Training Conditional Random Fields via Gradient Tree Boosting

Thomas G. Dietterich
Adam Ashenfelter
Yaroslav Bulatov
tgd@cs.orst.edu, ashenfad@engr.orst.edu, bulatov@cs.orst.edu

School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR 97331 USA

To appear, *International Conference on Machine Learning*, 2004

Abstract

Conditional Random Fields (CRFs; Lafferty, McCallum, & Pereira, 2001) provide a flexible and powerful model for learning to assign labels to elements of sequences in such applications as part-of-speech tagging, text-to-speech mapping, protein and DNA sequence analysis, and information extraction from web pages. However, existing learning algorithms are slow, particularly in problems with large numbers of potential input features. This paper describes a new method for training CRFs by applying Friedman's (1999) gradient tree boosting method. In tree boosting, the CRF potential functions are represented as weighted sums of regression trees. Regression trees are learned by stage-wise optimizations similar to Adaboost, but with the objective of maximizing the conditional likelihood $P(Y|X)$ of the CRF model. By growing regression trees, interactions among features are introduced only as needed, so although the parameter space is potentially immense, the search algorithm does not explicitly consider the large space. As a result, gradient tree boosting scales linearly in the order of the Markov model and in the order of the feature interactions, rather than exponentially like previous algorithms based on iterative scaling and gradient descent.

1 Introduction

Many applications of machine learning involve assigning labels to sequences of objects. For example, in part-of-speech tagging, the task is to assign a part of speech ("noun", "verb", etc.) to each word in a sentence (Ratnaparkhi, 1996). In protein secondary structure prediction, the task is to assign a secondary structure class to each amino acid residue in the protein sequence (Qian & Sejnowski, 1988).

We call this class of problems *sequential supervised learning* (SSL), and it can be formalized as follows:

Given: A set of training examples of the form (X_i, Y_i), where each $X_i = (\mathbf{x}_{i,1}, \ldots, \mathbf{x}_{i,T_i})$ is a sequence of T_i feature vectors and each $Y_i = (y_{i,1}, \ldots, y_{i,T_i})$ is a corresponding sequence of class labels, $y_{i,t} \in \{1, \ldots, K\}$.

Find: A classifier H that, given a new sequence X of feature vectors, predicts the corresponding sequence of class labels $Y = H(X)$ accurately.

Perhaps the most famous SSL problem is the NETtalk task of pronouncing English words by assigning a phoneme and stress to each letter of the word (Sejnowski & Rosenberg, 1987).

Early attempts to apply machine learning to SSL problems were based on *sliding windows*. To predict label y_t, a sliding window method uses features drawn from some "window" of the X sequence. For example, a 5-element window $w_t(X)$ would use the features $\mathbf{x}_{t-2}, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \mathbf{x}_{t+2}$. Sliding windows convert the SSL problem into a standard supervised learning problem to which any ordinary machine learning algorithm can be applied. However, in most SSL problems, there are correlations among successive class labels y_t. For example, in part-of-speech tagging, adjectives tend to be followed by nouns. In protein sequences, alpha helixes and beta structures always involve multiple adjacent residues. These correlations can be exploited to increase classification accuracy.

Recently, many new learning methods have been developed with the goal of capturing these $y \leftrightarrow y$ correlations. See Dietterich (2002) for a review. One of the most interesting new methods is the conditional random field (CRF) proposed by Lafferty et al. (2001). The CRF is a probabilistic model of the conditional probability that input sequence X will produce output label sequence Y: $P(Y|X)$. The CRF has the form of a Markov random field (Geman, 1998):

$$P(Y|X) = \frac{1}{Z(X)} \exp\left[\sum_t \Psi_t(y_t, X) + \Psi_{t-1,t}(y_{t-1}, y_t, X)\right],$$

where $\Psi_t(y_t, X)$ and $\Psi_{t-1,t}(y_{t-1}, y_t, X)$ are *potential functions* that capture (respectively) the degree to which y_t is compatible with X and the degree to which y_t is compatible with a transition from y_{t-1} and with X. These potential functions can be arbitrary real-valued functions. The exponential function ensures that $P(Y|X)$ is positive, and the normalizing constant $Z(X) = \sum_{Y'} \exp[\sum_t \Psi_t(y'_t, X) + \Psi_{t-1,t}(y'_{t-1}, y'_t, X)]$ ensures that $P(Y|X)$ sums to 1. This representation is completely general, subject to the assumption that $P(Y|X) > 0$ for all X and Y (Besag, 1974; Hammersley & Clifford, 1971). Normally, it is assumed that the potential functions do not depend on t, and we will adopt that assumption in this paper.

To apply a CRF to an SSL problem, we must choose a representation for the Ψ functions. Lafferty et al. studied Ψ functions that are weighted combinations of binary features:

$$\begin{aligned} \Psi(y_t, X) &= \sum_a \beta_a g_a(y_t, X) & (1) \\ \Psi(y_{t-1}, y_t, X) &= \sum_b \lambda_b f_b(y_{t-1}, y_t, X), & (2) \end{aligned}$$

where the β_a's and λ_b's are trainable weights, and the features g_a and f_b are boolean functions. For example, in part-of-speech tagging $g_{234}(y_t, X)$ might be 1 when $\mathbf{x}_t$ is the word "bank" and y_t is the class "noun" (and 0 otherwise). As with sliding window methods, it is natural to define features that depend only on a sliding window $w_t(X)$ of X values. This linear parameterization can be seen as an extension of logistic regression to the sequential case.

Once a parameterization is chosen, the CRF can be trained to maximize the log likelihood of the training data, possibly with a regularization penalty to prevent overfitting. Let $\Theta = \{\beta_1, \ldots, \lambda_1, \ldots\}$ denote all of the tunable parameters in the model. Then the objective function is to maximize

$$\begin{aligned} J(\Theta) &= \log \prod_i P(Y_i \mid X_i) \\ &= \sum_i \log \frac{1}{Z(X_i)} \exp\left[\sum_t \Psi_t(y_{i,t}, X_i) + \Psi_{t-1,t}(y_{i,t-1}, y_{i,t}, X_i)\right] \\ &= \sum_{i,t} \Psi_t(y_{i,t}, X_i) + \Psi_{t-1,t}(y_{i,t-1}, y_{i,t}, X_i) - \log Z(X_i) \end{aligned}$$

$$= \sum_{i,t} \sum_a \beta_a g_a(y_{i,t}, X_i) + \sum_b \lambda_b f_b(y_{i,t-1}, y_{i,t}, X_i) - \log Z(X_i)$$

Lafferty et al. introduced an iterative scaling algorithm for maximizing $J(\Theta)$, but they reported that it was exceedingly slow. Several groups have implemented gradient ascent methods, but naive implementations are also very slow, because the various β and λ parameters interact with each other: increasing one parameter may require compensating changes in others. McCallum's Mallet system (McCallum, 2003) employs the BFGS algorithm, which is an approximate second-order method that deals with these parameter interactions.

A drawback of this linear parameterization is that it assumes that each feature makes an independent contribution to the potential functions. Of course it is possible to define more features to capture combinations of the basic features, but this leads to a combinatorial explosion in the number of features, and hence, in the dimensionality of the optimization problem. For example, in protein secondary structure prediction, Qian and Sejnowski found that a 13-residue sliding window gave best results for neural network methods. There are $3^2 \times 13 \times 20 = 2340$ basic f_b features that can be defined over this window. If we consider fourth-order conjunctions of such features, we obtain more than 10^{12} features. This is obviously infeasible.

McCallum's Mallet system starts with a single constant feature and introduces new feature conjunctions by taking conjunctions of the basic features with features already in the model. Candidate conjunctions are evaluated according to their incremental impact on the objective function. He demonstrates significant improvements in speed and classification accuracy compared to a CRF that only includes the basic features.

In this paper, we introduce a different approach to training the potential functions based on Freidman's (2001) gradient tree boosting algorithm. In this approach, the potential functions are represented by sums of regression trees, which are grown stage-wise in the manner of Adaboost (Freund & Schapire, 1996). Each regression tree can be viewed as defining several new feature combinations—one corresponding to each path in the tree from the root to a leaf. The resulting potential functions still have the form of a linear combination of features, but the features can be quite complex. The advantage of the gradient boosting approach is that the algorithm is fast and straightforward to implement. In addition, there may be some tendency to avoid overfitting because of the "ensemble effect" of combining multiple regression trees.

2 Gradient Tree Boosting

Suppose we wish to solve a standard supervised learning problem, where the training examples have the form $(\mathbf{x}_i, y_i), i = 1, \ldots, N$ and $y_i \in \{1, \ldots, K\}$. We wish to fit a model of the form

$$P(y \mid \mathbf{x}) = \frac{\exp \Psi(y, \mathbf{x})}{\sum_{y'} \Psi(y', \mathbf{x})}.$$

Gradient tree boosting is based on the idea of functional gradient ascent. In ordinary gradient ascent, we would parameterize Ψ in some way, for example, as a linear function,

$$\Psi(y, \mathbf{x}) = \sum_a \beta_a g_a(y, \mathbf{x}).$$

Let $\Theta = \{\beta_1, \ldots\}$ represent all of the tunable parameters in this function. In gradient ascent, the fitted parameter vector after iteration m, Θ_m, is a sum of an initial parameter vector Θ_0 and a series of gradient ascent steps δ_m:

$$\Theta_m = \Theta_0 + \delta_1 + \cdots + \delta_m,$$

where each δ_m is computed as a step in the direction of the gradient of the log likelihood function:

$$\delta_m = \eta_m \, \frac{\partial}{\partial \Theta_{m-1}} \sum_i \log P(y_i \mid \mathbf{x}_i; \Theta_{m-1})$$

and η_m is a parameter that controls the step size.

Functional gradient ascent is a more general approach. Instead of assuming a linear parameterization for Ψ, it just assumes that Ψ will be represented by a weighted sum of functions:

$$\Psi_m = \Psi_0 + \Delta_1 + \cdots + \Delta_m.$$

Each Δ_m is computed as a *functional gradient*:

$$\Delta_m = \eta_m \, E_{\mathbf{x},y} \left[\frac{\partial}{\partial \Psi_{m-1}} \log P(y \mid \mathbf{x}; \Psi_{m-1}) \right].$$

The functional gradient indicates how we would like the function Ψ_{m-1} to change in order to increase the true log likelihood (i.e., on all possible points $(\mathbf{x}, y)$). Unfortunately, we do not know the joint distribution $P(\mathbf{x}, y)$, so we cannot evaluate the expectation $E_{\mathbf{x},y}$. We do have a set of training examples sampled from this joint distribution, so we can compute the value of the functional gradient at each of our training data points:

$$\Delta_m(y_i, \mathbf{x}_i) = \frac{\partial}{\partial \Psi_{m-1}} \sum_i \log P(y_i \mid \mathbf{x}_i; \Psi_{m-1}).$$

We can then use these point-wise functional gradients to define a set of *functional gradient training examples*, $((\mathbf{x}_i, y_i), \Delta_m(y_i, \mathbf{x}_i))$ and then train a function $h_m(y, \mathbf{x})$ so that it approximates $\Delta_m(y_i, \mathbf{x}_i)$. Specifically, we can fit a regression tree h_m to minimize

$$\sum_i [h_m(y_i, \mathbf{x}_i) - \Delta_m(y_i, \mathbf{x}_i)]^2.$$

We can then take a step in the direction of this fitted function:

$$\Psi_m = \Psi_{m-1} + \eta h_m.$$

Although the fitted function h_m is not exactly the same as the desired Δ_m, it will point in the same general direction (assuming there are enough training examples). So ascent in the direction of h_m will approximate true functional gradient ascent.

A key thing to note about this approach is that it replaces the difficult problem of maximizing the log likelihood of the data by the much simpler problem of minimizing squared error on a set of training examples. Friedman suggests growing h_m via a best-first version of the CART algorithm (Breiman et al., 1984) and stopping when the regression tree reaches a pre-set number of leaves L. Overfitting is controlled by tuning L (e.g., by internal cross-validation).

3 Gradient Tree Boosting for SSL

In principle, it is straightforward to apply functional gradient ascent to SSL. All we need to do is to represent and train $\Psi(y_t, X)$ and $\Psi(y_{t-1}, y_t, X)$ as weighted sums of regression trees. For historical reasons, we took a slightly different approach. Let

$$F^{y_t}(y_{t-1}, X) = \Psi(y_t, X) + \Psi(y_{t-1}, y_t, X)$$

Table 1: Derivation of the functional gradient

$$\begin{aligned}
\frac{\partial \log P(Y|X)}{\partial F^v(u, w_d(X))} &= \frac{\partial}{\partial F^v(u, w_d(X))} \sum_t F^{y_t}(y_{t-1}, w_t(X)) - \log Z(X) \\
&= I(y_{d-1} = u, y_d = v) - \frac{\partial \log Z(X)}{\partial F^v(u, w_d(X))} && (3) \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \frac{\partial Z(X)}{\partial F^v(u, w_d(X))} && (4) \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \frac{\partial}{\partial F^v(u, w_d(X))} \sum_k \left[\sum_{k'} \left[\exp F^k(k', w_d(X)) \right] \cdot \alpha(k', d-1) \right] \beta(k, d) && (5) \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \left[\exp F^v(u, w_d(X)) \right] \alpha(u, d-1) \beta(v, d) && (6) \\
&= I(y_{d-1} = u, y_d = v) - P(y_{d-1} = u, y_d = v \mid X) && (7)
\end{aligned}$$

be a function that computes the "desirability" of label y_t given values for label y_{t-1} and the input features X. There are K such functions F^k, one for each class label k. Then the CRF has the form

$$P(Y|X) = \frac{1}{Z(X)} \exp \sum_t F^{y_t}(y_{t-1}, X).$$

We now compute the functional gradient of $\log P(Y|X)$ with respect to $F^{y_t}(y_{t-1}, X)$. To simplify the computation, we replace X by $w_t(X)$, which is a window into the sequence X centered at $\mathbf{x}_t$. We will further assume, without loss of generality, that each window is unique, so there is only one occurrence of $w_t(X)$ in each sequence X.

Proposition 1 *The functional gradient of* $\log P(Y|X)$ *with respect to* $F^v(u, w_d(X))$ *is*

$$\frac{\partial \log P(Y|X)}{\partial F^v(u, w_d(X))} = I(y_{d-1} = u, y_d = v) - P(y_{d-1} = u, y_d = v \mid w_d(X)),$$

where $I(y_{d-1} = u, y_d = v)$ *is 1 if the transition* $u \rightarrow v$ *is observed from position* $d-1$ *to position* d *in the sequence* Y *and 0 otherwise, and where* $P(y_{d-1} = u, y_d = v \mid w_d(X))$ *is the predicted probability of this transition according to the current potential functions.*

To demonstrate this proposition, we must first introduce the forward-backward algorithm for computing $Z(X)$. We will assume that y_t takes the value $\perp$ for $t < 1$. Define the forward recursion by

$$\begin{aligned}
\alpha(k, 1) &= \exp F^k(\perp, w_1(X)) \\
\alpha(k, t) &= \sum_{k'} \left[\exp F^k(k', w_t(X)) \right] \cdot \alpha(k', t-1).
\end{aligned}$$

Define the backward recursion as

$$\begin{aligned}
\beta(k, T) &= 1 \\
\beta(k, t) &= \sum_{k'} \left[\exp F^{k'}(k, w_{t+1}(X)) \right] \cdot \beta(k', t+1)
\end{aligned}$$

The variables k and k' iterate over the possible class labels. The normalizer $Z(X)$ can be computed at any position t as

$$Z(X) = \sum_k \alpha(k,t)\beta(k,t).$$

If we unroll the α recursion one step, we can also write this as

$$Z(X) = \sum_k \left[\sum_{k'} \left[\exp F^k(k', w_t(X)) \right] \cdot \alpha(k', t-1) \right] \beta(k,t)$$

Table 1 shows the derivation of the functional gradient. In line 3, exactly one of the $F^{y_t}(y_{t-1}, w_t(X))$ terms will match $F^v(u, w_d(X))$, because $w_d(X)$ is unique. This term will have a derivative of 1, so we represent this by the indicator function $I(y_{d-1} = u, y_d = v)$. In line 5, we expand $Z(X)$ at position d using the forward-backward algorithm. Again because $w_d(X)$ is unique, only the product where $k' = u$ and $k = v$ will give a non-zero derivative, so this gives us line 6. The right-hand expression in 6 is precisely the joint probability that $y_{d-1} = u$ and $y_d = v$ given X. **Q.E.D.**

If $w_d(X)$ occurs more than once in X, each match contributes separately to the functional gradient.

This functional gradient has a very satisfying interpretation: It is our error on a probability scale. If the transition $u \rightarrow v$ is observed in the training example, then the predicted probability $P(u, v \mid X)$ should be 1 in order to maximize the likelihood. If the transition is not observed, then the predicted probability should be 0. Functional gradient ascent simply involves fitting regression trees to these residuals.

Table 2 shows pseudo code for our tree-boosting algorithm. The potential function for each class k is initialized to zero. Then M iterations of boosting are executed. In each iteration, for each class k, a set $S(k)$ of functional gradient training examples is generated. Each example consists of a window $w_t(X_i)$ on the input sequence, a possible class label k' at time $t-1$, and the target Δ value. A regression tree having at most L leaves is fit to these training examples to produce the function $h_m(k)$. This function is then added to the previous potential function to produce the next function. In other words, we are setting the step size $\eta_m = 1$. We experimented with performing a line search at this point to optimize η_m, but this is very expensive. So we rely on the "self-correcting" property of tree boosting to correct any overshoot or undershoot on the next iteration.

One way to improve upon this algorithm is to initialize the potential functions more intelligently. The *pseudo-likelihood* of (X, y_t) is $P(y_t \mid y_{t-1}, y_{t+1}, X)$. This is the probability of the correct label at position t *given the correct labels for* y_{t-1} *and* y_{t+1}. The pseudo-likelihood can be computed without performing any forward-backward iterations:

$$P(y_t \mid y_{t-1}, y_{t+1}, X) = \frac{\exp\left[F^{y_t}(y_{t-1}, w_t(X)) + F^{y_{t+1}}(y_t, w_{t+1}(X))\right]}{\sum_{y'} \exp\left[F^{y'}(y_{t-1}, w_t(X)) + F^{y_{t+1}}(y', w_{t+1}(X))\right]}.$$

The pseudo-likelihood—because it assumes that the correct labels are known for y_{t-1} and y_{t+1}—works well if our eventual error rate will be small. We found that it significantly sped up our training trials. It is known to be a consistent estimator of the likelihood (Besag, 1977). We perform three iterations of gradient tree boosting using the pseudo-likelihood to compute the boosting examples $S(k)$. Then we switch to using the full functional gradient.

The sets of generated examples $S(k)$ can become very large. For example, if we have 3 classes and 100 training sequences of length 200, then the number of training examples for each class k is $3 \times 100 \times 200 = 60,000$. Although regression tree algorithms are very fast, they still must consider all of the training examples! Friedman (2001) suggests two tricks for speeding up the computation: sampling and influence trimming. In sampling, a random sample of the training data is used for

Table 2: Gradient Tree Boosting for SSL

```
TreeBoost($Data, L$)
// $Data = \{(X_i, Y_i) : i = 1, \ldots, N\}$
for each class $k$, initialize $F_0^k(\cdot, \cdot) = 0$
for $m = 1, \ldots, M$ do
    for class $k$ from 1 to $K$ do
        $S(k) :=$ GenerateExamples($k, Data, Pot_{m-1}$)
            // where $Pot_{m-1} = \{F_{m-1}^u : u = 1, \ldots K\}$)
        $h_m(k) :=$ FitRegressionTree($S(k), L$)
        $F_m^k := F_{m-1}^k + h_m(k)$
    end
end
return $F_M^k$ for all $k$
end TreeBoost

GenerateExamples($k, Data, Pot_m$)
$S := \{\}$
for example $i$ from 1 to $N$ do
    execute the forward-backward algorithm on $(X_i, Y_i)$
        to get $\alpha(k, t)$ and $\beta(k, t)$ for all $k$ and $t$
    for $t$ from 1 to $T_i$ do
        for $k'$ from 1 to $K$ do
            $P(y_{i,t-1} = k', y_{i,t} = k \mid X_i) :=$
                $\dfrac{\alpha(k', t-1) \exp[F_m^k(k-1, w_t(X_i))] \beta(k, t)}{Z(X_i)}$
            $\Delta(k, k', i, t) := I(y_{i,t-1} = k', y_{i,t} = k) -$
                $P(y_{i,t-1} = k', y_{i,t} = k \mid X_i)$
            insert $((w_t(X_i), k'), \Delta(k, k', i, t))$ into $S$
        end
    end
end
return $S$
end GenerateExamples
```

training. In influence trimming, data points with Δ values close to zero are ignored. We did not apply either of these techniques in our experiments.

4 Making Predictions

Once a CRF model has been trained, there are (at least) two possible ways to define a classifier $Y = H(X)$ for making predictions. First, we can predict the *entire sequence* Y that has the highest probability:

$$H(X) = \operatorname*{argmax}_Y P(Y|X).$$

This makes sense in applications, such as part-of-speech tagging, where the goal is to make a coherent sequential prediction. This can be computed by the Viterbi algorithm (Rabiner, 1989), which has the advantage that it does not need to compute the normalizer $Z(X)$.

The second way to make predictions is to individually predict each y_t according to

$$H_t(X) = \operatorname*{argmax}_v P(y_t = v|X)$$

and then concatenate these individual predictions to obtain $H(X)$. This makes sense in applications where the goal is to maximize the number of individual y_t's correctly predicted, even if the resulting predicted Y sequence is incoherent. For example, a predicted sequence of parts of speech might not be grammatically legal, and yet it might maximize the number of individual words correctly classified. $P(y_t|X)$ can be computed by executing the forward-backward algorithm as

$$P(y_t|X) = \frac{\alpha(y_t,t)\beta(y_t,t)}{Z(X)}.$$

5 Experimental Studies

We implemented gradient tree boosting for CRFs and compared it to McCallum's Mallet system on four benchmark data sets. We will call our algorithm TreeCRF. We will use TreeCRF-V for the TreeCRF with Viterbi predictions and TreeCRF-FB for the TreeCRF with forward-backward predictions. Mallet implements McCallum's feature induction algorithm. Mallet makes its predictions using the Viterbi algorithm, so we will denote it by Mallet-V.

5.1 Data Sets

We tested these algorithms on four data sets: protein secondary structure prediction and three Usenet FAQs: `ai-general`, `ai-neural`, and `aix`.

The protein secondary structure benchmark was published by Qian & Sejnowski (1988). A protein consists of a sequence of amino acid residues. Each residue is represented by a single feature with 20 possible values (corresponding to the 20 standard amino acids). There are three classes: alpha helix, beta sheet, and coil (everything else). There is a training set of 111 sequences and a test set of 17 sequences.

Each of the FAQ data sets consists of Frequently Asked Questions files for a Usenet newsgroup (McCallum et al., 2000). The FAQs for each newsgroup are divided in separate files: `ai-general` has 7 files, `ai-neural` has 7 files, and `aix` has 5 files. Every line of an FAQ is labeled as either part of the header, a question, an answer, or part of the tail. Hence, each $\mathbf{x}_t$ consists of a line in the FAQ file, and the corresponding $y_t \in \{\text{header, question, answer, tail}\}$. The measure of accuracy is the number of individual lines correctly classified. McCallum provided us with the definitions of 20 features. We made a slight correction to one of the features, so our results are not directly comparable to his. For each newsgroup, performance was measured by leave-1-out cross-validation: the CRF was trained on all-but-one of the files and tested on the remaining file. This was repeated with each file, and the results averaged.

Both TreeCRF and Mallet have parameters that must be set by the user. For both algorithms, the user must set (a) the window size, (b) the order of the Markov model, and (c) the number of iterations to train. For TreeCRF, the only additional parameter is L, the depth limit for the regression trees. For Mallet the parameters are (a) the regularization penalty for squared weights (called the variance), (b) the number of iterations between feature inductions (kept constant at 8), (c) the number of features to add per feature induction (kept constant at 500), (d) the true label probability threshold (kept constant at 0.95), (e) the training proportions (kept constant at 0.2, 0.5, and 0.8), (f) the number of iterations to train. Except for the variance, we kept all of Mallet's parameters fixed at the values recommended by Andrew McCallum (personal communication). To set the remaining parameters, we manually tried a handful of settings and chose the setting that gave the best test set (or cross-validation) performance. Ideally, these would be set via internal cross-validation. However, because we did not perform a very careful search of the parameter settings, we believe that the parameters are not highly tuned.

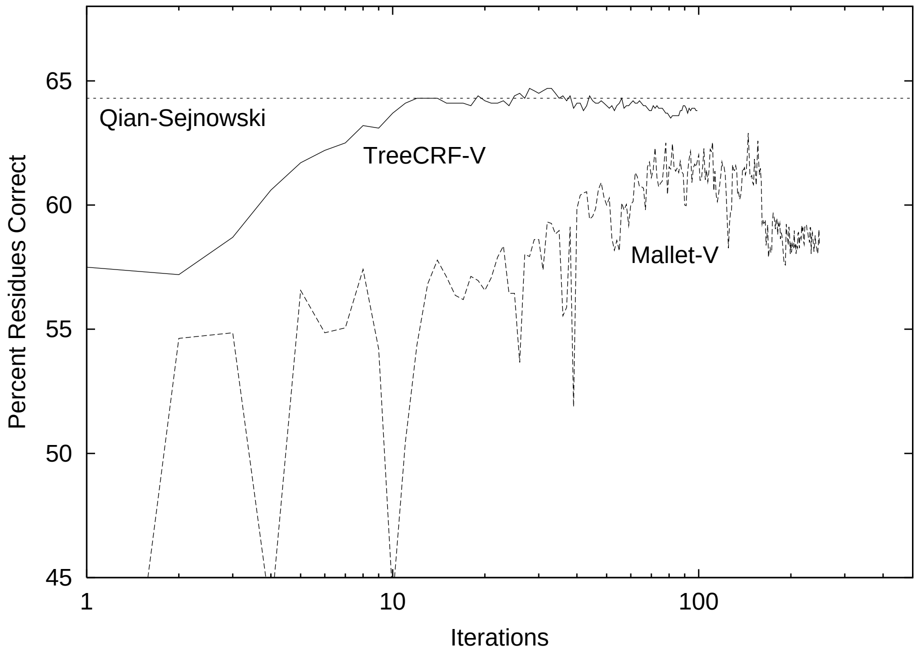

Figure 1: Protein secondary structure prediction

5.2 Results

Figure 1 shows the results on the protein task. In all cases (except Qian-Sejnowski), a first-order CRF was employed. The input features consisted of an 11-residue sliding window. The TreeCRF-FB attains its peak performance of 64.7% correct after 28 iterations. The next best method is the neural network sliding window of Qian and Sejnowski (1988), which attains 64.5%. Mallet-V reaches 62.9% after 145 iterations. A McNemar's test comparing the peak performance of TreeCRF-FB and Mallet-V shows that the difference is statistically significant ($p < 0.05$).

One worrying aspect of Mallet is that the performance curve exhibits a high degree of fluctuation. This is presumably due to the effect of introducing new features. But it also suggests that it will be difficult to find the optimal stopping point for avoiding overfitting. The peak performance of 62.9% is achieved in only one iteration. The second-highest performance is 62.6%, and a more realistic estimate of its achievable performance (i.e., by using cross-validation to determine the stopping point) would be around 61.5%.

It is difficult to compare the CPU time of the methods, because TreeCRF is written in C++ while Mallet is written in Java. Despite these differences the running times of the two programs are quite similar. The time required for TreeCRF to reach its peak performance is 1979.98 s; the time required for Mallet to reach its peak performance is 3634.37 s.

With an 11-residue window, it is not feasible to run LinearCRF on this problem. Table 3 compares the CPU time per iteration for smaller window sizes. We see that LinearCRF is faster for small window sizes, but that it slows down exponentially as the window size grows.

Figure 2 plots the percentage of lines correctly classified by the two algorithms on the `ai-general` FAQ. Again we see that Mallet's performance fluctuates wildly. A McNemar's test of the performance on the final iterations of the two methods concludes that TreeCRF is better ($p < 0.001$).

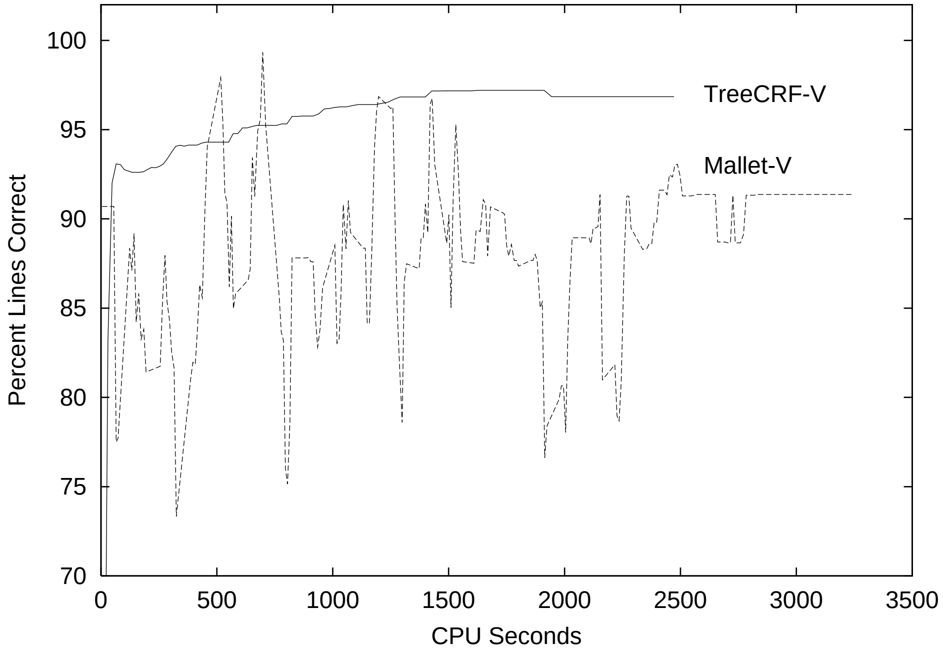

Figure 2: FAQ `ai-general`. Percentage of lines correct as a function of CPU time.

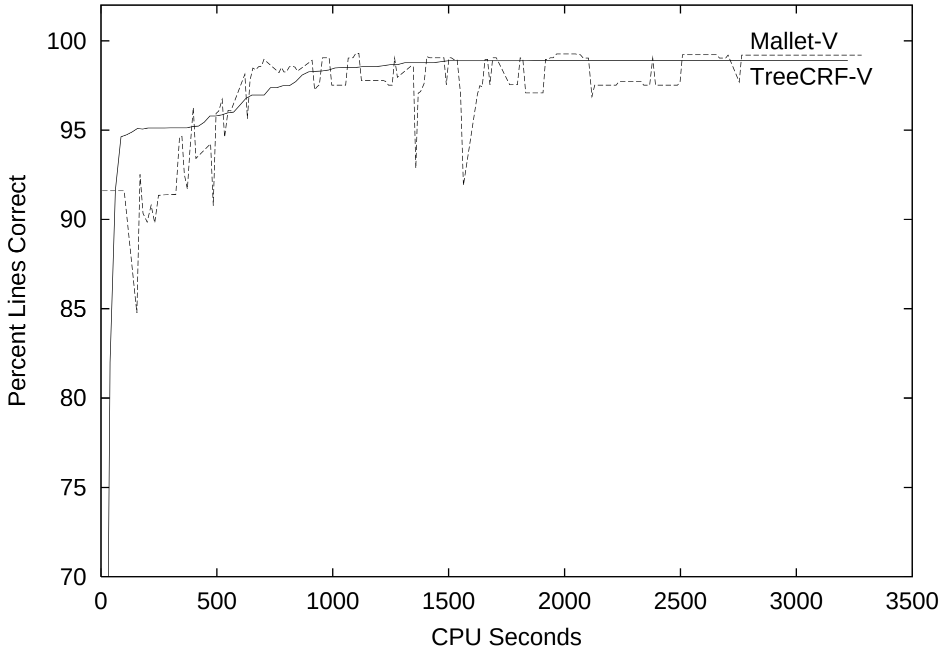

Figure 3: FAQ `ai-neural`. Percentage of lines correct as a function of CPU time

Table 3: Training iteration run time (seconds) for LinearCRF and TreeCRF

Window size	1	3	5	7
LinearCRF	0.04	0.66	41.2	1505
TreeCRF	0.8	1.11	1.2	1.4

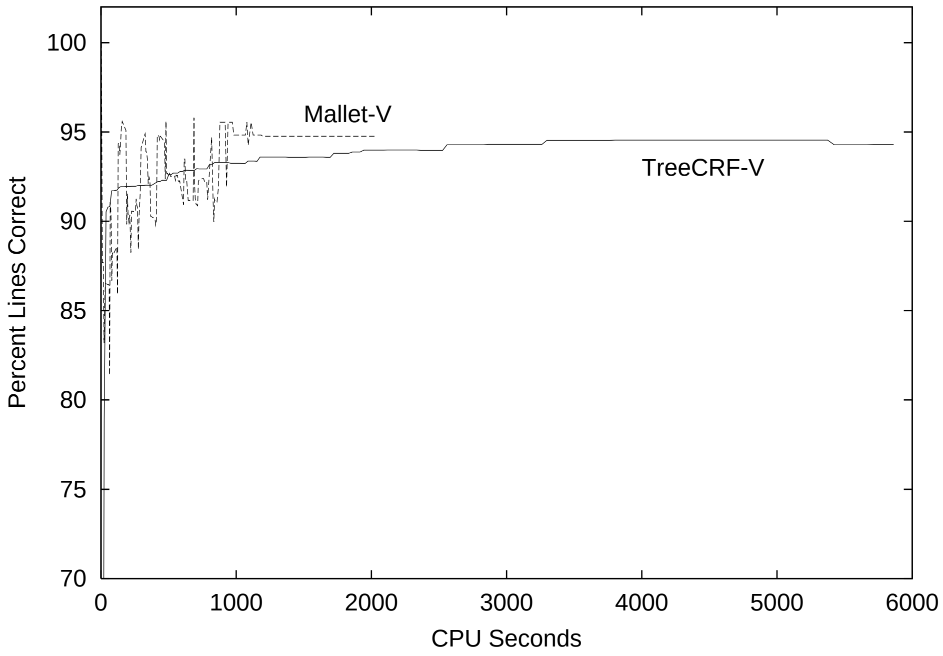

Figure 4: FAQ `aix`. Percentage of lines correct as a function of CPU time

Figure 3 plots the results for the `ai-neural` FAQ. This time, despite fluctuations, Mallet converges to a better classifier than TreeCRF according to McNemar's test ($p < 0.001$).

Finally, Figure 4 plots the results for the `aix` FAQ. Although it is difficult to see from the graph, Mallet again converges to a slightly better classifier ($p < 0.025$). Note that on this data set, TreeCRF required about twice as much time to reach peak performance.

6 Conclusions

This paper has introduced a novel method for training conditional random fields based on gradient tree boosting. We can evaluate it along several dimensions.

Ease of implementation: TreeCRF is simpler to implement than Mallet.

Ease of tuning: TreeCRF introduces only one tunable parameter, L, the maximum number of leaves permitted in each regression tree. Mallet has many more parameters to consider. Mallet's performance fluctuates wildly, while TreeCRF improves smoothly.

Scaling to large numbers of features: tree boosting scales much better than the original linearly-parameterized CRF method. It appears to match Mallet, which also gives dramatic speedups when there are many potential features.

Run time: In our experiments TreeCRF required run time within a factor of two of Mallet. Both are reasonable.

Accuracy: In our experiments, TreeCRF was more accurate on two data sets and less accurate on two data sets.

Scaling to large numbers of classes: In experiments not shown, we attempted to apply TreeCRF to the NETtalk text-to-speech problem, which has 140 classes. This is infeasible because the cost of performing the forward-backward algorithm (required by both TreeCRF and Mallet to compute gradients) scales as $T140^{n+1}$, where T is the length of the sequences and n is the order of the Markov model. For NETtalk, T is around 7, but previous research has suggested that n should be at least 3. This means that the forward-backward computation for each training sequence requires 2.7×10^9 operations, which means that it is very slow. An important challenge for SSL research is to develop methods that can handle large numbers of classes.

Gradient tree boosting may provide another advantage over methods based on standard parametric gradient ascent: the ability to handle missing values in the inputs. There are very good methods for handling missing values when growing regression trees including the surrogate split method of CART (Breiman et al., 1984) and the instance weighting method of C4.5 (Quinlan, 1993). In future work, we will evaluate whether these methods work well for training and evaluating CRFs.

Acknowledgements

The authors gratefully acknowledge the support of NSF grants IIS-0083292 and IIS-0307592.

References

Besag, J. (1974). Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society B*, *36*, 192–236.

Besag, J. (1977). Efficiency of pseudolikelihood estimation for simple Gaussian fields. *Biometrika*, *64*, 616–618.

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees.* Wadsworth International Group.

Dietterich, T. G. (2002). Machine learning for sequential data: A review. *Structural, Syntactic, and Statistical Pattern Recognition* (pp. 15–30). New York: Springer Verlag.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Proc. 13th International Conference on Machine Learning* (pp. 148–156). Morgan Kaufmann.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, *29*.

Geman, D. (1998). Random fields and inverse problems in imaging. In A. Ancona, D. Geman and N. Ikeda (Eds.), *École d'Été de probabilités de saint-flour xviii*, vol. Lecture Notes in Mathematics 1427, 117–196. Berlin: Springer-Verlag.

Hammersley, J. M., & Clifford, P. (1971). *Markov fields on finite graphs and lattices* (Technical Report). Unpublished.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proceedings of the 18th International Conference on Machine Learning* (pp. 282–289). San Francisco, CA: Morgan Kaufmann.

McCallum, A. (2003). Efficiently inducing features of conditional random fields. *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence* (pp. 403–410). San Francisco, CA: Morgan Kaufmann.

McCallum, A., Freitag, D., & Pereira, F. (2000). Maximum entropy Markov models for information extraction and segmentation. *Proc. 17th International Conf. on Machine Learning* (pp. 591–598). Morgan Kaufmann, San Francisco, CA.

Qian, N., & Sejnowski, T. J. (1988). Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, *202*, 865–884.

Quinlan, J. R. (1993). *C4.5: Programs for empirical learning.* San Francisco, CA: Morgan Kaufmann.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*, 257–286.

Ratnaparkhi, A. (1996). A maximum entropy model for part-of-speech tagging. *Proceedings of the conference on empirical methods in natural language processing* (pp. 133–142). Somerset, NJ: ACL.

Sejnowski, T. J., & Rosenberg, C. R. (1987). Parallel networks that learn to pronouce English text. *Complex Systems*, *1*, 145–168.