CRF++: Yet Another CRF toolkit

Introduction

CRF++ is a simple, customizable, and open source implementation of Conditional Random Fields (CRFs) for segmenting/labeling sequential data. CRF++ is designed for generic purpose and will be applied to a variety of NLP tasks, such as Named Entity Recognition, Information Extraction and Text Chunking.

Table of contents

Features

- Can redefine feature sets
- Written in C++ with STL
- Fast training based on LBFGS, a quasi-newton algorithm for large scale numerical optimization problem
- Less memory usage both in training and testing
- encoding/decoding in practical time
- Can perform n-best outputs
- Can perform single-best MIRA training
- Can output marginal probabilities for all candidates
- Available as an open source software

News

2013-02-13: CRF++ 0.58 Released
- Added createModelFromArray() method to load model file from fixed buffer.
- Added getTemplate() method to get template string.

2012-03-25

- Fixed build issue around libtool.
- Fixed C++11 compatible issue.

2012-02-24

- Added CRFPP:Tagger::set_model() method.
- Fixed minor bugs

2012-02-15: CRF++ 0.55

- Added new CRFPP:Model class so that multiple threads can share single CRF++ model.
- Added Tagger::set_penalty and Tagger::penalty() method for dual decompositon decoding
- Fixed crash bug on Windows
- Fixed minor bugs

2010-05-16: CRF++ 0.54 Released

- fixed the bug in L1 regularization. Reported by Fujii Yasuhisa

2009-05-06: CRF++ 0.5 Released

- fixed build failure on libtool

2009-04-19: CRF++ 0.52

- Code clean up
- replaced obsolete sstream with stringstream

2007-07-12: CRF++ 0.51

- Fixed a compilation error on gcc 4.3

2007-12-09: CRF++ 0.50

- Bug fix in --convert mode (Could not generate model from text file)

2007-08-18: CRF++ 0.49

- Added setter/getter for nbest, cost_factor and vlevel to API

2007-07-07: CRF++ 0.48 Released

- Support L1-CRF. use -a CRF-L1 option to enable L1 regularization.

2007-03-07: CRF++ 0.47 Released

- Fixed a bug in MIRA training

2007-02-12: CRF++ 0.46 Released

- Changed the licence from LGPL to LGPL/BSD dual license
- Perl/Ruby/Python/Java binding supports (see perl/ruby/python/java directory respectively)
- Code refactoring

2006-11-26: CRF++ 0.45

- Support 1-best MIRA training (use -a MIRA option)

2006-08-18: CRF++ 0.44

- Fixed a bug in feature extraction
- Allowed redundant spaces in training/test files
- Determined real column size by looking at template
- Added sample code of API (sdk/example.cpp)
- Described usage of each API function (crfpp.h)

2006-08-07: CRF++ 0.43

- implemented several API functions to get lattice information
- added -c option to control cost-factor

2006-03-31: CRF++ 0.42

- Fixed a bug in feature extraction

2006-03-30: CRF++ 0.41

- Support parallel training

2006-03-21: CRF++ 0.40
- Fixed a fatal memory leak bug
- make CRF++ API

2005-10-29: CRF++ 0.3
- added -t option that enables you to have not only binary model but also text model
- added -C option for converting a text model to a binary model

2005-07-04: CRF++ 0.2 Released
- Fixed several bugs

2005-05-28: CRF++ 0.1 Released
- Initial Release

Download

- **CRF++** is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License or new BSD License
- Please let me know if you use **CRF++** for research purpose or find any research publications where **CRF++** is applied.

Source

- CRF++-0.58.tar.gz: HTTP

Binary package for MS-Windows

- HTTP

Installation

- Requirements
  - C++ compiler (gcc 3.0 or higher)
- How to make

```
% ./configure
% make
% su
# make install
```

You can change default install path by using --prefix option of configure script.
Try --help option for finding out other options.

Usage

Training and Test file formats

Both the training file and the test file need to be in a particular format for **CRF++** to work properly. Generally speaking, training and test file must consist of multiple **tokens**. In addition, a **token** consists of multiple (but fixed-numbers) columns. The definition of tokens depends on tasks, however, in most of typical cases, they simply correspond to **words**. Each token must be represented in one line, with the columns separated by white space (spaces or tabular characters). A sequence of token becomes a **sentence**. To identify the boundary between sentences, an empty line is put.

You can give as many columns as you like, however the number of columns must be fixed through all tokens. Furthermore, there are some kinds of "semantics" among the columns. For example, 1st

column is 'word', second column is 'POS tag' third column is 'sub-category of POS' and so on.

The last column represents a true answer tag which is going to be trained by CRF.

Here's an example of such a file: (data for CoNLL shared task)

```
He        PRP  B-NP
reckons   VBZ  B-VP
the       DT   B-NP
current   JJ   I-NP
account   NN   I-NP
deficit   NN   I-NP
will      MD   B-VP
narrow    VB   I-VP
to        TO   B-PP
only      RB   B-NP
#         #    I-NP
1.8       CD   I-NP
billion   CD   I-NP
in        IN   B-PP
September NNP  B-NP
.         .    O

He        PRP  B-NP
reckons   VBZ  B-VP
..
```

There are 3 columns for each token.

- The word itself (e.g. reckons);
- part-of-speech associated with the word (e.g. VBZ);
- Chunk(answer) tag represented in IOB2 format;

The following data is invalid, since the number of columns of second and third are 2. (They have no POS column.) The number of columns should be fixed.

```
He        PRP  B-NP
reckons   B-VP
the       B-NP
current   JJ   I-NP
account   NN   I-NP
..
```

Preparing feature templates

As CRF++ is designed as a general purpose tool, you have to specify the feature templates in advance. This file describes which features are used in training and testing.

- Template basic and macro

  Each line in the template file denotes one *template*. In each template, special macro *%x[row,col]* will be used to specify a token in the input data. *row* specfies the relative position from the current focusing token and *col* specifies the absolute position of the column.

  Here you can find some examples for the replacements

  ```
  Input: Data
  He        PRP  B-NP
  reckons   VBZ  B-VP
  ```

```
the       DT   B-NP << CURRENT TOKEN
current   JJ   I-NP
account   NN   I-NP
```

template	expanded feature
%x[0,0]	the
%x[0,1]	DT
%x[-1,0]	reckons
%x[-2,1]	PRP
%x[0,0]/%x[0,1]	the/DT
ABC%x[0,1]123	ABCDT123

- Template type

  Note also that there are two types of templates. The types are specified with the first character of templates.

  - Unigram template: first character, **'U'**

    This is a template to describe unigram features. When you give a template "U01:%x[0,1]", CRF++ automatically generates a set of feature functions (func1 ... funcN) like:

    ```
    func1 = if (output = B-NP and feature="U01:DT") return 1 else return 0
    func2 = if (output = I-NP and feature="U01:DT") return 1 else return 0
    func3 = if (output = O and feature="U01:DT") return 1  else return 0
    ....
    funcXX = if (output = B-NP and feature="U01:NN") return 1  else return
    funcXY = if (output = O and feature="U01:NN") return 1  else return 0
    ...
    ```

    The number of feature functions generated by a template amounts to (L * N), where L is the number of output classes and N is the number of unique string expanded from the given template.

  - Bigram template: first character, **'B'**

    This is a template to describe bigram features. With this template, a combination of the current output token and previous output token (bigram) is automatically generated. Note that this type of template generates a total of (L * L * N) distinct features, where L is the number of output classes and N is the number of unique features generated by the templates. When the number of classes is large, this type of templates would produce a tons of distinct features that would cause inefficiency both in training/testing.

  - What is the diffrence between unigram and bigram features?

    The words unigram/bigram are confusing, since a macro for unigram-features does allow you to write word-level bigram like %x[-1,0]%x[0,0]. Here, unigram and bigram features mean uni/bigrams of output tags.

    - unigram: |output tag| x |all possible strings expanded with a macro|

    - bigram: |output tag| x |output tag| x |all possible strings expanded with a macro|

- Identifiers for distinguishing relative positions

  You also need to put an identifier in templates when relative positions of tokens must be distinguished.

  In the following case, the macro "%x[-2,1]" and "%x[1,1]" will be replaced into "DT". But they indicates different "DT".

```
The        DT  B-NP
pen        NN  I-NP
is         VB  B-VP << CURRENT TOKEN
a          DT  B-NP
```

  To distinguish both two, put an unique identifier (U01: or U02:) in the template:

```
U01:%x[-2,1]
U02:%x[1,1]
```

  In this case both two templates are regarded as different ones, as they are expanded into different features, "U01:DT" and "U02:DT". You can use any identifier whatever you like, but it is useful to use numerical numbers to manage them, because they simply correspond to feature IDs.

  If you want to use "bag-of-words" feature, in other words, not to care the relative position of features, You don't need to put such identifiers.

- Example

  Here is the template example for CoNLL 2000 shared task and Base-NP chunking task. Only one bigram template ('B') is used. This means that only combinations of previous output token and current token are used as bigram features. The lines starting from # or empty lines are discarded as comments

```
# Unigram
U00:%x[-2,0]
U01:%x[-1,0]
U02:%x[0,0]
U03:%x[1,0]
U04:%x[2,0]
U05:%x[-1,0]/%x[0,0]
U06:%x[0,0]/%x[1,0]

U10:%x[-2,1]
U11:%x[-1,1]
U12:%x[0,1]
U13:%x[1,1]
U14:%x[2,1]
U15:%x[-2,1]/%x[-1,1]
U16:%x[-1,1]/%x[0,1]
U17:%x[0,1]/%x[1,1]
U18:%x[1,1]/%x[2,1]

U20:%x[-2,1]/%x[-1,1]/%x[0,1]
U21:%x[-1,1]/%x[0,1]/%x[1,1]
U22:%x[0,1]/%x[1,1]/%x[2,1]
```

```
# Bigram
B
```

Training (encoding)

Use *crf_learn* command:

```
% crf_learn template_file train_file model_file
```

where *template_file* and *train_file* are the files you need to prepare in advance. *crf_learn* generates the trained model file in *model_file*.

crf_learn outputs the following information.

```
CRF++: Yet Another CRF Tool Kit
Copyright(C) 2005 Taku Kudo, All rights reserved.

reading training data: 100.. 200.. 300.. 400.. 500.. 600.. 700.. 800..
Done! 1.94 s

Number of sentences: 823
Number of features:  1075862
Number of thread(s): 1
Freq:                1
eta:                 0.00010
C:                   1.00000
shrinking size:      20
Algorithm:           CRF

iter=0 terr=0.99103 serr=1.00000 obj=54318.36623 diff=1.00000
iter=1 terr=0.35260 serr=0.98177 obj=44996.53537 diff=0.17161
iter=2 terr=0.35260 serr=0.98177 obj=21032.70195 diff=0.53257
iter=3 terr=0.23879 serr=0.94532 obj=13642.32067 diff=0.35138
iter=4 terr=0.15324 serr=0.88700 obj=8985.70071 diff=0.34134
iter=5 terr=0.11605 serr=0.80680 obj=7118.89846 diff=0.20775
iter=6 terr=0.09305 serr=0.72175 obj=5531.31015 diff=0.22301
iter=7 terr=0.08132 serr=0.68408 obj=4618.24644 diff=0.16507
iter=8 terr=0.06228 serr=0.59174 obj=3742.93171 diff=0.18953
```

- iter: number of iterations processed
- terr: error rate with respect to tags. (# of error tags/# of all tag)
- serr: error rate with respect to sentences. (# of error sentences/# of all sentences)
- obj: current object value. When this value converges to a fixed point, CRF++ stops the iteration.
- diff: relative difference from the previous object value.

There are 4 major parameters to control the training condition

- -a CRF-L2 or CRF-L1:
  Changing the regularization algorithm. Default setting is L2. Generally speaking, L2 performs slightly better than L1, while the number of non-zero features in L1 is drastically smaller than that in L2.
- -c float:
  With this option, you can change the hyper-parameter for the CRFs. With larger C value, CRF tends to overfit to the give training corpus. This parameter trades the balance between overfitting and underfitting. The results will significantly be influenced by this parameter. You can find an optimal value by using held-out data or more general model selection method such as cross validation.

- -f NUM:
  This parameter sets the cut-off threshold for the features. CRF++ uses the features that occurs no less than NUM times in the given training data. The default value is 1. When you apply CRF++ to large data, the number of unique features would amount to several millions. This option is useful in such cases.
- -p NUM:
  If the PC has multiple CPUs, you can make the training faster by using multi-threading. NUM is the number of threads.

Here is the example where these two parameters are used.

```
% crf_learn -f 3 -c 1.5 template_file train_file model_file
```

Since version 0.45, CRF++ supports single-best MIRA training. MIRA training is used when -a MIRA option is set.

```
% crf_learn -a MIRA template train.data model
CRF++: Yet Another CRF Tool Kit
Copyright(C) 2005 Taku Kudo, All rights reserved.

reading training data: 100.. 200.. 300.. 400.. 500.. 600.. 700.. 800..
Done! 1.92 s

Number of sentences: 823
Number of features:  1075862
Number of thread(s): 1
Freq:                1
eta:                 0.00010
C:                   1.00000
shrinking size:      20
Algorithm:           MIRA

iter=0 terr=0.11381 serr=0.74605 act=823 uact=0 obj=24.13498 kkt=28.00000
iter=1 terr=0.04710 serr=0.49818 act=823 uact=0 obj=35.42289 kkt=7.60929
iter=2 terr=0.02352 serr=0.30741 act=823 uact=0 obj=41.86775 kkt=5.74464
iter=3 terr=0.01836 serr=0.25881 act=823 uact=0 obj=47.29565 kkt=6.64895
iter=4 terr=0.01106 serr=0.17011 act=823 uact=0 obj=50.68792 kkt=3.81902
iter=5 terr=0.00610 serr=0.10085 act=823 uact=0 obj=52.58096 kkt=3.98915
iter=0 terr=0.11381 serr=0.74605 act=823 uact=0 obj=24.13498 kkt=28.00000
...
```

- iter, terr, serror: same as CRF training
- act: number of active examples in working set
- uact: number of examples whose dual parameters reach soft margin upper-bound C. 0 uact suggests that given training data was linear separable
- obj: current object value, ||w||^2
- kkt: max kkt violation value. When it gets 0.0, MIRA training finishes

There are some parameters to control the MIRA training condition

- -c float:
  Changes soft margin parameter, which is an analogue to the soft margin parameter C in Support Vector Machines. The definition is basically the same as -c option in CRF training. With larger C value, MIRA tends to overfit to the give training corpus.
- -f NUM:
  Same as CRF
- -H NUM:

Changes shrinking size. When a training sentence is not used in updating parameter vector NUM times, we can consider that the instance doesn't contribute training any more. MIRA tries to remove such instances. The process is called "shrinking". When setting smaller NUM, shrinking occurs in early stage, which drastically reduces training time. However, too small NUM is not recommended. When training finishes, MIRA tries to go through all training examples again to know whether or not all KKT conditions are really satisfied. Too small NUM would increase the chances of recheck.

Testing (decoding)

Use *crf_test* command:

```
% crf_test -m model_file test_files ...
```

where *model_file* is the file *crf_learn*creates. In the testing, you don't need to specify the template file, because the model file has the same information for the template. *test_file* is the test data you want to assign sequential tags. This file has to be written in the same format as training file.

Here is an output of *crf_test*:

```
% crf_test -m model test.data
Rockwell        NNP     B       B
International   NNP     I       I
Corp.   NNP     I       I
's      POS     B       B
Tulsa   NNP     I       I
unit    NN      I       I
..
```

The last column is given (estimated) tag. If the 3rd column is true answer tag , you can evaluate the accuracy by simply seeing the difference between the 3rd and 4th columns.

- verbose level

  The **-v** option sets verbose level. default value is 0. By increasing the level, you can have an extra information from CRF++

  - level 1
    You can also have marginal probabilities for each tag (a kind of confidece measure for each output tag) and a conditional probably for the output (confidence measure for the entire output).

    ```
    % crf_test -v1 -m model test.data| head
    # 0.478113
    Rockwell        NNP     B       B/0.992465
    International   NNP     I       I/0.979089
    Corp.   NNP     I       I/0.954883
    's      POS     B       B/0.986396
    Tulsa   NNP     I       I/0.991966
    ...
    ```

    The first line "# 0.478113" shows the conditional probably for the output. Also, each output tag has a probability represented like "B/0.992465".

  - level 2

You can also have marginal probabilities for all other candidates.

```
% crf_test -v2 -m model test.data
# 0.478113
Rockwell        NNP     B       B/0.992465      B/0.992465      I/0.00
International   NNP     I       I/0.979089      B/0.0105273     I/0.97
Corp.   NNP     I       I/0.954883      B/0.00477976    I/0.954883
's      POS     B       B/0.986396      B/0.986396      I/0.00655976
Tulsa   NNP     I       I/0.991966      B/0.00787494    I/0.991966
unit    NN      I       I/0.996169      B/0.00283111    I/0.996169
..
```

- N-best outputs

  With the **-n** option, you can obtain N-best results sorted by the conditional probability of CRF. With n-best output mode, CRF++ first gives one additional line like "# N prob", where N means that rank of the output starting from 0 and prob denotes the conditional probability for the output.

  Note that CRF++ sometimes discards enumerating N-best results if it cannot find candidates any more. This is the case when you give CRF++ a short sentence.

  CRF++ uses a combination of forward Viterbi and backward A* search. This combination yields the exact list of n-best results.

  Here is the example of the N-best results.

```
% crf_test -n 20 -m model test.data
# 0 0.478113
Rockwell        NNP     B       B
International   NNP     I       I
Corp.   NNP     I       I
's      POS     B       B
...

# 1 0.194335
Rockwell        NNP     B       B
International   NNP     I       I
```

Tips

- CRF++ uses the exactly same data format as YamCha uses. You may use both two toolkits for an input data and compare the performance between CRF and SVM
- The output of CRF++ is also compatible to CoNLL 2000 shared task. This allows us to use the perl script conlleval.pl to evaluate system outputs. This script is very useful and give us a list of F-measures for all chunk types

Case studies

In the example directories, you can find three case studies, baseNP chunking, Text Chunking, and Japanese named entity recognition, to use CRF++.

In each directory, please try the following commands

```
% crf_learn template train model
% crf_test  -m model test
```

To Do

- Support semi-Markov CRF
- Support piece-wise CRF
- Provide useful C++/C API (Currently no APIs are available)

References

- 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
- F. Sha and F. Pereira. Shallow parsing with conditional random fields, In Proc. of HLT/NAACL 2003
- NP chunking
- CoNLL 2000 shared task: Chunking

$Id: index.html,v 1.23 2003/01/06 13:11:21 taku-ku Exp $;

taku@chasen.org