AN ABSTRACT OF THE DISSERTATION OF

Guohua Hao for the degree of Doctor of Philosophy in Computer Science presented on July 21, 2009.

Title: Efficient Training and Feature Induction in Sequential Supervised Learning

Abstract approved: ________________________________

Thomas G. Dietterich

Sequential supervised learning problems arise in many real applications. This dissertation focuses on two important research directions in sequential supervised learning: efficient training and feature induction.

In the direction of efficient training, we study the training of conditional random fields (CRFs), which provide a flexible and powerful model for sequential supervised learning problems. Existing training algorithms for CRFs are slow, particularly in problems with large numbers of potential input features and feature combinations. In this dissertation, we describe a new algorithm, TREECRF, for training CRFs via gradient tree boosting. In TREECRF, the CRF potential functions are represented as weighted sums of regression trees, which provide compact representations of feature interactions. So the algorithm does not explicitly consider the potentially large parameter 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 as in previous algorithms based on iterative scaling and gradient descent. Detailed experimental results are provided to evaluate the performance of the TREECRF algorithm and possible extensions of this algorithm are discussed.

We also study the problem of handling missing input values in CRFs, which has been rarely discussed in the literature. Gradient tree boosting also makes it possible to use instance weighting (as in C4.5) and surrogate splitting (as in CART) to handle missing values in CRFs. Experimental studies of the effectiveness of these two methods (as well as standard imputation and indicator feature methods) show that instance weighting is the best method in most cases when feature values are missing at random.

In the direction of feature induction, we study the search-based structured learning framework and its application to sequential supervised learning problems. By formulating the label sequence prediction process as an incremental search process from one end of a sequence to the other, this framework is able to avoid complicated inference algorithms in the training process and thus achieves very fast training speed. However, for problems where there exist long range dependencies between the current position and future positions, at each search step, this framework is unable to exploit these dependencies to make accurate predictions. In this dissertation, a multiple-instance learning based algorithm is proposed to automatically extract useful features from future positions as a way to discover and exploit these long range dependencies. Integrating this algorithm with maximum entropy Markov models yields promising experimental results on both synthetic data sets and real data sets that have long range dependencies in sequences.

Efficient Training and Feature Induction in Sequential Supervised Learning

by

Guohua Hao

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented July 21, 2009
Commencement June 2010

ACKNOWLEDGEMENTS

First and foremost I owe my deepest gratitude to my major advisor, Professor Thomas Dietterich, for his tremendous support and great patience throughout my PhD study. Without his continuous encouragement, I would not have been able to finish my research work. It is my honor to work as his graduate student in the past six years.

I would like to thank all my current and previous PhD committee members, including Professor Alan Fern, Professor Xiaoli Fern, Professor Weng-Keen Wong, Professor Prasad Tadepalli, and Professor Jon Herlocker, for their kindly support and guidance to my PhD study and research. I would also like to thank Professor Margaret Niess and Professor Jack Barth for taking their time serving as GCRs on my committee.

I would like to show my gratitude to Dr. Stefan Riezler, my internship mentor at Google Inc., for giving me the opportunity to apply my research to solve industry problems. It is a valuable experience for my future career.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (Continued)

DEDICATION

To my family.

Chapter 1 – Introduction

In this chapter, we first formulate the sequential supervised learning problem and describe its application in a variety of domains. Then we give a brief review of previous work that has been done to solve this problem. After that, the contributions of this dissertation are summarized, and the outline of this dissertation is given at the end.

1.1 Sequential Supervised Learning

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

These kinds of problems can be formulated 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, where $y_{i,t} \in \mathcal{Y} = \{1, \ldots, L\}$.

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

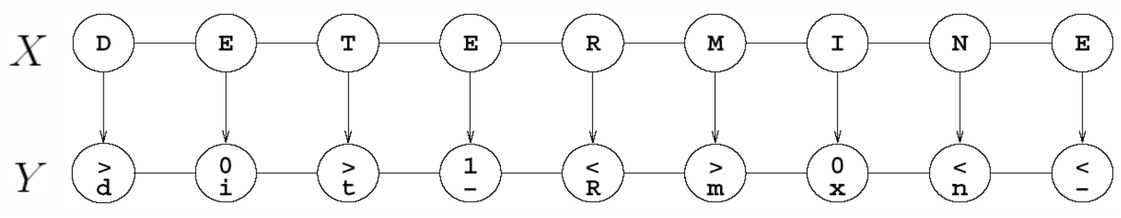


Figure 1.1: Graphical representation of a training example in the NETTalk data set. X is the English word "determine" and Y the corresponding sequence of stress/phoneme pairs.

These problems are called *Sequential Supervised Learning* (SSL) (Dietterich, 2002) or *Label Sequence Learning* (LSL) problems .

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 and Rosenberg, 1987). One training example in this task is shown in Figure 1.1. Other applications of SSL arise in information extraction (McCallum et al., 2000), handwritten word recognition (Taskar et al., 2004), and so on.

1.2 Previous Work

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 algorithms 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.

The best-known method for capturing the $y_{t-1} \leftrightarrow y_t$ correlation is the hidden Markov model (HMM) (Rabiner, 1989), which is a generative model of $P(X, Y)$, the joint distribution of the observation sequence and label sequence. In this model, the joint distribution is factored as

$$P(X, Y) = \prod_t P(y_t | y_{t-1}) P(\mathbf{x}_t | y_t) \quad ,$$

where $P(y_1|y_0) = P(y_1)$, and the observation distribution is further factored as

$$P(\mathbf{x}_t | y_t) = \prod_j P(\mathbf{x}_{t,j} | y_t) \quad .$$

This assumption of independence of each input feature $\mathbf{x}_{t,j}$ conditioned on y_t makes HMMs unable to model arbitrary, non-independent input features, and this limits the accuracy and "engineerability" of HMMs.

Recent research has instead focused on discriminative models, in which arbitrary and non-independent observation features can be easily incorporated. Much machine learning research has shown that discriminative models tend to be more accurate and more robust to incorrect modeling assumptions (Ng and Jordan, 2002). McCallum and his collaborators introduced maximum entropy Markov models (MEMMs) (McCallum et al., 2000) and conditional random fields (CRFs) (Lafferty et al., 2001). MEMMs are directed graphical models of the form

$$P(Y|X) = \prod_t P(y_t|y_{t-1}, w_t(X)) \ ,$$

where $w_t(X)$ is a sliding window over the X sequence centered at time t. They are easy to train, but they suffer from the *label bias problem* that results from the local normalization at each time step t. Conditional random fields are undirected models of the form

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

where $Z(X)$ is a global normalizing term and $\Psi(y_t, y_{t-1}, w_t(X))$ is a potential function that scores the compatibility of y_t, y_{t-1}, and $w_t(X)$. The global normalization avoids the label bias problem but makes training much more computationally expensive. CRFs have been applied to many problems with excellent results including POS tagging (Lafferty et al., 2001) and noun-phrase chunking (Sha and Pereira, 2003).

Kernel-based methods have also been extended to the SSL case. The hidden Markov SVM (Altun et al., 2003; Tsochantaridis et al., 2004) and max-margin Markov networks (Taskar et al., 2004) learn a discriminant function $F(X, Y')$ that assigns a real valued score to each possible label sequence Y' to maximize the margin between the correct label sequence Y and all competing incorrect label sequences.

More recently, the search based structured learning framework (Collins and Roark, 2004; Daumé III and Marcu, 2005) has become a promising approach to deal with complex structured learning problems. Unlike probabilistic graphical models such as CRFs, this approach does not rely on dynamic programming. In this framework, output labels are inferred incrementally by some approximate search methods (such as beam search), and the model parameters are learned to optimize this search process. It is straightforward to apply this approach to SSL problems.

1.3 Contributions

This dissertation describes the gradient tree boosting algorithm originally proposed by Dietterich et al. (2004) and proposes a new method for incorporating weight penalties into this algorithm. It then compares training time and generalization performance against McCallum's Mallet system. The results show that our implementation of tree boosting is competitive with Mallet in both speed and accuracy and that additional improvements in our implementation of the forward-backward algorithm would likely produce a system that is faster than both systems.

Within CRFs, we also perform experiments to evaluate the effectiveness of four methods for handling missing values (instance weighting, surrogate splits, indicator features, and imputation). The first two algorithms are specific to the tree boosting algorithm, while the other two are general missing values methods. The results show that instance weighting works best, but that imputation also works surprisingly well. This leads to two conclusions. First, for CRF models, instance weighting combined with gradient tree boosting can be recommended as a good algorithm for learning in the presence of missing values. Second, for all SSL methods, imputation can be employed to provide a reasonable missing values method.

Another contribution of this dissertation is that we analyze the limitations of the search based structured learning framework. This framework is unable to exploit long range dependencies between the current position and future positions at each search step, and thus can possibly fall into local ambiguity. In this dissertation, a multiple-instance learning algorithm is proposed to automatically extract useful features from future positions in the sequence as a way to capture these long range dependencies. Integrating this algorithm with maximum entropy Markov models gives promising experimental results on both synthetic data sets and real data sets, which have long range dependencies in sequences.

1.4 Outline

The remainder of this dissertation is organized as follows.

In chapter 2, we give a review of the conditional random field model, including the model representation, training algorithms, and inference algorithms. We also analyze the difficulty of training CRFs. Chapter 3 presents the TREECRF algorithm, which trains CRFs with gradient tree boosting. Detailed experimental comparisons against McCallum's Mallet system are given. In chapter 4, we empirically evaluate four methods for dealing with missing feature values in CRFs. The search based learning framework is analyzed in chapter 5, and a new learning algorithm is proposed to discover useful features at future positions. Chapter 6 concludes this dissertation with summary and future work.

Chapter 2 – Conditional Random Fields (CRFs)

Conditional random fields (CRFs) have been successfully applied to solve many sequential supervised learning problems. An introduction to CRFs is given by Sutton and McCallum (2007). In this chapter, we give a brief review of CRF, and point out the difficulty in training CRF models. This chapter serves as a background for the next few chapters.

2.1 Model Representation

Let (X, Y) be a sequential labeled training example, where $X = (\mathbf{x}_1, \ldots, \mathbf{x}_T)$ is the observation sequence and $Y = (y_1, \ldots, y_T)$ is the sequence of labels, where $y_t \in \mathcal{Y} = \{1, \ldots, L)$ for all t. A conditional random field is a linear chain Markov random field (Geman and Geman, 1984) over the label sequence Y globally conditioned on the observation sequence X. The conditional probability distribution $P(Y|X)$ can be written as

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

where $\Psi_t(y_t, X)$ and $\Psi_{t-1,t}(y_{t-1}, y_t, X)$ are *potential functions* defined on cliques y_t and (y_{t-1}, y_t), which 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

$$Z(X) = \sum_{Y'} \exp \left[\sum_t \Psi_t(y'_t, X) + \Psi_{t-1,t}(y'_{t-1}, y'_t, X) \right] \quad ,$$

ensures that $P(Y|X)$ sums to 1. If given sufficiently rich potential functions, this model can represent any first-order Markov distribution $P(Y|X)$ subject to the assumption that $P(Y|X) > 0$ for all X and Y (Besag, 1974; Hammersley and Clifford, 1971). Normally, it is assumed that the potential functions do not depend on t, and we will adopt this assumption in this paper.

To apply a CRF to an SSL problem, we must choose a representation for the potential functions. Lafferty et al. (2001) studied potential functions that are weighted combinations of binary features:

$$\Psi_t(y_t, X) = \sum_a \beta_a g_a(y_t, X) \quad , \tag{2.1}$$

$$\Psi_{t-1,t}(y_{t-1}, y_t, X) = \sum_b \lambda_b f_b(y_{t-1}, y_t, X) \quad , \tag{2.2}$$

where the β_a's and λ_b's are trainable weights, and the features g_a and f_b are boolean functions. In part-of-speech tagging, for example, $g_{234}(y_t, X)$ 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.

2.2 Training CRFs

CRFs can be trained by maximizing 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 we seek to maximize the objective function

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

Training CRFs is difficult for several reasons.

1. As with all collective classification problems, training requires performing inference. In particular, all algorithms must compute the conditional log likelihood $\log P(Y_i|X_i)$ for each training example (X_i, Y_i) in each iteration. This is expensive, and it dictates that training algorithms should try to minimize the number of iterations and maximize the amount of progress made in each iteration;

2. In many SSL applications, the space of potential features for describing the arguments of $\Psi_t(y_t, X)$ in Equation 2.1 and $\Psi_{t-1,t}(y_{t-1}, y_t, X)$ in Equation 2.2 is immense. As a result, there can be millions of weights β_a and λ_b to learn. In POS tagging and semantic role labeling, for example, it is common to have one feature (and hence, one weight) for every combination of a word and a pair of class labels.

Furthermore, in most applications, performance is improved if the algorithm can consider combinations of these basic features (e.g., word n-grams, feature conjunctions and disjunctions, and so on). If feature interactions are permitted, the number of parameters to be learned explodes;

3. In some problems, feature values can be missing, and this is difficult for discriminative training algorithms to handle.

There has been steady progress in algorithms for training CRFs. The initial paper (Lafferty et al., 2001) introduced an iterative scaling algorithm, which was reported to be exceedingly slow. Several groups have implemented gradient ascent methods (such as Sha and Pereira, 2003), but naive implementations are also very slow. McCallum's Mallet system (McCallum, 2002) employs the BFGS algorithm, which is an approximate second order method, to speed up the training of CRFs and improve the prediction accuracy. More recently, Vishwanathan et al. (2006) proposed to use stochastic gradient descent to train CRFs, and to accelerate this process via the Stochastic Meta-Descent (SMD), which is a gain adaptation method. The resulting algorithm is much faster than the BFGS algorithm and scales well on large data sets.

2.3 Inference in CRFs

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:

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

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

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

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 sequence Y 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

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

and the details of computing $\alpha(y_t, t)$ and $\beta(y_t, t)$ will be given in chapter 3.

Most of the existing training methods involve the repeated computation of the partition function $Z(X)$ and/or maximizing over label sequences, which is usually done using the forward-backward and Viterbi algorithms. The time complexity of these algorithms is $\mathcal{O}(T \cdot L^{k+1})$, where L is the number of class labels, k is the order of Markov model, and T is the sequence length. So even for first order CRFs, training and inference scale quadratically in the number of class labels, which becomes computationally demanding for large label sets.

Chapter 3 – Training CRFs with Gradient Tree Boosting

In this chapter, we describe the TREECRF algorithm originally proposed by Dietterich et al. (2004). This algorithm extends Friedman's gradient tree boosting algorithm (Friedman, 2001) to train CRFs. A new method for incorporating weight penalties into this algorithm is described. Characteristics of the TREECRF algorithm are analyzed based on detailed experimental comparisons against MaCallum's Mallet system (McCallum et al., 2000). The work presented in this chapter was published as a journal paper (Dietterich et al., 2008).

3.1 Motivation

A drawback of the linear parameterization in Equation 2.1 and Equation 2.2 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 (1988) 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 (McCallum, 2002) implements standard CRFs and CRFs with feature induction (McCallum, 2003). When feature induction is turned on, the learner starts with a single constant feature and (every 8 iterations) 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 chapter, we employ the gradient tree boosting method (Friedman, 2001) to construct complex features from the basic features as part of a stage-wise construction of the potential functions. The regression trees grown at each step are compact representations of complex features.

3.2 Functional 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 $y_i \in \{1, \ldots, L\}$. We wish to fit a model of the form

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

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,

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

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:

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

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

$$\delta_m = \eta_m \; \nabla_\Theta \sum_i \log P(y_i \mid \mathbf{x}_i; \Theta) \Big|_{\Theta_{m-1}} \qquad ,$$

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:

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

Each Δ_m is computed as step in the direction of the*functional gradient*:

$$\Delta_m = \eta_m \; E_{\mathbf{x},y} \left[\nabla_\Psi \, \log P(y \mid \mathbf{x}; \Psi)|_{\Psi_{m-1}} \right] \quad .$$

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 $E_{\mathbf{x},y}[\cdot]$. 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:

$$\Delta_m(y_i, \mathbf{x}_i) = \nabla_\Psi \sum_i \log P(y_i \mid \mathbf{x}_i; \Psi)\Big|_{\Psi_{m-1}} \quad .$$

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 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 the squared error

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

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

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

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 (2001) suggests growing h_m via a best-first version of the CART algorithm (Breiman et al., 1984; Friedman et al., 2000) and stopping when the regression tree reaches a pre-set number of leaves L. The pseudocode of this algorithm is shown in Table 3.1. Overfitting is controlled by tuning L (e.g., by internal cross-validation).

Table 3.1: Best-first version of the CART algorithm.

```
FITREGRESSIONTREE($Data, L$)
// $Data = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, N, \ \mathbf{x}_i = (x_{i1}, \ldots, x_{ip})\}$
// NodeQueue is a priority queue of tree nodes where the first node has the minimum $SplitScore$
$Root$ := FINDBESTSPLITATTRIBUTE($Data, NodeQueue$)
$NumLeaves := 1$
while (($NumLeaves < L$)  AND  NOTEMPTY($NodeQueue$))
    $Node$ := REMOVEFRONT($NodeQueue$)
    $TrueData$ := examples in $Node$ whose values of $SplitFeature$ are true
    $FalseData$ := examples in $Node$ whose values of $SplitFeature$ are false
    $TrueChild$ := FINDBESTSPLITATTRIBUTE($TrueData, NodeQueue$)
    $FalseChild$ := FINDBESTSPLITATTRIBUTE($FalseData, NodeQueue$)
    SETCHILDNODES ($Node, TrueChild, FalseChild$)
    $NumLeaves := NumLeaves + 1$
end
return $Root$
end FITREGRESSIONTREE

FINDBESTSPLITATTRIBUTE($Data, NodeQueue$)
$SplitScore := 0, \ SplitFeature := 0$
for $j$ from 1 to $p$
    $TrueData := \{(\mathbf{x}_i, y_i) \in Data : \ x_{ij} = 1\}$
    $FalseData := \{(\mathbf{x}_i, y_i) \in Data : \ x_{ij} = 0\}$
    $Gain$ := SQUAREDERROR($TrueData$) + SQUAREDERROR($FalseData$)
            −SQUAREDERROR($Data$)
    if $Gain < SplitScore$
        $SplitScore := Gain, \ SplitFeature := j$
    end
end
$Node$ := MAKELEAF(OUTPUT($Data$), $Data, SplitFeature, SplitScore$)
if $SplitFeature \geq 1$
    INSERT($Node, NodeQueue$)
end
return $Node$
end FINDBESTSPLITATTRIBUTE
```

In our experience, using L to control overfitting is a blunt tool that is hard to calibrate. In this paper, we instead introduce shrinkage into the algorithm for growing regression trees by adding a quadratic weight penalty. For each leaf in the regression tree h_m, the quantity that we minimize is the squared error of the examples $((\mathbf{x}_i, y_i), \Delta_m(y_i, \mathbf{x}_i))$ falling into this leaf plus a quadratic penalty:

$$\sum_i (\Delta_m(y_i, \mathbf{x}_i) - \hat{\delta})^2 + \lambda \hat{\delta}^2 \quad ,$$

where $\hat{\delta}$ is the output of this leaf and $\lambda > 0$ controls the strength of the penalty. Differentiating the above objective function with respect to $\hat{\delta}$ shows that the minimum is achieved at

$$\hat{\delta} = \frac{\sum_i \Delta_m(y_i, \mathbf{x}_i)}{\lambda + N} \quad , \tag{3.1}$$

where N is the total number of examples falling into this leaf. This has the nice interpretation that λ is an equivalent number of training examples with target values of 0. So this shrinks the leaf values (learned weights) toward zero. With this method, we can select a large number for L (the maximum number of leaves in the regression tree), and use λ to give fine control of overfitting. The algorithm shown in Table 3.1 can be adapted by using Equation 3.1 in the computation of function OUTPUT and function SQUAREDERROR. Experimental results show that this new algorithm works better and is more efficient than the original best-first version of the CART algorithm.

3.3 TREECRF Algorithm

In principle, it is straightforward to apply functional gradient ascent to train CRFs. 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. Let

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

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 L such functions F^k, one for each class label k. With this definition, the CRF has the form

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

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

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

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 the normalizing constant $Z(X)$. We will assume that y_t takes the value $\perp$ for $t < 1$. Define the forward recursion by

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

and the backward recursion by

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

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

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

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

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

Table 3.2: Derivation of the functional gradient.

$$\begin{aligned}
&\frac{\partial \log P(Y|X)}{\partial F^v(u, w_d(X))} \\
&= \frac{\partial}{\partial F^v(u, w_d(X))} \sum_t F^{y_t}(y_{t-1}, w_t(X)) - \log Z(X) \\
&= I(y_{d-1} = u, y_d = v) - \frac{\partial \log Z(X)}{\partial F^v(u, w_d(X))} && (3.2) \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \frac{\partial Z(X)}{\partial F^v(u, w_d(X))} \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \frac{\partial}{\partial F^v(u, w_d(X))} \sum_k \left[\sum_{k'} \left[\exp F^k(k', w_d(X))\right] \cdot \alpha(k', d-1) \right] \beta(k, d) && (3.3) \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \left[\exp F^v(u, w_d(X))\right] \alpha(u, d-1) \beta(v, d) && (3.4) \\
&= I(y_{d-1} = u, y_d = v) - P(y_{d-1} = u, y_d = v \mid X)
\end{aligned}$$

Table 3.2 shows the derivation of the functional gradient. In Equation 3.2, 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 Equation 3.3, 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 Equation 3.4. The right-hand expression in Equation 3.4 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.

The pseudo code for our gradient tree boosting algorithm is shown in Table 3.3, and we call this algorithm TREECRF. 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

Table 3.3: Gradient tree boosting algorithm for CRFs.

```
TreeBoost($Data, L$)
// $Data = \{(X_i, Y_i) : i = 1, \dots, N\}$
for each class $k$, initialize $F_0^k(\cdot, \cdot) = 0$
for $m = 1, \dots, M$
    for class $k$ from 1 to $K$
        $S(k) :=$ GenerateExamples($k, Data, Pot_{m-1}$)
            // where $Pot_{m-1} = \{F_{m-1}^u : u = 1, \dots K\}$)
        $h_m(k) :=$ 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 $N$
    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 $T_i$
        for $k'$ from 1 to $K$
            $P(y_{i,t-1} = k', y_{i,t} = k \mid X_i) :=$
                $\dfrac{\alpha(k', t-1) \exp[F_m^k(k', w_t(X_i))] \beta(k, t)}{Z(X_i)}$
            $\Delta(k, k', i, t) := I(y_{i,t-1} = k', y_{i,t} = k) -$
                $P(y_{i,t-1} = k', y_{i,t} = k \mid X_i)$
            insert $((w_t(X_i), k'), \Delta(k, k', i, t))$ into $S$
        end
    end
end
return $S$
end GenerateExamples
```

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.

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 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.

3.4 Related Work

The most related work to ours is the virtual evidence boosting (VEB) algorithm developed by Liao et al. (2007) for training CRFs. Both VEB and our approach use boosting for feature induction. However, VEB is a "soft" version of maximum pseudo-likelihood training, where the observed values of neighborhood labels are not used, but the probability distribution over neighborhood labels is used as virtual evidence. Our approach is a true maximum log likelihood method that does not depend on the pseudo-likelihood approximation. Another difference is that VEB only uses decision stumps to induce simple features, while our approach uses regression trees to induce more complex feature combinations.

3.5 Experimental Results

We implemented the gradient tree boosting algorithm for CRFs and compared it to McCallum's Mallet system (McCallum, 2002) on several data sets. We call our algorithm TREECRF. We use TREECRF-FB to denote TREECRF with forward-backward predictions and TREECRF-V to denote the TREECRF with Viterbi predictions. MALLET denotes the Mallet package with McCallum's feature induction algorithm (McCallum, 2003) turned on. Similarly, we use MALLET-FB and MALLET-V for the MALLET with forward-backward predictions and Viterbi predictions respectively. We also used the Mallet package to train standard CRFs without feature induction. We call it BASELINE, which serves as the baseline method. As before, BASELINE-FB donotes BASELINE with forward-backward predictions and BASELINE-V denotes BASELINE with Viterbi predictions. Note that the MALLET-FB and BASELINE-FB algorithms are not implemented in the original Mallet package. Instead we implemented them ourselves.

TREECRF, MALLET and BASELINE have parameters that must be set by the user. For all these algorithms, the user must set (a) the window size, (b) the order of the Markov model, which is set to be 1 in our experiments, and (c) the number of iterations to train. For TREECRF, the only additional parameter is either the maximum number of leaves L in the regression trees using the best-first version of CART, or the regularization constant λ for the shrinkage alternative. 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), and (e) the training proportions (kept constant at 0.2, 0.5, and 0.8). For BASELINE, the only additional parameter is the variance as in MALLET. Except for the variance, we kept all of MALLET's parameters fixed at the values recommended by Andrew McCallum (personal communication). We did not optimize the window size, but instead employed values that have been used in previous studies. The chosen sizes are given in the following section. To set the remaining parameters, we manually tried the following settings and chose the setting that gave the best internal cross-validation performance:

- Number of leaves in regression trees: 30, 50, 75, 100,
- TreeCRF regularization constant: 0, 5, 10, 20, 40, 80,
- Weight variance prior in Mallet package: 1, 5, 10, 20.

Throughout the experiments, we measured the performance by computing the prediction accuracy of individual labels, rather than individual sequences. McNemar's test is employed to assess the statistical significance of these results.

3.5.1 Data Sets

Protein Secondary Structure Benchmark (Qian and Sejnowski, 1988). Each observation sequence is a string of amino acid residues, and the corresponding output sequence is a string over the 3-letter alphabet $\{\alpha, \beta, \gamma\}$, where α indicates alpha helix, β indicates a beta sheet or beta turn, and γ indicates all other secondary structure types. There are 20 possible amino acid residues, and we represent each residue by a set of 20 indicator variables. There is a training set of 111 sequences and a test set of 17 sequences. An 11-residue sliding window is used in our experiments.

NETtalk Data Set. The original NETtalk task (Sejnowski and Rosenberg, 1987) is to assign a combination of phoneme and stress to each letter of the word so that the word is pronounced correctly. However, there are 140 legal phone-stress combinations, which gives a very large label space. Neither TREECRF nor MALLET is efficient enough to work with such a large label space. Hence, we chose to study only the problem of assigning one of five possible stress labels to each letter. The labels are '2' (strong stress), '1' (medium stress), '0' (light stress), '$<$' (unstressed consonant, center of syllable to the left), and '$>$' (unstressed consonant, center of syllable to the right).

Each input sequence is an English word, a string of letters over the 26 letter alphabet. Each input observation is represented by 26 boolean indicator variables. There are 1000 training words and 1000 test words in our standard training and test sets. We employed a window size of 13 (window width of 6).

Hyphenation Data Set. The hyphenation task is to insert hyphens into words at points where it is legal to break a word for a new line. This problem appears widely in many word processing programs. The input sequences are English words, encoded as for the NETtalk task. The output class label has only two values to indicate whether or not a hyphen may legally follow the current letter. We manually constructed a training set of 1951 words and a test set of 908 words. The input window size is set to be 6 (i.e., 3 letters on either side of the potential hyphen location).

Usenet FAQs Data Sets. 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 into a set of files: ai-general has 7 files, ai-neural has 7 files, and aix has 5 files. Every line of an FAQ file 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 for each line $\mathbf{x}_t$. We made a slight correction to one of the features, so our results are not directly comparable to his. The size of the sliding window used here is 1. For each newsgroup, performance was measured by leave-one-file-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.

3.5.2 Performance of Shrinkage in Regression Tree Generation

To evaluate the effectiveness of shrinkage in the regression tree fitting algorithm, we fixed L, the maximum number of leaves in regression trees, to be 100, and applied internal cross-validation to choose the best regularization constant λ. For purposes of comparison, we also implemented the original best-first regression tree generation algorithm. Internal cross-validation was employed to select the best value for L.

We ran these two implementations of TREECRF on each data set. The best performance of both forward-backward predictions and Viterbi predictions is reported as the percentage of correct predictions over the y_ts, as shown in Table 3.4. There are 12 pairs of comparisons (6 data sets with 2 prediction algorithms). In six of them, TREECRF with shrinkage does statistically better than TreeCRF without shrinkage. In five of them,

Table 3.4: Performance comparison of TREECRF with different regression tree fitting algorithms. Entries marked with one or more stars are statistically significantly better than the alternative method. Specifically, * means $p < 0.025$, ** means $p < 0.005$ and *** means $p < 0.001$ according to McNemar's test.

	TREECRF-FB		TREECRF-V	
	Shrinkage	Original	Shrinkage	Original
Protein	64.52**	62.70	62.05***	59.20
NETtalk	85.18***	84.08	85.20***	84.18
Hyphen	92.20	92.20	91.76	92.07
FAQ ai-general	95.65	95.69	95.72	96.02***
FAQ ai-neural	99.02	98.97	99.20***	99.05
FAQ aix	94.00	94.02	95.26*	95.15

the performance of these two versions of TREECRF is statistically indistinguishable. In only one of them, TREECRF without shrinkage does statistically better than TREECRF with shrinkage. Based on the results of these experiments, we decided to only employ TREECRF with shrinkage in the remaining experiments.

3.5.3 Comparison between TREECRF and MALLET

TREECRF and MALLET are the two leading CRF training methods that have feature induction capability. Here we compare the prediction accuracy and training speed of these two methods on each available data set. We also compare TREECRF and MALLET with the BASELINE method. For each method, internal cross-validation is applied to select the parameters that give the best performance of both forward-backward predictions and Viterbi predictions. The results reported here for each method are based on the prediction algorithm that gives higher prediction accuracy. All experiments were run on machines with 2.4 GHz Intel Xeon processors, 512KB cache, and 4GB memory.

Prediction Accuracy. Table 3.5 summarizes the prediction accuracy of TREECRF, MALLET, and BASELINE on each data set. McNemar's tests show that on four of the data sets, that is, protein, hyphen, FAQ ai-neural and FAQ aix, the difference between the prediction accuracy of TREECRF and MALLET is not statistically significant. On the FAQ ai-general data set, the prediction accuracy of TREECRF is statistically better than that of MALLET($p < 0.001$). Only on the NETtalk data set is the prediction accuracy of MALLET statistically better than that of TREECRF ($p < 0.05$). In comparison with the baseline method, the prediction accuracy of TREECRF and MALLET is statistically better than that of BASELINE in most cases. On the FAQ ai-general data set, the difference between MALLET and BASELINE is not statistically significant. Only on the FAQ ai-neural data set is the prediction accuracy of BASELINE statistically better than that of both TREECRF and MALLET.

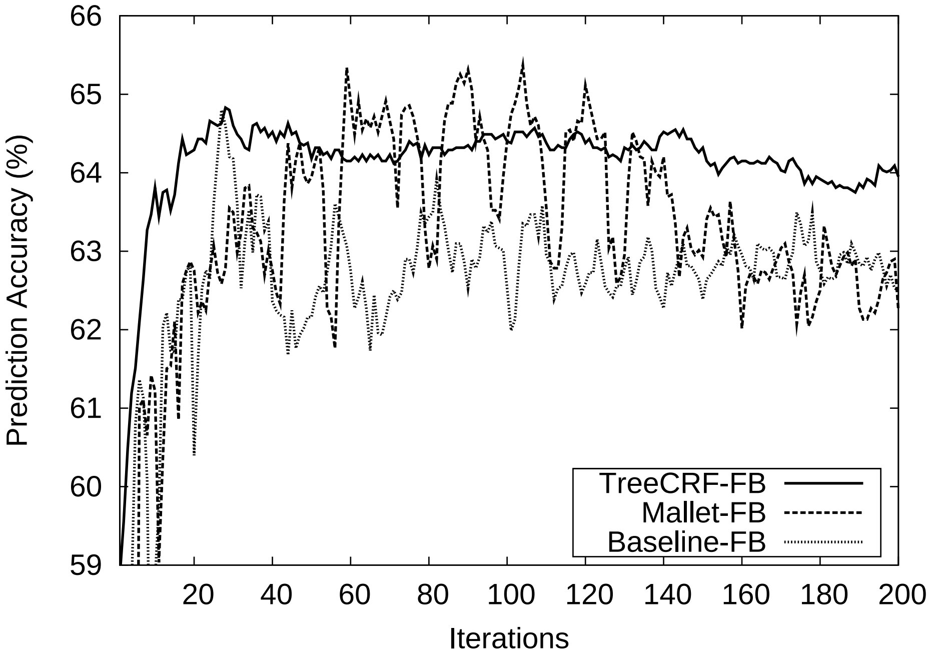
Figure 3.1 plots the prediction accuracy of TREECRF, MALLET and BASELINE as a function of the number of training iterations. One worrying aspect of MALLET is that the performance curve exhibits a high degree of fluctuation, which is clearly shown on Figure 3.1a, 3.1d, 3.1e and 3.1f. This is presumably due to the effect of introducing new features. But it also suggests that it will be difficult to find the optimal stopping points for avoiding overfitting.

Table 3.5: Performance of TREECRF, MALLET, and BASELINE on each data set. Entries marked with one or more stars are statistically significant than BASELINE. Specifically, * means $p < 0.005$, ** means $p < 0.001$ according to McNemar's test. Bolded numbers indicate the statistically better prediction accuracy between TREECRF and MALLET. The BASELINE method stops training if the optimization of loss functions converges. So for each FAQ data set, different training set may have different number of training iterations. Here we gave out the range of number of training iterations for each FAQ data set.

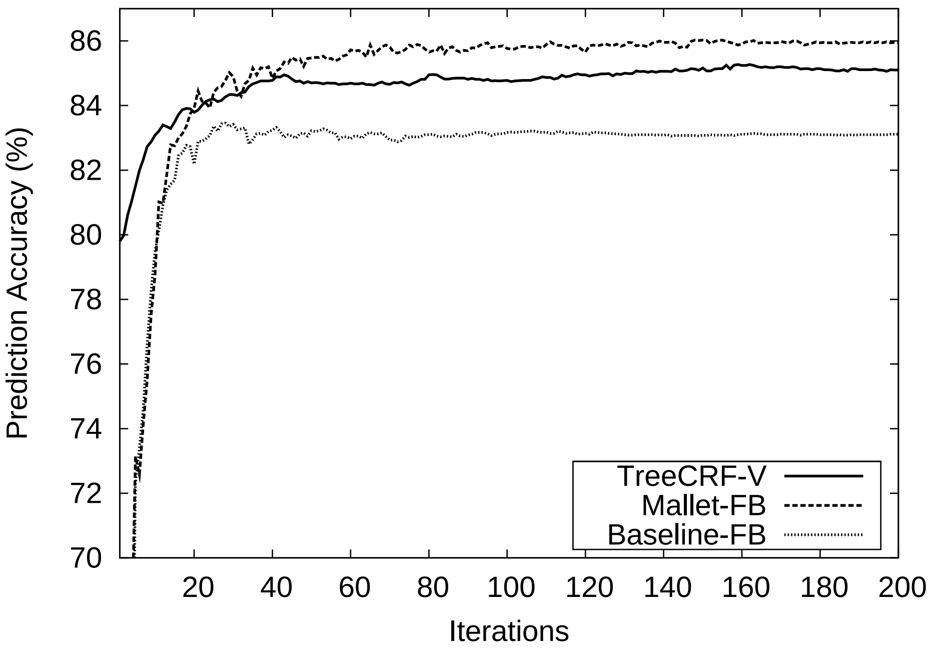
		Protein	NETtalk	Hyphen	FAQ ai-general	FAQ ai-neural	FAQ aix
Accuracy (%)	TREECRF	64.52*	85.20**	92.20**	**95.65** **	99.20**	95.26**
	MALLET	64.43*	**85.94****	92.10**	92.70	99.31*	95.28**
	BASELINE	62.44	82.81	88.86	92.70	99.41	94.04
Cumulative CPU Seconds	TREECRF	419.6	454.6	39.2	3921.9	2177.7	2636.1
	MALLET	786.9	941.4	66.4	484.1	237.2	125.5
	BASELINE	32.8	13.7	8.8	63.0	40.3	34.1
Iterations	TREECRF	142	169	58	214	84	158
	MALLET	123	167	69	188	181	150
	BASELINE	66	34	47	128–195	72–112	80–140

Training Speed. It is difficult to directly compare the CPU time of these two methods, because TREECRF is written in C++ while MALLET is written in Java. However, comparing the CPU time on different data sets can still give us some insight into the properties of these two methods. Figure 3.2 shows the number of cumulative CPU seconds consumed by these two methods on each data set. First, we can see that TREECRF scales linearly in the number of training iterations, because the cumulative CPU time has a constant slope. This makes sense, because for each potential function, only one regression tree is generated in each training iteration. Regression tree evaluations from previous iterations are cached so that they do not need to be re-evaluated. Without caching, the cumulative CPU curves for TREECRF would rise quadratically. Second, as shown in Figure 3.2a, 3.2b and 3.2c, TREECRF runs faster than MALLET on protein, NETtalk and hyphen data sets. But it is much slower than MALLET on FAQ data sets as shown in Figure 3.2d, 3.2e and 3.2f. The actual time required for each method to reach its peak performance on each data set is given in Table 3.5. Again we see that on the protein, NETtalk, and hyphen data sets, the time required for MALLET to reach its peak performance is about twice that of TREECRF. However, on the FAQ data sets, the time required for TREECRF to reach its peak performance is about 10-20 times more than for MALLET. BASELINE is faster than both TREECRF and MALLET as shown in Figure 3.2 and Table 3.5.

Analysis and Discussion. We can explain the training speed difference between TREECRF and MALLET by examining the details of these two methods. In both of them, most of the CPU time is spent on two major computations: forward-backward inference and feature induction/tree growing. The relative proportion of these two com-

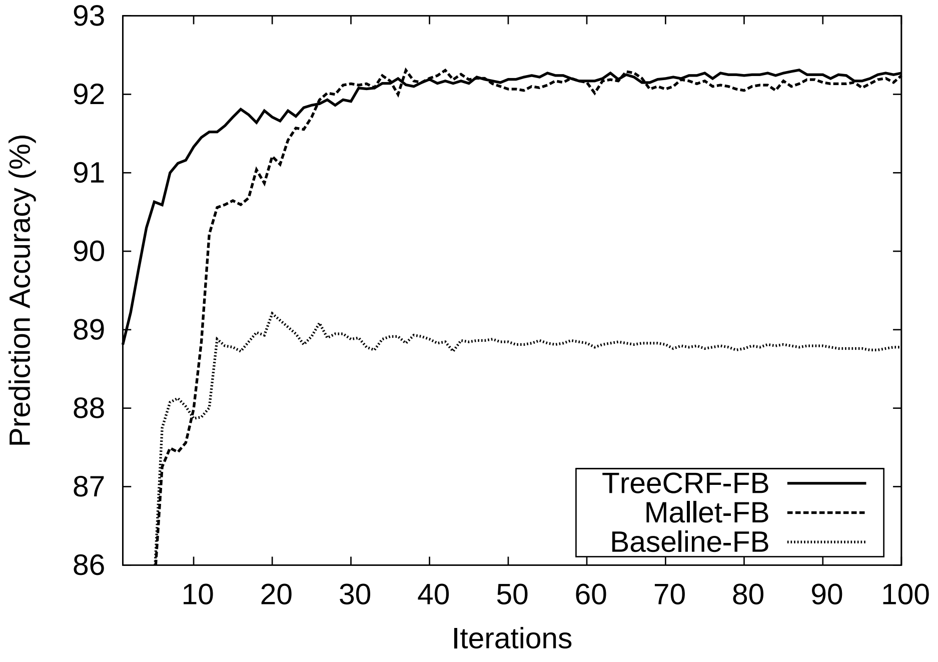


(a) Protein

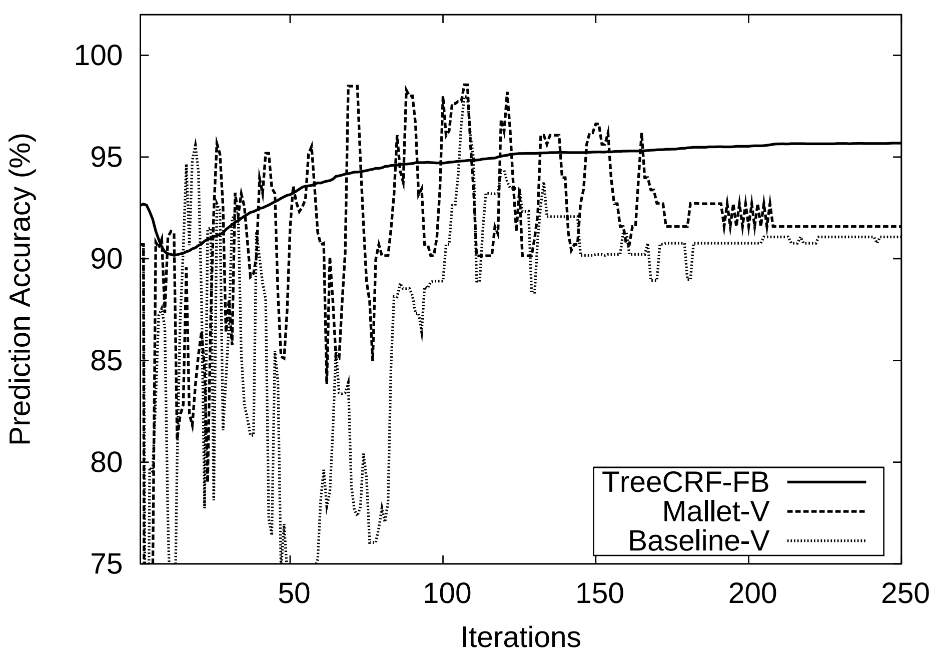


(b) NETtalk

Figure 3.1: Comparison of prediction accuracy on each data set.

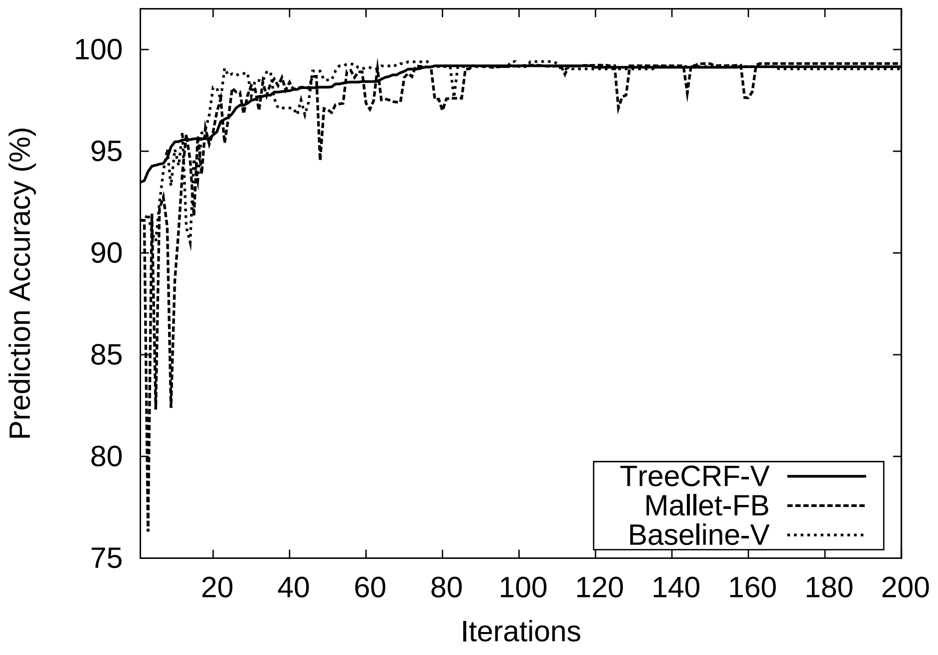


(c) Hyphen

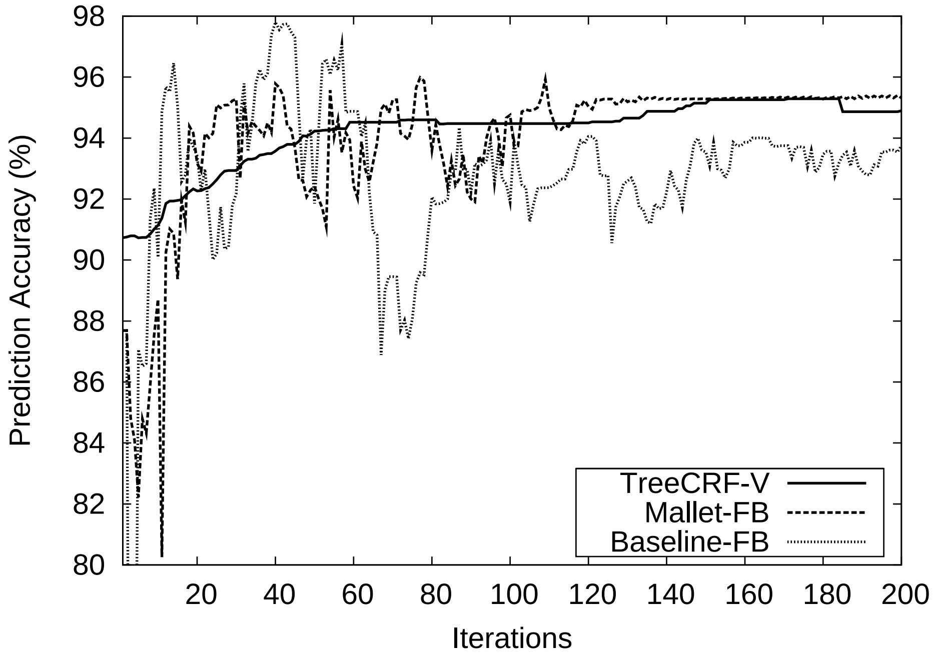


(d) FAQ ai-general

Figure 3.1: Comparison of prediction accuracy on each data set (Continued).

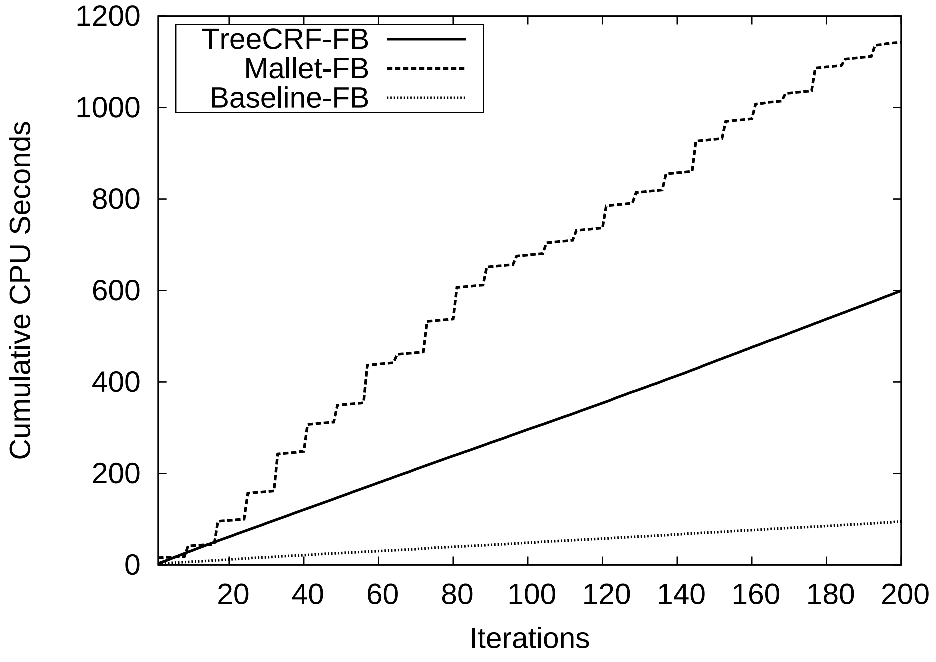


(e) FAQ ai-neural

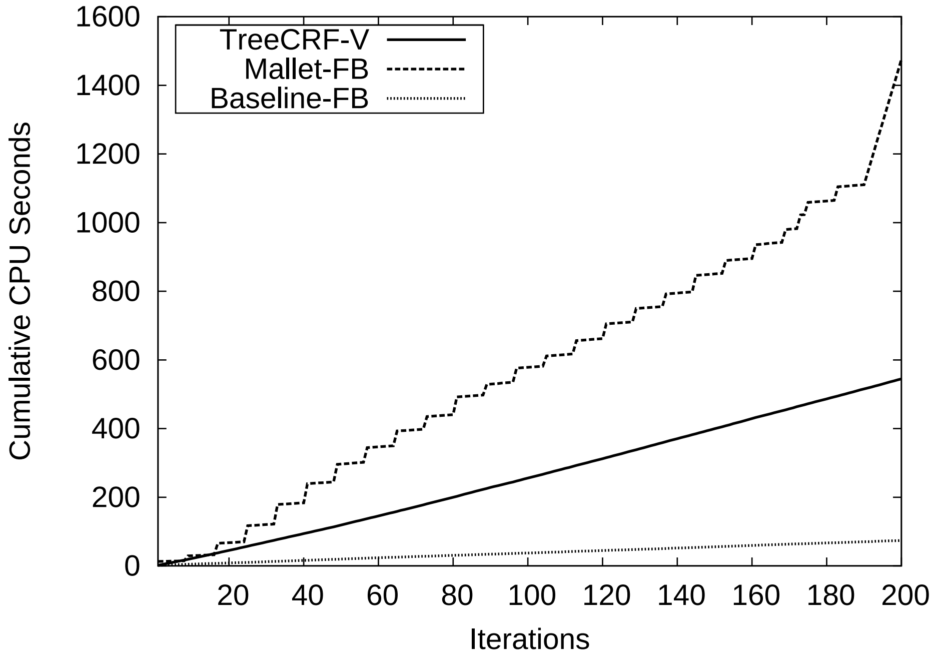


(f) FAQ aix

Figure 3.1: Comparison of prediction accuracy on each data set (Continued).

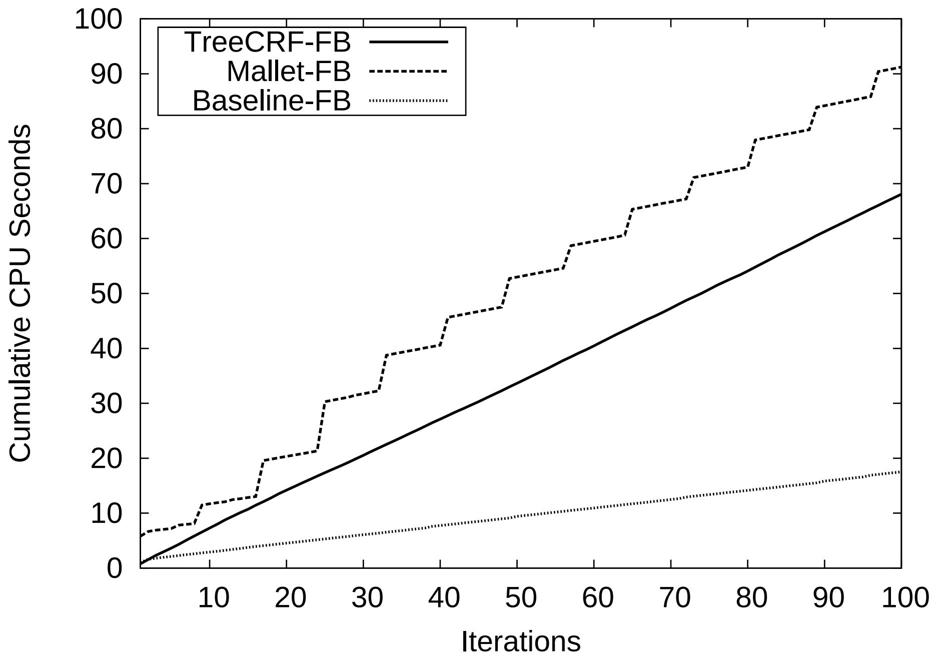


(a) Protein

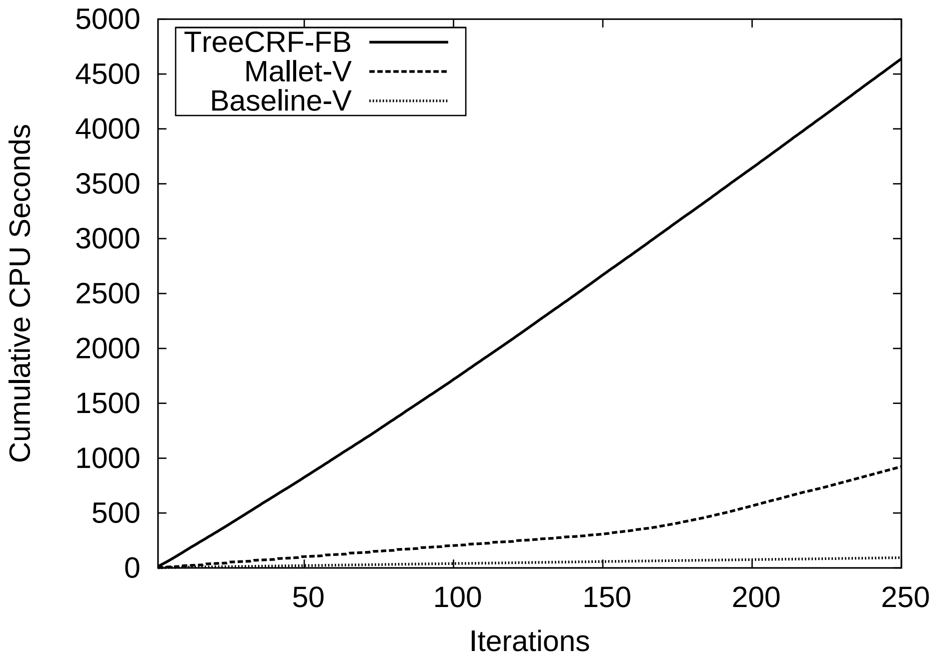


(b) NETtalk

Figure 3.2: Comparison of cumulative CPU time on each data set.

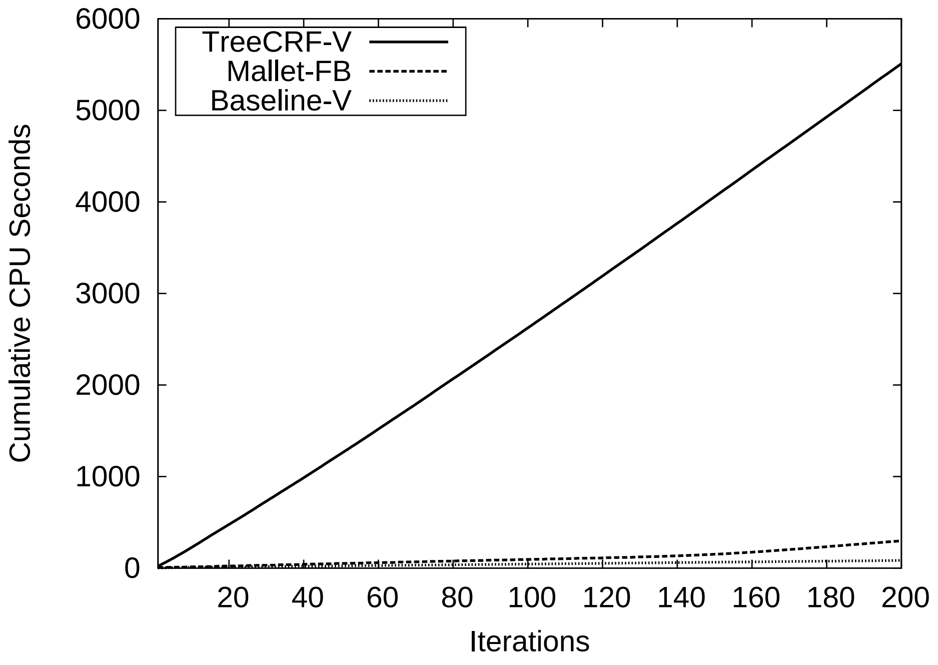


(c) Hyphen

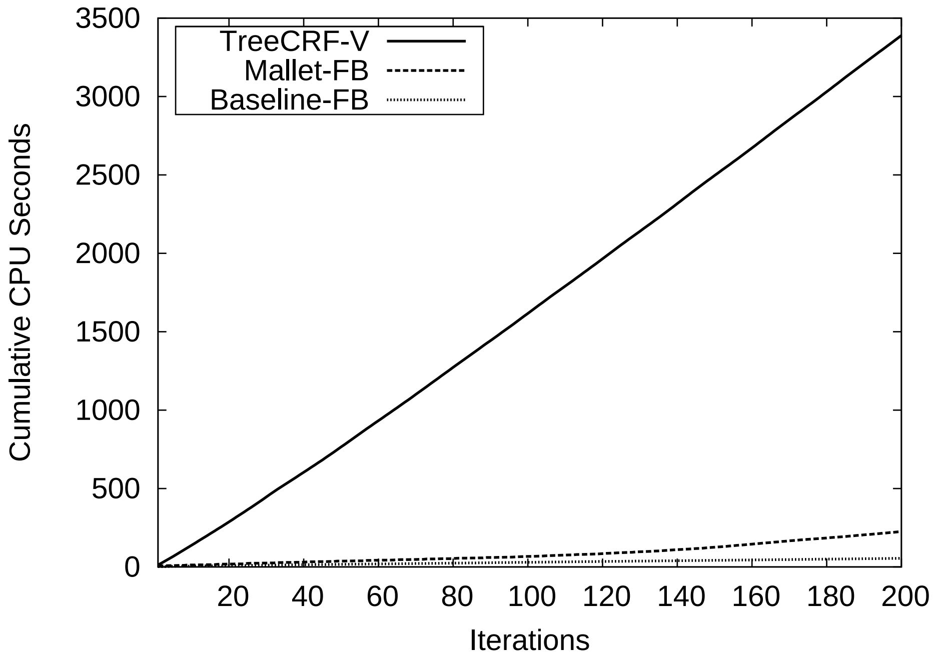


(d) FAQ ai-general

Figure 3.2: Comparison of cumulative CPU time on each data set (Continued).



(e) FAQ ai-neural



(f) FAQ aix

Figure 3.2: Comparison of cumulative CPU time on each data set (Continued).

Table 3.6: Comparison of average CPU seconds spent per iteration on forward-backward algorithm and feature induction algorithm in TREECRF and MALLET for each data set.

Data	Average	Number of	Forward-Backward Seconds		Feature Induction Seconds	
Set	Length	Features	TREECRF	MALLET	TREECRF	MALLET
Protein	163	231	1.493	0.736	1.433	48.889
NETtalk	7	351	0.622	0.589	2.049	25.983
Hyphen	6	162	0.324	0.307	0.332	4.621
FAQ ai-general	1580	20	18.927	0.780	1.562	1.211
FAQ ai-neural	1832	20	26.998	0.526	1.894	1.656
FAQ aix	1806	20	16.658	0.352	1.199	1.123

putations varies from problem to problem. To measure this, we instrumented both TREECRF and MALLET to track the amount of CPU time spent on each of these two computations. Table 3.6 shows that on domains with short sequences (Protein, NETtalk, and Hyphen), the time spent by both algorithms on forward-backward inference is about the same. But for domains with very long sequences, TREECRF consumes much more CPU time in forward-backward inference. Conversely, in domains with a small number of basic features (the FAQ data sets), the two methods consume roughly the same amount of CPU time in feature induction. But in domains with a large number of basic features, TREECRF is much more efficient than MALLET.

Why would the forward-backward cost of TREECRF be larger than for MALLET? TREECRF and MALLET use almost the same implementation of forward-backward algorithm except that in TREECRF the values of the potential functions at each position of the sequences are computed by evaluating the gradient regression trees generated in the current training iteration, while in MALLET those values are obtained by computing dot products of vectors, which is faster than tree evaluation. We hypothesize that the regression trees are more expensive to evaluate, not only because dot products are easier to compute than tree evaluations, but also possibly because of the reduced memory locality of regression trees.

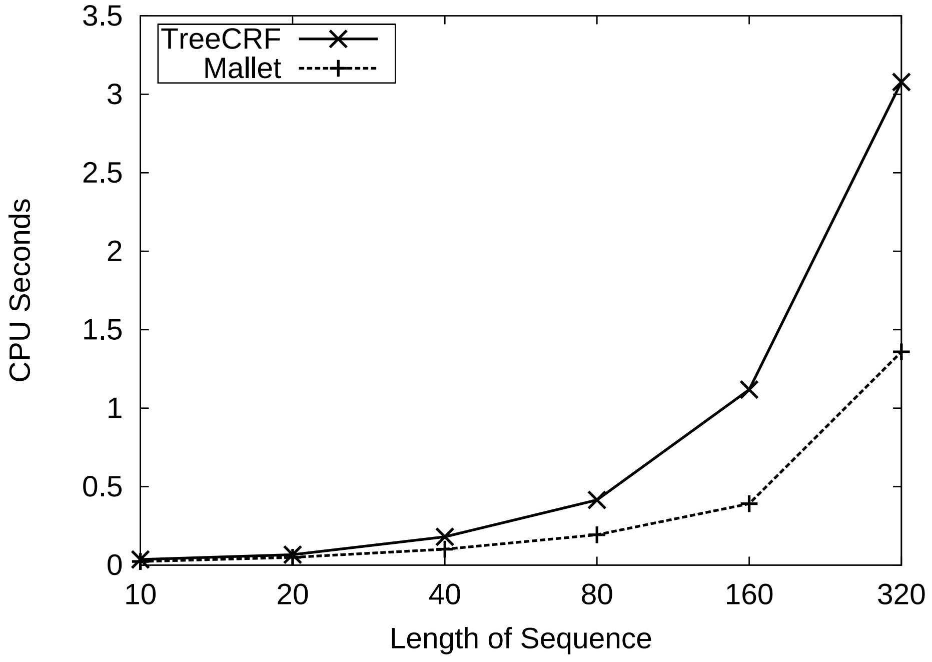
Why would feature induction be more expensive in MALLET? In each feature induction iteration, MALLET considers conjoining all of the basic features to each of the existing compound features. Hence, if there are n basic features and C compound features, this costs nC. Furthermore, C grows over time, so the cost of feature induction gradually increases. In the cumulative CPU time plots of Figure 3.2, the “steps’ in the “staircase” correspond to the feature induction iterations. In TREECRF, the cost of feature induction is the cost of growing a regression tree, which depends on the number of basic features n and the number of internal nodes in the tree L. This cost is nL, which remains constant across the iterations.

To verify our conjectures about the computational complexity of TREECRF and MALLET, we generated synthetic training data sets using a hidden Markov model (HMM) with 3 labels $\{l_1, l_2, l_3\}$ and 24 possible observations $\{o_1, \ldots, o_{24}\}$. To specify the observation distribution, for each label l_i, we randomly drew an observation from the set $\{o_{i*8-7}, \ldots, o_{i*8}\}$ with probability 0.6 and randomly drew an observation from the complement of this set with probability 0.4. The transition distribution was defined as $P(y_t = l_i \mid y_{t-1} = l_i) = 0.6$ and $P(y_t = l_j \mid y_{t-1} = l_i) = 0.2$ if $i \neq j$.

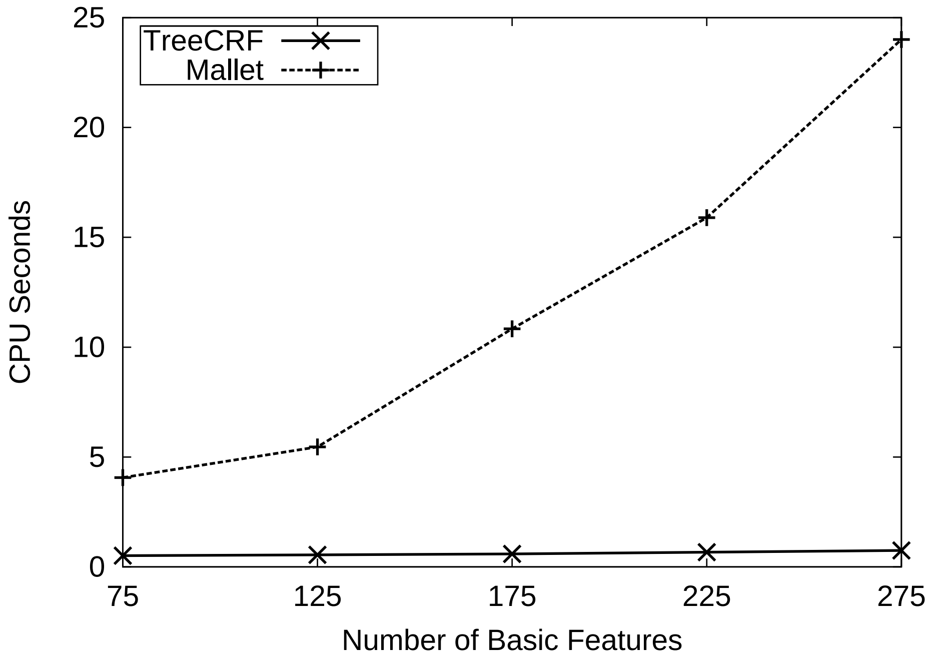
In order to measure the complexity of the forward-backward algorithm, we tried sequence lengths of 10, 20, 40, 80, 160 and 320. For each sequence length, we generated a training data set with 100 sequences and employed a sliding window of size 3. TREECRF and MALLET were run on each of these training data sets. Figure 3.3a shows the average CPU seconds spent per iteration on the forward-backward algorithm by these two methods. We see that the CPU cost of the forward-backward algorithm in TREECRF implementation rises faster than that in MALLET implementation as the length of sequence increases.

In order to measure the complexity of the feature induction algorithms, we generated a training data set with 100 sequences. The length of each sequences is 100. We tried sliding window sizes of 3, 5, 7, 9 and 11, so that the number of input features at each sequence position takes the values of 75, 125, 175, 225 and 275 (because each input observation is represented by 25 boolean indicator variables). TREECRF and MALLET were run for each sliding window size. Figure 3.3b shows the average CPU seconds spent per iteration on the feature induction algorithm by these two methods. It is clear that the feature induction algorithm in MALLET spends more and more CPU time than that in TREECRF as the number of basic features increases. In all the experiments on synthetic data sets, TREECRF uses regression trees of maximum 100 leaves and shrinkage constant 40. MALLET uses weight variance prior 20.

This analysis suggests that the performance of TREECRF could be improved by "flattening" the ensemble of regression trees to compute the corresponding vector of features and vector of weights. Then the cost of potential function evaluations would be similar to that of MALLET, and we would have a method that was faster than both the current TREECRF and MALLET implementations.



(a) Forward-backward algorithm



(b) Feature induction algorithm

Figure 3.3: Comparison of average CPU seconds spent per iteration on forward-backward algorithms and feature induction algorithms by TREECRF and MALLET.

Chapter 4 – Handling Missing Values in CRFs with Gradient Tree Boosting

In this chapter, we study the problem of missing feature values in sequential supervised learning problems. Four algorithms are compared and a guideline is given as to which method is preferred in a given situation. The work presented in this chapter was published in the Journal of Machine Learning Research (Dietterich et al., 2008).

4.1 Motivation

In some problem settings (e.g., activity recognition, sensor networks), the problem of missing values in the inputs can arise. The values of input features can be missing for a wide variety of reasons. Sensors may break or the sensor data feed may be lost or corrupted. Alternatively, input observations may not have been measured in all cases because, for example, they are expensive to obtain. Many methods for handling missing values have been developed for standard supervised learning, but many of them have not been tested on SSL problems. Recently, Sutton et al. (2006) developed a feature bagging method to deal with SSL problems where highly indicative features may be missing in the test data. A single CRF trained on all the features will be less robust, because the weights of weaker features will be undertrained. The feature bagging method divides all the original features into a collection of complementary and possibly overlapped feature subsets. Separate CRFs are trained on each subset and then combined.

With gradient tree boosting, a CRF is represented as a forest of regression trees. There exist very good methods for handing missing values when growing regression trees, which include the instance weighting method of C4.5 (Quinlan, 1993) and the surrogate splitting technique of CART (Breiman et al., 1984). An advantage of training CRFs with gradient tree boosting is that these missing values methods can be used directly in the process of generating regression trees over the functional gradient training examples.

4.2 Review of Instance Weighting

The instance weighting method (Quinlan, 1993), also known as "proportional distribution", assigns a weight to each training example, and all splitting decisions are based on weighted statistics. Initially, each example has a weight of 1.0. When selecting a feature to split on, each boolean feature x_j is evaluated based on the expected weighted squared error of the split using only the training examples for which x_j is not missing. The best feature x_{j*} is chosen, and the training examples for which x_{j*} is not missing are sent to the appropriate child node. Suppose that n_{left} examples are sent to the left child and n_{right} examples are sent to the right child. The remaining training examples (i.e., those for which x_{j*} is missing) are sent to *both* children, but with reduced weight. The weight of each example sent to the left child is multiplied by $n_{left}/(n_{left} + n_{right})$. Similarly, the weight of each example sent to the right child is multiplied by $n_{right}/(n_{left}+n_{right})$.

At test time, when the test example reaches the test on feature x_{j*}, if the feature value is present, then the example is routed left or right in the usual way. But if x_{j*} is missing, then the example is sent to both children (recursively). Let $\hat{y}_{left}$ be the predicted value computed by the left subtree and $\hat{y}_{right}$ be the predicted value computed by the right subtree. Then the value predicted by node $j*$ is the weighted average of these predictions:

$$\hat{y} = \frac{n_{left}\hat{y}_{left} + n_{right}\hat{y}_{right}}{n_{left} + n_{right}} \quad .$$

Instance weighting assumes that the training and test examples missing x_{j*} will on average behave exactly like the training examples for which x_{j*} is not missing.

4.3 Review of Surrogate Splitting

The surrogate splitting method (Breiman et al., 1984) involves separate procedures during training and testing. During training, as the regression tree is being constructed (in the usual top-down, greedy way), the key step in the learning algorithm is to choose which feature to split on. Each boolean feature x_j is evaluated based only on the training examples that have non-missing values for that feature, and the best feature, x_{j*} is chosen. Each of the remaining features $j' \neq j*$ is then evaluated to determine how accurately it can predict the value of x_{j*}, and the features are sorted according to their predictive power. This sorted list of features, called the surrogate splits, is stored in the node.

At test time, when test example x is processed through the regression tree, if x_{j*} is not missing, then the example is processed as usual by sending it to the left child if x_{j*} is false and to the right child if x_{j*} is true. However if x_{j*} is missing, then surrogate split features are examined in order until a feature j' is found that is not missing. The value of this feature determines whether to branch left or right.

4.4 Experimental Results

We performed a series of experiments to evaluate the effectiveness of methods for handling missing values in the TREECRF algorithm. In addition to the instance weighting and surrogate splitting methods described above, we also studied two simpler methods: imputation and indicator features. Let $x_{tj}, j = 1, \ldots, n$ be the input features describing a particular input observation $\mathbf{x}_t$. Imputation and indicator features are defined as follows:

Imputation: when a feature value x_{tj} is missing, it is replaced with the most common value for x_j in the training data among those feature values that are not missing. This strategy can be viewed as substituting the most likely value of x_j a priori or alternatively as substituting the value of x_j least likely to be informative.

Indicator Features: a boolean feature $\tilde{x}_{tj}$ is introduced for each feature x_{tj} such that if x_{tj} is present, $\tilde{x}_{tj}$ is false. But if x_{tj} is missing, then $\tilde{x}_{tj}$ is true and x_{tj} is set to a fixed chosen value, typically 0. Indicator features make sense when the fact that a value is missing is itself informative. For example, if x_{tj} represents a temperature reading, it may be that extremely cold temperature values tend to be missing because of sensor failure.

We adopted a first-order Markov model in all the following experiments and employed an internal hold-out method to set the other parameters: Two-thirds of the original training set was used as sub-training set and the other one third was used as development set to choose parameter values. Final training was performed using the entire training set.

For each learning problem, we took the chosen training and test sets and injected missing values at rates of 5%, 10%, 20% and 40%. For a given missing rate, we generated five versions of the training set and five versions of the test set. A CRF was then trained on each of the training sets and evaluated on each of the test sets (for a total of 5 CRFs and 25 evaluations per missing rate). The label sequences were predicted by the forward-backward algorithm (i.e., we computed $\hat{y}_t = \operatorname{argmax}_{y_t} P(y_t|X)$ for each t separately). Prediction accuracy was based on the number of individual labels correctly predicted in the label sequences. The final prediction accuracy was the average of all 25 combinations of damaged training and test sets.

To test the statistical significance of the differences among the four methods, we performed an analysis of deviance based on the generalized linear model discussed by Agresti (1996). We fit a logistic regression model

$$\log \frac{P(y_t = \hat{y}_t)}{1 - P(y_t = \hat{y}_t)} = \delta_1 m_1 + \delta_2 m_2 + \delta_3 m_3 + \sum_{\ell} \sigma_\ell S_\ell \ ,$$

where m_1, m_2, and m_3 are boolean indicator variables that specify which missing values method we are using and the S_ℓ's are indicator variables that specify which of the five training sets we are using. If $m_1 = m_2 = m_3 = 0$, then we are using instance weighting, which serves as our baseline method. If $m_1 = 1$, this indicates surrogate splitting, $m_2 = 1$ indicates imputation, and $m_3 = 1$ indicates the indicator feature method. Consequently, the fitted coefficients δ_1, δ_2, and δ_3 indicate the change in log odds (relative to the baseline) resulting from using each of these missing values methods. We can then test the hypothesis $\delta_i \neq 0$ against the null hypothesis $\delta_i = 0$ to determine whether missing values method i is statistically significantly different from the baseline method.

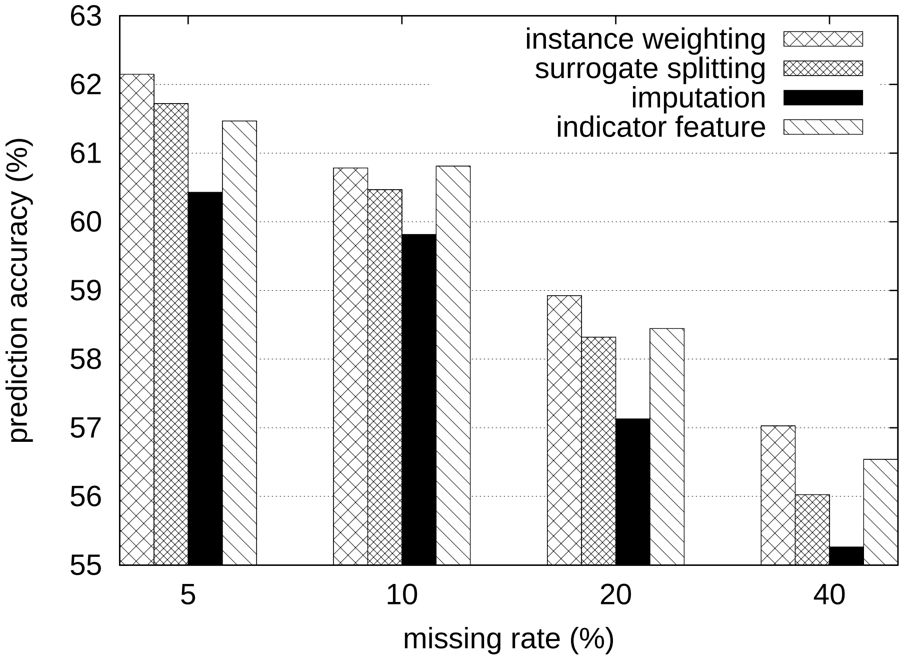
This statistical approach controls for variability due to the choice of the training set (through the σ_ℓ's) and variability due to the size of the test set.

4.4.1 Protein Secondary Structure Prediction

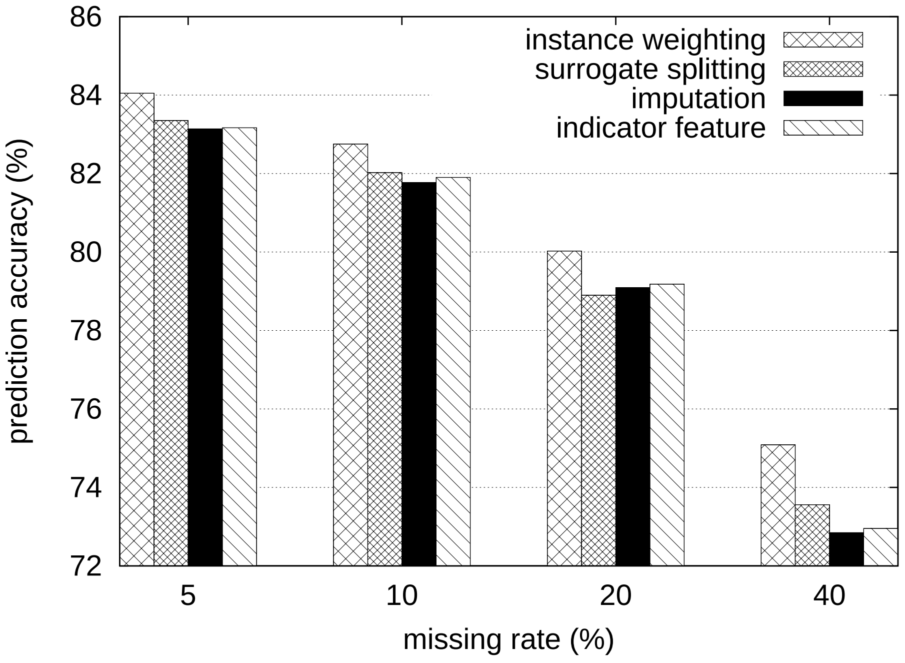
Figure 4.1a shows that instance weighting achieves the best prediction accuracy for each of the different missing rates. Table 4.2a shows that the base line missing values method, instance weighting, is statistically better than the other three missing values methods in most cases. In other cases, it is as good as other methods.

4.4.2 NETtalk Stress Prediction

In Figure 4.1b, we see that instance weighting does better than the other three missing values methods for all of the different missing rates. The statistical tests reported in Table 4.2b show that the baseline method, instance weighting, is statistically better than each of the other missing value methods in all cases.

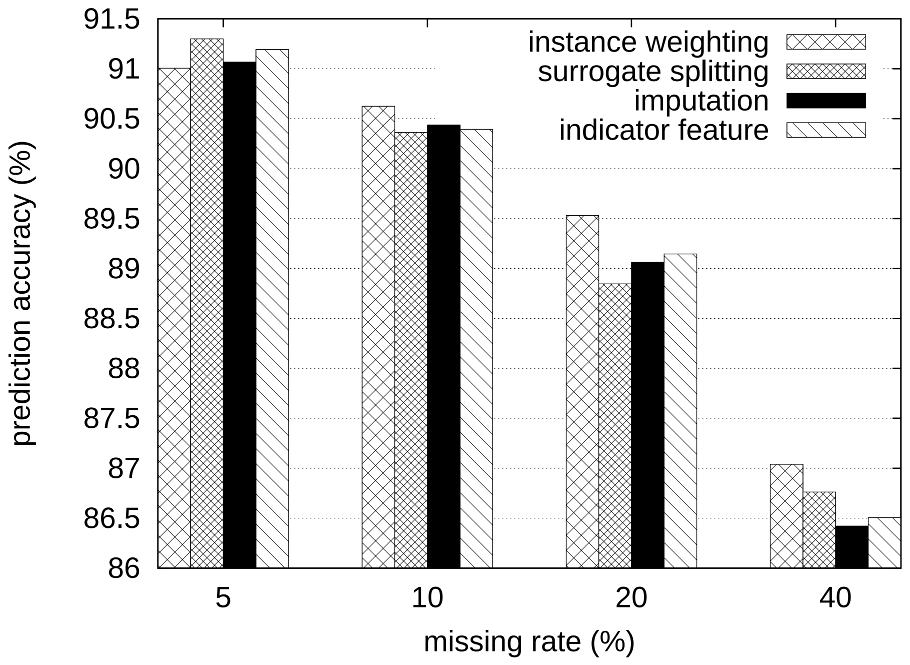


(a) Protein

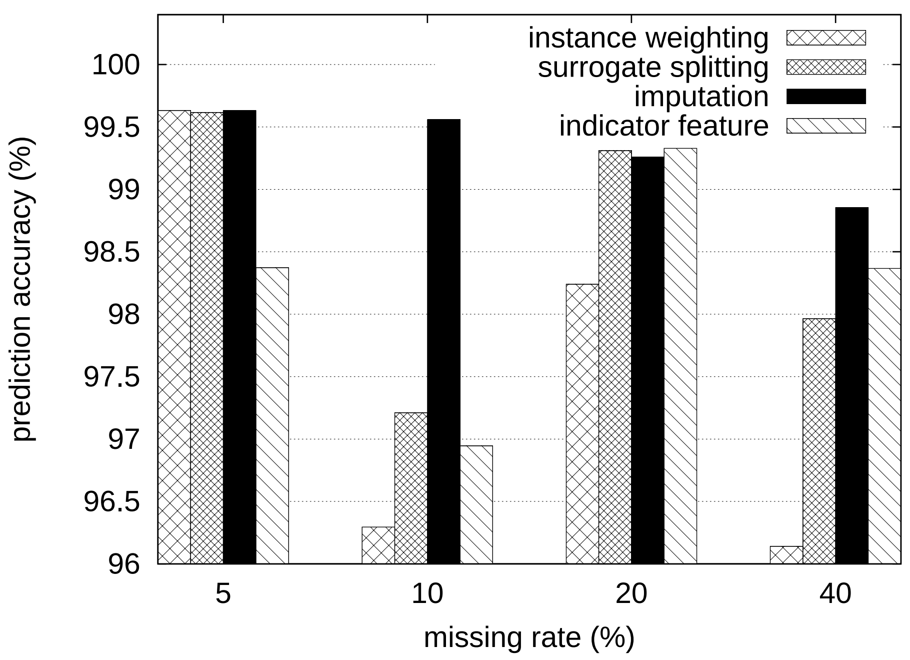


(b) NETtalk

Figure 4.1: Performance of missing values methods for different missing rates.



(c) Hyphen



(d) FAQ ai-general

Figure 4.1: Performance of missing values methods for different missing rates (Continued).

Table 4.1: Estimation of the coefficients corresponding to different missing values methods and statistical test results. In FAQ ai-general problem, imputation was the baseline method, so the coefficient values give the log odds of the change in accuracy relative to imputation. * means that the parameter value is statistically significantly different from zero ($p < 0.05$).

Missing rate	Surrogate splitting	Imputation	Indicator feature
5%	−0.018	−0.072*	−0.028*
10%	−0.013	−0.040*	0.001
20%	−0.025*	−0.074*	−0.020*
40%	−0.041*	−0.072*	−0.020*

(a) Protein

Missing rate	Surrogate splitting	Imputation	Indicator feature
5%	−0.051*	−0.066*	−0.064*
10%	−0.051*	−0.067*	−0.059*
20%	−0.069*	−0.057*	−0.052*
40%	−0.080*	−0.116*	−0.111*

(b) NETtalk

Missing rate	Surrogate splitting	Imputation	Indicator feature
5%	0.036*	0.007	0.023
10%	−0.031*	−0.022	−0.027*
20%	−0.071*	−0.049*	−0.040*
40%	−0.024*	−0.054*	−0.047*

(c) Hyphen

Missing rate	Instance weighting	Surrogate splitting	Indicator feature
5%	−8.824E−16	−0.043	−1.499*
10%	−2.161*	−1.867*	−1.961*
20%	−0.874*	0.072	0.100
40%	−1.243*	−0.584*	−0.359*

(d) FAQ ai-general

4.4.3 Hyphenation

Figure 4.1c shows that instance weighting is the best missing values method except for a missing rate of 5%. Statistical tests shown in Table 4.2c tell us that for missing rate of 5%, surrogate splitting is the best missing values method and the other three methods are not statistically significantly different from each other. For a missing rate of 10%, instance weighting and imputation are statistically better than the other two methods (and indistinguishable from each other). For missing rates of 20% and 40%, instance weighting is statistically better than the other three methods.

4.4.4 FAQ Document Segmentation

This task is based on the ai-general Usenet FAQ data set as we discussed in Chapter 3. We treat the first 6 files as the training set and the seventh file as the test set. The input window contains only the features corresponding to a single line in the file (window half-width of 0). Unlike in the previous data sets, instance weighting is no longer the best missing values method, as shown in Figure 4.1d. Instead, imputation performs very well for various missing value rates. Table 4.2d shows that imputation is statistically the best missing values method. For missing rates of 10% and 40%, it is statistically better than the other three methods. For a missing rate of 5%, it does as well as instance weighting and surrogate splitting. For a missing rate of 20%, it does as well as surrogate splitting and indicator features.

4.4.5 Analysis and Discussion

The four missing values methods are based on different assumptions about the input data. Imputation assumes that the most frequent value of a feature is the least informative and therefore presents the lowest risk of introducing errors into the learning process. Missing values are injected prior to converting the input features to binary. Hence, in the protein data set, missing values are introduced by choosing an amino acid residue position in the observation sequence and setting all 20 boolean indicator features that represent that position to missing. Similarly, in the NETtalk and hyphenation problems, a letter is made to be missing by setting all 26 indicator features for that letter to missing. Similarly, imputation is computed at the amino acid or letter level, not at the level of boolean features. However, in the Usenet FAQ data set, since the binary features are not exclusive, imputation is computed at the level of boolean features. In the case of protein sequences, imputation will replace missing values with the most frequently-occurring amino acid, which is alanine, code ‘A’. Alanine tends to form alpha helices, so this may cause the learning algorithms to over-predict the helix class, which may explain why imputation performed worst on the protein data set. In the case of English words, the most common letter is ‘E’, and it does not carry much information either about pronunciation or about hyphenation, so this may explain why imputation worked well in the NETtalk and hyphenation problems. Finally, in the ai-general FAQ data set, most of the features exhibit a highly skewed distribution, so that one feature value is much more common than another, as shown in Figure 4.2. Hence, in most cases, imputation with the most common feature value will supply the correct missing value. This may be why

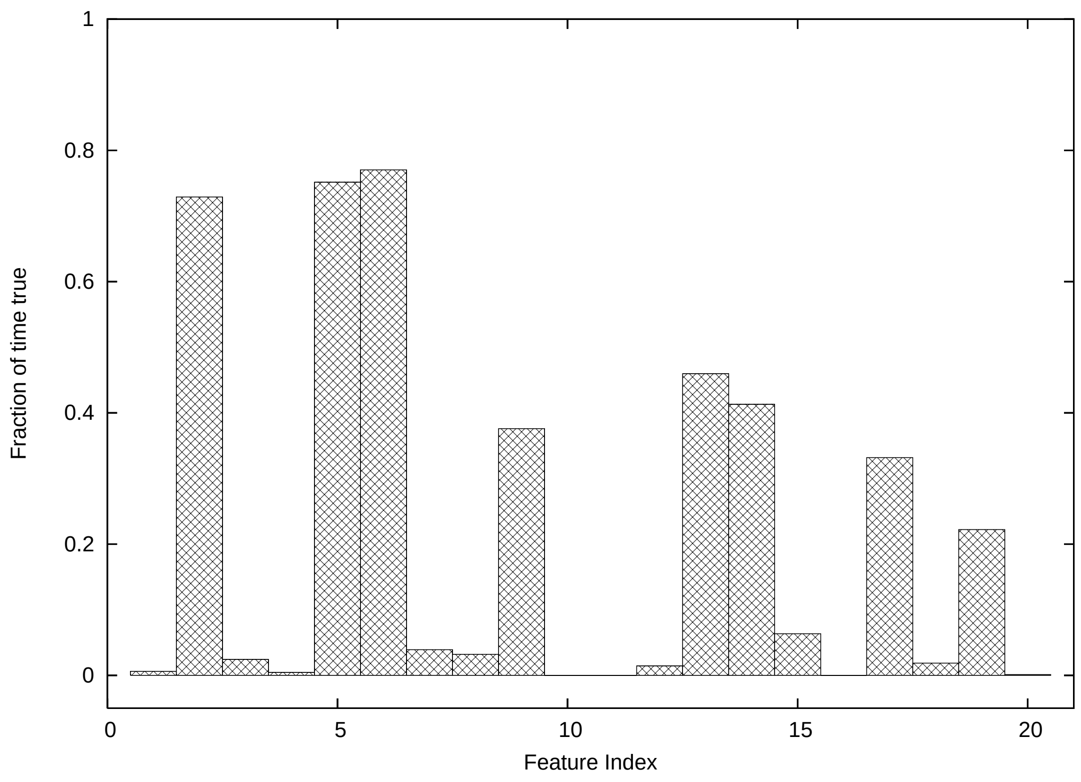


Figure 4.2: Fraction of the time that each FAQ feature is true (versus false). Features 1, 3, 4, 7, 8, 10, 11, 12, 16, 18, and 20 are rarely true.

it worked best on that data set.

The indicator feature approach is based on the assumption that the presence or absence of a feature is meaningful (e.g., in medicine, a feature could be missing because a physician explicitly chose not to measure it). Because features were marked as missing completely at random, this is not true, so the indicator feature carries no positive information about the class label. However, in cases where imputation causes problems, the indicator feature approach may help prevent those problems by being more neutral. The learning algorithm can learn that if the indicator feature is set, then the actual feature value should be ignored. This may explain why the indicator feature method works slightly better in most cases than the imputation method.

The surrogate splitting method assumes that the input features are correlated with one another, so that if one feature is missing, its value can be computed from another feature. The protein, NETtalk, and hyphenation data sets have a single input feature for each amino acid or letter. Hence, if this input feature is missing, then there is no information about that position in the sequence. The only exception to this would be if there were strong correlations between successive amino acids or letters. However, such strong correlations do not exist much either in protein sequences or in English, with the possible exception of the letter ‘q’, which is always followed by ‘u’. Note that the converse is not true: ‘u’ is not always preceded by ‘q’. Based on these considerations, we would not expect surrogate splitting to work well in these domains, and it does not.

In the FAQ data set, each line is described by 20 features computed from the words in that line. In the experiment, each of these 20 features could be independently marked as missing, which is a bit unrealistic, since presumably the real missing values would involve some loss or corruption of the words making up the line, and this would affect multiple features. The 20 features do have some redundancy, so we would expect that surrogate splitting should work well, and it does for 5% and 20% missing rates.

The instance weighting method assumes that the feature values are missing at random and that the other features provide no redundant information, so the most sensible thing to do is to marginalize away the uncertainty about the missing values. Our experiments show that this is a very good strategy in all cases except for the FAQ data set, where the features are somewhat redundant.

Chapter 5 – Discovering Future Features in Sequential Supervised Learning

Most existing models for structured learning problem are based on the Markovian dependency assumption and use dynamic programming techniques, for example, the Viterbi algorithm, for inference. However, the Markovian assumption makes it hard to capture long term dependencies among the output labels. Furthermore, dynamic programming is impractical in many complex structured learning problems, especially in natural language processing. Recently a new family of approaches have been proposed, which use approximate search methods, for example, beam search, to incrementally infer structured output labels, and model parameters are learned to optimize predictive accuracy when used with this search process. Each state s in the search space $\mathcal{S}$ consists of partially labeled structured outputs. A linear ranking function

$$f(s, X) = \mathbf{w} \cdot \Phi(X, s) \tag{5.1}$$

is defined to compute a score for the state s, where $\mathbf{w}$ is a weight vector and Φ is a vector of features. In each step of beam search with beam width b, the value of Eq. (5.1) is evaluated at each candidate state and only the b highest ranked states are kept and expanded to get the next set of candidate states. The weight parameter $\mathbf{w}$ is trained to ensure that at each search step there exists at least one state in the beam that can lead to a goal state. For a given state s, it is expanded by providing class labels for the next unlabeled position in the structured output. For example, when the output structure is a linear chain with length T, a state s_t can be represented as

$$s_t = (y_1, \ldots, y_t, \perp, \ldots, \perp) \ \ ,$$

which consists of class labels only for positions from 1 to t. In this state, positions from $t+1$ to T are not labeled, which is indicated by the special symbols $\perp$ at these positions. Expanding the state s_t will return a set of states. Each state in this set is in the form of

$$s_{t+1} = (y_1, \ldots, y_t, y_{t+1}, \perp, \ldots, \perp) \ \ ,$$

which means it provides a class label y_{t+1} for position $t+1$ while keeps the class labels for positions from 1 to t the same as in the state s_t.

The advantage of this method is that the decoding process does not rely on dynamic programming and thus it becomes possible to use non-local (i.e., higher order) output dependencies. Collins and Roark (2004) first used this idea for incremental parsing. Later on, Daumé III and Marcu (2005) proposed the LaSO framework to solve syntactic chunking and joint tagging/chunking tasks. Recently, Xu et al. (2007) improved the LaSO framework and applied it in a planning domain.

5.1 Motivation

Previous work on search based structured learning mostly focused on unidirectional (for example, left-to-right) greedy beam search, so the feature vector in Eq. (5.1) only depends on the input X and the preceding output labels (which are represented by the search state s) at each search step. However, this information is not sufficient to resolve local ambiguity in some cases. An example is given by Ashenfelter (2003), where the task is to pronounce "photograph" and "photography", whose pronunciations are totally different. If the search process runs from left to right, it will have exactly the same partial pronunciations for these two words before the step in which the position of the ending letter "y" is considered in the input features. Pronunciation errors made before this step cannot be recovered in the later search process.

This problem can possibly be avoided by running the search process from right to left, as was done by Bakiri and Dietterich (1997). However, in some cases, unidirectional search, either left-to-right or right-to left cannot make correct predictions. Here is an example. Consider a hidden Markov model where the label set is $\mathcal{L} = \{l_1, l_2, l_3\}$ and the observation set is $\mathcal{E} = \{e_1, e_2, e_3\}$. Define the transition model as

$$P(y_t = l_i \mid y_{t-1} = l_j) = \begin{cases} 0.9 & if \quad i = j \\ 0.05 & if \quad i \neq j \end{cases}$$

and the observation model as

$$P(x_t = e_1 \mid y_t) = \begin{cases} 0.45 & if \quad y_t = l_1 \\ 0.8 & if \quad y_t = l_2 \\ 0.1 & if \quad y_t = l_3 \end{cases},$$

$$P(x_t = e_3 \mid y_t) = \begin{cases} 0.45 & if \quad y_t = l_1 \\ 0.1 & if \quad y_t = l_2 \\ 0.8 & if \quad y_t = l_3 \end{cases},$$

$$P(x_t = e_2 \mid y_t) = 0.1 \text{ for any } y_t.$$

Given an observation sequence $(x_1, x_2, x_3) = (e_1, e_2, e_3)$, use the forward-backward algorithm to find the most probable label for x_2. In the forward pass, we have

$$\begin{aligned} \alpha_2(l_1) &= 0.1 * (0.45 * 0.9 + 0.8 * 0.05 + 0.1 * 0.05) = 0.045 \\ \alpha_2(l_2) &= 0.1 * (0.45 * 0.05 + 0.8 * 0.9 + 0.1 * 0.05) = 0.07475 \\ \alpha_2(l_3) &= 0.1 * (0.45 * 0.05 + 0.8 * 0.05 + 0.1 * 0.9) = 0.01525. \end{aligned}$$

In the backward pass, we have

$$\begin{aligned} \beta_2(l_1) &= 0.45 * 0.9 + 0.1 * 0.05 + 0.8 * 0.05 = 0.45 \\ \beta_2(l_2) &= 0.45 * 0.05 + 0.1 * 0.09 + 0.8 * 0.05 = 0.1525 \\ \beta_2(l_3) &= 0.45 * 0.05 + 0.1 * 0.05 + 0.8 * 0.9 = 0.7475. \end{aligned}$$

So the forward pass predicts the label of x_2 as l_2, while the backward pass predicts the label of x_2 as l_3. But both predictions are incorrect, because by combining the forward pass and backward pass together, the label of x_2 is found to be l_1. This example shows that both forward-pass-only and backward-pass-only will fail in some cases even when the full forward-backward algorithm works. The problem in this example cannot be solved by reversing the sequence.

One way to handle this problem is to use a large sliding window over the input X. But this will introduce a large number of features into the model and thus hurt generalization. Another drawback of using sliding windows is that features extracted from a sliding window are associated with specific positions within the sliding window, which gives us less flexibility in feature engineering. What we want is that at time t, we compute a score for the state s_t as

$$f(s_t, X_{1:t}, \mathbf{h}_t) \ ,$$

where $X_{1:t}$ represents the input features from position 1 to position t, and $\mathbf{h}_t$ is a feature vector, which is called *future feature vector*. A future feature c inside $\mathbf{h}_t$ takes the value 1 if and only if there exists a future position $t' > t$ such that feature c is true for input observation $\mathbf{x}_{t'}$. We formulate the problem of learning future features as a multiple-instance learning problem.

5.2 Review of Multiple-Instance Learning (MIL)

In a standard single instance learning scenario, the training data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ consists of N instances, where $\mathbf{x}_i \in \mathcal{X}$ is an instance (the feature vector) and $y_i \in \mathcal{Y} = \{0, 1\}$ is the corresponding *instance label*. The goal is to learn a classifier $f : \mathcal{X} \to \mathcal{Y}$.

Multiple-instance learning (MIL) is a generalization of standard supervised learning, and it can be formulated as follows:

Given: a training data set $\mathcal{D} = \{(B_i, y_i)\}_{i=1}^N$ consisting of N bags, where each bag $B_i = \{\mathbf{x}_{i1}, \ldots, \mathbf{x}_{in_i}\}$ consists of n_i instances and $y_i \in \mathcal{Y} = \{0, 1\}$ is the corresponding *bag label*.

Assume: there exists an unknown concept f that classifies each individual instance $\mathbf{x}_{ij}$ as 1 or 0 (positive or negative), and the bag label y_i is determined as follows:

- $y_i = 1$ if there is at least one instance $\mathbf{x}_{ij} \in B_i$ such that $f(\mathbf{x}_{ij}) = 1$, and
- $y_i = 0$ if $f(\mathbf{x}_{ij}) = 0$ for every instance $\mathbf{x}_{ij} \in B_i$.

Goal: learn the concept f to predict unseen instances and/or bags.

In multiple-instance learning, only bag labels are known, and instance labels are not directly provided. For instances in a negative bag, all of them are negative instances. However, for instances in a positive bag, it is unknown which ones are the actual positive instances and which ones are not. How to deal with this ambiguity is the key challenge in MIL.

Multiple-instance learning was first introduced by Dietterich et al. (1997) in the context of drug activity prediction. In that task, each molecule is represented as a bag of possible conformations. If a molecule shows drug-like activity (a positive bag), it can be inferred that at least one of its conformations is able to bind to the target binding site (a positive instance). On the other hand, if a molecule does not show drug-like activity (a negative bag), it can be inferred that none of its conformations is able to bind to the target site (negative instances). Experimentally, it is only possible to test the efficacy of a molecule, not of individual conformations. MIL has also been applied to many other real-world tasks, such as content-based image retrieval (Maron and Ratan, 1998; Zhang et al., 2002), text categorization (Andrews et al., 2002), face detection (Viola et al., 2006), and so on.

After the axis-parallel rectangle (APR) algorithms proposed by Dietterich et al. (1997), many algorithms have been developed to solve MIL problems. A framework called Diverse Density (DD) was proposed by Maron and Lozano-Pérez (1998). This algorithm was later combined with the EM algorithm by Zhang and Goldman (2001) to create the EM-DD algorithm. Many standard single instance learning algorithms have been adapted to the multiple-instance setting in the past, such as nearest neighbor (Wang and Zucker, 2000), decision trees (Chevaleyre and Zucker, 2001; Blockeel et al., 2005), SVMs (Andrews et al., 2002), neural networks (Ramon and De Raedt, 2000; Zhou and Zhang, 2002), logistic regression (Ray and Craven, 2005), boosting (Auer and Ortner, 2004; Xu and Frank, 2004; Viola et al., 2006), and so on. A good survey of multiple-instance learning was given by Zhou (2004).

In this chapter, we take the approach introduced by Viola et al. (2006) to extend single instance logistic regression to the multiple-instance setting. For each instance $\mathbf{x}_{ij}$

in the bag B_i, the probability of this instance being positive is given by

$$p_{ij} = P(y_{ij} = 1 \mid \mathbf{x}_{ij}) = \frac{1}{1 + exp[-F(\mathbf{x}_{ij})]} \quad , \tag{5.2}$$

where $F(\mathbf{x}_{ij})$ is the score function, which is usually represented as a linear combination of the features in $\mathbf{x}_{ij}$. The probability that the bag B_i is positive is given by

$$p_i = P(y_i = 1 \mid B_i) = 1 - \prod_{j=1}^{n_i} (1 - p_{ij}) \quad , \tag{5.3}$$

which is a noisy OR (Pearl, 1988). The score function $F(\cdot)$ can be trained by maximizing the conditional log-likelihood

$$\ell(\mathcal{D}) = \sum_{i=1}^{N} y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

over the training data set $\mathcal{D}$ using gradient based methods.

5.3 MIL-based Future Feature Discovery

As discussed in Chapter 1, each training example in a sequential supervised learning problem can be represented as a pair (X, Y), where $X = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$ is a sequence of T feature vectors, $Y = (y_1, y_2, \ldots, y_T)$ is a corresponding sequence of class labels, and $y_t \in \mathcal{Y} = \{1, \ldots, L\}$ for $1 \leq t \leq T$. Let

$$B_t = \{\mathbf{x}_{t+1}, \mathbf{x}_{t+2}, \ldots, \mathbf{x}_{t+u}\}$$

denote a bag of u future instances starting from position $t+1$, where u can take a fixed value or take the value $(T-t)$ so that it extends to the end of the sequence. Our intuition is that at position t in the sequence, if there exist dependencies between class label y_t and class labels at some future positions, some unknown pattern exhibited by at least one future instance $\mathbf{x}_{t'} \in B_t$ will be useful for predicting class label y_t. That is to say, some pattern exhibited in the bag B_t can be useful for predicting y_t. We call this pattern a *future feature*. We will use a MIL-based algorithm to discover a set of future features.

5.3.1 Future Feature Model

At each position t in a sequence, we consider three sources of information to the predict class label y_t: the feature vector $\mathbf{x}_t$ or $w_t(X)$ in general, the class labels at previous positions $y_{t-1}, \ldots, y_{t-k}$, and the bag of future instances B_t. We call the first two sources the *local features*. As in the Maximum Entropy Markov Model (MEMM) proposed by McCallum et al. (2000), we can represent the conditional probability of the class label y_t as

$$P(y_t \mid y_{t-1}, \ldots, y_{t-k}, \mathbf{x}_t, B_t) = \frac{\exp F^{y_t}(y_{t-1}, \ldots, y_{t-k}, \mathbf{x}_t, B_t)}{\sum_{l=1}^{L} \exp F^{l}(y_{t-1}, \ldots, y_{t-k}, \mathbf{x}_t, B_t)} \quad ,$$

where each function $F^l(\cdot)$ is the score function for class label $l \in \mathcal{Y}$. We call this model a *k-th order Future Feature MEMM*. The model is trained to maximize the log-likelihood

$$\log P(Y \mid X) = \sum_{t=1}^{T} \log P(y_t \mid y_{t-1}, \ldots, y_{t-k}, \mathbf{x}_t, B_t). \tag{5.4}$$

Unlike a MEMM, at each position t in the sequence, this model makes use of future features extracted from the bag B_t to resolve local ambiguity. The training algorithm for this model is discussed in detail in the next section.

5.3.2 Training Algorithm

In this section, we discuss the training of a first order Future Feature MEMM with binary class labels. That is to say, $k = 1$ and $\mathcal{Y} = \{0, 1\}$. In this case, the conditional probability of class label y_t being 1 is written as

$$P(y_t = 1 \mid y_{t-1}, \mathbf{x}_t, B_t) = \frac{\exp F(y_{t-1}, \mathbf{x}_t, B_t)}{1 + \exp F(y_{t-1}, \mathbf{x}_t, B_t)} \quad ,$$

where $F(\cdot)$ is the score function. This function will be learned by maximizing the log-likelihood function specified in Equation 5.4 with the functional gradient tree boosting method introduced in section 3.2.

Suppose the current estimation of the score function $F(\cdot)$ is $F_{m-1}(\cdot)$:

$$F(\cdot) = F_{m-1}(\cdot) \quad .$$

We use $\mathbf{h}_t^{(m)}$ to denote the future features extracted from bag B_t so far. The vector $\mathbf{h}_t$ is initialized to be an empty feature vector before training, and is extended incrementally during the training process. Two options are considered for the next training iteration.

First Option. Assume that the next update of the score function $F(\cdot)$ is of the form

$$F(\cdot) = F_m(\cdot) = F_{m-1}(\cdot) + \alpha_m \cdot f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t) \ . \tag{5.5}$$

Function $f(\cdot)$ is called the *top level update function*. It only depends on local features and existing future features.

Proposition 5.1 *The top level update function* $f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)$ *in Equation 5.5 can be computed as*

$$f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t) = I(y_t = 1) - P(y_t = 1 \mid y_{t-1}, \mathbf{x}_t, B_t) \ ,$$

where $I(y_t = 1)$ *is* 1 *if class label* y_t *is* 1 *and* 0 *otherwise, and where* $P(y_t = 1 \mid y_{t-1}, \mathbf{x}_t, B_t)$ *is computed according to the current score function* $F(\cdot) = F_{m-1}(\cdot)$.

The proof of this proposition is given in Table 5.1. In Equation 5.6, only one of the $P(y_t \mid y_{t-1}, \mathbf{x}_t, B_t)$ terms will contain $f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)$ if we assume that each $(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)$ is unique. Equation 5.8 is derived by writing the log-likehood function in Equation 5.7 in the cross-entropy form.

As in the TREECRF algorithm, we can fit a regression tree to approximate the function $f(\cdot)$. Thus in this option, we are adding a new *top level regression tree* to the score function $F(\cdot)$ in each training iteration, and this regression tree only depends on local features and existing future features. The step size α_m in Equation 5.5 can be determined by a line search.

Table 5.1: Derivation of the top level update function.

$$\begin{aligned}
& f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t) \\
= \quad & \frac{\partial \log P(Y|X)}{\partial f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)} \Bigg|_{F=F_m\ ,\ f=0} \\
= \quad & \frac{\partial}{\partial f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)} \sum_{t=1}^{T} \log P(y_t \mid y_{t-1}, \mathbf{x}_t, B_t) \Bigg|_{F=F_m\ ,\ f=0} && (5.6) \\
= \quad & \frac{\partial}{\partial f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)} \log P(y_t \mid y_{t-1}, \mathbf{x}_t, B_t) \Bigg|_{F=F_m\ ,\ f=0} && (5.7) \\
= \quad & \frac{\partial}{\partial f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)} \Big[y_t F(y_{t-1}, \mathbf{x}_t, B_t) - \log \big(1 + \exp F(y_{t-1}, \mathbf{x}_t, B_t)\big) \Big] && (5.8) \\
= \quad & \frac{\partial}{\partial f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)} \Big[y_t F_{m-1}(y_{t-1}, \mathbf{x}_t, B_t) + y_t f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t) \Big] \Bigg|_{f=0} \\
& - \frac{\partial}{\partial f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t)} \log \Big[1 + \exp \big[F_{m-1}(y_{t-1}, \mathbf{x}_t, B_t) + f(y_{t-1}, \mathbf{x}_t, \mathbf{h}_t) \big] \Big] \Bigg|_{f=0} \\
= \quad & I(y_t = 1) - P(y_t = 1 \mid y_{t-1}, \mathbf{x}_t, B_t)|_{F=F_{m-1}}
\end{aligned}$$

Second Option. Assume that the next update of score function $F(\cdot)$ is of the form

$$F(\cdot) = F_m(\cdot) = F_{m-1}(\cdot) + \eta_m \cdot Q(B_t) \tag{5.9}$$

That is to say, the update function $Q(\cdot)$ only depends on future instances in bag B_t. Inspired by the multiple-instance learning (MIL) introduced in section 5.2, we treat function $Q(\cdot)$ as the probability of bag B_t exhibiting some unknown pattern. Similar to Equation 5.2 and Equation 5.3, let p_{tj} be the probability that instance $\mathbf{x}_{t+j} \in B_t$ is positive, which can be written as

$$p_{tj} = \frac{\exp G(\mathbf{x}_{t+j})}{1 + \exp G(\mathbf{x}_{t+j})} \ , \tag{5.10}$$

where the function $G(.)$ is the score function corresponding to the unknown pattern, and $1 \leq j \leq u$. Then function $Q(B_t)$, the probability of bag B_t being positive, can be written as

$$Q(B_t) = p_t = 1 - \prod_{j=1}^{u} (1 - p_{tj}) \ . \tag{5.11}$$

In this training iteration, our goal is to update the score function $G(\cdot)$ based on its current definition $G_{n-1}(\cdot)$. That is to say, we want to compute a *lower level update function* $g(\cdot)$ such that

$$G(\cdot) = G_n(\cdot) = G_{n-1}(\cdot) + \beta_n \cdot g(\cdot) \tag{5.12}$$

maximizes the log-likelihood in Equation 5.4 based on the representation of the score function $F(\cdot)$ in Equation 5.9. This can be done by the functional gradient tree boosting algorithm as well.

Proposition 5.2 *The lower level update function $g(\mathbf{x}_{t+j})$ in Equation 5.12 can be computed as*

$$g(\mathbf{x}_{t+j}) = \eta_m \Big[I(y_t = 1) - p(y_t = 1 \mid y_{t-1}, \mathbf{x}_t, B_t) \Big] \cdot (1 - p_t) \cdot p_{tj} \ ,$$

where $I(y_t = 1)$ is 1 if the class label y_t is 1 and 0 otherwise, $P(y_t = 1 \mid y_{t-1}, \mathbf{x}_t, B_t)$ is computed according to the current definition of the score function $F_m(\cdot)$, and p_{tj} and p_t

Table 5.2: Derivation of the lower level update function.

$$
\begin{aligned}
& g(\mathbf{x}_{t+j}) \\
= \; & \left.\frac{\partial \log P(Y|X)}{\partial G(\mathbf{x}_{t+j})}\right|_{F=F_{m-1}+\eta_m\cdot Q_{n-1}} \\
= \; & \left.\frac{\partial}{\partial G(\mathbf{x}_{t+j})}\sum_{t=1}^{T}\log P(y_t \mid y_{t-1},\mathbf{x}_t,B_t)\right|_{F=F_{m-1}+\eta_m\cdot Q_{n-1}} \\
= \; & \left.\frac{\partial}{\partial G(\mathbf{x}_{t+j})}\log P(y_t \mid y_{t-1},\mathbf{x}_t,B_t)\right|_{F=F_{m-1}+\eta_m\cdot Q_{n-1}} \qquad (5.13) \\
= \; & \left.\frac{\partial \log P(y_t|y_{t-1},\mathbf{x}_t,B_t)}{\partial Q(B_t)}\right|_{F=F_{m-1}+\eta_m\cdot Q_{n-1},Q=Q_{n-1}} \\
& \left.\cdot\frac{\partial Q(B_t)}{\partial p_{tj}}\cdot\frac{\partial p_{tj}}{\partial G(\mathbf{x}_{t+j})}\right|_{G=G_{n-1}} \qquad (5.14) \\
= \; & \eta_m\Big[I(y_t=1)-p(y_t=1 \mid y_{t-1},\mathbf{x}_t,B_t)|_{F=F_{m-1}+\eta_m\cdot Q_{n-1}}\Big] \qquad (5.15) \\
& \left.\cdot\left[\prod_{i=1,i\neq j}^{u}(1-p_{ti})\right]\cdot p_{tj}(1-p_{tj})\right|_{G=G_{n-1}} \\
= \; & \eta_m\Big[I(y_t=1)-p(y_t=1 \mid y_{t-1},\mathbf{x}_t,B_t)\Big]\cdot(1-p_t)\cdot p_{tj}.
\end{aligned}
$$

are computed according to $G(\cdot)=G_{n-1}(\cdot)$.

The proof of this proposition is given in Table 5.2. Function $Q_{n-1}(\cdot)$ is computed according to Equation 5.10 and Equation 5.11 with $G(\cdot)=G_{n-1}(\cdot)$. The chain rule is applied to expand the partial derivative in Equation 5.13 into three terms as shown in Equation 5.14. As in the first option, a regression tree can be fit to approximate the function $g(\cdot)$. So in this option, a new *lower level regression tree* is added in one training iteration. A line search can be used to determine the step size β_n in Equation 5.12.

Training Algorithm. The overall training algorithm involves alternating between iterations with the first option and iterations with the second option. The basic idea is that we keep training the model based on the first option until the performance cannot be further improved. That means further exploiting local features and existing future features does not improve the log likelihood. At such a point, the algorithm needs to discover some new future features by performing training iterations based on the second option. When the log likelihood of these iterations stops improving, the value of $Q(B_t)$ serves to define a new future feature extracted from bag B_t. This new future feature is then added to the future feature vector $\mathbf{h}_t$ to obtain a new future feature vector for position t. In our current implementation, we evenly discretize the value range of $Q(\cdot)$ function into 20 bins and use the thermometer representation to represent the value of $Q(B_t)$. The step size η_m in Equation 5.9 is then determined by an additional line search. After introducing this new feature feature, the algorithm returns to performing training iterations based on the first option.

5.4 Experimental Results

We implemented the training algorithm for our Future Feature MEMM model. We call this algorithm FF-MEMM. In this section, we compare this algorithm to three other sequential supervised learning algorithms: 1) sliding windows method denoted by "SW", 2) the original MEMM algorithm proposed by McCallum et al. (2000), and 3) CRFs. We use "FF-MEMM-i", "MEMM-i", and "CRF-i" to denote the corresponding models with the i-th order Markov assumption. CRF models are trained by the TREECRF algorithm as described in Chapter 3. The implementation of TREECRF package is also able to train MEMM models via the functional gradient tree boosting method. Sliding windows models are treated as zeroth-order MEMM models and thus are trained using the same code.

For these four algorithms, a common set of parameters must be set by the user, which include (a) the window size, (b) the order of the Markov model, (c) the number of iterations to train, and (d) the maximum number of leaves in the regression trees, which is used to regularize trees. For the FF-MEMM algorithm, additional parameters include (a) the size of bag B_t for each position t, (b) the value range of step sizes α_m in the top level line search, (c) the value range of step sizes β_n in the lower level line search, (d) the value range of step sizes η_m in the top level line search when introducing a new future feature, and (e) the maximum number of iterations with the second option before the beginning of the next training iteration with the first option (kept constant at 10). The lower bound of the step size value ranges is always set to 0. Hold-out validation is used for parameter selection.

Throughout the experiments, the prediction accuracy is measured according to the fraction of correctly labeled sequence elements. For CRF models, we use the forward-backward algorithm to make label sequence predictions.

5.4.1 Synthetic Data Sets

We evaluated the performance of our algorithm on three synthetic data sets. Each data set consists of 400 training sequences, 200 validation sequences, and 100 test sequences.

The length of each sequence is 30. At each position t in a sequence, we generated a random feature vector $\mathbf{x}_t$, which consisted of 8 binary features. That is to say, $\mathbf{x}_t = (x_{t,1}, x_{t,2}, \ldots, x_{t,8})$.

For each position t in a sequence, we assigned a local label y_t^{local}. If at least 4 of the first 6 bits in $\mathbf{x}_t$ were true, then $y_t^{local} = 1$. Otherwise, $y_t^{local} = 0$. This is a *4-of-6* problem at each position t. As a result, using the sliding window method with window size 1, we can learn this SSL problem perfectly with simple learning algorithms such as logistic regression. The last 2 bits in $\mathbf{x}_t$ serve as random noise.

In the next three data sets, at each position t in a sequence, we consider a bag of 6 future instances, $B_t = \{\mathbf{x}_{t+1}, \ldots, \mathbf{x}_{t+6}\}$. An instance in the bag is a *positive instance* if and only if *at most* 1 of the first 6 bits in it is true. The bag label y_t^{bag} for B_t is 1 if it contains at least 1 positive future instance. Otherwise, $y_t^{bag} = 0$. The reason why we chose the bag of size 6 is that the probability of a bag being positive is about 0.5 if every feature vector $\mathbf{x}_t$ is generated randomly.

XOR Data Set. In this data set, we assigned the class label y_t^{xor} to each position t, where

$$y_t^{xor} = y_t^{local} \oplus y_t^{bag} \quad ,$$

and $\oplus$ is the XOR operator. This is a very hard problem for the sliding window method, because two identical feature vectors appearing at different positions in a sequence will have opposite class labels with probability of about 0.5.

OR Data Set. In this data set, we assigned the class label y_t^{or} to each position t, where

$$y_t^{or} = y_t^{local} \vee y_t^{bag} \quad ,$$

and $\vee$ is the OR operator.

EXTRA Data Set. In this data set, y_t^{bag} is used as an extra input feature at position t. We assigned the class label y_t^{extra} to each position t, where $y_t^{extra} = 1$ if the number of true features in $(y_t^{bag}, x_{t,1}, \ldots, x_{t,6})$ is at least 4. Otherwise, $y_t^{extra} = 0$.

In these three data sets, if the algorithm can correctly extract the *future feature* y_t^{bag} from bag B_t, then it can learn these SSL problems perfectly even with the sliding window method. Otherwise, it is doomed to fall into local ambiguity.

We tested the performance of the FF-MEMM algorithm over these synthetic data sets in order to get some insights about this algorithm's characteristics and how it works. In the experiments here, we set the window size to 1, the maximum number of leaves in the regression trees to 10, and the bag size to 6. For the score function $G(\cdot)$ in Equation 5.12, we chose different initial values G_0. For each G_0, we performed hold-out validation to select the other parameters, and the best values are reported.

Based on the results shown in Table 5.3. We have the following observations.

1. As shown in the columns "Instance Accuracy" and "Bag Accuracy", the algorithm can identify bag labels y_t^{bag} very well, because it can identify positive and negative instances in bags very well. This property is fairly robust to different initial values G_0. For initial values marked with "*", only one score function $G(\cdot)$ is learned. That is to say, one future feature is enough to correctly represent y_t^{bag}. For initial values marked with "**", two score functions $G(\cdot)$ are learned. For

Table 5.3: Performance of the FF-MEMM algorithm on synthetic data sets, where * means one future feature is extracted, ** means two future features are extracted, and *** means more than two future feature are extracted.

	Step Size Upper Bound				Prediction Accuracy (%)			Instance Accuracy (%)			Bag Accuracy (%)		
G_0	α	β	η	Iterations	Train	Val	Test	Train	Val	Test	Train	Val	Test
0**	40	10^5	10	29	100	100	100	100	100	100	100	100	100
-1***	40	10^4	10	52	99.17	98.67	98.97	N/A	N/A	N/A	N/A	N/A	N/A
-5**	40	5000	10	34	100	100	100	100	100	100	100	100	100
-7**	60	5000	10	32	99.98	99.92	99.97	100	100	100	100	100	100

(a) XOR data set

	Step Size Upper Bound				Prediction Accuracy (%)			Instance Accuracy (%)			Bag Accuracy (%)		
G_0	α	β	η	Iterations	Train	Val	Test	Train	Val	Test	Train	Val	Test
0**	20	5000	40	11	100	100	100	100	100	100	100	100	100
-1***	40	10^6	40	32	99.98	99.90	99.97	N/A	N/A	N/A	N/A	N/A	N/A
-3***	40	10^5	20	43	100	99.98	100	N/A	N/A	N/A	N/A	N/A	N/A
-5*	40	10^5	20	11	100	100	100	100	100	100	100	100	100
-7*	40	10^5	20	11	100	100	100	100	100	100	100	100	100
-10**	40	10^6	20	13	100	100	100	100	100	100	100	100	100

(b) OR data set

	Step Size Upper Bound				Prediction Accuracy (%)			Instance Accuracy (%)			Bag Accuracy (%)		
G_0	α	β	η	Iterations	Train	Val	Test	Train	Val	Test	Train	Val	Test
0***	60	10^5	10	49	100	100	100	100	100	100	100	100	100
-5*	40	10^4	10	43	100	100	100	100	100	100	100	100	100
-7*	40	10^4	10	43	100	100	100	100	100	100	100	100	100
-10***	60	10^5	20	51	99.99	99.97	99.93	100	100	100	100	100	100

(c) EXTRA data set

any positive instance or bag, at least one score function can identify it as positive. For any negative instance or bag, both score functions identify it as negative. So the combination of predictions made by these two score functions, that is to say, the combination of the two extracted future features, can represent bag labels y_t^{bag} correctly. For initial values marked with "***", it was sometimes difficult to determine whether the algorithm had correctly learned to predict the instances, so these were marked as N/A. However, the learned model can still make accurate predictions over test sequences. We believe this is because that the extracted future features can represent y_t^{bag} very well.

2. The step size β_m in Equation 5.12 can be very large, because the functional gradient residues $g(\cdot)$ are very small, and only large step sizes are able to make a significant update to the $G(\cdot)$ function.

3. We recommend setting G_0 to some negative value. In the current problem setting, most instances are negative instances. Setting G_0 to negative values treats all instances as negative instances at the beginning of the future feature extraction process.

We also compared the zeroth-order FF-MEMM model with the other three SSL methods over synthetic data sets. A zeroth-order FF-MEMM model is actually a sliding window model with future features. The results are shown in Table 5.4. We have the following observations.

1. Zeroth-order FF-MEMM does best in all cases by extracting future features.

2. On the OR and EXTRA data sets, CRFs with higher order Markov connections are able to improve the prediction accuracy. That means there exist long range dependencies in these two problems. By extracting future features, FF-MEMM can exploit these long range dependencies without using complicated inference algorithms such as the forward-backward algorithm.

3. On the XOR data set, even using higher order CRFs does not improve the performance. This is an example where even increasing the order of CRFs can not solve

Table 5.4: Prediction accuracy (%) of SW, MEMM, CRF, and FF-MEMM on synthetic data sets.

	SW	MEMM-1	CRF-1	CRF-2	CRF-3	CRF-4	FF-MEMM-0
XOR	51.6	50.7	56.0	57.3	57.6	58.4	**100**
OR	67.5	66.5	76.6	85.9	91.1	93.9	**100**
EXTRA	84.4	84.9	87.1	90.2	92.5	92.9	**100**

the problem, but FF-MEMM can solve the problems very well.

5.4.2 NETtalk Data Set

To evaluate FF-MEMM on a real data set, we transformed the original NETtalk data set into a binary classification problem. This is necessary because our current approach can only handle binary labels. For letters whose stress labels are '2' or '1', we assign class label 1. For letters whose stress labels are '0', '$<$', or '$>$', we assign class label 0. We employed a window size of 1 and a bag size of 2 in our experiments. The FF-MEMM method is compared with the other three SSL methods, and the results are shown in Table **??**.

The prediction accuracies of SW, MEMM-1, and CRF-1 are similar. The big improvement from CRF-1 to CRF-2 indicates that there exist high order dependencies within the sequences. By extracting future features, FF-MEMM-1 can capture this high order dependency to some extent. As a result, it can improve the performance of MEMM-1 to 89.4%, which is much better than both MEMM-1 and CRF-1. However, it is still inferior to CRF-2. This is probably because there is some dependency among

Table 5.5: Prediction accuracy (%) of SW, MEMM, CRF, and FF-MEMM on NETtalk data set.

SW	84.2			
	k=1	k=2	k=3	k=4
MEMM	85.7	87.1	88.9	89.0
CRF	85.7	90.4	91.8	91.4
FF-MEMM (bag = 2)	89.4	90.4	90.6	90.3
FF-MEMM (bag = 4)	88.9	89.6	90.5	89.23

output labels that cannot be represented by a first-order model and that cannot be easily captured by a multiple instance problem defined over the input features. This hypothesis is confirmed by the second order FF-MEMM-2 model, which is able to improve the performance to reach the same prediction accuracy as CRF-2. Without extracting future features, the prediction accuracy of MEMM-2 is only 87.1%. On this data set, increasing the size of the bags beyond 2 does not help much.

Chapter 6 – Summary and Future Work

6.1 Summary

In this dissertation, we presented TREECRF, a novel method for training conditional random fields based on gradient tree boosting. TREECRF has the ability to construct very complex feature conjunctions from basic features and scales much better than methods based on iterative scaling and simple gradient descent. It appears to match the L-BFGS algorithm implemented in MALLET, which also gives dramatic speedups when there are many potential features. In our experiments, TREECRF is as accurate as MALLET on four data sets, more accurate on one data set, and less accurate on one data set. Its feature induction method is faster than that of MALLET for problems with a large number of features. But its forward-backward implementation is slower than that of MALLET for very long sequences.

TREECRF is easier to implement and tune than MALLET. It introduces only one tunable parameter (either the maximum number of leaves permitted in each regression tree or the regularization constant), whereas MALLET has many more parameters to adjust. It is easier for the TREECRF to find the optimal stopping point to avoid overfitting, since its performance improves smoothly, while that of MALLET fluctuates wildly. An important direction to pursue in future research is to develop an approach that can use tree boosting for feature induction while still attaining the very fast inference of Mallet. This might be accomplished, for example, by converting the learned trees into flat feature vectors.

TreeCRF also has the ability to handle missing data via the instance weighting and surrogate splitting methods, which are not available in Mallet and other CRF training algorithms. Our experiments suggest that when the feature values are missing at random, the instance weighting approach works very well. In the one domain where instance weighting did not work well, imputation was the best method. The indicator feature method was also very robust. The method of surrogate splitting was the most expensive method to run and the least accurate. Hence, we do not recommend using surrogate splits with conditional random fields. The good performance of the indicator features and imputation methods is encouraging, because these methods can be applied with all known methods for sequential supervised learning, not only with gradient tree boosting. Since there is no one best method for handling missing values, as with many other aspects of machine learning, preliminary experiments on subsets of the training data are required to select the most appropriate method.

The search-based structured learning framework provides a powerful and efficient way to solve sequential supervised learning problems, particularly with large label vocabularies. However, the unidirectional search strategy that search-based methods employ make them unable to exploit long range dependencies between the current position and future positions. These long range dependencies can be essential for resolving local ambiguity in the search process. As a first step toward tackle this problem, we showed how to integrate a multiple-instance learning based algorithm into MEMM models to learn future features. Experimental results on synthetic data sets show that this method works very well on problems where a higher order CRF model is necessary or where no fixed-order CRF model can do well. Experiments on real data set show that this method is able to discover and exploit long range dependencies.

6.2 Future Work

Within the research direction of this dissertation, there are several open problems that need to be further explored.

In the direction of TREECRF algorithm, one problem is how to make TREECRF algorithm practical for problems where the number of features is very large. Many problems in natural language processing (NLP) have a huge number of features. However, in the current implementation of the TREECRF algorithm, in order to generate regression trees, we must consider all possible features at each internal node to decide the best split. This makes the current TREECRF algorithm impractical for solving those NLP problems. One possible solution is to use the random forest algorithm proposed by Breiman (2001). Instead of generating a single regression tree based on all features, we could generate a forest of random trees, where at each internal node of a random tree, only a much smaller number of features is considered. These random trees can even be generated in parallel to save more running time.

In the direction of the Future Feature MEMM model, several open problems need to be further studied.

- A more comprehensive evaluation of the binary class Future Feature MEMM model needs to be conducted. Potential benchmark problems include (1) intrusion detection in computer networks, where CRFs have been applied (Gupta et al., 2007; Gupta et al.), (2) the FAQs data sets as adapted by Cohen and Carvalho (2005), where "trailer" and "answer" are considered as separate tasks for each FAQ data set, and (3) the CMU seminar announcements data set used by Sutton and McCallum (2007), where we can treat identifying location names and identifying speaker names as two separate binary label tasks.

- The second open problem is how to extend the binary class Future Feature MEMM model to multiclass problems. This will make this algorithm much more useful, since many SSL problems are multiclass problems. One possible way is to extract a different set of future features for each class label and then combine all these future features together for the next round of top-level training.

- The third open problem is how to integrate the FF-MEMM model into the existing search-based structured learning framework, so that at each search step, we can automatically extract a set of useful future features to help resolve local ambiguity. Current search-based frameworks are not able to identify such information during the search process. We believe that this integration will make the search-based structured learning frameworks much more effective.

Bibliography

Alan Agresti. *An Introduction to Categorical Data Analysis*. Wiley, New York, 1996.

Yasemin Altun, Ioannis Tsochantaridis, and Thomas Hofmann. Hidden Markov support vector machines. In Tom Fawcett and Nina Mishra, editors, *Proceedings of the 20th International Conference on Machine Learning (ICML 2003)*, pages 3–10. AAAI Press, 2003.

Stuart Andrews, Ioannis Tsochantaridis, and Thomas Hofmann. Support vector machines for multiple-instance learning. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *NIPS*, pages 561–568. MIT Press, 2002.

Adam J. Ashenfelter. Sequential supervised learning and conditional random fields. Master's thesis, Oregon State University, 2003.

Peter Auer and Ronald Ortner. A boosting approach to multiple instance learning. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proceedings*, volume 3201 of *Lecture Notes in Computer Science*, pages 63–74. Springer, 2004.

Ghulum Bakiri and Thomas G. Dietterich. Achieving high-accuracy text-to-speech with machine learning. In *Data mining in speech synthesis*. Chapman and Hall, 1997.

Julian Besag. Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society B*, 36(2):192–236, 1974.

Hendrik Blockeel, David Page, and Ashwin Srinivasan. Multi-instance tree learning. In Luc De Raedt and Stefan Wrobel, editors, *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, volume 119 of *ACM International Conference Proceeding Series*, pages 57–64. ACM, 2005.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth Publishing Company, 1984.

Yann Chevaleyre and Jean-Daniel Zucker. Solving multiple-instance and multiple-part learning problems with decision trees and rule sets. application to the mutagenesis problem. In Eleni Stroulia and Stan Matwin, editors, *Advances in Artificial Intelligence, 14th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, AI 2001, Ottawa, Canada, June 7-9, 2001, Proceedings*, volume 2056 of *Lecture Notes in Computer Science*, pages 204–214. Springer, 2001.

William W. Cohen and Vitor R. Carvalho. Stacked sequential learning. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 671–676. Professional Book Center, 2005. ISBN 0938075934. URL `http://www.ijcai.org/papers/0378.pdf`.

Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 111–118, Barcelona, Spain, July 2004.

Hal Daumé III and Daniel Marcu. Learning as search optimization: Approximate large margin methods for structured prediction. In *International Conference on Machine Learning (ICML)*, Bonn, Germany, 2005.

Thomas G. Dietterich. Machine learning for sequential data: A review. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, London, UK, 2002. Springer-Verlag.

Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2): 31–71, 1997.

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)*, pages 217–224, Banff, Canada, 2004. ACM Press.

Thomas G. Dietterich, Guohua Hao, and Adam Ashenfelter. Gradient tree boosting for training conditional random fields. *Journal of Machine Learning Research (JMLR)*, 9(Oct.):2113–2139, 2008.

Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *The Annals of Statistics*, 38(2):337–374, 2000.

Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, Nov. 1984.

Kapil Kumar Gupta, Baikunth Nath, and Kotagiri Ramamohanarao. Layered approach using conditional random fields for intrusion detection. *IEEE Transactions on Dependable and Secure Computing*. In Press.

Kapil Kumar Gupta, Baikunth Nath, and Kotagiri Ramamohanarao. Conditional random fields for intrusion detection. In *Proceedings of 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW)*, pages 203–208. IEEE Press, 2007.

John M. Hammersley and Peter Clifford. Markov fields on finite graphs and lattices. Technical report, Unpublished, 1971.

John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning (ICML 2001)*, pages 282–289. Morgan Kaufmann, 2001.

Lin Liao, Tanzeem Choudhury, Dieter Fox, and Henry A. Kautz. Training conditional random fields using virtual evidence boosting. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2530–2535, Hyderabad, India, January 6-12 2007.

Oded Maron and Tomás Lozano-Pérez. A framework for multiple-instance learning. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

Oded Maron and Aparna Lakshmi Ratan. Multiple-instance learning for natural scene classification. In Jude W. Shavlik, editor, *ICML*, pages 341–349. Morgan Kaufmann, 1998.

Andrew McCallum. Efficiently inducing features of conditional random fields. In Christopher Meek and Uffe Kjaerulff, editors, *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2003)*, pages 403–410. Morgan Kaufmann, 2003.

Andrew McCallum, Dayne Freitag, and Fernando C. N. Pereira. Maximum entropy Markov models for information extraction and segmentation. In *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pages 591–598. Morgan Kaufmann, 2000.

Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

Andrew Y. Ng and Michael Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San mateo, California, 1988.

Ning Qian and Terrence J. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884, 1988.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA, 1993.

Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

Jan Ramon and Luc De Raedt. Multi instance neural networks. In *ICML-2000, Workshop on Attribute-Value and Relational Learning*, 2000.

Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In Eric Brill and Kenneth Church, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 133–142, Somerset, New Jersey, 1996. Association for Computational Linguistics.

Soumya Ray and Mark Craven. Supervised versus multiple instance learning: an empirical comparison. In Luc De Raedt and Stefan Wrobel, editors, *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, volume 119, pages 697–704. ACM, 2005.

Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168, 1987.

Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In Marti Hearst and Mari Ostendorf, editors, *HLT-NAACL 2003: Main Proceedings*, pages 213–220, Edmonton, Alberta, Canada, May 27 – June 1 2003. Association for Computational Linguistics.

Charles Sutton and Andrew McCallum. An introduction to conditional random fields for relational learning. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. 2007.

Charles Sutton, Michael Sindelar, and Andrew McCallum. Reducing weight undertraining in structured discriminative learning. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 89–95, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 25–32. MIT Press, Cambridge, MA, 2004.

Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *Proceedings of the 21st International Conference on Machine Learning (ICML 2004)*, pages 823–830, Banff, Canada, 2004. ACM Press.

Paul Viola, John Platt, and Cha Zhang. Multiple instance boosting for object detection. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 1417–1424. MIT Press, Cambridge, MA, 2006.

S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark W. Schmidt, and Kevin P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In William W. Cohen and Andrew Moore, editors, *Proceedings of the 23rd International Conference on Machine learning (ICML 2006)*, pages 969–976, New York, NY, USA, 2006. ACM.

Jun Wang and Jean-Daniel Zucker. Solving the multiple-instance problem: A lazy learning approach. In Pat Langley, editor, *ICML*, pages 1119–1126. Morgan Kaufmann, 2000.

Xin Xu and Eibe Frank. Logistic regression and boosting for labeled bags of instances. In Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang, editors, *Advances in*

Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004, Proceedings, volume 3056 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 2004.

Yuehua Xu, Alan Fern, and Sung Wook Yoon. Discriminative learning of beam-search heuristics for planning. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007)*, pages 2041–2046, Hyderabad, India, January 6-12 2007.

Qi Zhang and Sally A. Goldman. EM-DD: An improved multiple-instance learning technique. In Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani, editors, *NIPS*, pages 1073–1080. MIT Press, 2001.

Qi Zhang, Sally A. Goldman, Wei Yu, and Jason E. Fritts. Content-based image retrieval using multiple-instance learning. In Claude Sammut and Achim G. Hoffmann, editors, *ICML*, pages 682–689. Morgan Kaufmann, 2002.

Zhi-Hua Zhou. Multi-instance learning: A survey. Technical report, AI Lab, Department of Computer Science & Technology, Nanjing University, Nanjing, China, March 2004.

Zhi-Hua Zhou and Min-Ling Zhang. Neural networks for multi-instance learning. Technical report, AI Lab, Computer Science & Technology Department, Nanjing University, China, August 2002.