FlexCRFs: Flexible Conditional Random Fields

(Including **PCRFs -** a parallel version of FlexCRFs)

http://www.jaist.ac.jp/~hieuxuan/flexcrfs/flexcrfs.html

Copyright © 2004-2005 by

Hieu Xuan Phan & Minh Le Nguyen
{hieuxuan, nguyenml}@jaist.ac.jp

Graduate School of Information Science,
Japan Advanced Institute of Science and Technology (JAIST)

Table of Contents

1. Introduction

FlexCRFs is a conditional random field toolkit for segmenting and labeling sequence data written in C/C++ using STL library. It was implemented based on the theoretic model presented in (Lafferty et al., 2001) and (Sha and Pereira, 2003). The toolkit uses L-BFGS (Liu and Nocedal, 1989) – an advanced optimization procedure – to train CRF models. FlexCRFs was designed to deal with hundreds of thousand data sequences and millions of features. FlexCRFs supports both first-order and second-order Markov CRFs. We have tested FlexCRFs on Linux (Fedora), Sun Solaris, and MS Visual C++ 7.0 (MS Windows). Comments, suggestions, and error detections are highly appreciated.

PCRFs is the parallel version of FlexCRFs that allows to train conditional random fields on massively parallel processing systems supporting Message Passing Interface (MPI). PCRFs allows to train conditional random fields on large-scale datasets containing up to millions of data sequences. We have tested PCRFs on massively parallel systems such as Cray XT3, SGI Altix, and IBM SP.

1.1. License

FlexCRFs and PCRFs are free tools. You can redistribute them and/or modify them under the terms of GNU General Public License as published by the Free Software Foundation, either version 2 of the License, or (at your option) any later version.

FlexCRFs and PCRFs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY of FINESS FOR A PARTICULAR PURPOSE. Please see the GNU General Public License for more details.

1.2. Download

The source code of FlexCRFs and PCRFs can be downloaded from:

http://www.jaist.ac.jp/~hieuxuan/flexcrfs/flexcrfs.html

If you have any question, please feel free to contact us:

Email: {hieuxuan, nguyenml}@jaist.ac.jp or pxhieu@gmail.com

2. Building and Installation

2.2. Building and Install FlexCRFs

Source Code Organization:

Both FlexCRFs and PCRFs were written in C/C++ using STL library. The source code of FlexCRFs is organized in the directory tree as follows:

```
apps                    (user applications, e.g., chunking or POS tagging)
     Chunking           (chunking applications)
     NER                (named entity recognition)
     POSTagging         (part-of-speech tagging)
     <your own apps>    (create directories for your own applications here)
bin                     (outputs of the compiling process)
docs                    (documents, e.g., this manual)
include                 (header files)
lib                     (output library files)
src                     (source code of FlexCRFs)
     crfs               (source code of CRFs)
          evaluation    (source code of evaluation utilities)
          feature       (source code of CRF feature, feature generator)
          math          (source code of mathematic functions)
          misc          (source code of option class, etc.)
          trainer       (source code of the training procedures)
          viterbi       (source code of Viterbi algorithm for decoding
                         CRFs, including searching for n-best label paths)
     dataitf            (source code of data format and dictionary)
     feasel             (source code of feature selection utilities)
     utils              (source code of other utilities, e.g., strtokenizer)
```

System Requirements:

- Linux/Unix/Cygwin:
  - Compiler: GNU C Compiler (gcc) and GNU C++ Compiler (g++)
  - Library: STL
- MS Windows 2000, XP:
  - Compiler: MS Visual C++ 7.0
  - Library: STL

Building and Install FlexCRFs on Linux/Unix:

- Download FlexCRFs and unzip its source code:

```
$ gunzip FlexCRFs.tar.gz
$ tar –xf FlexCRFs.tar
```

- Compile (go to FlexCRFs directory):

`$ make clean` (remove any previous output)

`$ make all` (compile FlexCRFs)

- Install (you must login the system under the “root” privilege):

  `$ make install` (install FlexCRFs)

  `$ make uninstall` (uninstall FlexCRFs)

The outputs f the training process are several executable files in which the main program of FlexCRFs is “crf”. All of these files are copied to the “bin” directory. All the objective files will be put into a statistic library and copied to the “lib” directory.

Building and Install FlexCRFs on MS Windows with Visual C++ 7.0:

2.3. Building and Install PCRFs

Source Code Organization:

PCRFs was written in C/C++ using STL and Message Passing Interface (MPI). It can be compiled and run on any parallel systems (PC clusters, massively parallel processing systems) that support MPI. The source code organization is similar to that of FlexCRFs except that we have different versions of Makefile supporting different platforms.

System Requirements:

- GNU C/C++ compilers (gcc and g++) that support MPI
- STL library
- The parallel system must support a network file system that allows us to mount the working directory (that contain PCRFs and our applications) to all computing nodes in order that those computing nodes can read (or write) their training and testing data partitions independently and simultaneously.

Building PCRFs on Massively Parallel Processing Systems:

- Download PCRFs and unzip its source code:

  `$ gunzip PCRFs.tar.gz`

  `$ tar –xf PCRFs.tar`

We prepare two versions of Makefile for two parallel platforms: Cray XT3 and SGI Altix. Users can slightly modify the Makefiles for compiling PCRFs on other parallel systems other than Cray XT3 and SGI Altix.

- Compile (go to PCRFs directory):

  `$ cp Makefile.CrayXT3 Makefile` (if the system is Cray XT3)

  `$ cp Makefile.SGIAltix Makefile` (if the system is SGI Altix)

  `$ make clean` (remove any previous output)

  `$ make all` (compile PCRFs)

3. Introduction to Conditional Random Fields

Conditional random fields (CRFs - Lafferty et al., 2001) are probabilistic models that were designed for segmenting and labeling sequence data. In this section, we briefly introduce CRFs in order that users can work with FlexCRFs easily. For a complete theoretical presentation of CRFs, please see (Lafferty et al., 2001) and (Sha and Pereira, 2003).

3.1. Conditional Random Fields

Let $o = \{o_1,\ldots,o_T\}$ be some input data observation sequence. Let **S** be a finite set of states, each is associated with a label l ($\in$ **L** = { l_1, …,l_Q}). Let $s = \{s_1,\ldots,s_T\}$ be some state sequence. CRFs (Lafferty et al., 2001) are defined as the conditional probability of a state sequence given an input observation sequence as follows,

$$p_\theta(s \mid o) = \frac{1}{Z(o)} \exp\left(\sum_{t=1}^{T} F(s, o, t) \right) \tag{1}$$

where $Z(o) = \sum_{s'} \exp\left(\sum_{t=1}^{T} F(s', o, t) \right)$ is a normalized factor summing over all label sequences. F(s, o, t) is the sum of CRF features at time position t:

$$F(s, o, t) = \sum_i \lambda_i f_i(s_{t-1}, s_t) + \sum_j \lambda_j g_j(o, s_t) \tag{2}$$

in which f_i and g_j are *edge* and *state* feature functions, respectively. λ_i and λ_j ($\in \theta = \{\lambda_1, \lambda_2, \ldots\}$) are the feature weights associated with f_i and g_j.

$$f_i(s_{t-1}, s_t) \equiv [s_{t-1} = l'][s_t = l]$$

$$g_j(o, s_t) \equiv [x_j(o, t)][s_t = l]$$

where $s_t = l$ means that label l is associated with state s_t. And $x_j(o, t)$ is a logical context predicate that indicates whether or not the observation sequence o (at time t) holds a particular property or fact of empirical data. [*e*] is equal to 1 if the logical expression *e* is *true*, and zero otherwise.

3.2. Inference in CRFs

Inference in CRFs is to find the most likely state sequence s* given the input observation sequence o,

$$s^* = \arg\max_s \; p_\theta(s \mid o) = \arg\max_s \exp\left(\sum_{t=1}^{T} F(s, o, t) \right) \tag{3}$$

To find s*, one can apply the dynamic programming using the Viterbi algorithm (Rabiner, 1989). To avoid an exponential-time search over all possible settings of s, Viterbi stores the probability of the most likely path up to time t which accounts for the first t observations and ends in state s_t. We denote this probability to be $\varphi_t(s_i)$ ($0 \le t < T$) and $\varphi_0(s_i)$ to be the probability of starting in each state s_i. The recursion is given by:

$$\varphi_{t+1}(s_i) = \max_{s_j} \{ \varphi_t(s_j) \exp F(s, o, t+1) \} \tag{4}$$

The recursion terminates when t =T-1 and the biggest value is $p^* = \text{argmax}_i \varphi_T(s_i)$. At this time, we can backtrack through the stored information to find the most likely sequence s*.

3.3. Training CRFs

CRFs are trained by searching the set of weights $\theta = \{\lambda_1, \lambda_2, \ldots\}$ to maximize the log-likelihood, *L*, of a given training data set $\mathbf{D} = \{o^{(j)}, s^{(j)}\}_{j=1..N}$:

$$L = \sum_{j=1}^{N} \log\left(p_\theta(s^{(j)} \mid o^{(j)})\right) - \sum_k \frac{\lambda_k^2}{2\sigma^2} \tag{5}$$

where the second sum is a Gaussian prior over feature weights with variance σ^2, which provides smoothing to deal with sparsity in the training data (Chen & Rosenfeld, 1999).

When the labels make the state sequence unambiguous, the likelihood function in exponential models such as CRFs is convex, thus searching the global optimum is guaranteed. However, the optimum cannot be found analytically. Parameter estimation for CRFs requires an iterative procedure. It has been shown that quasi-Newton methods, such as L-BFGS (Liu and Nocedal, 1989), are most efficient (Sha and Pereira, 2003). This method can avoid the explicit estimation of the Hessian matrix of the log-likelihood by building up an approximation of it using successive evaluations of the gradient.

L-BFGS is a limited-memory quasi-Newton procedure for convex optimization that requires the value and the gradient vector of the function to be optimized. Let $s^{(j)}$ denote the state path of training sequence j in the training set **D**, then the log-likelihood gradient component of λ_k is

$$\frac{\delta L}{\delta \lambda_k} = \left[\sum_{j=1}^{N} C_k(s^{(j)}, o^{(j)})\right] - \left[\sum_{j=1}^{N} \sum_{s} p_\theta(s \mid o^{(j)}) C_k(s, o^{(j)})\right] - \frac{\lambda_k}{\sigma^2} \tag{6}$$

where $C_k(s, o)$ is the count of feature f_k given s and o. The first two terms correspond to the difference between the empirical and the model expected values of feature f_k. The last term is the first-derivative of the Gaussian prior.

3.4. Second-order Conditional Random Fields

Although the first-order Markov CRFs described above perform well for many segmenting and labeling tasks, they fail to encode the long-range interactions among states due to the limitation of the first-order Markov dependency (i.e., the current state depends only on one previous state). The second-order (Markov) CRFs are stronger in capturing such interactions, and thus perform better on labeling/segmenting tasks where the sequential dependencies are essential facts for inference. Sha and Pereira (2003) have also used second-order CRFs for text chunking, and achieved significant results. However, their description about this topology of CRFs is not general enough for an efficient feature selection and applications to other tasks. Here, we present this explicitly so that others can easily re-implement and apply this model to other labeling/segmenting problems.

Features in Second-order CRFs

In the second-order CRFs, we divide features into four categories: edge feature type 1 (e^1), state feature type 1 (s^1), edge feature type 2 (e^2), and state feature type 2 (s^2). Only

e^1 and s^1 are used for first-order CRFs and all of those four are used for second-order models. The sum of feature, F(s, o, t), is now rewritten as follows,

$$F(s,o,t) = \sum_i \lambda_i f_i(s_{t-1}, s_t) + \sum_j \lambda_j g_j(o, s_t) + \sum_k \lambda_k f_k(s_{t-2}, s_{t-1}, s_t) + \sum_h \lambda_h g_h(o, s_{t-1}, s_t) \quad (7)$$

where f_i (type e^1), g_j (type s^1), f_k (type e^2), and g_h (type s^2) are defined as follows,

$$f_i(s_{t-1}, s_t) \equiv [\, s_{t-1}s_t = l'l \,]$$

$$g_j(o, s_t) \equiv [\, x_j(o, t) \,][\, s_t = l \,]$$

$$f_k(s_{t-2}, s_{t-1}, s_t) \equiv [\, s_{t-2}s_{t-1} = l''l' \,][\, s_{t-1}s_t = l'l \,]$$

$$g_h(o, s_t) \equiv [\, x_h(o, t) \,][\, s_{t-1}s_t = l'l \,]$$

A feature of type e^1 is a special case of type s^2 if the logical predicate $x_h(o, t)$ is always true. Because t starts from 1, we need to add a pseudo-state s_0 at the beginning of each sequence. In principle, s_0 can be associated with any label l ($\in$ **L** = { $l_1, \ldots, l_Q$}). However, this would distort or influence the actual sequential dependencies among labels in training data. Therefore, it is better to use a pseudo-label l_0 for s_0. The label set is now **L** = { $l_0, l_1, \ldots, l_Q$}.

Training for and inference in CRFs need an efficient forward-backward computation which manipulates on *transition matrix* M_t at every time position t of each sequence (Lafferty et al., 2001). Unlike in first-order CRFs, the dimension of transition matrixes in second-order CRFs is $|\mathbf{L}|^2 \times |\mathbf{L}|^2$,

$$M_t[l''l'][l'l] = \exp F(s, o, t) \quad (8)$$

Supposing that labels l'', l', and l are represented in integer numbers, the real index of $l'l$ is $l'|\mathbf{L}| + l$, and similarly for $l''l'$. The four types of features can be summed to build the transition matrix M_t as follows: feature type e^2 is corresponding to matrix cell $[l''l'][l'l]$; feature type e^1 and s^2 are corresponding to matrix column $[l'l]$; and feature s^1 is corresponding to matrix columns $[*l]$ (where * is an arbitrary label l').

Please see (Lafferty et al., 2001) and (Sha and Pereira, 2003) for a concrete description of forward-backward and log-likelihood computations.

Inference in Second-order CRFs

Inference in second-order CRFs using Viterbi algorithm also involves the transition matrixes. The recursive variable for second-order CRFs is as,

$$\varphi_{t+1}(s_j, s_i) = \max_{s_k s_j} \{\varphi_t(s_k, s_j) \exp F(s, o, t+1)\} \quad (9)$$

where s_k, s_j, and s_i are states of time positions t -1, t, and t + 1, respectively.

If we have some constraints for Viterbi inference, we can apply them at this level. For example, every matrix cell $M_t[l''l'_1][l'_2l]$ must be zero if $l'_1 \neq l'_2$ because a state cannot be associated with two different labels on the same label path. Also, if we want to prevent the occurrence of a particular pair of consecutive labels $l^u l^v$, we only need to set the column $[l^u l^v]$ of the transition matrix to zero. This will disable all label paths going through this pair of labels.

3.5. Parallel Training of CRFs

While the inference for CRFs based on the Viterbi algorithm is quite efficient, the training process is much more expensive due to the heavy forward-backward computation to evaluate the log-likelihood function and its gradient vector for each iterative scaling step. The time complexity of the training process is $O(mNTQ^2nS)$, in which m is the number of training iterations; N is the number of training data sequences; T is the average length of training sequences; Q is the number of class labels; n is the number of CRF features; and S is the searching time of L-BFGS optimization at each step. In practical implementation, the computational time should be larger due to many other operations such as numerical scaling (to avoid numerical problems), smoothing, and mapping between data formats, etc. The time complexity of the second-order CRFs is even much larger, $O(mNTQ^4nS)$, because the number of labels is now squared.

When the number of labels is large, training CRFs on single computer is very time-consuming. For example, our C/C++ (first-order) CRFs took approximately 100 hours to train POS tagging task on all sections of WSJ (Penn TreeBank) on a single 2.4GHz Opteron processor (Linux OS, 8GB RAM).

Cohn et al. (2005) attempted to reduce the training time of CRFs by casting the original multi-label problem to many binary CRF models, training them independently, and then combining them using error-correcting codes. This method reduces computational time significantly. However, training two-label CRF models independently should lose many important (sequential) dependencies among labels. For example, dependencies and interactions among verb, adverb, adjective, noun, etc. in part-of-speech tagging problem are significant for finding the most likely tag path. Therefore, ignoring this type of information means that the binary CRF models would lose their accuracy considerably.

We, on the other hand, think of a parallel training solution for CRFs on PC clusters or massively parallel processing systems. Interestingly, the nature *sum* of the log-likelihood function allows us to divide the training dataset into different partitions and evaluate log-likelihood and its gradient on each partition independently. Thus, the parallelization is quite straightforward.

Input: Training data: $D = \{(o^{(j)}, l^{(j)})\}_{j=1..N}$; The # of parallel processes: P; And the # of training iterations: m **Output**: Optimal feature weights: $\theta^* = \{\lambda^*_1, \lambda^*_2, \ldots\}$
Initialization: - Generate features with intial weights: $\theta = \{\lambda_1, \lambda_2, \ldots\}$ - Each process loads its training data partitions D_i (and testing partition if need for evaluation)
Parallel Training (each training iteration): 1. The root process broadcasts θ to all parallel processes 2. Each process P_i compute the local log-likelihood L_i and local gradient vector $(\delta L/\delta\lambda_k)_i$ on D_i 3. The root process gathers and sums all Li and $(\delta L/\delta\lambda_k)_l$ to obtain the global L and $\delta L/\delta\lambda_k$ (note that smoothing is only performed on one process, e.g., the root process) 4. The root process performs L-BFGS search to update the new feature weights θ 5. If #iterations < m Then goto step 1 Else stop

Figure 1. Parallel algorithm for training CRFs

In the parallel algorithm (figure 1), the training dataset is randomly divided into equal partitions. At each iteration, the local log-likelihood and its gradient vector are evaluated

in parallel on distributed processes; the root process then gathers and sums those values to obtain the global log-likelihood and its gradient vector; the new setting of feature weights is computed on the root process using L-BFGS optimization; the root process then broadcasts the new setting or the feature weights to all the others for the next training iteration. Synchronization and data communication among processes are performed using a message passing mechanism. Because the L-BFGS search (even for millions of feature weights) is very fast and the communication among processes is usually high-speed link, the parallel algorithm is very efficient and its speed-up ratio approaches the number of parallel processes.

4. How to Use FlexCRFs

4.1. Format of Training and Testing Data

To use FlexCRFs for segmenting and labeling sequence data. Users must fisrt prepare training (and testing) data. Training and testing data have the format specified by the following rules:

```
<Data> := a list of <Data Sequences>
<A Data Sequence> := a list of <Data Observations>
<A Data Observation> := a list of <Context Predicates> + <A Label>
<A Context Predicate> := A string token
<A label> := A string token
```

In other words, training or testing data sets consist of a list of data sequences; and two consecutive data sequences are separated by a blank line. Each data sequence consists of a series of data observations; and each data observation is placed on a line. Each data observation contains a list of context predicates and a label that are separated by blank characters. Context predicates and labels are represented as string tokens, i.e., strings without blank characters. See the "case study" section for examples of training and testing data format.

4.2. FlexCRFs's Options

There are several options for FlexCRFs that control the training and testing process that are described in the following table.

Option	Default value	Description
model_dir	<current dir>	The directory contains the applications
trndata_file	train.tagged	Training data file (for training)
tstdata_file	test.tagged	Testing data file (for testing)
ulbdata_file	data.untagged	Unlabeled data file (for prediction)
model_file	model.txt	Containing the trained model, i.e. the outputs of the training process, including dictionary, feature weights, etc. This will be used for later prediction
trainlog_file	trainlog.txt	Training log file that save the status information of the training process
is_logging	1 (yes)	Logging or not, 1 (if logging), 0 (if no logging)
order	1	The order or CRF model, set to 1 if the first-order and 2 if the second-order Markov CRFs
label_of_first_observation	<automatic>	If this option is not set, a pseudo-label l_0 will be generated for the first observation (s_0).
f_rare_threshold	1	Rare threshold for features, i.e., those features whose occurrence frequency is smaller or equal to this threshold will be removed
cp_rare_threshold	1	Rare threshold for context predicates, i.e., those context predicates whose occurrence frequency is smaller or equal to this value will be removed
num_iterations	50	The number of training iterations
init_lambda_value	0.0	The initial value for the feature weights
sigma_square	100	The sigma square (the variance σ^2) for smoothing
m_for_hessian	7	The number of corrections used in L-BFGS search. The recommend value is between 3 and 7.
evaluate_during_training	0	Whether the testing is performed at every training iteration or not. Set to 1 if we would like to see the testing accuracy of the model at every iteration
chunk_evaluate_during_ training	0	Perform evaluation based on chunk/segment or not. Set to 1 if users perform chunk-based evaluation. In this case, users must specified the chunk information. FlexCRFs only supports four chunk types: IOB1, IOB2, IOE1, IOE2 of text

		chunking and NER.
chunktype	IOB2	The valid value is IOB1, IOB2, IOE1, and IOE2
chunk		Here an example of chunk specification: b-np:i-np:np. This means that the chunk starts with label "b-np", continues with label "i-np", and the chunk name is "np" (noun phrase). This option is repeated several times for all chunks.
nbest	1	This option specifies how many best label paths are inferred for a given input observation sequence. The default value is 1 (the best path). If users would like to infer more than one best path, please set this to larger values, e.g., 5 or 10
prevfixedlabels		This option will be used as a constraint for Viterbi inference. Here is an example for NP chunking: b-np:i-np|i-np (represented as IOB2 style). This means that the allowable labels preceding label "i-np" are only "b-np" or "i-np". This option can be repeated several times for all similar constraints.
nextfixedlabels		This option will also be used as a constraint for Viterbi inference. Here is an example for NP chunking: i-np|i-np:e-np (represented as IOE2 style). This means that the allowable labels going right after the label "i-np" are only "i-np" or "e-np". This option can be repeated several times for all constraints like that.

Figure 2. The list of options of FlexCRFs

Figure 2 shows the list of options of FlexCRFs. To train a CRF model using FlexCRFs, we store all the options in an option file named "option.txt". The "case study" section will show what an option file looks like.

4.3. Training, Testing, and Predicting for Unlabeled Data

Training

To train a CRF model, we must prepare the following files:

- Training data file (usually named as "`train.tagged`")
- Testing data file (usually named as "`test.tagged`") if we would like to perform the evaluation during training to see the testing accuracy at each training iteration
- Option file (usually named as "`option.txt`") that contains necessary options

Putting those files in a directory called the model directory (usually in sub-directory of the "`apps`" directory). From this directory (i.e., the current directory), use the following command to train (and test) the CRF model:

```
$ crf -all -d ./ -o option.txt
```

where "`-all`" means both training and testing. For training only, please replace "`-all`" by "`-trn`". And "`-d ./`" specifies the model directory (model_dir) is the current directory. If we are in another directory other than the model directory, please specify the relative path to it. For example, the model directory is "NPchunking" and we are in the parent directory of it, then replace "`-d ./`" by "`-d ./NPchunking`". "`-o option.txt`" means the option file is "`option.txt`".

The output of the training process is a trained CRF model that is saved in the model file "`model.txt`" in the model directory. The model file contains the model information, such as the mapping between context predicates and integer numbers, the dictionary of the context predicates, the mapping between labels (strings) and label indexes (integers),

and the trained CRF features (feature name, feature id, feature weight). This model file will be used to test or predict labels for unlabeled data. The training procedure also produces the training log file (usually named as "`trainlog.txt`").

Testing

To testing a already trained CRF model, we need three files as follows:

- The model file (i.e., "`model.txt`") of the previously trained CRF model
- The option file (i.e., "`option.txt`")
- And the testing data file (e.g., "`test.tagged`")

All those files are stored in the model directory. Supposing that we are in the model directory, use the following command to perform the testing and evaluation:

```
$ crf -tst -d ./ -o option.txt
```

The outputs of the testing process are the standard measures (precision, recall, and F1-score) based on label and chunks (if the option "chunk_evaluation_during_training" is set to 1 and chunktype and chunk information are specified in the option file). The training process also saves the output file "`test.tagged.tagged`" that contains the content of "`test.tagged`" plus the labels predicted by the CRF model.

Predicting for Unlabeld Data

To predict labels for unlabeled data, we need three following files:

- The model file (i.e., "`model.txt`") of the previously trained CRF model
- The option file (i.e., "`option.txt`")
- And the unlabeled data file (usually named as "`data.untagged`") containing unlabeled data whose labels need to be predicted by the CRF model.

The format of unlabeled data is the same as training and testing data except that the labels of data observations (at the end of each line) are missing. Suppose we are in the model directory, use the following command to perform the prediction:

```
$ crf -prd -d ./ -o option.txt
```

The output of the prediction process is a file (default name: "`data.untagged.model`") containing the content of the unlabeled data ("`data.untagged`") and the labels of data observations predicted by the CRF model.

4.4. Case Study: Noun Phrase Chunking with FlexCRFs

We describe a case study of noun phrase chunking (NP chunking) with FlexCRFs as an example of labeling and segmenting for sequence data.

NP Chunking

Text chunking (also known as phrase chunking, phrase recognition, or shallow parsing) - an intermediate step towards full parsing of natural language – recognizes phrase types (e.g., noun phrase – NP, verb phrase – VP, prepositional phrase – PP, etc.) in input text sentences. NP chunking deals with a part of this task: it involves recognizing the chunks that consist of noun phrases (NPs). Here is an example of a sentence with NP phrase

marking: "[NP *Rolls-Royce Motor Cars Inc.*] *expects* [NP *its U.S. sales*] *to remain steady at* [NP *about 1,200 cars*] *in* [NP *1990*]."

The standard dataset put forward by Ramshaw and Marcus consists of sections 15-18 of the Wall Street Journal corpus as training set and the section 20 of that corpus as testing set. The description of NP chunking task and the dataset can be downloaded from this site: http://staff.science.uva.nl/~erikt/research/np-chunking.html

The evaluation measures for this task are precision, recall, and F_1-score based on whole chunks: precision = a / b; recall = a / c; F_1-score = 2 x precision x recall / (precision + recall), in which a is the number of correctly predicted NP phrases by the CRF model, b is the number of NP phrases predicted by the CRF model, and c is the number of actual NP phrases annotated by humans.

		IOB2	**IOB1**	**IOE2**	**IOE1**
Confidence	NN	B-NP	I-NP	E-NP	I-NP
in	IN	O	O	O	O
the	DT	B-NP	I-NP	I-NP	I-NP
pound	NN	I-NP	I-NP	E-NP	I-NP
is	VBZ	O	O	O	O
widely	RB	O	O	O	O
expected	VBN	O	O	O	O
to	TO	O	O	O	O
take	VB	O	O	O	O
another	DT	B-NP	I-NP	I-NP	I-NP
sharp	JJ	I-NP	I-NP	I-NP	I-NP
dive	NN	I-NP	I-NP	E-NP	I-NP
if	IN	O	O	O	O
trade	NN	B-NP	I-NP	I-NP	I-NP
figures	NNS	I-NP	I-NP	E-NP	I-NP
for	IN	O	O	O	O
September	NNP	B-NP	I-NP	E-NP	I-NP
,	,	O	O	O	O
due	JJ	O	O	O	O
for	IN	O	O	O	O
release	NN	B-NP	I-NP	E-NP	E-NP
tomorrow	NN	B-NP	B-NP	E-NP	I-NP
,	,	O	O	O	O
fail	VB	O	O	O	O
to	TO	O	O	O	O
show	VB	O	O	O	O
a	DT	B-NP	I-NP	I-NP	I-NP
substantial	JJ	I-NP	I-NP	I-NP	I-NP
improvement	NN	I-NP	I-NP	E-NP	I-NP
from	IN	O	O	O	O
July	NNP	B-NP	I-NP	I-NP	I-NP
and	CC	I-NP	I-NP	I-NP	I-NP
August	NNP	I-NP	I-NP	E-NP	E-NP
's	POS	B-NP	B-NP	I-NP	I-NP
near-record	JJ	I-NP	I-NP	I-NP	I-NP
deficits	NNS	I-NP	I-NP	E-NP	I-NP
.	.	O	O	O	O

Observation sequence **Label sequence (according to four representation styles)**

The above table shows a sample data sequence (sentence) with NP phrase marking which will be used as training and testing data. The first two columns constitute the observation sequence containing tokens (English words or punctuations) and their part-

of-speech tags. The last four columns are the label sequences that are represented according to four representation styles (IOB2, IOB1, IOE2, IOE1). "B-" is the beginning of a NP phrase, "I-" is the inside of a NP phrase, "E-" marks the end of a NP phrase, and "O" is the outside of all NP phrases.

To perform NP chunking using FlexCRFs, we must first prepared training and testing data from raw data (i.e., the data from the CoNLL2000 shared task) by feature selection step. Then, we prepare the option file ("`option.txt`") and carry out the training process.

Feature Selection

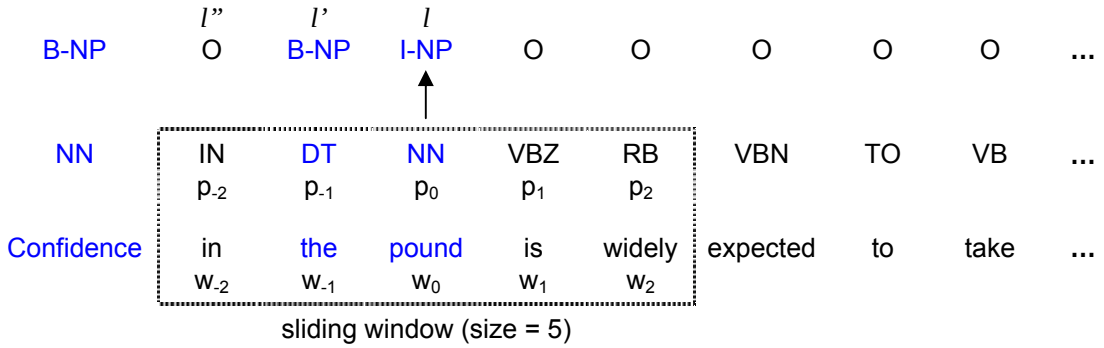

Figure 3. A sliding window (size = 5) moving over the data sequence

s_{t-2}	s_{t-1}	s_t	context preadiate templates $x_j(o, t)$ or (for type s^2)
Template for feature type e^1			
	l'	*l*	
Template for feature type e^2			
l"	*l'*	*l*	
Templates for feature type s^1			
		l	w_{-2}, w_{-1}, w_0, w_1, w_2, $w_{-1}w_0$, w_0w_1
		l	p_{-2}, p_{-1}, p_0, p_1, p_2, $p_{-2}p_{-1}$, $p_{-1}p_0$, p_0p_1, p_1p_2
		l	$p_{-2}p_{-1}p_0$, $p_{-1}p_0p_1$, $p_0p_1p_2$
		l	$p_{-1}p_0p_1w_0$
		l	$p_{-1}w_{-1}$, p_0w_0, $p_{-1}p_0w_{-1}$, $p_{-1}p_0w_0$, $p_{-1}w_{-1}w_0$, $p_0w_{-1}w_0$
Templates for feature type s_2			
	l'	*l*	w_{-1}, w_0, $w_{-1}w_0$, p_{-1}, p_0, $p_{-1}p_0$, $p_{-1}w_{-1}$, p_0w_0
	l'	*l*	$p_{-1}p_0w_{-1}$, $p_{-1}p_0w_0$, $p_{-1}w_{-1}w_0$, $p_0w_{-1}w_0$

Figure 4. Feature templates for NP chunking

Figure 3 shows a sample sequence including observation sequence (each observation consists of a token and its part-of-speech tag) and label sequence (represented in IOB2 style). A token is denoted as "w" and a part-of-speech tag is denoted as "p". Figure 4 shows a set of templates for CRF features. Edge features type e^1 or e^2 are automatically generated from the training data by FlexCRFs. For collecting state features (type s^1 or s^2) we must scan context predicates from raw training data using context predicate

templates $x_j(o, t)$ and $x_h(o, t)$ in figure 4. This can be done by moving a sliding window of size 5 (-2, -1, 0, 1, 2) over all the training data sequences.

For example, the context predicate template "w_{-1}" when being applied to the window in figure 3 will generate the context predicate represented as the string "`w:-1:the`". This string means that "the word at the position -1 (relative to the current window) is "`the`". Similarly, the template "p_2" will generate the context predicate "`p:2:rb`" (note that all tokens and POS tags are converted to lower case or upper case for consistency). The template "$p_{-1}w_{-1}w_0$" will generate the context predicate "`pww:-1:-1:0:dt:the:pound`", the combination of the POS tag ("DT") of the previous word, the previous word ("the"), and the current word ("pound"). In the three above examples, "`w:-1:`", "`p:2:`", and "`pww:-1:-1:0:`" can be seen as prefixes. One can use any other prefix convention provided that the prefixes should help to distinguish any two different context predicates.

For those context predicates belong to feature type s2, we put a '#' character at the beginning to let FlexCRFs know. Thus, we rewrite the three examples of context predicates above as: "`#w:-1:the`", "`p:2:rb`", and "`#pww:-1:-1:0:dt:the:pound`". The first and the third belong to both feature type s^1 and s^2 while the second (generated from template "p_2") only belongs to feature type s^1.

```
Sequence 1
    #w:0:confidence … ww:0:1:confidence:in … #p:0:nn … #pw:0:0:nn:confidence b-np
    #w:-1:confidence … #ww:-1:0:confidence:in … #pww:-1:0:0:confidence:in:in o
    w:-2:confidence … #w:0:the … ppp:-1:0:1:in:dt:nn … #pww:-1:0:0:in:the:dt b-np
    w:-2:in #w:-1:the #w:0:pound w:1:is w:2:widely … p:-2:in #p:-1:dt #p:0:nn … i-np
    w:-2:the #w:-1:pound #w:0:is … #pp:-1:0:nn:vbz … #pww:-1:0:-1:pound:is:nn … o
    …
    w:-2:'s #w:-1:near-record #w:0:deficits w:1:. ww:0:1:deficits:. p:0:nns … i-np
    w:-2:near-record … #w:0:. … #pp:-1:0:nns:. … #pww:-1:0:0:deficits:.:. o
<a blank line>
Sequence 2
    #w:0:chancellor w:1:of w:2:the … #p:0:nnp p:1:in … #pw:0:0nnp:chancellor o
    #w:-1:chancellor #w:0:of … pp:0:1:in:dt … pppw:-1:0:1:0:nnp:in:dt:of … o
    w:-2:chancellor #w:-1:of #w:0:the … #p:0:dt … #pww:-1:0:0:of:the:dt b-np
    w:-2:of #w:-1:the #w:0:exchequer w:1:nigel … #p:0:nnp p:1:nnp p:2:nnp … i-np
    w:-2:the #w:-1:exchequer #w:0:nigel w:1:lawson … ppp:-2:-1:0:dt:nnp:nnp … i-np
    …
    w:-2:the #w:-1:past #w:0:week … #pp:-1:0:jj:nn … #pww:-1:0:0:past:week:nn … i-np
    W:-2:past #w:-1:week #w:0:. … ppp:-2:-1:0:jj:nn:. … #pww:-1:0:0:week:.:. … o
<a blank line>
    …
```

Figure 5. Input training and testing data format for FlexCRFs

Figure 5 shows the format of training and testing data of NP chunking that serve as input data for FlexCRFs. This data format is obtained by moving the sliding window (of size 5) over all training and testing data sequences and collecting context predicates as well as class labels. In figure 5, each data observation is put on each line that contains a list of context predicates and a class label (**b-np**, **i-np**, or **o**); the three-dot "…" character in each line in figure 5 stands for many other context predicates that we do not show explicitly due to the space limitation. Two consecutive data sequences are separated by a blank line.

Although It seems to be a little bit complicated to prepare the training and testing data for FlexCRFs, this operation can be done quite easily by using our feature selection utility called "`chunkingfeasel`" (chunking feature selection) in "`src`" or "`bin`" directory. This utility was written only for text chunking (and NP chunking). The corresponding source code is "`chunkingfeasel.cpp`" in the directory "`src/feasel/Chunking`". Users can

modify this source code according to their feature templates in order to select whatever kind of context predicates from data as they want. The command line of the feature selection utility is as follows:

```
$ chunkingfeasel –lbl/-ulb <raw input data file> <output data file> [tolower]
```

In which, "`-lbl`" is used for training and testing data (i.e., for data that have class labels) and "`-ulb`" is used when we want to prepare unlabeled data for FlexCRFs. <`raw input data file`> is the name of file containing raw data (must be in format of the data of the CoNLL2000 shared task) and <`output data file`> is the output file name. [`tolower`] is used if we want to convert all characters in the output file into lower case ones. For example, if the raw training and testing data files of the NP chunking task are "`train.txt`" and "`test.txt`", respectively. And, the output training and testing files are named "`train.tagged`" and "`test.tagged`", respectively. And, supposing that all data are converted to lower case. The two command lines below will perform the feature selection for training and testing data sets:

```
$ chunkingfeasel –lbl train.txt train.tagged tolower
$ chunkingfeasel –lbl test.txt test.tagged tolower
```

Training Options

In order to train a second-order CRF model for the NP chunking, we must set several options from the option list in figure 2 as follows.

```
traindata_file=train.tagged
testdata_file=test.tagged
order=2
num_iterations=130
f_rare_threshold=1
cp_rare_threshold=1
init_lambda_val=0.05
evaluate_during_training=1
chunk_evaluate_during_training=1
chunktype=IOB2
chunk=b-np:i-np:np
```

The above option-value pairs say that:

- The name of training data file is "`train.tagged`".
- The name of testing data file is "`test.tagged`".
- The order or CRF model is 2 (i.e., the second-order Markov CRFs). The default value is 1 (i.e., the first-order Markov CRFs).
- The number of training iterations is 130.
- The rare thresholds for features (`f_rare_threshold`) and context predicates (`cp_rare_threshold`) are 1 and 1. These thresholds mean that context predicates and features whose occurrence frequencies are smaller than or equal to 1 will be removed.
- The initial value for all feature weights is 0.05 (the default value is zero).
- The evaluation is performed during training (`evaluate_during_training=1`). Use zero if we want to train only.

- Chunk-based evaluation is also performed during training (`chunk_evaluate_during_training=1`). Use zero if we want to disable this functionality.
- If chunk-based evaluation is set, we must provide the chunk type and the chunk information. Here, we use the chunk type IOB2 (`chunktype=IOB2`), and each NP chunk is marked by using two labels "b-np" and "i-np" (`chunk=b-np:i-np:np`).

The above option-pair values are put in the option file ("`option.txt`"). All the other options take their default values.

Training and Performance Evaluation

To train the CRF model for the above NP chunking task, we create a model directory that contains the training and testing data files ("`train.tagged`" and "`test.tagged`") as well as the option file ("`option.txt`"). Supposing that the model directory is put in the "`apps`" directiory: "`apps/Chunking/NPChunking/CoNLL2000`". And suppose the "crf" (after compiling FlexCRFs) is in the "`bin`" directory. We execute the following command to train the CRF model:

```
$ ../../../../bin/crf -all -d ./ -o option.txt
```

The outputs of the training process are two three following files:

- The model file ("`model.txt`") that contains all the information about the trained CRF model including the mapping between the context predicates (string) and their indexes (integer), the mapping between labels (string) and their indexes (integer), the dictionary of used context predicates, and the features (with their trained weights). This file can be used to test or predict labels for unlabeled data.
- The log file of the training process ("`trainlog.txt`") that records all kinds of training information, such as option values, the log-likelihood, the norm of the weight vector and the gradient vector at each training iterations, the training time of each iteration, and the ouput of performance evaluation at each iteration if applicable.
- The test data file ("`test.tagged.model`") with two label columns: one is of the old testing data file (annotated by humans) and another is predicted by the trained CRF model using the Viterbi algorithm. Users can use this file to evaluate the learning performance using their own evaluator.

We provide an evaluation utility ("`evaluatechk`") for computing the precision, recall, and F1-measure for both label-based and chunk-based format. This utility is put in the "`bin`" and was compiled from three source code files "`include/evaluatechunk.h`", "`src/crfs/evaluation/evaluatechunk.cpp`", and "`src/evaluatechk.cpp`". To compute the precision, recall, and F1-measure for "`test.tagged.model`" file using this utility, we must prepare an evaluation option file including the following options:

```
label=b-np
label=i-np
label=o
chunktype=IOB2
chunk=b-np:i-np:np
```

The above option-value pairs let the "`evaluatechk`" know the set of labels (b-np, i-np, o), the chunktype (IOB2), and the chunk information (b-np:i-np:np). These option-pair values are stored in evaluation option file, e.g., "`evaloption.txt`", and suppose that this file is also in the model directory. To perform the evaluation, we execute the command below.

```
../../../../bin/evaluatechk -o evaloption.txt test.tagged.model
```

The output of the evaluation is the highest testing performance (precision, recall, and F1-measure) as follows:

```
Label-based performance evaluation:

        Label   Manual  Model   Match   Pre.(%) Rec.(%) F1-Measure(%)
        -----   ------  -----   -----   ------- ------- -------------
        b-np    12422   12387   12055    97.32   97.05   97.18
        i-np    14376   14350   13974    97.38   97.20   97.29
        o       20579   20640   20263    98.17   98.46   98.32
        -----   ------  -----   -----   ------- ------- -------------
        Avg1.                           97.62   97.57   97.60
        Avg2.   47377   47377   46292    97.71   97.71   97.71

Chunk-based performance evaluation:

        Chunk   Manual  Model   Match   Pre.(%) Rec.(%) F1-Measure(%)
        -----   ------  -----   -----   ------- ------- -------------
        np      12422   12387   11731    94.70   94.44   94.57
        -----   ------  -----   -----   ------- ------- -------------
        Avg1.                           94.70   94.44   94.57
        Avg2.   12422   12387   11731    94.70   94.44   94.57
```

The following text shows the log information of the training iteration 70th at which the highest performance were achieved:

```
Iteration: 70
        Log-likelihood                          =     -10223.317157
        Norm (log-likelihood gradient vector) =       1001.564067
        Norm (lambda vector)                    =        108.228691
        Log-likelihood and gradient computational time: 2 seconds
        Training iteration elapsed: 3 seconds
        Label-based performance evaluation:

                Label   Manual  Model   Match   Pre.(%) Rec.(%) F1-Measure(%)
                -----   ------  -----   -----   ------- ------- -------------
                b-np    12422   12387   12055    97.32   97.05   97.18
                o       20579   20640   20263    98.17   98.46   98.32
                i-np    14376   14350   13974    97.38   97.20   97.29
                -----   ------  -----   -----   ------- ------- -------------
                Avg1.                           97.62   97.57   97.60
                Avg2.   47377   47377   46292    97.71   97.71   97.71

        Chunk-based performance evaluation:

                Chunk   Manual  Model   Match   Pre.(%) Rec.(%) F1-Measure(%)
                -----   ------  -----   -----   ------- ------- -------------
                np      12422   12387   11731    94.70   94.44   94.57
                -----   ------  -----   -----   ------- ------- -------------
                Avg1.                           94.70   94.44   94.57
                Avg2.   12422   12387   11731    94.70   94.44   94.57


        Current max chunk-based F1:  94.57 (iteration 70)
        Training iteration elapsed (including evaluation time): 3 seconds
```

(Note that the above training log file is that of the parallel training of PCRFs on a Cray XT3 system using 45 parallel processes so the training time is very small. The training log file on serial systems is exactly the same).

Methods	F_1-score
Ando & Zhang 2005 (semi-supervised learning, using additional 15 million words as unlabeled data)	94.70
Ours (majority voting among 16 CRFs)	**94.73**
Ours (CRFs, 417,831 features)	**94.57**
Kudo & Matsumoto 2001 (voting among SVMs)	94.39
Kudo & Matusmoto 2001 (SVMs)	94.11
Carreras & Marquez 2003 (perceptrons)	94.41
Sha & Pereira 2003 (CRFs, 3.8 million features)	94.38
Zhang et al. 2002 (generalized winnow + enhanced linguistic features from a full parser	93.89 94.38

Table 1. NP chunking performance comparison

Table 1 shows the F_1-score of the previous systems and ours for NP chunking task on the CoNLL2000 shared task dataset. Ando and Zhang (2005) proposed a nice semi-supervised learning framework that can gain additional information from thousands of auxiliary learning problems relevant to the main learning task. They used additional 15 million words from TREC corpus as unlabeled data to improve this task and obtained the highest F_1-score of 94.70. Kudo and Matsumoto (2001) used SVM combination for this task. They obtained the highest F_1 of 94.11. They voted among SVMs trained according to different label styles (IOB1, IOB2, IOE1, IOE2) and forward-backward inference and achieved the highest F_1 of 94.39. Carreras and Marquez (2003) used two-layer perceptrons and achieved the highest F_1 of 94.41. Sha and Pereira (2003) also used the second-order CRFs for NP chunking and they achieved the highest F_1 of 94.38 but using up to 3.8 million features. Zhang et al. (2002) used generalized winnow for this task and obtained F_1-score of 93.89. They exploited extra enhanced linguistic features from a full parser and got the highest F_1 of 94.38.

Our second-order CRF models used only 417,831 features and achieved the highest F_1 score of 94.57. This is the highest score among normal machine learning techniques (i.e., without using auxiliary tools or unlabeled data). We voted among 16 CRF models trained according to different label styles (IOB2, IOE2) and different initial values for feature weights (0.00, 0.01, 0.02, 0.03, 0.04, 0.05. 0.06. 0.07), and we obtained the highest F_1-score of 94.73. This is the best result for this task.

5. How to Use PCRFs

The format of training and testing data for PCRFs is the same as that of FlexCRFs. This means that we also have to convert raw data into the input training ("`train.tagged`") and testing data ("`test.tagged`") using feature templates before training the CRF model using PCRFs. The list options in figure 2 are also applied for PCRFs.

5.1. Data Partitioning and Initialization

Before training the CRF model using PCRFs on parallel systems. We have to divide the training (and testing) data sets into partitions as well as generate some common data structures, such as context predicate mapping, context predicate dictionary, and feature set, that are shared among parallel processes.

Data Partitioning

Suppose that we have two data files: `train.tagged` and `test.tagged`, and we want to train the CRF model on *m* parallel processes, we have to divide them into *m* partitions by the following commands:

```
$ partition train.tagged m
$ partition test.tagged m
```

For example, if m = 10, the above commands will generate partitions and save them on files as follows: `train.tagged.0, train.tagged.1, …, train.tagged.9, test.tagged.0, test.tagged.1, …, test.tagged.9`. If m = 100, the output files are: `train.tagged.00, train.tagged.01, …, train.tagged.99, test.tagged.00, test.tagged.01, …, test.tagged.99`. The `partition` command is in the "`src`" and "`bin`" directories after compiling PCRFs. Thus, users must specify the relative path to this utility if the model directory is elsewhere in the "`apps`" directory.

Initialization

Before training the model with PCRFs on parallel systems, we must initialize some data structured that are shared among parallel processes, such as the mapping between context predicates (string) and their integer indexes, the dictionary of context predicates, and the CRF features (note that the feature weights are set according the value of the parameter "`init_lambda_val`", the default value is zero). The initialization process is performed using the "`pcrfinit`" utility (this utility is generated after compiling PCRFs):

```
$ pcrfinit –d <model directory> -o option.txt
```

In which `<model directory>` is the directory that contains the application, and the option file is "`option.txt`".

The output of the initialization process is the model file "`model.txt`". This file si similar to that of FlexCRFs, i.e., contains all information about a CRF models, except that all feature weights are set to the value of the parameter "`init_lambda_val`".

5.2. Parallel Training with PCRFs

To train a CRF model using PCRFs, we must prepare the following files:

- The local training data files generated after partitioning data: `train.tagged.01`, `train.tagged.02`, etc.
- The local testing data files generated after partitioning (i.g., `test.tagged.01`, `test.tagged.02`, etc.) if we would like to perform the evaluation during training to see the testing accuracy at each training iteration.
- Option file (usually named as "`option.txt`") that contains necessary options.

Putting those files in a directory called the model directory (usually in sub-directory of the "`apps`" directory). From this this directory (i.e., the current directory), use the following command to train (and test) the CRF model:

```
$ <mpirun> -np <#processes> pcrf -all -d ./ -o option.txt
```

where `<mpirun>` is a MPI utility that is needed to launch the parallel training process; The name of this utility depends on platform of the parallel processing systems; "`-np` <#processes>" is the option that specifies the number of parallel processes. "`pcrf`" is the main program; "`-all`" means we perform both training and testing. For training only, please replace "`-all`" by "`-trn`". And "`-d ./`" specifies the model directory (model_dir) is the current directory. If we are in another directory other than the model directory, please specify the relative path to it. For example, the model directory is "Chunking" and we are in the parent directory of it, then replace "`-d ./`" by "`-d ./Chunking`". "`-o option.txt`" means the option file is "`option.txt`".

The output of the parallel training process using PCRFs is the same as the output of FlexCRFs: a trained CRF model that is saved in the model file "`model.txt`" in the model directory. The model file contains the model information, such as the mapping between context predicates and integer numbers, the dictionary of the context predicates, the mapping between labels (strings) and label indexes (integers), and the trained CRF features (feature name, feature id, feature weight). This model file will be used to test or predict labels for unlabeled data. The model files of FlexCRFs and PCRFs can be used interchangeably because they have the same format. The training procedure also produces the training log file (usually named as "`trainlog.txt`").

5.3. Case Study: Large-Scale Text Chunking with PCRFs

Text Chunking

Text chunking (also known as phrase chunking, phrase recognition, or shallow parsing) - an intermediate step towards full parsing of natural language – recognizes phrase types (e.g., noun phrase – NP, verb phrase – VP, prepositional phrase – PP, etc.) in input text sentences. Here is an example of a sentence with phrase marking: "[NP *Rolls-Royce Motor Cars Inc.*] [VP *expects*] [NP *its U.S. sales*] [VP *to remain*] [ADJP *steady*] [PP *at*] [NP *about 1,200 cars*] [PP *in*] [NP *1990*]."

The training and testing data can be downloaded from the CoNLL2000 shared task[1]. This data consists of the same sections of the Wall Street Journal corpus (Penn TreeBank WSJ) as widely used data for noun phrase chunking: sections from 15 to 18

[1] CoNLL2000 shared task: http://www.cnts.ua.ac.be/conll2000/chunking/

as training data (8936 sentences, 211727 tokens) and section 20 as testing data (2012 sentences, 47377 tokens).

The evaluation measures for this task are precision, recall, and F_1-score based on whole chunks: precision = a / b; recall = a / c; F_1-score = 2 x precision x recall / (precision + recall), in which a is the number of correctly predicted phrases by the CRF model, b is the number of phrases predicted by the CRF model, and c is the number of actual phrases annotated by humans.

		IOB2	**IOB1**	**IOE2**	**IOE1**
Confidence	NN	B-NP	I-NP	E-NP	I-NP
in	IN	B-PP	I-PP	E-PP	I-PP
the	DT	B-NP	I-NP	I-NP	I-NP
pound	NN	I-NP	I-NP	E-NP	I-NP
is	VBZ	B-VP	I-VP	I-VP	I-VP
widely	RB	I-VP	I-VP	I-VP	I-VP
expected	VBN	I-VP	I-VP	I-VP	I-VP
to	TO	I-VP	I-VP	I-VP	I-VP
take	VB	I-VP	I-VP	E-VP	I-VP
another	DT	B-NP	I-NP	I-NP	I-NP
sharp	JJ	I-NP	I-NP	I-NP	I-NP
dive	NN	I-NP	I-NP	E-NP	I-NP
if	IN	B-SBAR	I-SBAR	E-SBAR	I-SBAR
trade	NN	B-NP	I-NP	I-NP	I-NP
figures	NNS	I-NP	I-NP	E-NP	I-NP
for	IN	B-PP	I-PP	E-PP	I-PP
September	NNP	B-NP	I-NP	E-NP	I-NP
,	,	0	0	0	0
due	JJ	B-ADJP	I-ADJP	E-ADJP	I-ADJP
for	IN	B-PP	I-PP	E-PP	I-PP
release	NN	B-NP	I-NP	E-NP	E-NP
tomorrow	NN	B-NP	B-NP	E-NP	I-NP
,	,	0	0	0	0
fail	VB	B-VP	I-VP	I-VP	I-VP
to	TO	I-VP	I-VP	I-VP	I-VP
show	VB	I-VP	I-VP	E-VP	I-VP
a	DT	B-NP	I-NP	I-NP	I-NP
substantial	JJ	I-NP	I-NP	I-NP	I-NP
improvement	NN	I-NP	I-NP	E-NP	I-NP
from	IN	B-PP	I-PP	E-PP	I-PP
July	NNP	B-NP	I-NP	I-NP	I-NP
and	CC	I-NP	I-NP	I-NP	I-NP
August	NNP	I-NP	I-NP	E-NP	E-NP
's	POS	B-NP	B-NP	I-NP	I-NP
near-record	JJ	I-NP	I-NP	I-NP	I-NP
deficits	NNS	I-NP	I-NP	E-NP	I-NP
.	.	0	0	0	0
Observation sequence		**Label sequence (according to four representation styles)**			

The above table shows a sample data sequence (sentence) with phrase marking which will be used as training and testing data. The first two columns constitute the observation sequence containing tokens (English words or punctuations) and their part-of-speech tags. The last four columns are the label sequences that are represented according to four representation styles (IOB2, IOB1, IOE2, IOE1). “B-“ is the beginning of a phrase, “I-“ is the inside of a phrase, “E-“ marks the end of a phrase, and “O” is the outside of all phrases.

To perform text chunking using PCRFs, we must first prepared training and testing data from raw data (i.e., the data from the CoNLL2000 shared task) by feature selection step. Then, we prepare the option file ("`option.txt`") and carry out the training process.

Feature Selection

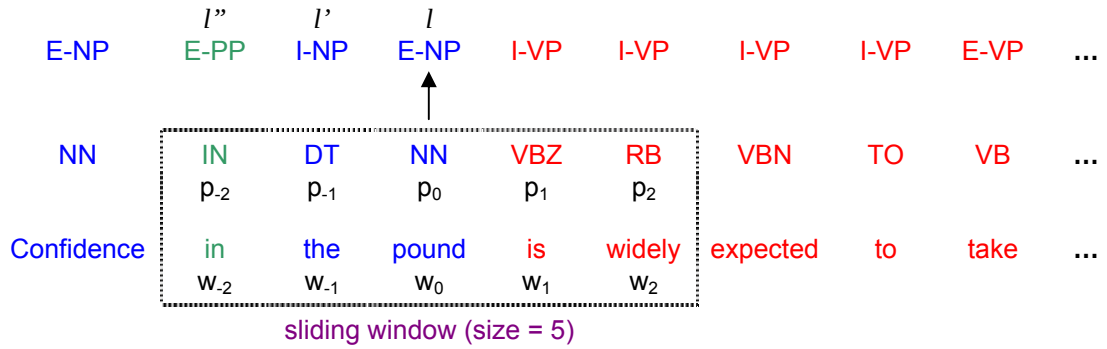

Figure 6. A sliding window (size = 5) moving over the sequence

s_{t-2}	s_{t-1}	s_t	context preadiate templates $x_j(o, t)$ or (for type s^2)
Template for feature type e^1			
	l'	*l*	
Template for feature type e^2			
l"	*l'*	*l*	
Templates for feature type s^1			
		l	w_{-2}, w_{-1}, w_0, w_1, w_2, $w_{-1}w_0$, w_0w_1
		l	p_{-2}, p_{-1}, p_0, p_1, p_2, $p_{-2}p_{-1}$, $p_{-1}p_0$, p_0p_1, p_1p_2
		l	$p_{-2}p_{-1}p_0$, $p_{-1}p_0p_1$, $p_0p_1p_2$
		l	$p_{-1}p_0p_1w_0$
		l	$p_{-1}w_{-1}$, p_0w_0, $p_{-1}p_0w_{-1}$, $p_{-1}p_0w_0$, $p_{-1}w_{-1}w_0$, $p_0w_{-1}w_0$
Templates for feature type s_2			
	l'	*l*	w_{-1}, w_0, $w_{-1}w_0$, p_{-1}, p_0, $p_{-1}p_0$, $p_{-1}w_{-1}$, p_0w_0
	l'	*l*	$p_{-1}p_0w_{-1}$, $p_{-1}p_0w_0$, $p_{-1}w_{-1}w_0$, $p_0w_{-1}w_0$

Figure 7. Feature templates for text chunking

Figure 6 shows a sample sequence including observation sequence (each observation consists of a token and its part-of-speech tag) and label sequence (represented in IOE2 style). A token is denoted as "w" and a part-of-speech tag is denoted as "p". Figure 7 shows a set of templates for CRF features. Edge features type e^1 or e^2 are automatically generated from the training data by PCRFs. For collecting state features (type s^1 or s^2) we must scan context predicates from raw training data using context predicate templates $x_j(o, t)$ and $x_h(o, t)$ in figure 7. This can be done by moving a sliding window of size 5 (-2, -1, 0, 1, 2) over all the training data sequences.

For example, the context predicate template "w_{-1}" when being applied to the window in figure 6 will generate the context predicate represented as the string "`w:-1:the`". This

string means that “the word at the position -1 (relative to the current window) is “the”. Similarly, the template “p_2” will generate the context predicate “p:2:rb” (note that all tokens and POS tags are converted to lower case or upper case for consistency). The template “$p_{-1}w_{-1}w_0$” will generate the context predicate “pww:-1:-1:0:dt:the:pound”, the combination of the POS tag (“DT”) of the previous word, the previous word (“the”), and the current word (“pound”). In the three above examples, “w:-1:”, “p:2:”, and “pww:-1:-1:0:” can be seen as prefixes. One can use any other prefix convention provided that the prefixes should help to distinguish any two different context predicates.

For those context predicates belong to feature type s2, we put a ‘#’ character at the beginning to let FlexCRFs know. Thus, we rewrite the three examples of context predicates above as: “#w:-1:the”, “p:2:rb”, and “#pww:-1:-1:0:dt:the:pound”. The first and the third belong to both feature type s^1 and s^2 while the second (generated from template “p_2”) only belongs to feature type s^1.

Although It seems to be a little bit complicated to prepare the training and testing data, this operation can be done quite easily by using our feature selection utility called “chunkingfeasel” (chunking feature selection) in “src” or “bin” directory. This utility was written only for text chunking (and NP chunking). The corresponding source code is “chunkingfeasel.cpp” in the directory “src/feasel/Chunking”. Users can modify this source code according to their feature templates in order to select whatever kind of context predicates from data as they want. The command line of the feature selection utility is as follows:

```
$ chunkingfeasel –lbl/-ulb <raw input data file> <output data file> [tolower]
```

in which, “-lbl” is used for training and testing data (i.e., for data that have class labels) and “-ulb” is used when we want to prepare unlabeled data. <raw input data file> is the name of file containing raw data (must be in format of the data of the CoNLL2000 shared task) and <output data file> is the output file. [tolower] is used if we want to convert all characters in the output file into lower case ones. For example, if the raw training and testing data files of the chunking task are “train.txt” and “test.txt”, respectively. And, the output training and testing files are named “train.tagged” and “test.tagged”, respectively. And, supposing that all data are converted to lower case. The two command lines below will perform the feature selection for training and testing data sets:

```
$ chunkingfeasel –lbl train.txt train.tagged tolower
$ chunkingfeasel –lbl test.txt test.tagged tolower
```

Training Options

In order to train a second-order CRF model for the NP chunking, we must set several options from the option list in figure 2 as follows.

```
traindata_file=train.tagged
testdata_file=test.tagged
order=2
num_iterations=130
f_rare_threshold=1
cp_rare_threshold=1
init_lambda_val=0.05
evaluate_during_training=1
chunk_evaluate_during_training=1
```

```
chunktype=IOE2
chunk=i-np:e-np:np
chunk=i-pp:e-pp:pp
chunk=i-vp:e-vp:vp
chunk=i-sbar:e-sbar:sbar
chunk=i-adjp:e-adjp:adjp
chunk=i-advp:e-advp:advp
chunk=i-prt:e-prt:prt
chunk=i-lst:e-lst:lst
chunk=i-intj:e-intj:intj
chunk=i-conjp:e-conjp:conjp
chunk=i-ucp:e-ucp:ucp
```

The above option-value pairs say that:

- The name of training data file is "`train.tagged`".
- The name of testing data file is "`test.tagged`".
- The order or CRF model is 2 (i.e., the second-order Markov CRFs). The default value is 1 (i.e., the first-order Markov CRFs).
- The number of training iterations is 130.
- The rare thresholds for features (`f_rare_threshold`) and context predicates (`cp_rare_threshold`) are 1 and 1. These thresholds mean that context predicates and features whose occurrence frequencies are smaller than or equal to 1 will be removed.
- The initial value for all feature weights is 0.05 (the default value is zero).
- The evaluation is performed during training (`evaluate_during_training=1`). Use zero if we want to train only.
- Chunk-based evaluation is also performed during training (`chunk_evaluate_during_training=1`). Use zero if we want to disable this functionality.
- If chunk-based evaluation is set, we must provide the chunk type and the chunk information. Here, we use the chunk type IOE2 (`chunktype=IOE2`), and each phrase type (i.e., chunk type) is marked by using two labels. For example, NP phrase is marked by "i-np" and "e-np" (`chunk=i-np:e-np:np`), VP phrase is marked using two labels "i-vp" and "e-vp" (`chunk=i-vp:e-vp:vp`).

The above option-pair values are put in the option file ("`option.txt`"). All the other options take their default values.

Parallel Training

To train the CRF model for the above NP chunking task, we create a model directory that contains the training and testing data files ("`train.tagged`" and "`test.tagged`") as well as the option file ("`option.txt`"). Supposing that the model directory is put in the "`apps`" directiory: "`apps/Chunking/Chunking/CoNLL2000`". And suppose the partitioning utility "`partition`", initialization utility "`pcrfinit`" and PCRFs main program "`pcrf`" (after compiling PCRFs) are all in the "`bin`" directory. Suppose we train PCRFs on 90 parallel processors of a Cray XT3 system. We execute the following commands to partition data, initialize, and train the CRF model:

- **Data Partitioning** (suppose we are in "`apps/Chunking/Chunking/CoNLL2000`" directory):

  ```
  $ ../../../../bin/partition train.tagged 90
  $ ../../../../bin/partition test.tagged 90
  ```

  The outputs of data partitioning are: `train.tagged.00, train.tagged.01, …, train.tagged.89`, and `test.tagged.00, test.tagged.01, …, test.tagged.89`.

- **Initialization**:

  ```
  $ ../../../../bin/pcrfinit –d ./ -o option.txt
  ```

  The output of initialization is the model file "`model.txt`"

- **Parallel Training** (the model file "`model.txt`" and all the local training and testing data files must be available on all computing nodes of the parallel system in order that those nodes can read the initialization information of the CRF model and their data partitions independently and simultaneously. This can be done using a network file system such as NFS. Almost all massively parallel processing systems support this function):

  ```
  $ yod –np 90 ../../../../bin/pcrf –all –d ./ -o option.txt
  ```

  The above command launches the PCRFs on a Cray XT3 system using 90 parallel processors. On other systems such as Altix, users can replace "`yod`" by "`mpirun`".

The outputs of the training process are two three following files:

- The model file ("`model.txt`") that contains all the information about the trained CRF model including the mapping between the context predicates (string) and their indexes (integer), the mapping between labels (string) and their indexes (integer), the dictionary of used context predicates, and the features (with their trained weights). This file can be used to test or predict labels for unlabeled data.
- The log file of the training process ("`trainlog.txt`") that records all kinds of training information, such as option values, the log-likelihood, the norm of the weight vector and the gradient vector at each training iterations, the training time of each iteration, and the output of performance evaluation at each iteration if applicable.

Figure 8 shows the training information (including log-likelihood, the norm of the gradient vector, the norm of the feature weights, training time, and the performance evaluation) of the training iteration 63 at which the highest performance (F_1 = 94.05) was achieved.

```
Iteration: 63
        Log-likelihood                          =     -17513.435276
        Norm (log-likelihood gradient vector) =        564.693971
        Norm (lambda vector)                    =        200.513450
        Log-likelihood and gradient computational time: 78 seconds
        Training iteration elapsed: 78 seconds
        Label-based performance evaluation:

                Label   Manual  Model   Match   Pre.(%) Rec.(%) F1-Measure(%)
                -----   ------  -----   -----   ------- ------- -------------
                e-np    12422   12429   12118    97.50   97.55   97.53
                e-pp    4811    4869    4731     97.17   98.34   97.75
                i-np    14376   14440   13988    96.87   97.30   97.08
                i-vp    2646    2663    2538     95.31   95.92   95.61
                e-vp    4658    4683    4510     96.31   96.82   96.56
                e-sbar  535     500     465      93.00   86.92   89.86
                o       6180    6173    5973     96.76   96.65   96.71
                e-adjp  438     396     343      86.62   78.31   82.25
                i-advp  89      81      57       70.37   64.04   67.06
                e-advp  866     823     706      85.78   81.52   83.60
                i-adjp  167     135     112      82.96   67.07   74.17
                i-sbar  4       15      3        20.00   75.00   31.58
                i-pp    48      37      31       83.78   64.58   72.94
                e-prt   106     106     86       81.13   81.13   81.13
                e-lst   5       0       0         0.00    0.00    0.00
                i-intj  0       0       0         0.00    0.00    0.00
                e-intj  2       1       1       100.00   50.00   66.67
                i-conjp 13      14      8        57.14   61.54   59.26
                e-conjp 9       10      4        40.00   44.44   42.11
                i-prt   0       0       0         0.00    0.00    0.00
                i-ucp   0       0       0         0.00    0.00    0.00
                e-ucp   0       0       0         0.00    0.00    0.00
                -----   ------  -----   -----   ------- ------- -------------
                Avg1.                           76.71   74.29   75.48
                Avg2.   47375   47375   45674    96.41   96.41   96.41

        Chunk-based performance evaluation:

                Chunk   Manual  Model   Match   Pre.(%) Rec.(%) F1-Measure(%)
                -----   ------  -----   -----   ------- ------- -------------
                np      12422   12430   11730    94.37   94.43   94.40
                pp      4811    4869    4723     97.00   98.17   97.58
                vp      4658    4683    4399     93.94   94.44   94.19
                sbar    535     500     453      90.60   84.67   87.54
                adjp    438     396     324      81.82   73.97   77.70
                advp    866     823     696      84.57   80.37   82.42
                prt     106     106     86       81.13   81.13   81.13
                lst     5       0       0         0.00    0.00    0.00
                intj    2       1       1       100.00   50.00   66.67
                conjp   9       10      4        40.00   44.44   42.11
                ucp     0       0       0         0.00    0.00    0.00
                -----   ------  -----   -----   ------- ------- -------------
                Avg1.                           76.34   70.16   73.12
                Avg2.   23852   23818   22416    94.11   93.98   94.05


        Current max chunk-based F1:  94.05 (iteration 63)
        Training iteration elapsed (including evaluation time): 99 seconds
```

Figure 8. Training information of the iteration yielding the highest performance

Methods	F_1-score
Ando & Zhang 2005 (semi-supervised learning, using additional 15 million words as unlabeled data)	94.39
Ours (majority voting among 16 CRFs)	**94.15**
Ours (CRFs, 450,063 features)	**94.05**
Kudo & Matsumoto 2001 (voting among SVMs)	93.91
Kudo & Matusmoto 2001 (SVMs)	93.85
Carreras & Marquez 2003 (perceptrons)	93.74
Zhang et al. 2002 (generalized winnow + enhanced linguistic features from a full parser	93.57 94.17

Table 2. Text chunking performance comparison

Table 2 shows the F_1-score of the previous systems and ours for text chunking task on the CoNLL2000 shared task dataset. Ando and Zhang (2005) proposed a nice semi-supervised learning framework that can gain additional information from thousands of auxiliary learning problems relevant to the main learning task. They used additional 15 million words from TREC corpus as unlabeled data to improve this task and obtained the highest F_1-score of 94.39. Kudo and Matsumoto (2001) used SVM combination for this task. They obtained the highest F_1 of 93.85. They voted among SVMs trained according to different label styles (IOB1, IOB2, IOE1, IOE2) and forward-backward inference and achieved the highest F_1 of 93.91. Carreras and Marquez (2003) used two-layer perceptrons and achieved the highest F_1 of 93.74. Sha and Pereira (2003) did not report results on text chunking task. Zhang et al. (2002) used generalized winnow for this task and obtained F_1-score of 93.57. They exploited extra enhanced linguistic features from a full parser and got the highest F_1 of 94.17.

Our second-order CRF models used only 450,063 features and achieved the highest F_1 score of 94.05. This is the highest score among normal machine learning techniques (i.e., without using auxiliary tools or unlabeled data). We voted among 16 CRF models trained according to different label styles (IOB2, IOE2) and different initial values for feature weights (0.00, 0.01, 0.02, 0.03, 0.04, 0.05. 0.06. 0.07), and we obtained the highest F_1-score of 94.15.

We also performed large-scale training for chunking and NP chunking tasks on larger datasets as follows:

Dataset	Description
CoNLL2000-L	- Training set: sections from 02 to 21 of WSJ corpus (Penn TreeBank III) - Testing set: section 00 of that corpus
CV-test WSJ	25-fold cross-validation tests on all 25 sections of WSJ Each fold: - Take one section as the testing set, - The other 24 sections the training set

We performed both chunking and NP chunking on CoNLL2000-L, and NP chunking on 25-foldCV-test WSJ.

F.	IOB2 $F_{\beta=1}$	IOE2 $F_{\beta=1}$	Max $F_{\beta=1}$	F.	IOB2 $F_{\beta=1}$	IOE2 $F_{\beta=1}$	Max $F_{\beta=1}$
00	96.56	96.54	*96.56*	13	97.17	97.17	*97.17*
01	96.72	96.76	*96.76*	14	96.29	96.51	*96.51*
02	96.76	96.81	*96.81*	15	96.04	96.19	*96.19*
03	96.56	96.53	*96.56*	16	96.42	96.33	*96.42*
04	96.65	96.67	*96.67*	17	96.50	96.52	*96.52*
05	96.55	96.48	*96.55*	18	96.46	96.62	*96.62*
06	96.07	96.78	*96.78*	19	96.90	96.92	*96.92*
07	95.42	95.54	*95.54*	20	95.91	96.05	*96.05*
08	96.79	97.12	*97.12*	21	96.28	96.25	*96.28*
09	96.08	96.06	*96.08*	22	96.47	96.52	*96.52*
10	96.59	96.61	*96.61*	23	96.45	96.43	*96.45*
11	96.01	96.06	*96.06*	24	95.42	95.26	*95.42*
12	95.68	95.97	*95.97*	Avg	**96.35**	**96.42**	***96.45***

Table 3. 25-fold cross-validation test of NP chunking on all 25 sections of WSJ corpus

	NP chunking						Chunking					
Init	IOB2, #features: 1,351,627			IOE2, #features: 1,350,514			IOB2, #features: 1,471,004			IOE2, #features: 1,466,312		
θ	Pre.	Rec.	$F_{\beta=1}$	Pre.	Rec.	$F_{\beta=1}$	Pre.	Rec.	$F_{\beta=1}$	Pre.	Rec.	$F_{\beta=1}$
.00	96.54	96.37	96.45	96.49	96.37	96.43	96.09	96.04	96.06	96.10	96.10	96.10
.01	96.50	96.32	96.41	96.51	96.44	96.48	96.09	96.04	96.06	96.12	96.09	96.11
.02	96.63	96.31	96.47	96.59	96.36	96.47	96.11	96.10	96.10	96.19	96.09	96.14
.03	96.53	96.31	96.42	96.50	96.44	96.47	96.09	96.01	96.05	96.13	96.08	96.11
.04	96.67	96.35	96.51	96.57	96.33	96.45	96.07	95.98	96.03	96.16	96.04	96.10
.05	96.59	96.29	96.44	96.63	96.55	**96.59**	96.12	96.01	96.07	96.13	96.04	96.09
.06	96.54	96.40	96.47	96.72	96.43	96.58	96.10	96.00	96.05	96.20	97.17	**96.18**
.07	96.59	96.33	96.46	96.49	96.54	96.51	96.03	96.07	96.05	96.12	96.17	96.15
	Voting: Pre = 96.80, Rec = 96.68, $F_{\beta=1}$ = **96.74**						Voting: Pre = 96.33, Rec = 96.33, $F_{\beta=1}$ = **96.33**					

Table 4. Results of chunking and NP chunking with different initial feature weights on the CoNLL2000-L (training: sections 02-21, training: section 00 of WSJ corpus)

Methods	NP Chunking F1-score	Chunking F_1-score
Ours (majority voting among 16 CRFs)	**96.74**	**96.33**
Ours (CRFs, 450,063 features)	**96.59**	**96.18**
Kudo & Matsumoto 2001 (voting among SVMs)	95.77	-
Kudo & Matusmoto 2001 (SVMs)	95.34	-
Sang 2000 (system combination)	94.90	-

Table 5. Accuracy comparison of chunking and NP chunking on CoNLL2000-L dataset

In order to investigate NP chunking performance on the whole WSJ, we performed a 25-fold cross-validation test on all 25 sections of it. We train 50 CRF models for 25 folds

using two label representation styles (IOB2, IOE2) and the same intial value of θ (= {0.00, 0.00, …, 0.00}) for feature weights. The numbers of features of these models are about 1.5 million. Table 3 show the higest F_1-score of the 50 CRF models. The *italic* columns show the maximum F_1 between the two models using different label styles, i.e., using IOB2 and IOE2. The last row (right hand side) gives the average F_1 scores of those CRF models, the highest is 96.45.

Table 4 gives the results of NP chunking and chunking on the CoNLL2000-L dataset. We achieved the highest F_1 scores of 96.59 (using 1,350,514 features, initial θ = {0.05, 0.05, …, 0.05}, IOE2) for NP chunking and of 96.18 (using 1,466,312 features, θ = {0.06, 0.06, …, 0.06}, IOE2) for chunking. The highest F_1 scores after voting among CRFs are 96.74 and 96.33 for NP chunking and chunking, respectively.

Table 5 shows the accuracy comparison of NP chunking and chunking on CoNLL2000-L dataset. Sang (2000) performed majority voting among classifiers using different label styles and got the highest F_1 of 94.60. Kudo and Matsumoto (2001) also reported the F_1 of 95.34 on this dataset using SVMs. They performed the voting among SVMs and obtained the highest score of 95.77. No previous work reported results of chunking on this dataset. Our CRFs used from 1.3 to 1.5 million features and achieved the F_1 scores of 96.59 for NP chunking and 96.18 for chunking. We also voted among CRFs and obtained the highest scores of 96.74 and 96.33, respectively. Our method reduces 22.93% error relative to the best NP chunking result (i.e., that of Kudo and Matsumoto).

Computational Time

Task	# training iterations	Training time	
NP Chunking		Single process	45 parallel processes
CoNLL2000	130	4h50’	6’52’’
CoNLL2000-L	130	38h57’	56’
CV test of WSJ	150	55h59’ (estimated)	1h21’
Chunking		Single process	90 parallel processes
CoNLL2000	140	190h32’ (estimated)	2h29’
CoNLL2000-L	200	1348h26’	17h46’

Table 6. Training time of the second-order CRFs on single and parallel processes

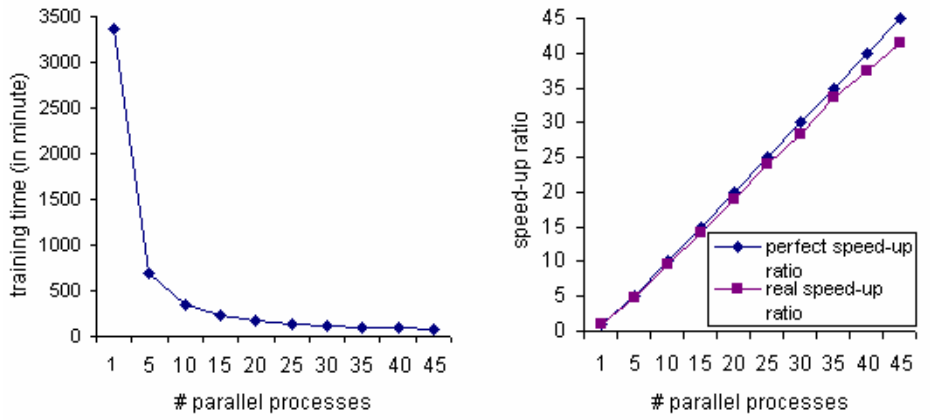

Figure 9. Training time and speed-up ratio of the first-fold of 25-fold CV-test on WSJ

We also measure the computational time of the second-order CRF models. Table 6 reports the training time of 5 tasks using a single process and parallel processes. For example, 130 training iterations of NP chunking on CoNLL2000 dataset using a single process took 4h50' while they took only 6'52" if using 45 parallel processes. Similarly, each fold of 25-fold CV test of WSJ took an average training time of 1h21' on 45 parallel processes while it took approximately 56h if using one process. All-phrase chunking are much more time-consuming than NP chunking. This is because the numbers of labels are 23 on CoNLL2000 and 24 on CoNLL2000-L (including the pseudo-label l_0). For instance, chunking on the CoNLL2000-L dataset requires about 1348h (more than 56 days) for 200 iterations on a single process whereas it took only 17h46' on 90 parallel ones.

Figure 9 shows the change of training time and the speed-up ratio when we train on different numbers of parallel processes. We can see that the training time decreases dramatically and the real speed-up ratio approaches the perfect line. This is because the data exchanged at each training iteration include only log-likelihood function, its gradient vector, and the vector of feature weights. This amount of data is much smaller (even we use millions of features) in comparison with high-speed links among processors.

6. Developing Applications upon FlexCRFs and PCRFs

In order to build sequential label applications upon FlexCRFs and PCRFs. Users should follow the steps below:

- Obtain the raw training (and testing) data
- Choosing feature templates for the CRF model
- Preparing the training (and testing) data ("`train.tagged`" and "`test.tagged`") for FlexCRFs or PCRFs by applying those feature templates to the raw training (and testing) data. Users must design and write their own feature selection utility.
- Preparing option file "`option.txt`" including option-value pairs that are necessary for the application.
- If using PCRFs, users must partition data and initialize the CRF model.
- Then, training the CRF model using FlexCRFs or PCRFs
- The trained CRF model can then be used to predict labels for unlabeled data. The unlabeled data must be the same format as training or testing data except that the label column (at the end of each line) is missing.

Also, users can modify the source code of FlexCRFs and PCRFs to fit their applications.

Acknowledgements

We would like to thank professor Jorge Nocedal, Department of Electrical and Computer Engineering, School of Engineering and Applied Science, Northwestern University, for his provision of L-BFGS FORTRAN source code. www.ece.northwestern.edu/~nocedal/

The C based L-BFGS used in this project is borrowed from CRF++ project developed by Taku Kudo (www.chasen.org/~taku/software/CRF++/). We would like to thank him for his open source project.

A part of this project, the training section (e.g., the computation of log-likelihood function and its gradient vector), is based on the Java source code of CRF project developed by professor Sunita Sarawagi, KR School of Information Technology, IIT Bombay. We would like to thank prof. Sunita Sarawagi for sharing her CRF package and answering related questions. www.it.iitb.ac.in/~sunita/

References

(Ando and Zhang, 2005): R. Ando and T. Zhang. A high-performance semi-supervised learning methods for text chunking. *In Proc. of ACL*, 2005.

(Berger et al., 1996): A. Berger, A. Della Pietra, and J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39-71, 1996.

(Carreras and Marquez, 2003): X. Carreras and L. Marquez. Phrase recognition by filtering and ranking with perceptrons. *In Proc. of RANLP*, 2003.

(Chen and Rosenfeld, 1999): S. Chen and R. Rosenfeld. A gaussian prior for smoothing maximum entropy models. *Technical Report CMU-CS-99-108*. Carnegie Mellon University, 1999.

(Cohn et al., 2005): T. Cohn, A. Smith, M. Osborne. Scaling conditional random fields using error-correcting codes. *In Proc. of ACL*, 2005.

(Kudo and Matsumoto, 2001): T. Kudo and Y. Matsumoto. Chunking with support vector machines. *In Proc. of ACL-NAACL*, 2001.

(Lafferty et al., 2001): J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: probabilistic models for segmenting and labeling sequence data. *In Proc. of ICML*, pp.282-289, 2001.

(Liu and Nocedal, 1989): D. Liu and J. Nocedal. On the limited memory BFGS method for large-scale optimization. *Mathematical Programming*, 45:503-528, 1989.

(Rabiner, 1989): L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *In Proc. of IEEE*, 77(2):257-286, 1989.

(Sang, 2000): E. Sang. Noun phrase representation by system combination. *In Proc. of ANLP-NAACL*, 2000.

(Sha and Pereira, 2003): F. Sha and F. Pereira. Shallow parsing with conditional random fields. *In Proc. of HLT/NAACL*, 2003.

(Zhang et al., 2002): T. Zhang, F. Damerau, and D. Johnson. Text chunking based on a generalization of winnow. *Journal of Machine Learning Research*, 2:615-637.