for the form of the expression (1). First, a linear combination $\sum_j w_j F_j(x, y)$ 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.

A function of the form

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

is called a softmax function because the exponentials enlarge the bigger a_k values compared to the smaller a_k values. Other functions have the same property of being similar to the maximum function, but differentiable. Softmax is widely used now, perhaps because its derivative is especially simple; see Section 4 below.

2 Conditional random fields

A conditional random field (CRFs) is an important special case of a log-linear model. First, 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. We will 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 features 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 each feature-function can depend on only one tag, or on two neighboring tags.

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. Define the log-linear model

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

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 $\bar{x}$:

$$F_j(\bar{x},\bar{y}) = \sum_i f_j(y_{i-1}, y_i, \bar{x}, i).$$

This notation means that each low-level feature-function f_j can depend on the whole sentence, the current tag and the previous tag, and the current position i within the sentence. A feature-function f_j may depend on only a subset of these four possible influences. Examples of features are "the current tag is NOUN and the current word is capitalized," "the word at the start of the sentence is Mr." and "the previous tag was SALUTATION."

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

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

$$\bar{y}^* = \text{argmax}_{\bar{y}}\ p(\bar{y}|\bar{x}; w) \tag{2}$$

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 argmax 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

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

The difficulty here is that the denominator again ranges over all tag sequences $\bar{y}$: $Z(\bar{x}, w) = \sum_{\bar{y}'} \exp \sum_j w_j F_j(\bar{x}, \bar{y}')$. For both these tasks, we will need tricks to account for 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.

3 Inference algorithms for linear-chain CRFs

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

$$\bar{y}^* = \text{argmax}_{\bar{y}}\ p(\bar{y}|\bar{x}; w) = \text{argmax}_{\bar{y}}\ \sum_j w_j F_j(\bar{x}, \bar{y}).$$

Use the definition of F_j to get

$$\bar{y}^* = \text{argmax}_{\bar{y}}\ \sum_j w_j \sum_i f_j(y_{i-1}, y_i, \bar{x}, i) = \text{argmax}_{\bar{y}} \sum_i g_i(y_{i-1}, y_i)$$

where $g_i(y_{i-1}, y_i) = \sum_j w_j f_j(y_{i-1}, y_i, \bar{x}, i)$. Note that the $\bar{x}$ and i arguments of f_j have been dropped in the definition of g_i. Each g_i is a different function for each i, and depends on w as well as on $\bar{x}$ and i.

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.

Let v range over the tags. Define $U(k, v)$ to be the score of the best sequence of tags from 1 to k, where tag k is required to be v. This is a maximization over $k-1$ tags because tag number k is fixed to have value v. Formally,

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

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

$$U(k, v) = \max_{y_{k-1}} [U(k-1, y_{k-1}) + g_k(y_{k-1}, v)]$$

With this recurrence we can compute $\bar{y}$ for any $\bar{x}$ in $O(m^2 n)$ time, where n is the length of $\bar{x}$ and m is the cardinality of the set of tags. This algorithm 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:

$$Z(\bar{x}, w) = \sum_{\bar{y}} \exp \sum_j w_j F_j(\bar{x}, \bar{y}).$$

Remember that

$$\sum_j w_j F_j(\bar{x}, \bar{y}) = \sum_i g_i(y_{i-1}, y_i),$$

where i ranges over all positions 1 to n of the input sequence $\bar{x}$, so we can write

$$Z(\bar{x}, w) = \sum_{\bar{y}} \exp \sum_i g_i(y_{i-1}, y_i) = \sum_{\bar{y}} \prod_i \exp g_i(y_{i-1}, y_i).$$

We can compute the expression above efficiently by matrix multiplication. For $t = 1$ to $t = n + 1$ let M_t be a square m by m matrix such that $M_t(u, v) = \exp g_t(u, v)$ 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[1]

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

Similarly,

$$\begin{aligned} M_{123}(\text{START}, x) &= \sum_w M_{12}(\text{START}, w) M_3(w, x) \\ &= \sum_w [\sum_v M_1(\text{START}, v) M_2(v, w)] M_3(w, x) \\ &= \sum_{v,w} M_1(\text{START}, v) M_2(v, w) M_3(w, x) \end{aligned}$$

and so on. Consider the $\langle \text{START}, \text{STOP} \rangle$ entry of the entire product $M_{123\ldots n+1}$. This is

$$M_{123\ldots n+1}(\text{START}, \text{STOP}) = T = \sum_{\bar{y}} M_1(\text{START}, y_1) M_2(y_1, y_2) \ldots M_{n+1}(y_n, \text{STOP}).$$

We have

$$\begin{aligned} T &= \sum_{\bar{y}} \exp[g_1(\text{START}, y_1)] \exp[g_2(y_1, y_2)] \ldots \exp[g_{n+1}(y_n, \text{STOP})] \\ &= \sum_{\bar{y}} \prod_i \exp[g_i(y_{i-1}, y_i)] \end{aligned}$$

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 is $O(nm^3)$. We have reduced a sum over an exponential number of alternatives to a polynomial-time computation. However, even though polynomial, this is worse than the time needed by the Viterbi algorithm. An interesting question is whether computing the partition function is harder in some fundamental way than computing the most likely label sequence.

[1]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 $\bar{x}$.

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 $\sum_{\bar{y}} h_i(y_{i-1}, y_i)$.

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 an image for example, or a collection of separate items, e.g. telephone customers.

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 interactions between parts are short-distance.

4 Training by gradient ascent

The learning task for a log-linear model is to choose values for the weights (also called parameters). Given a set of training examples, our goal is to choose parameter values w_j to maximize the conditional probability of the training examples.

With this functional form for probabilities, we can choose parameter values w_j to maximize the conditional probability of the training examples.

To learn values for the parameters w_j by gradient-following we need to be able to evaluate the objective function and its gradient. The standard objective function is the conditional log-likelihood (CLL). We want to maximize CLL, so we do gradient ascent as opposed to descent.

The objective function used for training is not the same one that we really want to maximize on test data. Instead of maximizing CLL we could maximize (all on training data): yes/no accuracy of the entire predicted $\bar{y}$, or pointwise conditional log likelihood, or we could minimize mean-squared error if tags are numerical, or some other measure of distance between true and predicted tags.

A fundamental issue 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.

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.) Start with

$$\begin{aligned}
\frac{\partial}{\partial w_j} \log p(y|x;w) &= F_j(x,y) - \frac{\partial}{\partial w_j} \log Z(x,w) \\
&= F_j(x,y) - \frac{1}{Z(x,w)} \sum_{y'} \frac{\partial}{\partial w_j} \exp \sum_{j'} w_{j'} F_{j'}(x,y') \\
&= F_j(x,y) - \frac{1}{Z(x,w)} \sum_{y'} [\exp \sum_{j'} w_{j'} F_{j'}(x,y')] F_j(x,y') \\
&= F_j(x,y) - \sum_{y'} F_j(x,y') \frac{\exp \sum_{j'} w_{j'} F_{j'}(x,y')}{\sum_{y''} \exp \sum_{j''} w_{j''} F_{j''}(x,y'')} \\
&= F_j(x,y) - \sum_{y'} F_j(x,y') p(y'|x;w) \\
&= F_j(x,y) - E_{y' \sim p(y'|x;w)}[F_j(x,y')].
\end{aligned}$$

In words, the partial derivative with respect to weight number i is the value of feature-function i for the true training label y, minus the average value of the feature-function for all possible labels y'. Note that this derivation allows feature-functions to be real-valued, not just zero or one.

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

$$\sum_{\langle x,y \rangle \in T} F_j(x,y) = \sum_{\langle x,\cdot \rangle \in T} E_{y \sim p(y|x;w)}[F_j(x,y)].$$

This equality is true only for the whole training set, not for training examples individually.

The left side above is the total value 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 just on those examples for which the feature-function is nonzero.

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, check for each feature-function F_j that

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

Second, check that each partial derivative is correct by comparing it numerically to the value obtained by finite differencing of the CLL objective function.

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 $\frac{\partial}{\partial w_j} \log p(y|x; w) = 0$ 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.

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

$$w_j := w_j + \alpha(F_j(x, y) - E_{y' \sim p(y'|x;w)}[F_j(x, y')]) \tag{3}$$

where α is a learning rate parameter.

5 Efficient CRF training

The partial derivative for stochastic gradient training of a CRF model is

$$\begin{aligned}\frac{\partial}{\partial w_j} \log p(\bar{y}|\bar{x}; w) &= F_j(\bar{x}, \bar{y}) - \sum_{\bar{y}'} F_j(\bar{x}, \bar{y}')p(\bar{y}'|\bar{x}; w) \\ &= F_j(\bar{x}, \bar{y}) - \sum_{\bar{y}'} F_j(\bar{x}, \bar{y}') \frac{\exp \sum_{j'} w_{j'} F_{j'}(\bar{x}, \bar{y}')}{Z(\bar{x}, w)}.\end{aligned}$$

The first term $F_j(\bar{x}, \bar{y})$ is fast to compute because $\bar{x}$ and its training label $\bar{y}$ are fixed. Section 3 shows how to compute $Z(\bar{x}, w)$ efficiently. The remaining difficulty is to compute $\sum_{\bar{y}'} F_j(\bar{x}, \bar{y}') \exp \sum_j w_j F_j(\bar{x}, \bar{y}')$.

If the set of alternative labels $\{y\}$ is large, then it is computationally 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 $p(y|x; w)$. In

this section we discuss three different approximations. The corresponding training methods are called the Collins perceptron, Gibbs sampling, and contrastive divergence.

The Collins perceptron. Suppose we place all the probability mass on the most likely y value, i.e. we use the approximation $\hat{p}(y|x;w) = I(y = \hat{y})$ where $\hat{y} = \text{argmax}_y \ p(y|x;w)$ as before. Then the update rule (3) simplifies to the following rule:

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

Given a training example x, the label $\hat{y}$ can be thought of as an "impostor" compared to the genuine label y. The concept to be learned is those vectors of feature-function values $\langle F_1(x, y), \ldots \rangle$ that correspond to correct $\langle x, y \rangle$ pairs. The vector $\langle F_1(x, y), \ldots \rangle$, where $\langle x, y \rangle$ is a training example, is a positive example of this concept. The vector $\langle F_1(x, \hat{y}), \ldots \rangle$ is a negative example of the same concept. 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 $\hat{y}$.

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 $p(y_2|x, y'_1, y'_2, y_4, \ldots, y_n; w)$;
- and so on until y'_n.

(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 $1/s$.

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.

How to do Gibbs sampling. Gibbs sampling relies on drawing samples efficiently from marginal distributions. Let y_{-i} be an abbreviation for the set $\{y_1, \ldots, y_{i-1}, j_{i+1}, \ldots, y_n\}$. 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

$$p(y|x; w) \propto \prod_{m=1}^{M} \phi_m(y^m|x; w) \tag{4}$$

where each ϕ_m is a potential function that depends on just a subset y^m of components of y. Linear-chain conditional random fields are a special case of Equa-

tion (4). In this case

$$p(y_i|x, y_{-i}; w) \propto \prod_{m \in C} \phi_m(y^m|x; w) \tag{5}$$

where C indexes those potential functions y^m that include the 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 $y_{-i} = \{y_1, \ldots, y_{i-1}, i_{i+1}, \ldots, y_n\}$. 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.