Conditional Random Fields for Word Hyphenation

Tsung-Yi Lin and Chen-Yu Lee
Department of Electrical and Computer Engineering
University of California, San Diego
{tsl008, chl260}@ucsd.edu
February 12, 2013

Abstract

Word hyphenation is an important problem which has many practical applications. The problem is challenging because of the vast amount of English words. We use linear-chain Conditional Random Fields (CRFs) that has efficient algorithms to learn and to predict hyphen of English words that do not appear in the training dictionary. In this report, we are interested in finding 1) an efficient optimization technique to learn linear-chain CRFs model and 2) a good feature representation for word hyphenation. We compare the convergence time of three optimization techniques 1) Collins Perceptron; 2) Contrastive Divergence; 3) limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS). We design two feature representation 1) relative binary encoding (RBE) and 2) absolute binary encoding (ABE) and compare their performance. The experiment results show that Collins Perceptron is the most efficient method for training linear-chain CRFs and ABE is a better feature representation scheme that outperforms RBE by 7.9% accuracy. We show our design is reasonable by comparing it to the state-of-the-art [2] which outperforms this work only by 4.66% accuracy.

1 Introduction

The objective of this project is to learn a model to predict syllables of novel English words correctly. A linear-chain Conditional Random Fields is an efficient way to apply a log-linear model to this type of task. We model the states of two consecutive tags y_{i-1} and y_i at ith letter position has the posterior probability $p(y_{i-1}, y_i|\bar{x}; w)$ given observed a substring $\bar{x}$ with the model parameter w in a English word. Training a CRF means finding the parameter of the model that gives the best possible prediction for each training example. The gradient based optimization method is a common tool to approach the optimal parameter vector w^* with the iterative process. In this report, we implement three gradient-based methods 1) Collins Perceptron 2) Contrastive Divergence 3) limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) to solve the maximum likelihood problem of linear-chain CRFs, and we implement all needed CRF-specific algorithms: Viterbi algorithm, Gibbs sampling, and Forward-Backward algorithm for each training method respectively.

For each input word, there is always one output tag per letter. We tag each letter with either 1, for hyphen allowed following this letter, or 0, for hyphen not allowed after this letter. We design two feature encoding schemes 1) relative binary encoding (RBE) by considering relative position of substring $\bar{x}$ to the tag y_i and 2) absolute binary encoding (ABE) by considering absolute position of substring $\bar{x}$ and tag y_i. We test our implementation by using the dataset[1] available online which contains of 66,001 English words with syllables separated by hyphens. The experiment results show that Collins Perceptron is the most efficient method for training linear-chain CRFs and ABE is a better feature representation scheme that outperforms RBE by 7.9% accuracy. We show our design

[1] http://www.cs.ucsd.edu/users/elkan/hyphenation/

is reasonable by comparing it to the state-of-the-art [2] which outperforms this work only by 4.66% accuracy.

2 Feature Representation

The feature representation is the key for training CRFs efficiently. We use indicator feature function $F_j(\bar{x}, \bar{y}, p)$ to capture the relationship between tags y, substrings $\bar{x}$, and the position argument p within a length n word. We define $y_i = 1$ for the ith letter if hyphen is allowed for the following letter. We define $F_j(y_{i-1}, y_i, \bar{x}, p) = 1$ if y_{i-1} or y equal to 1; otherwise, $F_j(y_{i-1}, y_i, \bar{x}, p) = 0$. The substring $\bar{x}$ is any subtring with length from 2 to 5 that overlaps the position where $y_i = 1$. We define the tag equals to 0 and substring character equals to $-$ at the start and end position ($i = 0$ and $i = n + 1$). In the following section, two different designs are introduced to encode position argument p: 1) Relative Binary Encoding (RBE) and 2) Absolute Binary Encoding (ABE).

2.1 Relative Binary Encoding

RBE encodes position argument with the relative distance between the position of the first letter in the substring $\bar{x}$ and the position of the tag y_i. For example, **hy-phen-a-tion** has feature function $F_j(y_{i-1} = 0, y_i = 1, \bar{x} = hy, p = 1)$ and a feature $F_j(y_{i-1} = 0, y_i = 1, \bar{x} = yp, p = 0)$. The position argument p in the former example is 1 because $p = 2 - 1$ is the subtraction of the the tag position at y and first letter position of h of the substring. Two words can have the same feature function $F_j(y_{i-1}, y_i, \bar{x}, p)$ at the different tag positions. This scheme can capture the characteristics for suffix hyphenation, e.g., **-ing**, **-ment**, etc. RBE produces 249,815 different binary indicator functions involve a substring that appears at least once in the training dataset.

2.2 Absolute Binary Encoding

ABE encodes position argument with the absolute position of the first letter of the substring. For example, **hy-phen-a-tion** has feature function $F_j(y_{i-1} = 0, y_i = 1, \bar{x} = hy, p = 1)$ and $F_j(y_{i-1} = 0, y_i = 1, \bar{x} = yp, p = 1)$. ABE produce larger amount of indicator functions than RBE since it distinguish the same substring and tag at different position. in other word, same suffix, e.g., **-ing**, has different feature function given the different prefix. ABE has 335,569 different binary indicator functions involve a substring that appears at least once in the training dataset.

3 Algorithm Design And Analysis

In this section, we introduce the principle of linear-chain CRFs. Collins Perceptron, Constractive Diverfence, and L-BFGS are introduced to optimize the log likelihood of linear-chain CRFs. In this report, we focus on analyzing the algorithm complexity for our implementation. We use the same notation as in [1]. For detail equation derivations please refer to [1].

3.1 Linear-chain Conditional Random Field

A linear conditional random field is a way to apply a log-linear model to this type of work. We first define the terminologies for the model: let $\bar{x}$ be a sequence of words and let $\bar{y}$ be a corresponding sequence of tags. Here $\bar{x}$ is an example, $\bar{y}$ is a label, and a component y_i is a tag. The standard log-linear model is

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

In this project, we assume that each feature function F_j is a sum along the output label, for $i = 1$ to $i = n$ where n is the length of y:

$$F_j(\bar{x}, \bar{y}) = \sum_{i=1}^{n} f_j(y_{i-1}, y_i, \bar{x}, i) \tag{2}$$

We can then have a fixed set of feature functions F_j, even though the training examples are not of fixed length. The equation above indicates that each low-level feature function f_j can depend on the whole sentence $\bar{x}$, the current tag y_i and the previous tag y_{i-1}, and the current position i within the sentence. Each low-level feature function is well-defined for all tag values in positions 0 and $n+1$.

3.2 Inference of CRFs

The best possible prediction could be obtained by solving the argmax problem

$$\hat{y} = \arg\max_{\bar{y}} p(\bar{y}, \bar{x}; w) \tag{3}$$

We implement the Viterbi algorithm to solve the argmax problem efficiently. First, we can ignore the denominator because it is the same when $\bar{x}$ and w are fixed. We can also ignore the exponential inside the numerator because the exponential function is a monotonic increasing function. Now we want to compute

$$\hat{y} = \arg\max_{\bar{y}} p(\bar{y}, \bar{x}; w) = \arg\max_{\bar{y}} \sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}) \tag{4}$$

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

$$\begin{aligned} \hat{y} &= \arg\max_{\bar{y}} \sum_{j=1}^{J} w_j \sum_{i=1}^{n} f_j(y_{i-1}, y_i, \bar{x}, i) \\ &= \arg\max_{\bar{y}} \sum_{i=1}^{n} g_i(y_{i-1}, y_i) \end{aligned} \tag{5}$$

where we define

$$g_i(y_{i-1}, y_i) = \sum_{j=1}^{J} w_j f_j(y_{i-1}, y_i, \bar{x}, i) \tag{6}$$

for $i = 1$ to $i = n$. the $\bar{x}$ argument of f_j has been dropped in the definition of g_i because we are considering only a single fixed input $\bar{x}$. For each i, g_i is a different function. The arguments of each g_i are just two tag values, because everything else is fixed.

Let v range over the set of tags. Define $U(k, v)$ to be the score of the best sequence of tags from position 1 to position k, where tag number k is required to equal v. The score here means the sum of g_i functions taken from $i = 1$ to $i = k$. This is maximization over $k - 1$ tags because tag numbeer k is fixed to have value v. After the U matrix has been filled in for all k and v, the final entry in the optimal output sequence $\hat{y}$ can be computed as $\hat{y_n} = \arg\max_v U(n, v)$. Each previous entry can then be computed as

$$\hat{y_{k-1}} = \arg\max_{u} [U(k-1, u) + g_k(u, \hat{y}_k)] \tag{7}$$

The time complexity of the Viterbi algorithm is $O(m^2nJ + m^2n)$ time since we need $O(m^2J)$ time to compute all g_i functions and $O(m^2)$ for all U scores at each position i, and we have n positions in total. Here m is the number of tags, n is the length of $\bar{y}$ and J is the number of feature functions. In our experiment, we only use feature function f_j that has been fired at least once, so the factor J is much smaller than then the actual number J.

3.3 Training of CRFs

The training task for a log-linear model is to choose values for the weights (parameters). Because of the non-linear characteristic of the exponential function in the model, we could only use numerical approaches to find those weights. As shown in [1], we have to find the weights that maximize the log likelihood function of the training data. At the global maximum the entire gradient is zero, so we have

$$\sum_{<x,y>\in T} F_j(x,y) = \sum_{<x,y>\in T} E_{y\tilde{}p(y|x;w)}[F_j(x,y)] \tag{8}$$

In order to compute the gradient efficiently, we could get the gradient value using different computational schemes are as follows.

3.3.1 Collins Perceptron

We could approximate the probability mass function as a indicator function that has value 1 on the most likely y value. This means that we use the approximation

$$\hat{p}(y|x;w) = I(y = \hat{y}), \text{where } \hat{y} = \arg\max_y p(y|x;w) \tag{9}$$

Then the gradient update rule simplifieds to the following rule:

$$w_j := w_j + \lambda F_j(x,y) - \lambda F_j(x,\hat{y}) \tag{10}$$

where $\hat{y}$ could be found using the Viterbi algorithm as shown above. One update by the Collins perceptron method cause a net increase in w_j for features F_j whose value is higher for y than for $\hat{y}$. It thus modifies the weights to directly increase the probability of y compared to the probability of $\hat{y}$. If $\hat{y} = y$, then there is no change in the weight vector, and this is reason why the computational time for one epoch is decreasing as the weights are getting better. The time complexity of Collins perceptron is $O(m^2nJ + nJ) = O(m^2nJ)$ because for each update iteration we need to spend $O(m^2nJ)$ for Viterbi algorithm, and we need to spend $O(nJ)$ to compute F_j for n positions to update the J weights. Note that the number of updating iteration decreases as the model becomes better because only few $\hat{y} \neq y$.

3.3.2 Contrastive Divergence

The idea of contrastive divergence is to obtain a single value y^* that is somehow similar to the training label y, but also has high probability according to $p(y|x;w)$. We implement the Gibbs sampling to obtain the "evil twin" y^*. Gibbs sampling relies on drawing samples efficiently from marginal distributions as shown in [1]. We can get a stream of samples by the following process:

(1) Select an arbitrary initial guess $y =< y_1, y_2, ..., y_n >$.
(2) Draw y_1' according to $p(y_1|x, y_2, ..., y_n)$;
draw y_2' according to $p(y_2|x, y_1', y_3, ..., y_n)$;
draw y_3' according to $p(y_3|x, y_1', y_2', y_4..., y_n)$;
and so on until y_n'.
(3) Replace $y_1, y_2, ..., y_n$ by $y_1', y_2', ..., y_n'$ and repeat from (2)

In our implementation, we randomly select a training label y as the initial guess instead of arbitrary initial guess, and then we only execute the process one round for efficiency. The time complexity of Gibbs sampling is $O(m)$ for a single tag y_i once all g_i matrices have been computed and stored.

3.3.3 L-BFGS

L-BFGS is a quasi-Newton optimization method which approximates Hessian matrix by the gradient. The gradient g of linear-chain CRFs is:

$$g = \sum_{<x,y>\in T} F_j(x,y) - \sum_{<x,y>\in T} E_{y\text{-}p(y|x;w)}[F_j(x,y)] \tag{11}$$

The expectation of feature function is the computation bottleneck because it involves computing partition function and $p(y_{i-1}, y_i|\bar{x}; w)$. Forward and backward algorithm is an efficient method to compute linear-chain graph. Forward algorithm computes $\alpha(k,u)$ that starts from $k = 0$ and ends at $k = n$:

$$\alpha(k+1, v) = \sum_u \alpha(k,u)[exp(g_{k+1}(u,v))] \tag{12}$$

$\alpha(k,u)$ means the unnormalized probability that has state u at position k given observed fist $k-1$ nodes. Note that $\alpha(0,u)$ Note that we initialize α such that $\alpha(0,y) = I(y = START)$.

Backward algorithm computes $\beta(u,k)$ that starts from $k = n+1$ and ends at $k = 1$:

$$\beta(u,k) = \sum_v [exp(g_{k+1}(u,v))]\beta(v,k+1) \tag{13}$$

$\alpha(k,u)$ means the unnormalized probability that has state u at position k given observed last $n-k$ nodes. Note that $\alpha(0,u)$ Note that we initialize β such that $\beta(u,n+1) = I(y = STOP)$. The partition function now can be computed as $Z(\bar{x}, w) = \sum_v \alpha(n,v)$. The expectation of feature function j is

$$E_{y\text{-}p(y|x;w)}[F_j(x,y)] = \sum_{i=1}^{n} \sum_{y_{i-1}} \sum_{y_i} f_j(y_{i-1}, y_i, \bar{x}, i) \frac{\alpha(i-1, y_{i-1})[exp(g_i(y_{i-1}, y_i)]\beta(y_i, i))}{Z(\bar{x}, w)} \tag{14}$$

We need $O(m^2 J)$ to compute $g_i(u,v)$. Both forward and backward algorithms take the advantage of linear-chain property and only require $O(mn)$ to compute. Partition function only needs $O(m)$ after $\alpha(n,v)$ is available. The bottleneck is to evaluate expectation of feature function which requires $O(Jnm^2)$ to compute. To sum up, compute true gradient is much more expensive than Collins Perceptron and Constractive Divergence.

4 Experimental Results

In this section, we discuss the setup and result of two experiments. We compare the performance of three different optimization techniques introduced in Section 3 and two feature representation schemes introduced in section 2.

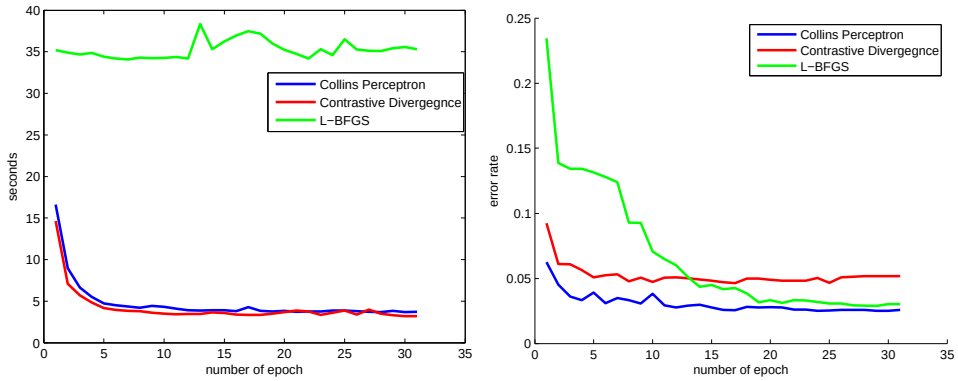
4.1 Experimental Design and Setup

?We first design experiment to find the most efficient algorithms to train CRFs described in Section 3. First 5000 words in the training dictionary are selected as the training and validating data to evaluate the optimization performance. We use 90% of total words for the training data and the rest 10% for validation. We measure the time spent and the error rate of different methods for each epoch to compare their performance. We measure performance through a relative long duration (30 epoch) to ensure the training process converges.

We measure the hyphenation accuracy with the whole dataset that contains 66,001 words. We use 90% as training data and 10% as validation set which is the same experiment setup as [2]. Collins perceptron, which is the best algorithm among the methods we try in the previous experiment, is used for training. We measure the miss rate at the end of every epoch and run optimization through 40 epochs for ABE and RBE feature representation schemes introduced in section 2.

4.2 Convergence Analysis of Training CRFs

Figure 1 shows the performance of three different optimization techniques. Figure 1a shows Constractive Divergence is slightly faster than Collins perceptron and both of them greatly outperforms



(a) Time spent for each epoch.

(b) Error rate.

Figure 1: The performance of different optimization techniques is compared by 1a the time to run for each epoch and 1b error rate of word hyphenation.

L-BFGS. We observe L-BFGS spends 3 more times than other algorithms for an epoch which indicates L-BFGS is much slower than Collins perceptron and Constractive Divergence. The result makes sense because the latter methods approximates the true gradient computation which is the computation bottleneck of L-BFGS. We also observe the time spent of Collins perceptron and Constractive Divergence decreases with training epoch from 15 seconds to 4 seconds an epoch. This observation suggests that the skip of parameter update when $\hat{y} = y$ saves the significant computation time.

Figure 1a shows the error rate of different algorithms. We can find Collins perceptron and L-BFGS converge to the same error rate but Constractive Divergence converges to the error rate about 3% higher. This can be explained by the suboptimal $\hat{y}$ found by Gibbs sampling to approximate true gradient distribution. Since we only run one iteration for Gibbs sample, it is not surprising that Gibbs sample stuck at the suboptimal point. The problem may be able to solve by adding more iterations for Gibbs sampling but it is not computational economics to do that since . We can conclude from Figure 1a that Collins perceptron has the advantage of fast convergence and stable convergence to the global optimal point. As the result, we will apply Collins perceptron to evaluate hyphenation performance on the whole dataset.

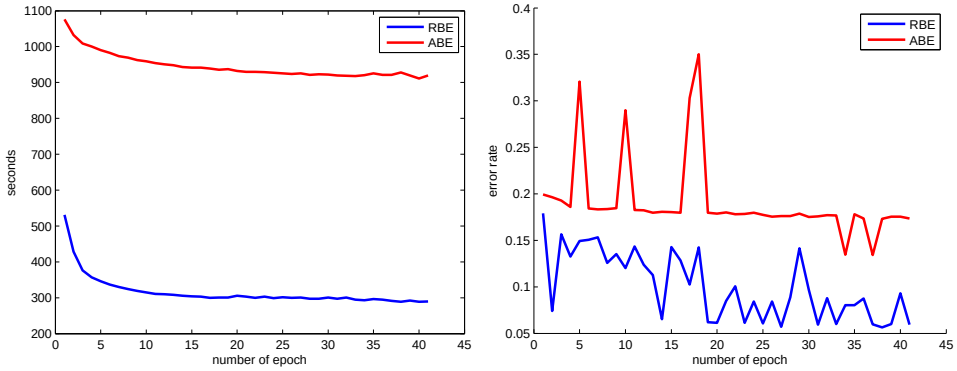
4.3 Accuracy Evaluation

Figure 2 shows the performance of different feature representation schemes. We use 90% data for training and 10% data for testing on whole dataset. Figure 2a shows RBE representation is faster than ABE representation at training step. This is because RBE only produces 249,815 different binary indicator functions while ABE has 335,569 different binary indicator functions and the complexity of Collins perceptron is linear to the dimension of feature. Figure 2b shows RBE has lower error rate than ABE by 7.9%. The reason might be that the relative position of the input string to the tag can capture the suffix of a word and thus generalize better than encoding scheme of ABE.

In Table 1, we report the performance of different methods and our implementation. Here we use RBE as feature representation scheme and Collins perceptron as our training approach. We compare our result with commercial products and the algorithm listed in [2] by the same evaluation method. Our implementation could achieve 3rd lowest error rate compared to all methods. The algorithm in [2] has lowest error rate and it may be because they use 2,916,942 different indicator functions which is 11 times as ours.

5 Discussion

In this report, we solve text hyphenation prediction with linear-chain CRFs. Three different optimization techniques are implemented and compared. We conclude Collins perceptron is the most efficient and stable algorithm to optimize CRFs in this problem. We apply Collins perceptron to



(a) Time spent for each epoch.

(b) Error rate.

Figure 2: The performance of different feature representation schemes is compared by 2a the time to run for each epoch and 2b error rate of word hyphenation.

Method	TP	FP	TN	FN	% error rate
Place no hyphen	0	0	439062	111228	20.21
TEX (hyphen.tex)	75093	1343	437719	36135	6.81
TEX (ukhyphen.tex)	70307	13872	425190	40921	9.96
TALO	104266	3970	435092	6962	1.99
PATGEN	74397	3934	435128	36831	7.41
[2]	108859	2253	436809	2369	0.84
Our implementation	108790	28170	411730	2100	5.50

Table 1: Performance on the English dataset.

solve the 66,001 English words dataset with two feature representation schemes. We find RBE feature representation scheme has the advantage of less binary indicators and lower error rate than ABE. We obtain 5.5% error rate for RBE. Compared the error rate to existing methods in [2], the result is competitive to the commercial products listed in Table 1. However, the well-tuned state-of-the-art [2] still outperforms by 4.66% the implementation in this report.

References

[1] C. Elkan. Log-linear models and conditional random fields. In *UCSD CSE250B Lecture Note*, 2013.

[2] N. Trogkanis and C. Elkan. Conditional random fields for word hyphenation. In *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden*, pages 366–374. The Association for Computer Linguistics, 2010.