Log-linear models and conditional random fields

Charles Elkan
elkan@cs.ucsd.edu

February 9, 2012

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. The examples can be drawn from any set X and the labels can be drawn from any finite set Y. In general an example x is not a real-valued vector, unlike before. The set Y can have very large cardinality, but it must be finite, so that sums, maximizations, and other aggregate operations over it are well-defined. A log-linear model posits that the probability of any particular label y, given the example x, is

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

Each expression $F_j(x, y)$ is called a feature function. Intuitively, $F_j(x, y)$ is a specific measure of the compatibility of the example x and the label y. Intuitively, each of the J feature functions measures a different type of compatibility. The corresponding parameter w_j, also called a weight, describes the influence of this feature function. If $w_j > 0$ then a positive value for the feature function makes y more likely as the true label of x, holding everything else fixed. Conversely, if

$w_j < 0$ then $F_j(x, y) > 0$ makes y less likely as the true label of x. If $w_j = 0$ then F_j is irrelevant as a predictor of y. Feature functions are defined in advance by a human, while weights are learned by a training algorithm. Feature functions are real-valued in general, but an important special case is when $F_j(x, y) \in \{0, 1\}$ for all x and y.

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

$$Z(x, w) = \sum_{y' \in Y} \exp \sum_{j=1}^{J} w_j F_j(x, y').$$

Here, we wrote y' to emphasize that it is not the same as y above. In the terminology of mathematical logic or of programming languages, y' is a local variable, so its name could be changed to any other name. The outer sum is over all members of the set 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

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

Note that if all weights are multiplied by the same nonzero constant, then the highest-probability label $\hat{y}$ is unchanged. However, the actual probabilities of different labels do depend on the absolute magnitudes of the weights, not just on their relative magnitudes.

Mathematically, log-linear models are simple: there is one real-valued weight for each feature function, no more and no fewer. The justification for the form of the righthand side of Equation 1 is similar to the justification for logistic regression. 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, and the division makes the results between 0 and 1, i.e. makes them be valid probabilities. Note that the ranking of the probabilities is the same as the ranking of the linear values.

In general, a function of the form

$$b(y) = \frac{\exp a(y)}{\sum_{y'} \exp a(y')}$$

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 $a(y)$ values compared to the smaller $a(y)$ 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 5 below.

2 Feature functions

As mentioned, in general a feature function can be any real-valued function of the data space X to which examples x belong and of the label space Y. Formally, a feature function is any mapping $F_j : X \times Y \to \mathbb{R}$. As a special case, a feature function can be Boolean, that is a mapping $F_j : X \times Y \to \{0, 1\}$.

Usually we do not define feature functions one at a time. Instead, we define classes of feature functions using templates of some sort. The integer j is then defined to range over all members of the classes. For example, suppose that $x \in \mathbb{R}^d = X$ and $y \in \{1, 2, \ldots, C\} = Y$. Then we can define a class of cardinality $J = dC$ of feature functions indexed from $j = 1$ to $j = J$ via the template

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

Here x_i is the ith component of the vector x and $j = i + (c - 1)d$, 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 x_i. 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 fixed concrete way of assigning a unique index j to each member of each class of feature functions.

The previous paragraph is just one example of how feature functions can be constructed. It corresponds to the multiclass version of logistic regression, which is often called multinomial logistic regression. The example has an important sparsity property that other sets of feature functions often have also: for any particular x and y the value of the great majority of feature functions is zero.

Often, a class of feature functions is defined as

$$F_j(x, y) = A_a(x)B_b(y) \tag{2}$$

where the subscript a indexes a set of functions of x, and the subscript b indexes a set of functions of y. The example above is a special case of this situation, with d different A_a functions and C different B_b functions. Often, the functions

A_a and B_b are binary presence/absence indicators. In this situation, the product $A_a(x)B_b(y)$ is a logical conjunction.

With log-linear models, anything and the kitchen sink can be used to define a feature function. A single feature function can involve many candidate label values y, and many attributes or components of x. 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

$$\begin{aligned} A_1(x) &= I(x \text{ starts with a capital letter}) \\ A_2(x) &= I(x \text{ starts with the letter G}) \\ A_3(x) &= I(x \text{ is the exact string ``Graham''}) \\ A_4(x) &= I(x \text{ is six letters long}) \end{aligned}$$

and so on. 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 look at conditional random fields (CRFs), specifically so-called linear-chain CRFs. First, we present linear-chain CRFs through an example application. Next, Section 4 explains the special algorithm that makes inference tractable for linear-chain CRFs. Section 5 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 quite 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 POS taggers with highest accuracy currently use over 100,000 feature functions. Of course, these feature functions are defined via templates, not one at a time. 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). The word "structured" refers to the fact that labels have internal structure, in this case being sequences. POS tagging is a difficult task that is 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 words and let $\bar{y}$ be a corresponding sequence of 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

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

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

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

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. The

notation above indicates 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 START is a special fixed tag value. Similarly, if necessary we assume that $y_{n+1} = \text{STOP}$. Each low-level feature function must be well-defined for all tag values in positions 0 and $n + 1$.

A low-level feature function f_j may depend on only a subset of its four allowed arguments. Examples of legal low-level feature functions are "the current tag is NOUN and the current input word is capitalized," "the first input word is Mr. and the second tag is PROPER NOUN," and "the previous tag is SALUTATION and the current tag is PROPER NOUN."

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

$$\hat{y} = \text{argmax}_{\bar{y}}\ p(\bar{y}|\bar{x}; w) \tag{3}$$

for each training example $\bar{x}$. However, before we can talk about training there is a major inference problem to solve. How can we do the argmax computation in Equation 3 efficiently, for any $\bar{x}$ and any parameter vector w? This computation is difficult since the number of alternative tag sequences $\bar{y}$ is exponential. We need a trick in order to consider 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 this trick exist. The next section explains how to solve the inference problem 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 log conditional likelihood (LCL), with regularization, on the training data. However, instead of maximizing LCL we could maximize yes/no accuracy of the entire predicted $\hat{y}$, 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 an objective that depends only on a single predicted $\hat{y}$. Instead, we might want to maximize an objective that depends on multiple predictions. 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.

4 Inference algorithms for linear-chain CRFs

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

$$\hat{y} = \text{argmax}_{\bar{y}}\ p(\bar{y}|\bar{x}; w) = \text{argmax}_{\bar{y}}\ \sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}).$$

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

$$\begin{aligned} \hat{y} &= \text{argmax}_{\bar{y}}\ \sum_{j=1}^{J} w_j \sum_{i=1}^{n} f_j(y_{i-1}, y_i, \bar{x}, i) \\ &= \text{argmax}_{\bar{y}}\ \sum_{i=1}^{n} g_i(y_{i-1}, y_i) \end{aligned} \tag{4}$$

where we define

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

for $i = 1$ to $i = n$. 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.

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. Here, score means the sum in Equation 4 taken from $i = 1$ to $i = k$. This is a maximization over $k - 1$ tags because tag number k is fixed to have value v. Formally,

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

Expanding the equation above gives

$$U(k, v) = \max_{y_{k-1}} \max_{y_1, \ldots, y_{k-2}} \sum_{i=1}^{k-2} g_i(y_{i-1}, y_i) + g_{k-1}(y_{k-2}, y_{k-1}) + g_k(y_{k-1}, v).$$

Writing u instead of y_{k-1} gives a recursive relationship 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.

After the U matrix has been filled in for all k and v, the final entry in the optimal output sequence $\hat{y}$ can be computed as $\hat{y}_n = \text{argmax}_v U(n, v)$. Each previous entry can then be computed as

$$\hat{y}_{k-1} = \text{argmax}_u [U(k-1, u) + g_k(u, \hat{y}_k)].$$

Note that $\hat{y}$ must be computed from right to left, and this can be done only after the whole U matrix has been filled in from left to right.

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

Notice that it is the length of $\bar{y}$, not the length of $\bar{x}$, that is important. The input $\bar{x}$ in fact does not even need to be a sequence, because it is treated as a unit. 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, each $\bar{y}$ 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.

5 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 logarithm of the conditional likelihood (LCL) of the set of training examples. Since we want to maximize LCL, we do gradient ascent as opposed to descent.

For stochastic gradient ascent (also called online gradient ascent) we update parameters based on single training examples. Therefore, we evaluate the partial derivative of the LCL for a single training example, with respect to each w_j. The partial derivative of the LCL is

$$\begin{aligned}\frac{\partial}{\partial w_j}\log p(y|x;w) &= F_j(x,y) - \frac{\partial}{\partial w_j}\log Z(x,w) \\ &= F_j(x,y) - \frac{1}{Z(x,w)}\frac{\partial}{\partial w_j}Z(x,w).\end{aligned}$$

Above, y is the known true label of the training example x, and j is the index of the parameter for which the partial derivative is being computed. The bar notation for x and y is not used, because the derivations in this section are valid for all log-linear models, not just for conditional random fields. Also note that the derivations allow feature functions to be real-valued; they are not restricted to being binary.

Expanding the partition function $Z(x,w)$ gives

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

where the sum over y' is a sum over all candidate labels, inside which there is a sum over all feature functions $F_{j'}$. Simplifying yields

$$\begin{aligned}\frac{\partial}{\partial w_j}Z(x,w) &= \sum_{y'}\frac{\partial}{\partial w_j}[\exp\sum_{j'} w_{j'}F_{j'}(x,y')] \\ &= \sum_{y'}[\exp\sum_{j'} w_{j'}F_{j'}(x,y')]\frac{\partial}{\partial w_j}[\sum_{j'} w_{j'}F_{j'}(x,y')] \\ &= \sum_{y'}[\exp\sum_{j'} w_{j'}F_{j'}(x,y')]F_j(x,y').\end{aligned}$$

So, the partial derivative of the LCL is

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

Now, note that

$$\frac{\exp \sum_{j'} w_{j'} F_{j'}(x, y')}{Z(x, w)} = p(y'|x; w)$$

so

$$\begin{aligned} \frac{\partial}{\partial w_j} \log p(y|x; w) &= F_j(x, y) - \sum_{y'} F_j(x, y') p(y'|x; w) \\ &= F_j(x, y) - E_{y' \sim p(y'|x;w)}[F_j(x, y')]. \end{aligned} \tag{5}$$

In words, the partial derivative with respect to parameter number j for training example $\langle x, y \rangle$ is the value of feature function j for x and y, minus the weighted average value of the feature function for x and all possible labels y', where the weight inside the average of y' is its conditional probability given x.

The gradient of the LCL 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

$$\sum_{\langle x,y \rangle \in T} F_j(x, y) = \sum_{\langle x,\cdot \rangle \in T} E_{y \sim p(y|x;w)}[F_j(x, y)]. \tag{6}$$

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 lefthand side of Equation 6 is the total value (mass) of feature function j on the whole training set. The righthand 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 derivatives that are given symbolically by Equation 5. Obviously the difficult aspect is to evaluate the expectations. 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 LCL objective function. Second, after doing the optimization, check for each feature function F_j that both sides of Equation 6 are numerically equal.

6 Forward and backward vectors and their uses

This section explains how to evaluate partial derivatives efficiently for linear-chain CRFs. Consider all possible unfinished sequences of tags that end at position k with tag v. The unnormalized probability of this set is called its score $\alpha(k, v)$. For each k, $\alpha(k, v)$ is a vector of length m with a component for each tag value v. This vector is called a forward vector.

The unnormalized probability of all complete sequences y_1 to y_n is

$$Z(\bar{x}, w) = \sum_{\bar{y}} \exp \sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}) = \sum_{\bar{y}} \exp \sum_{j=1}^{J} w_j \sum_{i=1}^{n} f_j(y_{i-1}, y_i, \bar{x}, i).$$

Remember the definition

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

It follows that

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

Correspondingly,

$$\alpha(k+1, v) = \sum_{y_1, \ldots, y_k} \exp[\sum_{i=1}^{k} g_i(y_{i-1}, y_i) + g_{k+1}(y_k, v)].$$

Rewriting,

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

We have the recursive definition

$$\alpha(k+1, v) = \sum_{u} \alpha(k, u)[\exp g_{k+1}(u, v)].$$

The summation is over all possible tag values u in position k. In words, the total probability of all tag sequences ending with v is the sum of the total probability

of all shorter tag sequences ending with u, times the probability of v following u. The base case is

$$\alpha(0, y) = I(y = \text{START}).$$

The first use of the forward vectors is to write

$$Z(\bar{x}, w) = \sum_v \alpha(n, v).$$

which lets us evaluate the partition function in polynomial time.

The backward vector for position k captures the unnormalized probabilities of partial sequences starting at that position. For each tag value u and position k, the backward vector is

$$\beta(u, k) = \sum_v [\exp g_{k+1}(u, v)]\beta(v, k + 1)$$

with base case

$$\beta(u, n + 1) = I(u = \text{STOP}).$$

The difference in the order of arguments between $\alpha(k, v)$ and $\beta(u, k)$ is a reminder that in the former the partial sequence ends with v, while in the latter the partial sequence starts with u. We can compute the partition function using the backward vector also:

$$Z(\bar{x}, w) = \beta(\text{START}, 0) = \sum_v [\exp g_1(\text{START}, v)]\beta(v, 1).$$

Verifying that $Z(\bar{x}, w)$ is numerically the same using the forward vectors and using the backward vectors is one way to check that they have been computed correctly.

The forward and backward vectors have many uses. For example, the probability of a particular tag u at position k, summing over all possibilities for all other positions, is

$$p(Y_k = u|\bar{x}; w) = \frac{\alpha(k, u)\beta(u, k)}{Z(\bar{x}, w)}.$$

Note that for all positions k it should be true that

$$\sum_u \alpha(k, u)\beta(u, k) = Z(\bar{x}, w).$$

The probability of the specific tags u and v at positions k and $k+1$ is

$$p(Y_k = u, Y_{k+1} = v|\bar{x}; w) = \frac{\alpha(k,u)[\exp g_{k+1}(u,v)]\beta(v,k+1)}{Z(\bar{x},w)}.$$

For training, we need to compute the expectation $E_{\bar{y}\sim p(\bar{y}|\bar{x};w)}[F_j(\bar{x},\bar{y})]$ where $\bar{y}$ ranges over all labels, that is over all entire tag sequences. This is

$$\begin{aligned} E_{\bar{y}\sim p(\bar{y}|\bar{x};w)}[F_j(\bar{x},\bar{y})] &= E_{\bar{y}\sim p(\bar{y}|x;w)}[\sum_{i=1}^{n} f_j(y_{i-1},y_i,\bar{x},i)] \\ &= \sum_{i=1}^{n} E_{\bar{y}\sim p(\bar{y}|\bar{x};w)}[f_j(y_{i-1},y_i,\bar{x},i)]. \end{aligned}$$

For each position i, we do not need to take the expectation over all $\bar{y}$. Instead, we can compute the expectation just over all tags y_{i-1} and y_i at positions $i-1$ and i within the label $\bar{y}$. That is,

$$\begin{aligned} E_{\bar{y}\sim p(\bar{y}|\bar{x};w)}[F_j(\bar{x},\bar{y})] &= \sum_{i=1}^{n} E_{y_{i-1},y_i}[f_j(y_{i-1},y_i,\bar{x},i)] \\ &= \sum_{i=1}^{n}\sum_{y_{i-1}}\sum_{y_i} p(y_{i-1},y_i|\bar{x};w) f_j(y_{i-1},y_i,\bar{x},i) \\ &= \sum_{i=1}^{n}\sum_{y_{i-1}}\sum_{y_i} f_j(y_{i-1},y_i,\bar{x},i)\frac{\alpha(i-1,y_{i-1})[\exp g_i(y_{i-1},y_i)]\beta(y_i,i)}{Z(\bar{x},w)}. \end{aligned}$$

The last equation above gives an $O(nm^2)$ time method of computing the partial derivatives needed for gradient following.

7 Stochastic gradient ascent

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

$$w_j := w_j + \lambda(F_j(x,y) - E_{y'\sim p(y'|x;w)}[F_j(x,y')]) \tag{7}$$

where λ is a learning rate. If the log-linear model is a CRF, then the expectation in Equation 7 is computed using forward and backward vectors as described in the previous section.

Suppose that, as in Equation 2, every feature function F_j is the product of a function $A_a(x)$ of x only and a function $B_b(y)$ of y only. Then $\frac{\partial}{\partial w_j} \log p(y|x; w) = 0$ if $A_a(x) = 0$, regardless of y. This implies that with stochastic gradient ascent, for each example x parameters must be updated *only* for feature functions for which $A_a(x) \neq 0$. Not updating other parameters can be a great saving of computational effort.

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.

It is instructive to work out the time complexity of stochastic gradient training. In the update to each parameter, the expectation is computed using forward and backward vectors. Computing the g_i matrices for $i = 1$ to $i = n$ for one training example takes time $O(Jm^2n)$ time ignoring sparsity. Computing the forward and backward vectors takes $O(m^2n)$ time. These and the g_i matrices are the same for all j, given x and the current parameter values. They do not depend on the training label y. A different expectation must be computed for each j. Computing each one of these requires $O(m^2n)$ time. For each j, doing the actual update after computing the expectation takes only constant time. Putting everything together, the total time complexity of the updates for all j, for a single training x and its label y, is $O(Jm^2n)$. Interestingly, this is the same as the order-of-magnitude complexity of computing the highest-probability prediction $\hat{y}$. Therefore, stochastic gradient training is not more expensive than the Collins perceptron described below.

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 sampling, and the third is a heuristic called contrastive divergence.

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

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

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, and no special trick is available, then it is computationally expensive to evaluate the second term that is

the expectation $E[F_j(x, y')|y' \sim p(y'|x; w)]$. 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 approximation.

9 The Collins perceptron

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

$$\hat{p}(y|x; w) = I(y = \hat{y}) \quad \text{where} \quad \hat{y} = \text{argmax}_y \, p(y|x; w).$$

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

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

For a given training example $\langle x, y \rangle$, this rule is applied for every weight w_j. Given a training example x, the label $\hat{y}$ can be thought of as an "impostor" compared to the genuine label y.

The simplified update rule 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 to classify 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.

One update by 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.

As mentioned in Section 1, if all weights are multiplied by the same nonzero constant, then which label $\hat{y}$ has highest probability is unchanged. The Collins perceptron method relies only on the identity of $\hat{y}$, and not on its probability, so the method will give the same behavior regardless of the value of the learning rate λ, assuming that it is constant. Therefore, we can fix $\lambda = 1$.

10 Gibbs sampling

Computing the most likely label $\hat{y}$ does not require computing the partition function $Z(x, w)$, or any derivatives. Nevertheless, sometimes identifying $\hat{y}$ is still too difficult. In this case one option for training is to estimate the expectations $E_{y \sim p(y|x;w)}[F_j(x, y)]$ approximately by sampling y values from their 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 that the entire label y can be written as a set of parts $y = \{y_1, \ldots, y_n\}$. A linear-chain CRF is an obvious special case of this situation. Suppose also that 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) Replace $\{y_1, \ldots, y_n\}$ by $\{y'_1, \ldots, y'_n\}$ 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 reduce dependence on the starting point, 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 more or less 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 $1/s$. The next section describes a more efficient application for Gibbs sampling, while the rest of this section explains how to do Gibbs sampling efficiently, regardless of what the samples are used for.

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.

We have

$$p(v|x, y_{-i}; w) = \frac{p(v, y_{-i}|x; w)}{\sum_{v'} p(v', y_{-i}|x; w)}$$

Any value v for y_i and y_{-i} together are an entire label. Let y be v combined with y_{-i} and let y' be v' combined with y_{-i}. For a log linear model we get

$$p(v|x, y_{-i}; w) = \frac{[\exp \sum_j w_j F_j(x, y)]/Z(x, w)}{\sum_{v'} [\exp \sum_j w_j F_j(x, y')]/Z(x, w)}.$$

The partition function $Z(x, w)$ can be canceled, giving

$$p(v|x, y_{-i}; w) = \frac{\prod_j \exp w_j F_j(x, y)}{\sum_{v'} \prod_j \exp w_j F_j(x, y')}.$$

A further simplification is also often possible. Suppose that for some j the feature function $F_j(x, y)$ does not depend on y_i, meaning that $F_j(x, y) = F_j(x, y')$. All such factors can be canceled also.

For linear chain CRFs, following Equation 4,

$$p(y|x; w) = \frac{\exp \sum_{i=1}^{n} g_i(y_{i-1}, y_i)}{Z(x, w)}.$$

Therefore

$$p(v|x, y_{-i}; w) = \frac{[\exp g_i(y_{i-1}, v)][\exp g_{i+1}(v, y_{i+1})]}{\sum_{v'} [\exp g_i(y_{i-1}, v')][\exp g_{i+1}(v', y_{i+1})]}.$$

After the g_i matrices have been computed and stored, the time cost of evaluating the equation above for all v, for a single i, is just $O(m)$.

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.

1. Hanna M. Wallach. Conditional Random Fields: An Introduction. Technical Report MS-CIS-04-21. Department of Computer and Information Science, University of Pennsylvania, 2004.

2. Charles Sutton and Andrew McCallum. An Introduction to Conditional Random Fields for Relational Learning. In Introduction to Statistical Relational Learning. Edited by Lise Getoor and Ben Taskar. MIT Press, 2006.

3. Rahul Gupta. Conditional Random Fields. Unpublished report, IIT Bombay, 2006.

4. Roland Memisevic. An Introduction to Structured Discriminative Learning. Technical Report, University of Toronto, 2006.

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.

1. John D. Lafferty, Andrew McCallum, Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data, In Proceedings of the 18th International Conference on Machine Learning (ICML), 2001, pp. 282-289.

2. Michael Collins. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing, pp. 1-8, 2002.

3. Sham Kakade, Yee Whye Teh, Sam T. Roweis. An alternate objective function for Markovian fields. In Proceedings of the 19th International Conference on Machine Learning (ICML), 2002.

4. Andrew McCallum. Efficiently inducing features of conditional random fields. In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI-2003), 2003.

5. Sanjiv Kumar and Martial Hebert. Discriminative random fields: A discriminative framework for contextual interaction in classification. In Proceedings of the Ninth IEEE International Conference on Computer Vision, 2003.

6. Ben Taskar, Carlos Guestrin and Daphne Koller. Max-margin Markov networks. In Advances in Neural Information Processing Systems 16 (NIPS), December 2003.

7. Thomas G. Dietterich, Adam Ashenfelter and Yaroslav Bulatov. Training conditional random fields via gradient tree boosting. In Proceedings of the 21st International Conference on Machine Learning (ICML), 2004.

8. Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts? In IEEE Transactions on Pattern Analysis and Machine Intelligence, February 2004.

9. Charles Sutton, Andrew McCallum. Collective segmentation and labeling of distant entities in information extraction. ICML Workshop on Statistical Relational Learning, 2004.

10. Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, Yasemin Altun. Large margin methods for structured and interdependent output variables. Journal of Machine Learning Research, December 2005.

11. Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. Submitted for publication, 2006.

12. Samuel Gross, Olga Russakovsky, Chuong Do, and Serafim Batzoglou. Training conditional random fields for maximum labelwise accuracy. In Advances in Neural Processing Systems 19 (NIPS), December 2006.

CSE 250B Quiz 7, February 18, 2010

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 Quiz 8, February 25, 2010

[3 points] What is the order-of-magnitude time complexity of the Collins perceptron algorithm for training a CRF model?

Use the following notation:

- J is the number of low-level feature functions,
- m is the number of alternative tags,
- n is the length of each training sequence (assume that all sequences have the same length),
- S is the number of training examples,
- T is the number of epochs of training.

CSE 250B Quiz 6, February 10, 2011

The basic equation for a log-linear model is

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

This model can be used for multilabel classification, where an example x is a vector in $\mathbb{R}^d$ and a label y is a subset of the finite set $A = \{a_1, a_2, \ldots, a_K\}$. Suppose that some feature functions are of the form

$$F_j(x,y) = x_i I(a_k \in y)$$

while other feature functions are of the form

$$F_j(x,y) = I(a_{k_1} \in y) I(a_{k_2} \in y).$$

How many feature functions are there of the second form? What is the purpose and meaning of these feature functions?

CSE 250B Quiz 7, February 17, 2011

In today's *Financial Times*, Richard Waters discusses the IBM system named Watson that just won the quiz show called Jeopardy. He writes "Rather than just picking the highest-ranked result, it [Watson] ascribes a probability to whether a given answer is correct – a potentially useful technique for real-world situations, where things are seldom black and white."

Explain the author's technical misunderstanding when he contrasts "the highest-ranked result" with "a probability."

Here is an optional no-credit additional question for those who watched the show. In fact, it is not exactly true that Watson "ascribes a probability to whether a given answer is correct." Explain how you can know that this is not exactly true from watching the show.

CSE 250B project assignment 4, due in class on March 11, 2010

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.

CSE 250B project assignment 3, due in class on March 1, 2011

As before, for this project 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 lecture on Tuesday March 1.

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 segment novel Zulu words correctly. The available dataset consists of 10,000 Zulu words with correct segmentations indicated. The word list is available at `http://cseweb.ucsd.edu/~elkan/250B/ZuluWordList.txt`, copied from `http://www.cs.bris.ac.uk/Research/MachineLearning/Morphology/Resources/Ukwabelana/2010.07.17.WordListSegmented.txt`.

You should define and implement 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.

Define and implement your own set of feature functions that identify characteristics of words that are potentially relevant. 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. 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.

Implement and do experiments with two different training methods, namely Collins' perceptron algorithm and one of the following:

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

For each training method, you should implement all needed CRF-specific algorithms yourself. The perceptron training approach needs only the Viterbi algorithm to be working, while contrastive divergence needs you to understand and im-

plement Gibbs sampling. Nonlinear optimization and stochastic gradient following 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 basic algorithms are implemented correctly. Your report must convince the reader that this is true. For methods that use gradients, use a function such as the one available at `http://people.csail.mit.edu/jrennie/matlab/checkgrad2.m` to verify that computed derivatives are correct. After you are sure that your algorithm implementations are correct, it is also vital to make them fast. Use a small random subset of words and the Matlab profiler to maximize the speed of your Matlab code.

After you have a correct, fast and stable implementation of a learning algorithm, tune its settings 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 segment boundary occurs after this letter.

Do experiments with two different methods of coding segment boundaries. The first coding method uses the tag 1 to indicate that a letter is the last one in a segment, and the tag 0 otherwise. The second coding method uses the tag 1 for the first letter in a segment, the tag 2 for the second letter, 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.

There are 107 words in the corpus that have more than one correct segmentation. Ignoring these, it is possible to achieve letter-level precision and recall both over 0.94, measured using ten-fold cross-validation.

CSE 250B project assignment 2, due on February 16, 2012

As before, for this project 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 lecture on Thursday February 16.

The objective of this project is to understand the basic algorithms for conditional random fields (CRFs) thoroughly, by implementing them yourself. The experimental task is to learn a CRF model that can divide novel English words into syllables correctly. The dataset, which consists of 66,001 English words with syllables separated by hyphens, is available at `http://www.cs.ucsd.edu/users/elkan/hyphenation/`.

Design a method for generating feature functions automatically from high-level specifications such as "all consecutive sequences of four specific letters." In your report, explain this method. The implementation of the method does not have to be general-purpose, that is the code does not need to be usable for other applications. However, the method itself should be systematic, meaning that a relatively small program should enumerate all feature functions in some large classes. Make sure that each feature function has value zero most of the time, and that this common case is handled efficiently both during data generation and during CRF training.

Use the BIO method of coding syllables, where the tag B for "begin" indicates that a letter is the first one in its syllable, the tag O for "out" indicates that a letter is the last one in its syllable, and the tag I for "in" indicates that a letter is in the middle of its syllable. Make sensible decisions about what tags to use or positions immediately before and after a word.

Implement and do experiments with two different training methods, namely Collins' perceptron algorithm and one of the following:

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

For each training method, you should implement all needed CRF-specific algorithms yourself. The perceptron training approach needs only the Viterbi algorithm to be working, while contrastive divergence needs you to understand and implement Gibbs sampling. Nonlinear optimization and stochastic gradient following 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 basic algorithms are implemented correctly. Your report must convince the reader that this is true. For methods that use gradients, use a function such as the one available at `http://people.csail.mit.edu/jrennie/matlab/checkgrad2.m` to verify that computed derivatives are correct. After you are sure that your algorithm implementations are correct, it is also vital to make them fast. Use a small random subset of words and the Matlab profiler to maximize the speed of your Matlab code.

After you have a correct, fast and stable implementation of a learning algorithm, tune its settings using cross-validation. Use word-level accuracy as the performance metric. This is simply the fraction of words that a method gets exactly right.