A Practitioner Guide to Conditional Random Fields for Sequential Labelling

Tran The Truyen, Dinh Phung
Department of Computing
Curtin University of Technology
thetruyen.tran@postgrad.curtin.edu.au, d.phung@curtin.edu.au

April 18, 2008

Contents

1 Introduction

There has been much interest in Conditional Random Fields (CRFs) [4], a recent advance in statistical machine learning. Its applications are wide, ranging from information extraction to computer vision. For those familiar with neural networks, here is the link: if neural networks can recognise a single digit, the CRFs can recognise a sequence of digits at the same time. However, this power does come with a price: its formulation requires some understanding of probability theory and dynamic programming, which may not readily available for most programmers.

This introduction is meant to give a practical tutorial for those who want to implement CRFs without deep understanding of the theories. For general introduction, please see [10].

2 The Sequential Labelling Problem

This note focuses on the simplest labelling problem that involves sequential data. This type of data is very popular in the real world. For example, in a natural language processing task known as noun-phrase (NP) chunking, we are given a sentence, or a sequence of words, we want to provide correct phrasal labels for each word. In our case, the phrasal labels are `B-NP` for begin-noun-phrase, `I-NP` for inside-noun-phase and `O` for others. For instant, a labelled sentence in the CoNLL2000 dataset [8] reads:

Chancellor/O of/O the/B-NP Exchequer/I-NP Nigel/B-NP Lawson/I-NP 's/B-NP restated/I-NP commitment/I-NP to/O a/B-NP firm/I-NP monetary/I-NP policy/I-NP has/O helped/O to/O prevent/O a/B-NP freefall/I-NP in/O sterling/B-NP over/O the/B-NP past/I-NP week/I-NP ./O

This problem may appear straightforward on the surface because we can always manually specify textual patterns that correspond to certain labels. However, this quickly becomes a very time-consuming approach because the texts are often highly ambiguous. For example, the word restated may refer to a verb in the past tense, but it is inside a noun-phrase in this case. We may have to create millions of patterns. The second problem is that even if the patterns are available, they may conflict with each other. In some cases, we have to rely on the fact that some patterns are more likely to occur than others. Such 'likelihood' suggests a better approach: we should rely on the statistical properties of the labels given the text. This is the foundation of the CRFs.

2.1 The Statistical Pattern Recognition Approach

In the statistical pattern recognition approach, there are two phases:

- In the *training phase*, we are given a dataset with labels. Then we collect the statistical properties of the raw data and the association patterns between these statistics with the known labels. Typically, in a long data sequence, we extract only the local patterns around each position. These patterns are then weighed in a process known as learning (or parameter estimation). The resulting local patterns and their parameters are collectively called a model.
- In the *testing phase*, we are given the raw data and ask the trained model to output the labels.

It may appear that this approach takes too much time to manually prepare the training data. However, giving the labels typically does not require programming knowledge, and anyone familiar with the domain can do. For example, in information extraction, most adults can easily give correct labels after some initial guidance. Second, this training data is independent of any specific modelling methods or training algorithms.

2.2 Some Notations

Denote by z the data and $x = (x_1, x_2, ..., x_T)$ the sequence of labels of length T. The subscript $t \in [1, T]$ will be referred to as time, even though our data may have nothing to do with time.

2.3 CRFs for Sequential Labelling (CRF-SL)

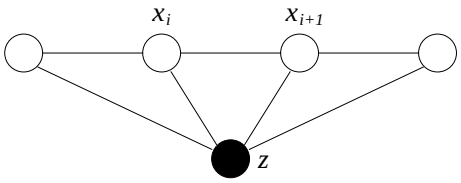

Figure 1: A CRF for sequential labelling. Empty circles denote state variables x and filled circle denotes data z.

A CRF-SL is a formulation that supports joint modelling of the data and its sequential labels. Formally, x given z is represented by a *undirected* Markov chain[1]. The conditional probability of x given z is defined as

$$P(x|z) = \frac{1}{Z(z)} \prod_c \psi_c(x_c, z)$$

where $\psi_c(x_c, z)$ are positive functions known as potentials, and $Z(z) = \sum_x \prod_c \psi_c(x_c, z)$ is the normalisation constant known as the partition-function.

In a typical implementation, there are two types of potentials:

- The *node potential* $\phi_t(x_t, z)$ takes care of the label x_t at time t. This usually accounts for the contribution of the local information around time t in the global model. It can be implemented as a 2D array of size $S \times T$.
- The *edge potential* $\psi_t(x_t, x_{t+1}, z)$ realises the relationship between the two nearby labels x_t and x_{t+1}. We may not need the data z, and sometimes, this does not even depend on time t. If the data is involved, then we can implement it as a 3D array of size $S \times S \times T$. Otherwise, a 2D array of size $S \times S$ is needed.

The potentials refer to some quantities that tie the labels and the local data patterns at particular time together. The data patterns are commonly known as *features* in the CRF literature. In the simplest case which is often used in practice, potentials are exponential of sum of weighted features. The node potential is given as

$$\phi_t(x_t, z) = \exp(\sum_k w_k f_k(x_t, z, t)) \tag{1}$$

The edge potential is formally computed as

$$\psi_t(x_t, x_{t+1}, z) = \exp(\sum_k w_k f_k(x_t, x_{t+1}, z, t)) \tag{2}$$

In a typical implementation of CRFs, once the labelled data is available for training, we often have to perform the following tasks:

1. Extracting features from training data.
2. Formulating the data likelihood and maximising it to obtain the parameters.
3. Using the learnt parameters, performing feature extracting in the test data, and decoding the best label sequence for each test data instance.

[1] A undirected chain is Markovian if $P(x_t|x_1, x_2, ..., x_{t-1}, x_{t+1}, ..., x_T) = P(x_t|x_{t-1}, x_{t+1})$. In words, a variable, when given its neighbours, does not depend on the rest.

3 Feature Extraction

3.1 Feature APIs

Features are perhaps the most important element of the model. Good *feature engineering* can often increase the labelling accuracy very significantly. Feature extraction may follow certain basic rules, which are application specific. However, the common practice is to deal with a segment of raw data falling within a window of length W. For information extraction, W may range from 3 to 11. So typically, at a particular time t, we have a vector of features. The size of the vector greatly varies with applications. In chunking tasks, the vector size may be up to several millions [9] but it may be few hundreds in speech recognition.

In typically implementation, there is a function of input called *data pattern* $g_m(z, t)$ that takes care of the raw data and returns a (real or binary) value. For example, in noun-phrase chunking, we may have a feature

$$g_{100}(z, 10) = \begin{cases} 1 & z_9 = today. \\ 0 & \text{otherwise} \end{cases}$$

that is, the 100th data feature receives value of 1 at current time $t = 10$ if the previous word is *today*.

Since the number of patterns may be potentially very large, representing the whole pattern vector at each time is not desirable. We should, however, implement a spare vector in that only non-empty patterns and their indices are stored. So we assume that there exists an API equivalent to

```
getNextDataPattern(int dataID, int t)
```

This function returns both the data pattern value ant its index, where `dataID` is the index of the data instance and `t` is the time index.

Once data patterns are extracted, we need to associate them with labels to make up the model features. Usually, one of the following associations is implemented:

- *Node association* that is realised in the node feature $f_k(x_t, z, t)$ which takes into account the current label x_t and an element of $g_m(z, t)$.
- *Edge association* that is realised in the edge features $f_k(x_t, x_{t+1}, z, t)$ which takes into account the two nearby labels x_t and x_{t+1} and an element of $g_m(z, t)$.

3.2 Node Feature

Depending on the role of the node, we may associate it with data patterns in that the node feature is given as

$$f_k(x_t, z, t) = \delta[x_t, s] g_m(z, t) \tag{3}$$

where $\delta[x_t, s]$ returns 1 if $x_t = s$ and 0 otherwise. Continued from the previous example, we may have

$$f_{1000}(x_{10}, z, 10) = \begin{cases} g_{100}(z, 10) & x_{10} = \text{B-NP} \\ 0 & \text{otherwise} \end{cases}$$

that is, the 1000th node feature received a value returned by $g_{100}(z, 10)$ if the current label is `B-NP`.

Here s is any label in the label set. So the number of node features will be $S \times M$, where M is the number of data patterns. We do not usually need to store the f_k. Instead, whenever we need to use f_k, we repeatedly call `getNextDataPattern(int dataID, int t)` until a `NULL` is returned.

If we need only to account for the label bias, and leave the data association to the edge, then we need only the following node feature

$$f_k(x_t, z, t) = \delta[x_t, s]$$

3.3 Edge Feature

The simplest implementation of edge feature is not to implement anything! This is because we need only indicator function:

$$f_k(x_t, x_{t+1}, z, t) = \delta[x_t, s]\delta[x_{t+1}, s'] \tag{4}$$

This does not depend on time t nor data z. So simply, the edge feature set is just an identity matrix of size $S \times S$. For example, we choose

$$f_9(x_{10}, x_{11}z, 10) = \begin{cases} 1 & x_{10} = \text{B-NP} \,\&\, x_{11} = \text{I-NP} \\ 0 & otherwise \end{cases}$$

This models *co-occurrence* patterns of label pairs. This is important to capture the strong relations between label pairs, like those in the noun-phrase chunking example: `I-NP` must follow `B-NP`.

Another option is to choose

$$f_k(x_t, x_{t+1}, z, t) = \delta[x_t, x_{t+1}] \tag{5}$$

which models the *continuity* of labels. This may be suitable for problems where the same label persists for a long time, like those in activity recognition. This special feature also enables the fast computation: the complexity is $\mathcal{O}(ST)$ rather than $\mathcal{O}(S^2T)$.

However, if we believe that data association to the node may not enough to capture the relationship between data and labels then we may want to tie up the transition from x_t to x_{t+1} and the data. The typically implementation is

$$f_k(x_t, x_{t+1}, z, t) = \delta[x_t, s]\delta[x_{t+1}, s']g_m(z, t) \tag{6}$$

For example, the feature may be

$$f_{10000}(x_{10}, x_{11}, z, 10) = \begin{cases} g_{100}(z, 10) & x_{10} = \text{B-NP} \,\&\, x_{11} = \text{I-NP} \\ 0 & otherwise \end{cases}$$

As with node features, we do not need store $f_k(x_t, x_{t+1}, z, t)$ but rather repeatedly call `getNextDataPattern(int dataID, int t)` until a `NULL` is returned.

3.4 Feature Selection

Since our method relies on statistical properties of the data, for the model to work effectively, features need to occur in the training data frequently. Rare features may degrade the performance of the model. The easiest way is to keep only those features with occurrences larger than a threshold. In many cases, if often results in a compact feature set and improves the performance. A typical threshold is 4.

4 The Training Phase

We a given a set of D data instances of the form $(\tilde{x}^d, z^d)$ for $d \in [1, D]$, where z^d is the raw input and $\tilde{x}^d$ is the label sequence. Except for the numerical optimisation described in the next subsection, we deal with each data instance separately so we drop the superscript d for clarity.

4.1 Numerical Optimisation

Numerical optimisation refers to the process of finding the optimal point of a given function. In our cases, the function is the *log-likelihood*, a quantity that characterises how much the current parameters are supported by the training data. Maximising the log-likelihood means finding the

optimal parameters so that the support by the training data is strongest. Since there is no direct algebraic solution for this maximisation problem, we rely on iterative method to improve the log-likelihood step-by-step. Typically, numerical optimisation methods require two elements: the evaluation of the log-likelihood and its gradient at a particular parameter. We will show how to compute these two quantities in the next subsections.

Limited memory quasi-Newton method

Theoretically any optimisation methods that are capable of finding the locally optimal solution for a function will do. In practice, however, most CRFs implementation use a method known as L-BFGS, after the work of [9]. We will not go into the details of the methods and assume that there exists an efficient implementation for you to use. L-BFGS is an iterative method in that each step the objective function is improved. So we need to know when to stop the improvement loop. Typically, we set some thresholds on the number of steps (e.g. ≥ 500) or the relative improvement of the objective function (e.g. $\leq 10^{-5}$).

Stochastic gradient method

For those who do not want to deal with other people's implementation of L-BFGS, there is a simple but effective alternative [12]. Unlike L-BFGS which requires to loop through all the data in one integration before updating the parameters, we can update the parameters after seeing only one data instance

$$w_k \leftarrow w_k + \omega G_k^z$$

where G_k^z is the kth element of the gradient of the log-likelihood (see Equations 8 and 9) with respect to the data instance z and $\omega > 0$ is the learning rate. This is an example of *online learning*, where we update parameters as new data arrives. Typically, we choose $\omega \in [0.001, 0.1]$. This method is fast and easy to implement but its final performance may be slightly less than the L-BFGS. The number of iterations through the training data is quite small, about $5 - 20$ in practice. However, this method is potentially unstable numerically. So use it with care or use it for pilot study.

Voted perceptron

Voted perceptron (VP) was originated in [7, 3] for general classifiers and adapted in [2] for classifiers with structured output patterns. The perceptron family aims to find classifiers that can achieve zeros errors in training data, and thus hope to achieve the similar performance in unseen data.

Perceptron is another example of online learning. First, it tries to predict the label sequence x^* (using techniques described in Section 5) for the instance z and then checks if $x^* \neq \tilde{x}$. The parameters are updated as follows

$$\begin{aligned} w_k &\leftarrow w_k + \omega \sum_{t \in [1,T]} \Big(f_k(\tilde{x}_t, z, t) - f_k(x_t^*, z, t) \Big) \\ w_{k'} &\leftarrow w_{k'} + \omega \sum_{t \in [1,T-1]} \Big(f_{k'}(\tilde{x}_t, \tilde{x}_{t+1}, z, t) - f_{k'}(x_t^*, x_{t+1}^*, z, t) \Big) \end{aligned}$$

for any $\omega > 0$. Typically we choose a small ω for numerical stability (e.g. $\omega \in [0.001, 0.01]$). The data is passed through multiple times until all predicted labels are correct, or until time runs out.

As proposed in [2], we may want to use the average of these parameters over all steps for prediction rather than the parameter at the last step. The idea is that we consider the model learned at each step is an 'expert' and the final model is the 'voted' version of all experts. This technique appears to improve the prediction accuracy and is more stable.

In general, voted perceptron may be slightly less accurate that the maximum likelihood methods but it is quite fast (about 5-10 times faster) and is easy to implement. So it may be a good candidate for system testing purposes.

4.2 Computing the Log-likelihood and Its Gradient

The log-likelihood is the quantity we want to optimise in the learning process. It is given as

$$\mathcal{L} = \sum_{t\in[1,T]} \sum_{k} w_k f_k(\tilde{x}_t, z, t) - \sum_{t\in[1,T-1]} \sum_{k'} w_{k'} f_{k'}(\tilde{x}_t, \tilde{x}_{t+1}, z, t) - \log Z(z) - \sum_{k} \frac{w_k^2}{2\sigma^2} \tag{7}$$

where $Z(z)$ is a special quantity known as partition function with respect to the input z, and σ is the standard deviation of the Gaussian distribution. The last term is to 'regularise' the objective function as it penalises large parameter w_k. Choosing the right σ often requires trials-and-errors, probably through different values like $0.1, 10$, and 10.

The gradient is given

$$G_k^z = \sum_{t\in[1,T]} \left(f_k(t) - \sum_{x_t} P_t(x_t|z) f_k(x_t, z, t) \right) - \frac{w_k}{\sigma^2} \tag{8}$$

$$G_{k'}^z = \sum_{t\in[1,T-1]} \left(f_{k'}(\tilde{x}_t, \tilde{x}_{t+1}, z, t) - \sum_{x_t} \sum_{x_{t+1}} P_t(x_t, x_{t+1}|z) f_{k'}(x_t, x_{t+1}, z, t) \right) - \frac{w_{k'}}{\sigma^2} \tag{9}$$

where $P_t(x_t|z)$ is the probability of the label x_t occurs at time t given the data instance z. This can be implemented as a 2D array of size $S \times T^z$. Likewise, $P_t(x_t, x_{t+1}|z)$ is the probability that both the labels x_t and x_{t+1} co-occur at time t and t+1, respectively. This can be implemented as a 3D array of size $S \times S \times T^z$. The two arrays are computed in a two-pass procedure through the data sequence. This procedure is known as *forward-backward*.

In the case where each node feature k is associated with only one assignment of x_t (as in Equation 3) then we do not need to make the sum over x_t in Equation 8. Likewise, if each edge feature k' is associated with the assignments of the pair (x_t, x_{t+1}) (as in Equations 4 and 6) then we can drop the sum over x_t and x_{t+1} in Equation 9.

4.3 The Forward Pass

The forward-pass refers to the scanning from time $t = 1$ to time $t = T$. At the end of the pass, we are able to compute the partition function Z. We need another pass in the reverse order to compute all the local probabilities.

Assume that we have computed all the local potentials $\phi_t(x_t, z)$ and $\psi_t(x_t, x_{t+1}, z)$. We need a bookkeeper known as the forward variable, denoted by $\alpha_t[x_t]$. This is a 2D array of size $S \times T$. When $t = 1$, we initialise $\alpha_1(x_1) = 1/S$ for all labels x_1. When $t > 2$, we have the following recursion

$$\alpha_t[x_t] = \kappa_t \sum_{x_{t-1}} \alpha_{t-1}[x_{t-1})]\phi(x_t, z)\psi_{t-1}(x_{t-1}, x_t, z) \tag{10}$$

where $\kappa_t > 0$ is the scaling factor. This is important to avoid the so-call numerical overflow problem which happens when T is large. Typically, we set κ_t so that $\sum_{x_t} \alpha_t[x_t] = 1$.

That is, for each x_t, we need to sum over all the previous labels x_{t-1}. So to fill in the forward bookkeeper, we need $S \times S \times T$ steps.

The log-partition function can be computed as

$$\log Z(z) = \sum_{t\in[1,T]} \log \kappa_t + \log \sum_{x_T} \alpha_T[x_T]\phi_T(x_T, z) \tag{11}$$

4.4 The Backward Pass

The backward pass is almost identical to the forward pass. The only difference is that we scan from time $t = T$ to time $t = 1$. We also need to maintain a 2D array bookkeeper known as backward

Algorithm 1 Computing log-likelihood and gradient

Input: data patterns APIs, parameter vector w
Output: real evaluation of log-likelihood and gradient vector

```
For d = 1 to D
  Compute the node potentials φ_t(x_t, z) for t ∈ [1,T] using Equation 1
  Compute the edge potentials ψ_t(x_t, x_{t+1}, z) for t ∈ [1,T−1] using Equation 2
  Compute forward variables α_t[x_t]
        and store scaling factors κ_t for t ∈ [1,T] using Equation 10
  Compute backward variables β_1[x_t] for t ∈ [1,T] using Equation 12
  Compute the node probabilities P_t(x_t, z) for t ∈ [1,T] using Equation 13
  Compute the edge probabilities P_t(x_t, x_{t+1}, z) for t ∈ [1,T−1] using Equation 14
  Compute the log-likelihood using Equations. 11 and 7
  Compute the gradient using Equations 9 and 8
End
```

variables, denoted by $\beta_t[x_t]$. When $t = T$, we initialise $\beta_T[x_T] = 1/S$ for all labels x_T. When $t < T$, the recursion is as follows

$$\beta_t[x_t] = \mu_t \sum_{x_{t+1}} \beta_{t+1}[x_{t+1}]\phi_{t+1}(x_{t+1}, z)\psi_t(x_t, x_{t+1}, z) \tag{12}$$

The is another way to compute the partition function

$$\log Z(z) = \sum_{t\in[1,T]} \log \mu_t + \log \sum_{x_1} \beta_1(x_1)\phi_1(x_1, z)$$

4.5 Local Probabilities

Once the forward and backward arrays have been filled, local probabilities are straightforward

$$P_t(x_t|z) = \lambda_t \alpha_t[x_t]\phi_t(x_t, z)\beta_t[x_t] \tag{13}$$
$$P_t(x_t, x_{t+1}|z) = \gamma_t \alpha_t[x_t]\phi_t(x_t, z)\psi_t(x_t, x_{t+1}, z)\phi_{t+1}(x_{t+1}, z)\beta_{t+1}[x_{t+1}] \tag{14}$$

where λ_t, γ_t are the normalisation factors to ensure that

$$\sum_{x_t} P_t(x_t|z) = 1$$
$$\sum_{x_t}\sum_{x_{t+1}} P_t(x_t, x_{t+1}|z) = 1$$

respectively.

4.6 Put Everything Together

We assume that features have been computed, or there exists an API to get any feature we want to use. The algorithm to compute the log-likelihood and the gradient vector is summarised in Algorithm 1.

5 The Testing Phase

In the testing phase, we want to provide the labels for the whole sequence, not just a single node. Theoretically, we want to find the sequence of labels that are most likely to happen given the input. There are several ways to do, but here we describe the well-known method called Viterbi decoding [6]. This is a two-step procedure

Algorithm 2 Viterbi decoding

Input: data patterns APIs, parameter vector w
Output: sequence of optimal labels
`Compute the node potentials` $\phi_t(x_t, z)$ `for` $t \in [1, T]$ `using Equation 1`
`Compute the edge potentials` $\psi_t(x_t, x_{t+1}, z)$ `for` $t \in [1, T-1]$ `using Equation 2`
`Compute maximal forward variables` $\alpha_t^{\max}[x_t]$ `using Equation 15`
`and update bookkeeper` $Y_t[x_t]$ `using Equation 16`
`Backtrack to decode the optimal label sequence using Eqs.17 and 18.`

1. The *maximal forward* pass. This is almost identical to the forward pass described above. The only difference is that we replace the summation by the maximisation. We need to maintain a bookkeeper to hold the local optimal labels.

2. The *backtracking* pass. Given the bookkeeper, we track backward to decode the most likely labels.

5.1 The Maximal Forward Pass

We need to maintain a 2D array of maximal forward variables $\alpha_t^{\max}[x_t]$. We also need a 2D bookkeeper $Y_t[x_t]$ of size $S \times T$. When $t = 1$, we initialise $\alpha_1^{\max}[x_1] = 1/S$ for all labels x_1. When $t > 2$, we have the following recursion

$$\alpha_t^{\max}[x_t] = \kappa_t \max_{x_{t-1}} \left(\alpha_{t-1}[x_{t-1}^{\max}] \phi_t(x_t, z) \psi_{t-1}(x_{t-1}, x_t, z) \right) \tag{15}$$

$$Y_t[x_t] = \arg\max_{x_{t-1}} \left((\alpha[x_{t-1}^{\max}] \phi_t(x_t, z) \psi_{t-1}(x_{t-1}, x_t, z) \right) \tag{16}$$

where $\kappa_t > 0$ is any scaling factor. The purpose of the bookkeeper is to store the optimal label at time $t-1$ according to the label x_t.

5.2 Backtracking

The purpose of backtracking is to recover the optimal labels. Since at each time step t and label x_t, we store the previous optimal label x_{t-1}, if we know x_t^*,we can always recover x_{t-1}^*. At the end of the sequence, i.e. $t = T$, the optimal label is

$$x_T^* = \arg\max_{x_T} \left(\alpha_T^{\max}[x_T] \phi_T(x_T, z) \right) \tag{17}$$

Then for general case: for $t = T-1, T-2, ..., 1$

$$x_t^* = Y_t[x_{t+1}^*] \tag{18}$$

The sequence $(x_1^*, x_2^*, ..., x_T^*)$ is the optimal label sequence.

6 Other Practical Issues

6.1 Trade-off between Time and Space

One important issue is memory requirement and training time. In standard implementation, each pass through a training sequence takes $S \times S \times T$ steps. In typical text processing applications, we may have to deal with millions of sentences. Besides, the L-BFGS may takes around $100 - 1000$ iterations to converge. The L-BFGS stores about $10 - 20$ gradient vectors (the size of the gradient

vector is the same as the number of features). This is quite significant because the number of features (or equivalently the size of the gradient vector) can be as large as several millions. So we may not have the option to pre-compute all the features for large data set (e.g. about a million sentences).

6.2 Build and Test

Often we have to deal with large-scale data, especially in natural language processing. The most time-consuming part in using the system is perhaps learning. Theoretically, we need to learn the model only once and use it theoretically forever. Practically, however, we do not have such luxury because we need to test the system for correctness and performance. So it is best to extract small subsets of training and testing data for debugging purposes. Bear in mind that the performance on small datasets may not necessarily reflect that on the full-scale dataset.

6.3 Parallel Implementation

Since we are dealing with independent data, parallelisation is quite straightforward. We can first randomly decompose the training data $\mathcal{D}$ into equal partitions $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_N$. Each process is in charge of one partition, the gradient and log-likelihood are passed to the master process to perform parameter update. See [5] for implementation details.

7 Case Study: Noun-Phrase Chunking

In this experiment we apply the CRFs to the task of noun-phrase (NP) chunking. The data is from the CoNLL-2000 shared task [8], in which 8926 English sentences from the Wall Street Journal corpus are used for training and 2012 sentences are for testing. Each word in a sentence is annotated by two labels: the part-of-speech (POS) and the noun-phrase (NP). There are 44 POS labels and 3 NP labels (B-NP for beginning of a noun-phrase, I-NP for inside a noun-phrase or O for others). Each noun-phrase generally has more than one word. The POS tags are actually the output of the Brill's tagger [1], while the NPs are manually labeled.

In our implementation, we separate the data pattern extraction task from the CRF model. We extract data patterns from the text in the way similar to that in [11]. Pre-processing steps include extraction of unigram and bigram lists and finding the association between words and POS tags. We consider only a limited vocabulary extracted from the training data in that we only select unigrams and bigrams with more than 3 occurrences. This reduces the vocabulary and the feature size significantly. Data patterns are extracted from words and POS tags falling within a sliding window of size 5. This setting gives rise to about 32+K data patterns.

Since most data patterns for each word position are zeros (this is due to the fact that we use only indicator features), we need not to include them in the feature database. So a sparse representation is used. We store the pre-computed features in a text file, where each line has the following format:

```
v1:ID1 v2:ID2 v3:ID3 ...
```

where v1, v2 and v3 are feature values (which are 1 in this case) and ID1, ID2 and ID3 are respective feature indices.

As outlined in Section 3, there are two alternatives of associating data patterns: with the nodes and with the edges. In the first choice, edge features are simply indicator functions. Likewise in the second choice, node features are set to indicators. The number of parameters are $S \times M + S^2$ and $S + S^2 M$, respectively[2]. Clearly, the number of parameters in the first choice is much smaller when S is large.

[2] Recall that S is the size of the label set, and M is the number of data patterns.

Associating data patterns with nodes.

Training data	10%	100%
VP	83.11/83.04/83.07	88.10/88.21/88.16
SGA	90.39/90.70/90.54	91.66/92.61/92.13
L-BFGS	90.48/91.07/90.77	92.66/92.93/92.80

Associating data patterns with edges.

Training data	10%	100%
VP	90.51/90.63/90.57	93.18/93.43/93.30
SGA	91.06/91.15/91.11	93.01/93.43/93.22
L-BFGS	90.85/91.50/91.17	93.01/93.37/93.19

Table 3: Results of noun-phrase chunking on the CoNLL2000 data: Recall/Precision/F_1.

For optimisation, we implement Collin's voted perceptron (VP), stochastic gradient ascent (SGA) and L-BFGS. The L-BFGS code is borrowed from Taku Kudo[3]. The voted perceptron is stopped after 100 iterations, or when the training error increases or reaches zero. The learning rate for the stochastic gradient is 0.1, and it is stopped after 20 iterations or when the log-likelihood decreases. The L-BFGS is terminated after 100 iterations or after the convergence rate falls below 10^{-5}. The log-likelihood is regularised by a Gaussian prior with standard deviation of $\sigma = 3$.

The performance of the three training algorithms the two pattern association methods are reported in Table 3. We evaluate these combinations on two training sets: a small set which is about 10% of the original set, and the full set. Three measure scores are used: recall/precision/F_1.

In our experiments, the voted perceptron is the fastest training algorithm but its performance may be worse than the others. The stochastic gradient ascent is generally fast as it requires less iterations to reach reasonable performance. The L-BFGS may achieve a better performance at the cost of running time. It also requires good machine precision to be numerically stable. In our study, when converting `double` data to `float`, the L-BFGS by Kudo crashes.

8 Conclusion

We have attempted to introduce the CRFs in the way that may be helpful for practitioners. However, there is no substitute for deep understanding of the probability and statistics theory behind the formulation of CRFs.

References

[1] E. Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–566, 1995.

[2] M. Collins. Discriminative training methods for hidden Markov models: Theory and experiments with the perceptron algorithm. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2002.

[3] Y. Freund and R.E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, 1999.

[3] http://crfpp.sourceforge.net/

[4] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the International Conference on Machine learning (ICML)*, pages 282–289, 2001.

[5] X.-H. Nguyen Phan, L.-M. Inoguchi, and S. Y. Horiguchi. High-performance training of conditional random fields for large-scale applications of labeling sequence data. *IEICE Transactions on Information and Systems*, E90, Series E(1):13–21, Jan 2007.

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

[7] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol Rev*, 65(6):386–408, Nov 1958.

[8] Erik F. Tjong Kim Sang and Sabine Buchholz. Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of the 2nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning*, volume 7, pages 127–132, Lisbon, Portugal, 2000. http://www.cnts.ua.ac.be/conll2000/chunking/.

[9] Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In Marti Hearst and Mari Ostendorf, editors, *Proceedings of Human Language Technology (NAACL)*, pages 213–220, Edmonton, Alberta, Canada, May 27 - June 1 2003. Association for Computational Linguistics.

[10] 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*, chapter 4, pages 93–128. MIT Press, 2006.

[11] Charles Sutton, Andrew McCallum, and Khashayar Rohanimanesh. Dynamic conditional random fields: Factorized probabilistic models for labeling and segmenting sequence data. *Journal of Machine Learning Research*, 8:693–723, Mar 2007.

[12] 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 *Proceedings of the International Conference on Machine learning (ICML)*, pages 969–976, 2006.