CRF-OPT: An Efficient High-Quality Conditional Random Field Solver

Abstract

Conditional random field (CRF) is a popular graphical model for sequence labeling. The flexibility of CRF poses significant computational challenges for training. Using existing optimization packages often leads to long training time and unsatisfactory results. In this paper, we develop CRF-OPT, a general CRF training package, to improve the efficiency and quality for training CRFs.

We propose two improved versions of the forward-backward algorithm that exploit redundancy and reduce the time by several orders of magnitudes. Further, we propose an exponential transformation that enforces sufficient step sizes for quasi-Newton methods. The technique improves the convergence quality, leading to better training results. We evaluate CRF-OPT on a gene prediction task on pathogenic DNA sequences, and show that it is faster and achieves better prediction accuracy than both the HMM models and the original CRF model without exponential transformation.

Introduction

Conditional random field (CRF) (Lafferty, McCallum, & Pereira 2001) is a major model for sequential data labeling. It significantly relaxes the independence assumptions of the popular hidden Markov model (HMM). However, the added flexibility of CRF greatly increases the optimization difficulties. In most applications, an optimization package is called to learn the CRF weights. We find that for large problems this approach can be slow and the solution quality is not satisfactory.

In this paper, we propose to enhance generic optimization solvers by a number of techniques that are designed specifically for improving the speed and quality of CRF training. First, we observe that, in gradient-based optimization, most time is spent on the evaluations of objective and gradients, which are done by the forward-backward method. However, even with forward-backward, the optimization can be prohibitively expensive. For example, TAO (Benson *et al*. 2005), a state-of-the-art quasi-Newton solver, requires up to two days on a PC in a gene prediction task with only 1615 bases. The real problem may have more than 100K bases. We reveal that, there are much computational redundancy, largely overlooked by existing work, in the standard implementation of the forward-backward algorithm. We propose two techniques that can improve the speed by several orders of magnitude through exploiting the redundancy.

CRF training is a convex continuous optimization problem for which gradient-based search algorithms have solid theoretical convergence guarantees. However, we observe that, gradient-based search algorithms on CRF models usually terminate prematurely without reaching the actual optimal point. Moreover, different starting points usually lead to very different solutions other than the unique global optimum. Such a behavior is caused by an nearly flat search terrain along certain directions near the optimal point. We propose an exponential transformation technique that can force sufficient reduction in each step and achieve much better convergence quality.

Parameter Estimation for CRF

As shown in Figure 1(a), a CRF is a graphical model based on an undirected graph $G = (V, E)$, where $Y = (Y_v)_{v \in V}$ is the set of hidden variables, and X is the set of observation variables. The random variables Y_v obey the Markov property with respect to the graph: $p(Y_v|X, Y_w, w \neq v) = p(Y_v|X, Y_w, w \sim v)$, where $w \sim v$ indicates that w and v are neighbors in G (Lafferty, McCallum, & Pereira 2001). Each clique in G defines a *feature* for the CRF.

In this paper, we focus on linear-chain CRF, a class of CRF that is most widely used, in which each feature only involves two consecutive hidden states as shown in Figure 1(b). A linear-chain CRF defines the conditional distribution of a label sequence **y**, given the observation sequence **x**, as

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \exp\left\{\sum_{k=1}^{F} \lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\right\}, \quad (1)$$

where $\Lambda = \{\lambda_k\} \in \mathcal{R}^F$ is the weight vector, and $\{f_k(y_t, y_{t-1}, \mathbf{x}_t)\}_{k=1}^{F}$ is a set of feature functions, and Z(**x**) is a normalization function

$$Z(\mathbf{x}) = \sum_{\mathbf{y'}} \exp\left\{\sum_{k=1}^{F} \lambda_k f_k(y'_t, y'_{t-1}, \mathbf{x}_t)\right\}. \quad (2)$$

The predefined *feature* $f_k(\mathbf{y}_c, \mathbf{x}_c)$ typically returns a binary value. For example, a feature can be defined as:

$$f_k(\mathbf{y}_c, \mathbf{x}_c) = 1_{\{y_{t-1}=NC\}} 1_{\{y_t=C\}} 1_{\{\mathbf{x}_{t\cdots t+2}='ATG'\}}$$

It is 1 when y_{t-1} is NC, y_t is C, and the observations at time t to $t+2$ are 'ATG'.

Given training data $\mathcal{D} = \left\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\right\}_{i=1}^{N}$, to estimate the weights $\{\lambda_k\}$, we typically maximize a penal-

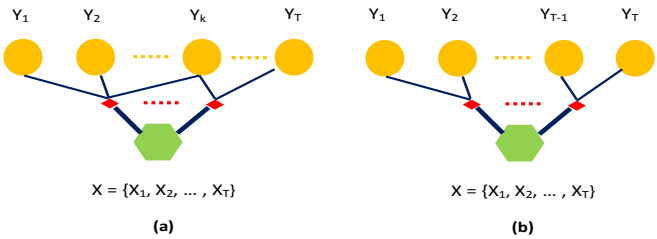


Figure 1: (a) A general CRF model. (b) A linear-chain CRF model.

ized conditional log likelihood:

$$\text{maximize}_\Lambda \quad \mathcal{O}(\Lambda) = \sum_{i=1}^{N} \log p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}) - \sum_{k=1}^{F} \frac{\lambda_k^2}{2\delta^2}. \quad (3)$$

The second term is a zero-mean, δ^2-variance gaussian prior to penalize large weight norm (McCallum 2003).

It is a convex optimization problem with only one global optimum. Generally, the objective function cannot be maximized in closed form and numerical optimization is used.

Efficient Function Evaluations

We have developed CRF-OPT, a package for general CRF training, on top of the Limited Memory Variable Metric (LMVM) solver in the Toolkit for Advanced Optimization (TAO) (Benson *et al.* 2005). LMVM is a quasi-Newton method that requires only function and gradient information, which are evaluated by the forward-backward algorithm.

The forward-backward algorithm

We briefly review the standard forward-backward algorithm which is a core routine for CRF training. Without loss of generality, we assume that there is only one ($N = 1$) training sequence.

We discuss the forward process as the backward process is symmetric. Let S be the set of all possible states, and *start* be an arbitrary fixed starting state. For each index $t \in 1, 2, \cdots, T$, and label $s \in S$, the forward variables $\alpha_t(s)$ can be defined recursively as:

$$\alpha_t(s) = \sum_{s' \in S} \alpha_{t-1}(s') \exp(\Phi_t(s', s, \mathbf{x}_t)), \quad (4)$$

where $\Phi_t(s', s, \mathbf{x}_t) = \sum_{k \in K_t(s',s)} \lambda_k$, α_0 is 1 for *start* and 0 for other states, and $K_t(s', s) = \{k \mid f_k(s', s, \mathbf{x}_t) = 1,\ k = 1 \cdots F\}$ is the set of active features for a given transition at time t.

In practice, to make the code numerically stable, the forward variables α_t are processed in logarithmic forms by taking logarithms for both sides of equation (4) :

$$\log(\alpha_t(s)) = \log(\alpha_{t-1}(s_0)) + \Phi_t(s_0, s, \mathbf{x}_t) + \log\left(1 + \sum_{s' \neq s_0} \exp\left(\log(\alpha_{t-1}(s')) + \Phi_t(s', s, \mathbf{x}_t) - \log(\alpha_{t-1}(s_0)) - \Phi_t(s_0, s, \mathbf{x}_t)\right)\right) \quad (5)$$

operation	exp	+	*	/	FE
time (sec)	8.093	0.203	0.203	0.203	0.448

Table 1: Timing results for 10^8 runs of some operations and a 7-th order feature evaluation (FE) on a PC with Xeon 2.4 GHz CPU and 2GB memory running Linux.

where $s_0 \in S$ can be any state. We set s_0 to be the state with the maximum $\log(\alpha_{t-1}(s))$ to further improve the numerical stability when calculating the exponential part.

We analyze the complexity of various operations used by forward-backward. Let T be the length of the training sequence, F the total number of features, M the number of possible states ($M = |S|$), and I be the number of iterations taken by the optimization algorithm to converge.

Most existing CRF packages, such as MALLET (McCallum 2002), MinorThird (Cohen 2004), Sarawagi's package (Sarawagi *et al.* 2004), and Murphy's package (Murphy & Schmidt 2006), iterate through all features to find the set of indices of active features $K_t(s', s)$ for each step t, and each transition (s', s). Therefore, a full forward computation requires $\Theta(TFM^2)$ feature evaluations. Each iteration of a gradient-based optimization algorithm requires an objective and gradient evaluation, resulting a total of $\Theta(TFM^2I)$ feature evaluations. Each time we compute $\log(\alpha_t(s))$ using (5), we need to perform $\Theta(M)$ exponentiations. Therefore, the optimization requires in total $\Theta(TM^2I)$ exponentiations.

We examine the complexity of each basic operation. The results are shown in Table 1. We see that in our platform, exp is 40 times more expensive than simple operations. This motivates us to reduce the number of exponentiations in the following two implementations of forward-backward.

Feature-vector fast forward (FVFF)

We propose a feature-vector fast forward (FVFF) method shown in Algorithm 1. It implements the forward process to compute $\alpha_t(s)$. In this method, $K_t(s, s')$ is the set of *active features*: $K_t(s, s') = \{k | f_k(s', s, \mathbf{x}_t) = 1, k = 1, \cdots, F\}$. $\Phi_t(s', s, \mathbf{x}_t)$ is the sum of weights of all *active features* at step t, with transition (s', s): $\Phi_t(s', s, \mathbf{x}_t) = \sum_{k \in K_t(s',s)} \lambda_k$.

Since the active feature sets $K_t(s, s')$ remains identical in each iteration of the optimizer, a preprocessing phase (Line 2) is used in Algorithm 1 to construct and store $K_t(s, s')$ for all the t, s' and s.

The forward computation is performed once in each iteration of the optimizer. The exponential of each feature weight E_k is precomputed and stored to avoid future exponential operations (Line 4). As in Line 6, $C(t, s', s)$ is computed using the production of E_k. The logarithm of each forward variable $\log(\alpha_t(s))$ is stored in $A(t, s), \forall t = 1, \ldots, T, s \in S$.

Proposition 1. The FVFF algorithm correctly computes the forward variables and we have

$$A(t, s) = \log(\alpha_t(s)), \forall t = 1, \cdots, T, s \in S.$$

	preprocessing time	preprocessing space	running space	feature evaluations	number of exponentiations
standard	$\Theta(1)$	$\Theta(1)$	$\Theta(TM)$	$\Theta(TFM^2I)$	$\Theta(TM^2I)$
FVFF	$\Theta(TFM^2)$	$\Theta(TVM^2)$	$\Theta(TVM^2)$	$\Theta(1)$	$\Theta(FI)$
OSFF	$\Theta(T+JFM^2)$	$\Theta(T+JVM^2)$	$\Theta(TM+JVM^2)$	$\Theta(1)$	$\Theta(JM^2I)$

Table 2: Comparison of the standard implementation of forward-backward and the two proposed schemes.

We omit the full proof but give some key equations here. The key for reducing exponential operations is to compute $C(t, s', s)$, which is the exponential of $\Phi_t(s', s, \mathbf{x}_t)$, using precomputed E_k:

$$C(t, s', s) = \exp\left(\Phi_t(s', s, \mathbf{x}_t)\right) = \prod_{k \in K_t(s',s)} E_k$$

$B(t, s)$ corresponds to the first part of (5):

$$B(t, s) = \log(\alpha_{t-1}(s_0)) + \Phi_t(s_0, s, \mathbf{x}_t)$$

$L(t, s)$ stores the term within the second log of (5):

$$L(t,s) = 1 + \sum_{s' \neq s_0} \exp\Big(\log(\alpha_{t-1}(s')) + \Phi_t(s', s, \mathbf{x}_t) - \log(\alpha_{t-1}(s_0)) - \Phi_t(s_0, s, \mathbf{x}_t)\Big)$$

Let V be the maximum number of active features for any given transition (s', s) and t. Typically V is very small comparing to F or T. The complexity of FVFF is shown in the second line of Table 2. Comparing to the cost of the standard implementation of forward-backward algorithm, we see several differences. First, in the preprocessing phase, FVFF takes $\Theta(TFM^2)$ time and $\Theta(TVM^2)$ space to figure out and store the $K_t(s', s)$ sets. This strategy saves the feature evaluations happened in each iteration of standard implementation. Second, in FVFF, the $\exp$ operation only takes place in the calculation of the E_k values in each iteration of the optimizer. Hence, there are $\Theta(FI)$ $\exp$ operations in contrast to $\Theta(TM^2I)$ for standard implementation. In most applications, we have $F \ll TM^2$.

Observation-substring fast forward (OSFF)

Through experimentation, we have found that a great number of $K_t(s', s)$ sets remain the same for different time steps t. Certain observation patterns repeatedly occur in the observation sequence, which motivate a new observation-substring fast forward (OSFF) algorithm.

For each feature $f_k(s', s, \mathbf{x}_t)$, we first find out the observation pattern p. For example, the observation pattern p_1 for feature $f_1(s', s, \mathbf{x}_{t-1} = a, \mathbf{x}_t = b, \mathbf{x}_{t+1} = c)$ is (ab, c), where the comma here indicates the position of t. Another feature $f_2(s', s, \mathbf{x}_t = a, \mathbf{x}_{t+1} = b, \mathbf{x}_{t+2} = c)$ has a different pattern (a, bc). The observation pattern for $f_3(s', s, \mathbf{x}_{t-2} = a, \mathbf{x}_{t-1} = a, \mathbf{x}_{t+1} = c)$ is $(aa*, c)$, where $*$ means this position is not defined. Let P be the set of all observation patterns.

Algorithm 1: feature_vector_fast_forward (FVFF)

A. preprocessing phase (executed once)
  compute $K_t(s', s)$,$\forall\, t = 1..T$, $s, s' \in S$;
B. forward computation (for each iteration)
  compute $E_k = \exp(\lambda_k)\ \forall k = 1, \cdots, F$;
  foreach state $s \in S$ **do**
    set $A(1, s) = \Phi_1(\textit{start}, s, \mathbf{x}_1)$, $L(1, s) = 1$, $C(1, \textit{start}, s) = \prod_{k \in K_1(\textit{start}, s)} E_k$;
  end_do;
  set $s'_0 = \textit{start}$;
  for $t = 2$ to T **do**
    find $s_0 \in S$ that maximizes $A(t-1, s)$;
    foreach state $s \in S$ **do**
      set $C(t, s_0, s) = \prod_{k \in K_t(s_0, s)} E_k$;
    end_do;
    foreach state $s \in S$ **do**
      /* using precomputed $K_t(s', s)$ */
      $B(t, s) = A(t-1, s_0) + \Phi_t(s_0, s, \mathbf{x}_t)$;
      $L(t, s) = 1$;
      foreach state $s' \in S, s' \neq s_0$ **do**
        set $C(t, s', s) = \prod_{k \in K_t(s', s)} E_k$;
        set $L(t, s) = L(t, s) + \frac{L(t-1, s')C(t-1, s'_0, s')C(t, s', s)}{L(t-1, s_0)C(t-1, s'_0, s_0)C(t, s_0, s)}$;
      end_do;
      set $A(t, s) = B(t, s) + \log(L(t, s))$;
    end_do;
    set $s'_0 = s_0$;
  end_do;

If a pattern p_2 must be satisfied when pattern p_1 is satisfied, we say that p_1 implies $p2$. For example, (a, bc) implies (a, b). We construct a *pattern implication forest* $\mathcal{F}_p$ as shown in Figure 4. Each node in $\mathcal{F}_p$ represents a pattern in P, and is distributed in different layers of $\mathcal{F}_p$ according to the number of defined positions in the pattern. There is an edge $p_1 \rightarrow p_2$ if p_1 implies p_2.

We then collect the active pattern set Ω, a subset of P, by mapping each step t of the training sequence into patterns. For each $t \in 1, 2, \cdots, T$, we search $\mathcal{F}_p$ in a top-down fashion to find those highest-level patterns matching the observations at t. For example, if the observations at t is (AT,G), then (AT,G) is selected but (AT,) is not. The patterns are added to Ω, and the mapping of step t is saved in $\pi(t)$, the set of indices of matched patterns in Ω. Let $J = |\Omega|$.

For each of the pattern ω_j, we define the *pattern implication set* $K^\omega_j, j = 1..J$ as:

$$K^\omega_j = \{m | p_m \in P, p_m \text{ is implied by } \omega_j\}$$

The set can be easily constructed by collecting all the descendants of pattern ω_j in $\mathcal{F}_p$.

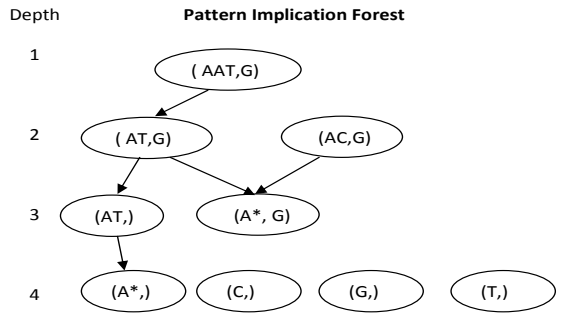


Figure 2: A pattern implication forest illustrating the implication relations between observation patterns.

Further, we define the *pattern-transition set* $K_j^\omega(s', s), \forall s', s \in S, j = 1..J$ as:

$$K_j^\omega(s', s) = \{k | f_k(s', s, p) \text{ is a feature }, p \in K_j^\omega)\}$$

Normally, there are lots of implication relationships between the observation patterns appeared in the features and we have $J \ll F$.

The OSFF algorithm is shown in Algorithm 2. In the preprocessing phase, it creates the Ω and $K_j^\omega(s', s)$ sets. The forward computation differs from FVFF in that, instead of computing $C(t, s', s)$ for each t, OSFF computes and stores $C^\omega(j, s', s)$, the exponential of the total weight for each $K_j^\omega(s', s)$ set.

Proposition 2. The OSFF correctly computes the forward variables and we have $A(t, s) = \log(\alpha_t(s))$, $\forall\ t = 1, \cdots, T, s \in S$.

The correctness of OSFF can be shown in a similar way as FVFF. The main difference is that, instead of precomputing $C(t, s', s)$ at the beginning of each iteration, OSFF precomputes $C^\omega(j, s', s)$ (Line 6) and come up with $C(t, s', s)$ from $C^\omega(j, s', s)$ using the mapping $\pi(t)$ (Line 14).

The last line of Table 2 shows the time complexity of OSFF. For the preprocessing time, OSFF is more efficient than FVFF when $J \ll T$. For the exponentiations, the OSFF is more efficient when $JM^2 \leq F$.

Discussions and validation

As listed in Table 2, both FVFF and OSFF are much faster than the original algorithm. Although the problem parameters vary by applications, there are some typical observations. First, T, the sequence length, is always orders of magnitude larger than F and M.

For example, in a gene prediction CRF, T can be as large as $300K$ to millions. Second, F tends to be smaller than T but greater than M. For example, in gene prediction, there are typically M=7 to 50 states. We have $F = M \times 4^{n+1} + M^2$, where n is the order of features whose typical value ranges from 1 to 7.

For a given problem with fixed T, F, M and other parameters, we can also estimate which of FVFF and OSFF has a lower complexity. As a general guideline, FVFF is better than OSFF when F is smaller than JM^2,

	$M = 7, F = 28721, J = 4096$	
	$T = 1615$	$T = 266225$
standard	$\approx 48hours$	-
FVFF	84 seconds	$\approx 4hours$
OSFF	282 seconds	$\approx 2.5hours$

Table 3: The training time of different algorithms on a PC with 2.4GHz CPU.

while OSFF is more favorable when JM^2 is smaller than F or when J is much smaller than T.

Algorithm 2: observation_substring_fast_forward

1 **A. preprocessing phase** (executed once)
2 create the Ω set and the mapping $\pi(t)$;
3 compute the set $K_j^\omega(s', s), \forall\ j = 1..J, s, s' \in S$;
4 **B. forward computation** (for each iteration)
5 **for** $j = 1$ to $J, \forall s', s \in S$ **do**
6 set $C^\omega(j, s', s) = \exp(\sum_{k \in K_j^\omega(s', s)} \lambda_k)$;
7 **end_do**;
8 **foreach** state $s \in S$ **do**
9 set $A(1, s) = \Phi_1(\textit{start}, s, \mathbf{x}_1)$, $L(1, s) = 1$, $C(1, \textit{start}, s) = \prod_{j \in \pi(1)} C^\omega(j, \textit{start}, s)$;
10 **end_do**;
11 set $s'_0 = \textit{start}$;
12 **for** $t = 2$ to T **do**
13 find $s_0 \in S$ that maximizes $A(t-1, s)$;
14 $C(t-1, s'_0, s_0) = C^\omega(\pi(t-1), s'_0, s_0)$
15 **foreach** state $s \in S$ **do**
16 $B(t, s) = A(t-1, s_0) + \Phi_t(s_0, s, \mathbf{x}_t)$;
17 set $C(t, s_0, s) = C^\omega(\pi(t), s_0, s)$;
18 set $L(t, s) = 1$;
19 **foreach** state $s' \in S, s' \neq s_0$ **do**
20 $C(t-1, s'_0, s') = \prod_{j \in \pi(t-1)} C^\omega(j, s'_0, s')$;
21 $C(t, s', s) = \prod_{j \in \pi(t)} C^\omega(j, s', s)$;
22 $L(t, s) = L(t, s) + \frac{L(t-1, s')C(t-1, s'_0, s')C(t, s', s)}{L(t-1, s_0)C(t-1, s'_0, s_0)C(t, s_0, s)}$;
23 **end_do**;
24 set $A(t, s) = B(t, s) + \log(L(t, s))$;
25 **end_do**;
26 set $s'_0 = s_0$;
27 **end_do**;

Table 3 validates the complexity saving. As shown in Table 3, for a problem with a large number of features, even for a short sequence with only 1615 symbols, the time for solving it using the standard forward-backward algorithm is prohibitive.

According to Table 2, the training time is propositional to T. Hence, it will take roughly one year for standard implementation to solve the one with length $T = 266225$. Parallel computers are often required for such tasks. In contrast, FVFF and OSFF can solve the problem much faster. A comparison of the solving time of FVFF and OSFF further proves that FVFF is faster when $F \ll TM^2$, and OSFF is most efficient when $J \ll T$.

Both FVFF and OSFF use $A(t, s) = B(t) +$

$\log(L(t,s))$ to compute the logarithmic form of $\alpha_t(s)$. By leaving $B(t) = A(t-1,s_0) + \Phi_t(s_0,s,\mathbf{x}_t)$ out of the log, the algorithms are as numerically stable as the standard implementation.

Exponential Transformation

We address another issue regarding the optimization quality. We have found that the quasi-Newton method often terminates prematurely when optimizing a CRF. We illustrate the observation by an example. The model we use is a toy casino gambling model, where there is a fair dice with even probability (1/6) for rolling each number, and a loaded dice with a higher probability (1/2) for rolling a six. There are two possible underlying states, fair or loaded, with different emission probabilities. Using a total of 16 features, the task is to infer which dice is used given a sequence of numbers.

In our experiment, we found that, different starting points lead to different, non-optimal solutions. To visualize the search, in Figure 3, we plot two variables, with each axis corresponding to one variable. For these two variables, the optimal values are located at $[0,0]$. We can see that, when the optimization terminates, the solutions from different starting points, marked by black circles in Figure 3a, are very different and far from the optimal point. We find that, the search often terminates when the difference of the objective between two consecutive iterations is smaller than a threshold 1.0e-04. Experimental results show that reducing the threshold will only cause negligibly small numerical differences.

To address this issue, we add a second round of optimization after LMVM terminates. In this round, we apply an exponential transformation to the original objective function formulation and optimize the transformed objective. Let Λ_0 be the solution of LMVM and let its function value be $\mathcal{O}_0 = \mathcal{O}(\Lambda_0)$, where the CRF objective function $\mathcal{O}(\Lambda)$ is defined in (3). In the transformation, we use the following new objective:

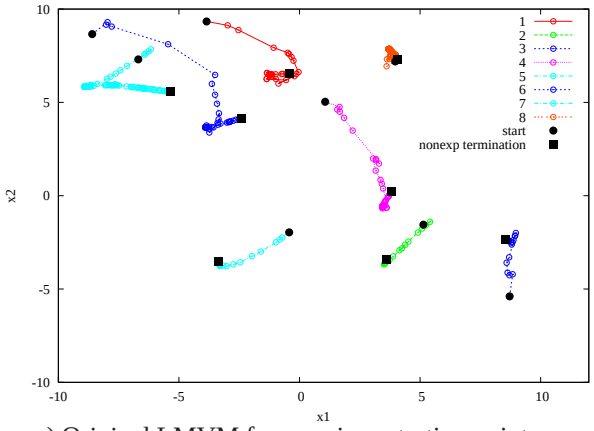
$$\chi(\Lambda) = \exp\left(\mathcal{O}(\Lambda) - \mathcal{O}_0\right) \tag{6}$$

It is obvious that the optimal Λ maximizing $\mathcal{O}(\Lambda)$ also maximizes $\chi(\Lambda)$, thus guaranteeing the correctness of our approach. The complexity for computing $\chi(\Lambda)$ is low since it only requires one more exponentiation after $\mathcal{O}(\Lambda)$ is obtained. The gradient of $\chi(\Lambda)$ is, $\forall k = 1, \cdots, F$,

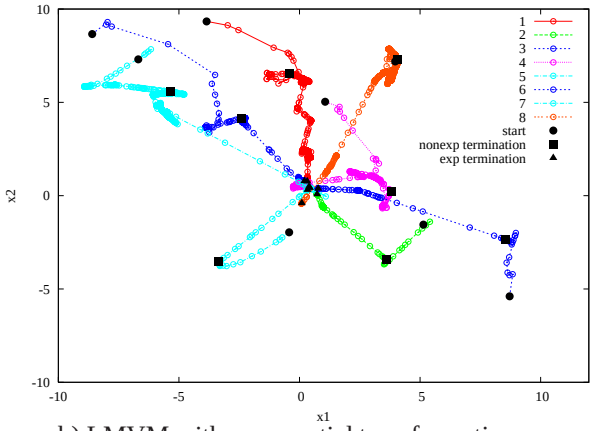
$$\frac{\partial \chi(\Lambda)}{\partial \lambda_k} = \exp\left(\mathcal{O}(\Lambda) - \mathcal{O}_0\right) \frac{\partial \mathcal{O}(\Lambda)}{\partial \lambda_k}. \tag{7}$$

Therefore, there is little additional overhead for computing the gradients.

We can see from (7) that, for any new points with a high objective value than that of Λ_0, we have $\frac{\partial \chi(\Lambda)}{\partial \lambda_k} > \frac{\partial \mathcal{O}(\Lambda)}{\partial \lambda_k}$. Thus, the exponential transform magnifies the improvements along the search trajectory and allows the optimizer to converge closely to an optimal point without early termination. This effect is illustrated in Figure 3b, where the termination points after using the exponential transformation are marked by triangles. We can clearly see that the exponential transformation can enforce the search to converge very closely to the same optimal solution, regardless of the starting point.



a) Original LMVM from various starting points.



b) LMVM with exponential transformation.

Figure 3: Search trajectories before and after using the exponential transformation.

Experimental Results

We have developed a software package, named CRF-OPT, that integrates the fast forward algorithms and exponential transformation into the TAO package.

We present results on a gene prediction task. Given a DNA observation sequence, consisting of 'A','T','G','C' bases, the aim of gene prediction is to find out the protein coding regions, known as genes, and their associated components, including coding exons, start/stop exons, promoters, and poly-adenylation sites (Burge. 1997).

Figure 4 shows a finite state machine representation of the structure of genomic sequences in our implementation. In this model, each circle represents a hidden state, such as exon (C), intron (I), and intergenic region (NC). In our model, the introns and exons are further divided into three phases according to the reading frame. Each edge represents a possible hidden state transition with required base observation.

The features used in our model is the state transition

Table 4: The performance of HMM and CRF on the gene prediction data sets.

Measure	Set 1			Set 2			Set 3			Set 4		
	HMM	CRF	eCRF	HMM	CRF	eCRF	HMM	CRF	eCRF	HMM	CRF	eCRF
Gene Sensitivity(%)	2.50	**7.50**	**7.50**	0	2.50	**5.00**	2.17	**4.35**	**4.35**	**4.08**	2.04	**4.08**
Gene Specificity(%)	1.89	6.82	**7.14**	0	2.13	**5.88**	2.00	4.35	**4.44**	**4.00**	1.79	**4.00**
Transcript Sensitivity(%)	2.50	**7.50**	**7.50**	0	2.50	**5.00**	2.17	**4.35**	**4.35**	**4.08**	2.04	**4.08**
Transcript Specificity(%)	1.89	6.82	**7.14**	0	2.13	**5.88**	2.00	4.35	**4.44**	**4.00**	1.79	**4.00**
Exon Sensitivity(%)	15.42	**37.00**	35.68	**35.44**	25.32	34.18	**36.63**	34.43	34.43	31.72	28.62	**32.41**
Exon Specificity(%)	16.43	**43.30**	41.33	32.56	30.00	**40.30**	34.36	39.66	**39.83**	32.39	33.88	**37.01**
Nucleotide Sensitivity(%)	81.70	**90.35**	89.96	78.26	84.21	**90.13**	86.06	90.62	**90.79**	82.66	92.84	**93.45**
Nucleotide Specificity(%)	**89.33**	88.54	87.89	**92.33**	89.64	90.91	**92.02**	89.54	90.45	**94.19**	90.67	92.38

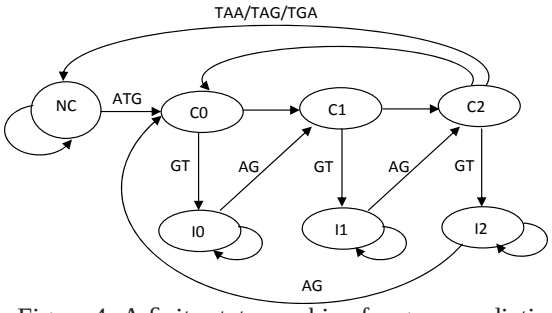


Figure 4: A finite state machine for gene prediction

and base observation requirement as represented by the edges in Figure 4 and a 5-th order bases emission information. In total we have around 7×4^6 features.

We test our algorithm on a sequence with 570K bases for the pathogenic fungus cryptococcus neoformans. We evaluate performance at the following four different levels: nucleotide-level, exon-level, transcript level, and gene level. For each level, we use the standard sensitivity (Sn) and specificity (Sp) measures defined as $Sn = \frac{TP}{TP+FN}$ and $Sp = \frac{TP}{TP+FP}$, where TP, FN, FP denote the numbers of true positive, false negative, and false positive labels, respectively (Burge. 1997).

We compare the performance of the CRF model before and after using the exponential transformation. Also, since our CRF model uses fifth order features, we compare the results to the corresponding fifth order Hidden Markov Model (HMM) implementation with the same state transition model and splice sites information.

We estimate the accuracy of our predictor using Holdout Validation and show the results in Table 4. There are 170 genes in the data set. We use all these genes to construct four different data sets. We randomly split the data into 75% training data and 25% testing data. For each test set, we show the performance of HMM and CRF. For each set, CRF denotes the original CRF without exponential transformation and eCRF denotes the CRF using exponential transformation. Both use the OSFF implementation of the forward-backward algorithm. We do not show the results of the standard implementation of forward-backward because it will take more than a month to finish on our workstation with Intel Xeon 2.4 GHz CPU and 2GB memory running Linux.

As shown in the result, CRF-OPT gives better performance than HMM using the same feature set. Moreover, our proposed exponential transformation further improves the prediction accuracy, especially for the higher-level gene, transcript, and exon level measures. The higher level accuracy is typically more difficult to achieve than the nucleotide level accuracy. The improvement shows that it is beneficial to achieve high-precision convergence to the global optimal solution using exponential transformation.

The gene prediction model used in our experiment is still a much simplified one. Yet, the results show the effectiveness of the proposed work. We are now working on integrating these techniques to CONTRAST, a state-of-the-art CRF-based gene predictor (Gross *et al.* 2007).

References

Benson, S. J.; McInnes, L. C.; Moré, J.; and Sarich, J. 2005. TAO user manual (revision 1.8). Technical Report ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory.

Burge., C. 1997. *Identification of genes in human genomic DNA*. Ph.D. Dissertation, Stanford Univerisity.

Cohen, W. W. 2004. Minorthird: Methods for identifying names and ontological relations in text using heuristics for inducing regularities from data. http://minorthird.sourceforge.net.

Gross, S. S.; Do, C. B.; Sirota, M.; and Batzoglou, S. 2007. Contrast: a disriminative, phylogeny-free approach to multiple informant de novo gene prediction. *Genome Biology* 8:R269.

Lafferty, J.; McCallum, A.; and Pereira, F. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *International Conference on Machine Learning*.

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

McCallum, A. 2003. Efficiently inducing features of conditional random fields. *Conference on Uncertainty in Artificial Intelligence*.

Murphy, K., and Schmidt, M. 2006. Crf toolbox for matlab. http://www.cs.ubc.ca/ murphyk/Software/CRF/crf.html.

Sarawagi, S.; Jaiswal, A.; Tawari, S.; Mansuri, I.; Mittal, K.; and Tiwari, C. 2004. Sunita sarawagi's crf package. http://crf.sourceforge.net.