Log-linear models and conditional random fields

Notes for a tutorial at CIKM'08

Charles Elkan
elkan@cs.ucsd.edu

October 20, 2008

Contents

Chapter 1

Likelihood and logistic regression

Logistic regression is the simplest example of a log-linear model, so this section examines logistic regression in detail. All log-linear models are based on the idea of maximizing likelihood, so we shall discuss that general idea first of all.

1.1 Principle of maximum likelihood

Consider a family of probability distributions defined by a set of parameters θ. The distributions may be either probability mass functions (pmfs) or probability density functions (pdfs). Suppose we have a random sample drawn from a fixed but unknown member of this family. The random sample is a training set of n examples x_1 to x_n. We assume that the examples are independent so the probability of the set is the product of the probabilities of the individual examples:

$$f(x_1, \ldots, x_n; \theta) = \prod_j f_\theta(x_j; \theta).$$

Usually we think of the distribution θ as fixed and the examples x_j as unknown, or varying. However, we can think of the training data as fixed and consider alternative parameter values. This is the point of view behind the definition of the likelihood function:

$$L(\theta; x_1, \ldots, x_n) = f(x_1, \ldots, x_n; \theta).$$

Note that if $f(x; \theta)$ is a probability mass function, then the likelihood is always less than one, but if $f(x; \theta)$ is a probability density function, then the likelihood can be greater than one, since densities can be greater than one.

The principle of maximum likelihood says that we should use as our model the distribution $f(\cdot;\hat{\theta})$ that gives the greatest possible probability to the training data. Formally,

$$\hat{\theta} = \text{argmax}_\theta L(\theta; x_1, \ldots, x_n).$$

This value $\hat{\theta}$ is called the maximum likelihood estimator (MLE) of θ. Note that in general each x_j is a vector of values, and θ is a vector of real-valued parameters. For example, for a Gaussian distribution $\theta = \langle \mu, \sigma^2 \rangle$.

Notational note: In the expression $p(y|x;\beta)$ the semicolon indicates that β is a parameter, not a random variable that is being conditioned on, even though it is to the right of the vertical bar. Viewed as a mapping, this expression is simply a function of three arguments. Viewed as a probability, it is a property of two random variables. In a Bayesian framework, parameters are also viewed as random variables, and one can write expressions such as $p(\beta|x)$. We are not doing a Bayesian analysis, so we indicate that β is not a random variable.

1.2 Maximum likelihood for Bernoulli distributions

As a first example of finding a maximum likelihood estimator, consider the parameter of a Bernoulli distribution. A random variable with this distribution is a formalization of a coin toss. The value of the random variable is 1 with probability θ and 0 with probability $1-\theta$. Let X be a Bernoulli random variable. We have

$$P(X = x) = \begin{array}{c} \theta \text{ if } x = 1 \\ 1-\theta \text{ if } x = 0 \end{array}.$$

For mathematical convenience write this as

$$P(X = x) = \theta^x (1-\theta)^{1-x}.$$

Suppose the training data are x_1 through x_n where each $x_j \in \{0, 1\}$. We maximize the likelihood function

$$L(\theta; x_1, \ldots, x_n) = f(x_1, \ldots, x_n; \theta) = \theta^h (1-\theta)^{n-h}$$

where $h = \sum_i x_i$. The maximization is over the possible values $0 \leq \theta \leq 1$.

We can do the maximization by setting the derivative with respect to θ equal to zero. The derivative is

$$\begin{aligned} \frac{\partial}{\partial p}\theta^h (1-\theta)^{n-h} &= h\theta^{h-1}(1-\theta)^{n-h} + \theta^h (n-h)(1-\theta)^{n-h-1}(-1) \\ &= \theta^{h-1}(1-\theta)^{n-h-1}[h(1-\theta) - (n-h)\theta] \end{aligned}$$

which has solutions $\theta = 0$, $\theta = 1$, and $\theta = h/n$. The solution which is a maximum is clearly $\theta = h/n$ while $\theta = 0$ and $\theta = 1$ are minima. So we have the maximum likelihood estimate $\hat{\theta}_{\text{MLE}} = h/n$.

The log likelihood function is simply the logarithm of the likelihood function. Because logarithm is a monotonic strictly increasing function, maximizing the log likelihood is precisely equivalent to maximizing the likelihood, or to minimizing the negative log likelihood.

1.3 Conditional likelihood

An important extension of the idea of likelihood is *conditional* likelihood. The conditional likelihood of θ given data x and y is $L(\theta; y|x) = f(y|x; \theta)$. Intuitively, y follows a probability distribution that is different for different x, but x itself is never unknown, so there is no need to have a probabilistic model of it. Technically, for each x there is a different distribution $f(y|x; \theta)$ of y, but all these distributions share the same parameters θ.

Given training data consisting of $\langle x_i, y_i \rangle$ pairs, the principle of maximum conditional likelihood says to choose a parameter estimate $\hat{\theta}$ that maximizes the product $\prod_i f(y_i|x_i; \theta)$. Note that we do not need to assume that the x_i are independent in order to justify the conditional likelihood being a product; we just need to assume that the y_i are independent conditional on the x_i. For any specific value of x, $\hat{\theta}$ can be used to predict values for y; we assume that we never want to predict values of x.

1.4 Logistic regression

If y is a binary outcome and x is a real-valued vector, then the conditional model

$$p = p(y|x; \alpha, \beta) = \frac{1}{1 + \exp -[\alpha + \sum_{j=1}^{d} \beta_j x_j]}$$

is called logistic regression. We use j to index over the feature values x_1 to x_d of a single example of dimensionality d, since we use i below to index over training examples 1 to n.

The logistic regression model is easier to understand in the form

$$\log \frac{p}{1-p} = \alpha + \sum_j \beta_j x_j.$$

The ratio $p/(1-p)$ is called the odds of the event y given x, and $\log[p/(1-p)]$ is called the log odds. Since probabilities range between 0 and 1, odds range between 0 and $+\infty$ and log odds range unboundedly between $-\infty$ and $+\infty$. A linear expression of the form $\alpha + \sum_j \beta_j x_j$ can also take unbounded values, so it is reasonable to use a linear expression as a model for log odds, but not as a model for odds or for probabilities. Essentially, logistic regression is the simplest possible model for a random yes/no outcome that depends linearly on predictors x_1 to x_d.

For each feature j, $\exp(\beta_j x_j)$ is a multiplicative scaling factor on the odds $p/(1-p)$. If the predictor x_j is binary, then $\exp(\beta_j)$ is the extra odds of having the outcome $y = 1$ when $x_j = 1$, compared to when $x_j = 0$.

Note that it is acceptable, and indeed often beneficial, to include a large number of features in a logistic regression model. Some features may be derived, i.e. computed as deterministic functions of other features. One great advantage of logistic regression in comparison to other classifiers is that the training process will find optimal coefficients for features regardless of whether the features are correlated. Other learning methods, in particular naive Bayes, do not work well when the feature values of training or test examples are correlated.

A second major advantage of logistic regression is that it gives well-calibrated probabilities. The numerical values $p(y = 1|x)$ given by a logistic regression model are not just scores where a larger score means that the example x is more likely to have label $y = 1$; they are meaningful conditional probabilities. This implies that given a set of n test examples with numerical predictions v_1 to v_n, the number of examples in the set that are truly positive will be close to $\sum_{i=1}^{n} v_i$, whatever this sum is.

Last but not least, a third major advantage of logistic regression is that it is not sensitive to unbalanced training data. What this means is that even if one class (either the positive or negative examples) is much larger than the other (correspondingly, the negative or positive examples), logistic regression training encounters no difficulties and the final classifier will still be well-calibrated. The conditional probabilities predicted by the trained classifier will range below and above the base rate, i.e. the unconditional probability $p(y = 1)$.

Chapter 2

Stochastic gradient training

All training algorithms for log-linear models are based on the gradient of the conditional likelihood function, or on a closely related idea. The simplest of these training algorithms, which are often the fastest and most useful in practice, use the gradient computed from one training example at a time. These algorithms are called stochastic gradient methods.

2.1 Logistic regression gradient

We shall continue with the special case of logistic regression. Given a single training example that consists of x and y values, the conditional log likelihood is $\log L(\beta; x, y) = \log p$ if $y = 1$ and $\log L(\beta; x, y) = \log(1 - p)$ if $y = 0$. The goal of training is to maximize the conditional log likelihood. So, let us evaluate its partial derivative with respect to each parameter β_j. To simplify the following discussion, assume that $\alpha = \beta_0$ and $x_0 = 1$ for every example x from now on. If $y = 1$ the partial derivative is

$$\frac{\partial}{\partial \beta_j} \log p = \frac{1}{p} \frac{\partial}{\partial \beta_j} p$$

while if $y = 0$ it is

$$\frac{\partial}{\partial \beta_j} \log(1 - p) = \frac{1}{1 - p} \Big(- \frac{\partial}{\partial \beta_j} p \Big).$$

Let $e = \exp[-\sum_j \beta_j x_j]$ where the sum ranges from $j = 0$ to $j = d$, so $p = 1/(1 + e)$ and $1 - p = (1 + e - 1)/(1 + e) = e/(1 + e)$. With this notation we

have

$$\begin{aligned}
\frac{\partial}{\partial \beta_j} p &= (-1)(1+e)^{-2} \frac{\partial}{\partial \beta_j} e \\
&= (-1)(1+e)^{-2}(e) \frac{\partial}{\partial \beta_j} [-\sum_j \beta_j x_j] \\
&= (-1)(1+e)^{-2}(e)(-x_j) \\
&= \frac{1}{1+e} \frac{e}{1+e} x_j \\
&= p(1-p)x_j.
\end{aligned}$$

So $(\partial/\partial\beta_j) \log p = (1-p)x_j$ and $(\partial/\partial\beta_j) \log(1-p) = -px_j$. Given training examples $\langle x_1, y_1 \rangle$ to $\langle x_n, y_n \rangle$, the total partial derivative of the log likelihood with respect to β_j is

$$\sum_{i:y_i=1} (1-p_i)x_{ij} + \sum_{i:y_i=0} -p_i x_{ij} = \sum_i (y_i - p_i)x_{ij}$$

where x_{ij} is the value of the jth feature of the ith training example. Setting the total partial derivative to zero yields

$$\sum_i y_i x_{ij} = \sum_i p_i x_{ij}.$$

We have one equation of this type for each parameter β_j. The equations can be used to check the correctness of a trained model.

2.2 Gradient ascent, one example at a time

There are several sophisticated ways of actually doing the maximization of the total conditional log likelihood, i.e. the conditional log likelihood summed over all training examples $\langle x_i, y_i \rangle$. However, here we consider a method called stochastic gradient ascent. This method changes the parameter values to increase the log likelihood based on one example at a time. It is called stochastic because the derivative based on a randomly chosen single example is a random approximation to the true derivative based on all the training data.

Consider a single training example $\langle x, y \rangle$, where again we drop the subscript i for convenience. Consider the jth parameter for $0 \leq j \leq d$. The partial derivative

of the log likelihood given this single example is

$$\frac{\partial}{\partial \beta_j} \log L(\beta; x, y) = (y - p)x_j$$

where $y = 1$ or $y = 0$. For each j, we increase the log likelihood incrementally by doing the update $\beta_j := \beta_j + \lambda(y - p)x_j$. Here λ is a multiplier called the learning rate that controls the magnitude of the changes to the parameters.

Stochastic gradient ascent (or descent, for a minimization problem) is a method that is often useful in machine learning. Experience suggests some heuristics for making it work well in practice.

- The training examples are sorted in random order, and the parameters are updated for each example sequentially. One complete update for every example is called an epoch. Typically, a small constant number of epochs is used, perhaps 3 to 100 epochs.

- The learning rate is chosen by trial and error. It can be kept constant across all epochs, e.g. $\lambda = 0.1$ or $\lambda = 1$, or it can be decreased gradually as a function of the epoch number.

- Because the learning rate is the same for every parameter, it is useful to scale the features x_j so that their magnitudes are similar for all j. Given that the feature x_0 has constant value 1, it is reasonable to normalize every other feature to have mean zero and variance 1, for example.

For the state of the art in guidelines for applying the stochastic gradient idea, see `http://leon.bottou.org/p̃rojects/sgd`.

2.3 Properties of stochastic gradient training

Stochastic gradient ascent (or descent) has some properties that are very useful in practice. First, suppose that $x_j = 0$ for most features j of a training example x. Then updating β_j based on x can be skipped. This means that the time to do one epoch is $O(nfp)$ where n is the number of training examples, p is the number of features, and f is the average number of nonzero feature values per example. If an example x is the bag-of-words representation of document, then p is the size of the vocabulary but fp is the average length of a document.

Second, suppose that the number n of training examples is very large, as is the case in many modern applications. Then, a stochastic gradient method may converge to good parameter estimates in less than one epoch of training. In contrast, a training method that computes the log likelihood of all data and uses this in the same way regardless of n will be inefficient in how it uses the data.

For each example, a stochastic gradient method updates all parameters once. The dual idea is to update one parameter at a time, based on all examples. This method is called coordinate ascent (or descent). For feature j the update rule is

$$\beta_j := \beta_j + \lambda \sum_i (y_i - p_i) x_{ij}.$$

The update for the whole parameter vector $\bar{\beta}$ is

$$\bar{\beta} := \bar{\beta} + \lambda (\bar{y} - \bar{p})^T X$$

where the matrix X is the entire training set and the column vector $\bar{y}$ consists of the 0/1 labels for every training example. Often, coordinate ascent converges too slowly to be useful. However, it can be useful to do one update of $\bar{\beta}$ after all epochs of stochastic gradient ascent.

Regardless of the method used to train a model, it is important to remember that optimizing the model perfectly on the training data usually does not lead to the best possible performance on test examples. There are several reasons for this:

- The model with best possible performance may not belong to the family of models under consideration. This is an instance of the principle "you cannot learn it if you cannot represent it."
- The training data may not be representative of the test data, i.e. the training and test data may be samples from different populations.
- Fitting the training data as closely as possible may simply be overfitting.
- The objective function for training, namely log likelihood or conditional log likelihood, may not be the desired objective from an application perspective; for example, the desired objective may be classification accuracy.

Chapter 3

Log-linear models

This chapter describes the general log-linear model, which are a far-reaching extension of logistic regression. Conditional random fields (CRFs), which are explained in the next chapter, are a special case of log-linear models. Section 3.1 in this chapter explains what a log-linear model is, and then Section 3.2 explains a very important representational idea, the generalization from features to feature-functions.

3.1 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

$$p(y|x;w) = \frac{\exp \sum_j w_j F_j(x,y)}{Z(x,w)} \tag{3.1}$$

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.

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 possible justifications for the form of the expression (3.1). 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 4.4 below.

3.2 Feature functions

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 $F_j : X \times Y \to \mathbb{R}$.

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.

Chapter 4

Conditional random fields

Now that we understand log-linear models, at last, we can explain conditional random fields (CRFs), specifically so-called linear-chain CRFs. First, Section 4.1 presents linear-chain CRFs through an example application. Next, Section 4.2 generalizes the example, and Section 4.3 explains the special algorithms that make inference tractable for linear-chain CRFs. Section 4.4 gives a general derivation of the gradient of a log-linear model; this is the foundation of all log-linear training algorithms.

4.1 A typical CRF application

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. 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."

4.2 Linear-chain CRFs in general

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

$$\bar{y}^* = \text{argmax}_{\bar{y}}\, p(\bar{y}|\bar{x}; w) \tag{4.1}$$

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 4.1 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.

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 CLL on the training data. However, instead of maximizing CLL we could maximize 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 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.

4.3 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

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

[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 $\bar{x}$.

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.

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.

4.4 Training CRFs by stochastic 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, 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.) Start with

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

$$= F_j(x,y) - \frac{1}{Z(x,w)} \sum_{y'} \frac{\partial}{\partial w_j} \exp \sum_{j'} w_{j'} F_{j'}(x,y')$$

$$= F_j(x,y) - \frac{1}{Z(x,w)} \sum_{y'} [\exp \sum_{j'} w_{j'} F_{j'}(x,y')] F_j(x,y')$$

$$= F_j(x,y) - \sum_{y'} F_j(x,y') \frac{\exp \sum_{j'} w_{j'} F_{j'}(x,y')}{\sum_{y''} \exp \sum_{j''} w_{j''} F_{j''}(x,y'')}$$

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

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

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

$$w_j := w_j + \alpha(F_j(x, y) - E_{y' \sim p(y'|x;w)}[F_j(x, y')]) \tag{4.2}$$

where α is a learning rate parameter.

Chapter 5

Alternative CRF training methods

This chapter explains three special CRF training algorithms. One is a variant of the perceptron method, the second is called contrastive divergence, and the third is an approximate method named Gibbs sampling.

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 4.3 above 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 $p(y|x; w)$. Each section below describes a method based on a different approximation.

5.1 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 (4.2) at the end of the previous chapter 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 $\hat{y}$.

5.2 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 $1/s$.

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

$$p(y|x; w) \propto \prod_{m=1}^{M} \phi_m(y^m|x; w) \tag{5.1}$$

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 Equation (5.1). In this case

$$p(y_i|x, y_{-i}; w) \propto \prod_{m\in C} \phi_m(y^m|x; w) \tag{5.2}$$

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 (5.2) 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.

5.3 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.

Chapter 6

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. 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.

2. 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.

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

4. 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.

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

6. 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.

7. 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.

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

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

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

11. 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.