# CS 284 PA2: Conditional Random Fields

Isabelle Baguio Tingzon
M.S. Computer Science
Department of Engineering
University of the Philippines – Diliman
Instructor: Sir Pros Naval
ibtingzon2@gmail.com

## 1. INTRODUCTION

In this programming assignment, we tackle the problem of intra-word syllable boundary detection (or *hyphenation*) using conditional random fields (CRF). Hyphenation is the task of learning the proper placement of hyphens within words for the purpose of syllabification. The hyphenation method applied in this paper is based on the works of Trogkanis and Elkan [2] which uses linear-chain CRFs for segmentation and sequential label prediction. Using a dataset of over 66000 hyphenated English words, I demonstrate the application of linear-chain CRF to the hyphenation problem and achieve error rates of nearly 1%.

## 2. OBJECTIVES

In this programming project, I aim to demonstrate my knowledge of conditional random fields by applying the method to the hyphenation problem. The specific objectives of this study are as follows

- To write a Python script that generates the training and test data from the given English Hyphenation Dataset. The two datasets are to be of equal sizes, and the entries of these sets are to be randomly chosen from the original dataset.
- To use the open source implementation of conditional random fields CRF++ [1] to predict the hyphenation of words taken from the English Hyphenation Dataset. This task can be broken down further into the following subtasks:
    - Converting the training and test data into CRF++ readable format
    - Designing a template that describes the features to be used in training and testing
    - Training and testing using the `crf_learn` and `crf_test` commands, respectively
- To evaluate the results obtained based on a confusion matrix (or *error matrix*) that outlines the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) results.

## 3. METHODOLOGY

The following sections will be divided as follows: Section 3.1 will provide the preliminary background necessary to understanding the problem; Section 3.2 will discuss pre-processing techniques used in the generation of training and test sets; and Section 3.3 will present an extensive discussion on the construction of the feature template in CRF++. Section 3.4. will briefly discuss training and testing in CRF++.

### 3.1.Preliminaries

In this section, I discuss in-depth the mathematical background of conditional random fields (Section 3.1.1), the hyphenation problem in the context of CRFs (Section 3.1.2), and the CRF++ toolkit (Section 3.1.3).

#### *3.1.1. Conditional Random Fields*

As discussed in [6], conditional random fields are a type of undirected probability graphical models that can be seen as a supervised learning approach to predicting class labels based on a known dataset. However, unlike most classifiers, CRFs are capable of taking the neighboring predicted labels into account. More precisely, CRFs use log-linear models to encode a distribution over label sequences given some observation sequence [2,5]. Because of this, CRFs are becoming a more popular method for natural language processing (NLP) and sequence prediction tasks.

An example is denoted as $x = (x_1, x_2, \dots, x_d)$ where the $x_i$'s are the features, and the labels $y$ are taken from a finite set $\{y_1, y_2, \dots y_C\}$. In linear chain CRFs, $x$ represents a sequence of observations and $y$ represents the labels to predicted given observations $x$. The linear-chain CRF is structured such that an edge is formed between each $y_i$ and $y_{i-1}$ as illustrated in Figure 1.

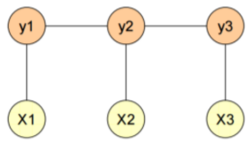

Figure 1: An example of a linear-chain CRF [3]

Feature functions of the form $f(y_{i-1}, y_i, x, i)$ define the conditional dependency of each $y_i$ on $x$. A weight $w_i$ is assigned to each feature to determine the probability that $y_i$takes a certain value and is determined through maximum likelihood estimation. The log-linear model is thus defined as

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

Where Z is the normalizing constant

$$Z(x; w) = \sum_{y' \in Y} \exp \left\{ \sum_{j=1}^{J} w_j F_j(x, y) \right\}$$

and $F_j(x, y)$'s denote the feature functions.

Training a CRF is equivalent to finding weight vector $w$ that give the best possible predictions given by

$$y^* = \arg\max_{y} p(y|x; w)$$

### 3.1.2. *Hyphenation Problem*

Based on the works of Trogkanis and Elkan [2], we define the log-linear model for the hyphenation problem as follows:

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

where $\bar{x}$ represent a sequence of length $n$ letters, $\bar{y}$ represents the corresponding of $n$ tags, and index $j$ ranges over the length of feature functions $F_j(\bar{x}, \bar{y})$, which we define as follows:

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

Each feature function $f_j$ is a binary indicator function that chooses the values for neighboring tags $y_i$ , $y_{i-1}$, and substring $\bar{x}$. Note that the substring $x$can range from length 2 to 5 covering up to 4 letters to the left and right of the current letter.

The normalizing constant Z is defined as

$$Z(\bar{x}; w) = \sum_{\bar{y}} \exp \left\{ \sum_{j=1}^{J} w_j F_j(\bar{x}, \bar{y}) \right\}$$

The weight vector $w$that gives the best possible prediction is given by

$$\bar{y}^* = \arg\max_{\bar{y}} p(\bar{y}|\bar{x}; w)$$

### 3.1.3. *CRF++ Toolkit*

The CRF++ toolkit [1] developed by Taku Kudo is an open-source implementation of conditional random fields as well as a general-purpose toolkit that can be applied to a wide range of NLP tasks, including text chunking [1] and information extraction [4]. CRF++ was written in C++ language with Standard Template Library (STL). The toolkit claims to provide a number of attractive features, including fast training based on L-BFGS (a quasi-Newton method for large scale optimization problems), less memory usage both in training and testing, and encoding and decoding in practical time [1].

In order to properly train and test data in CRF++, datasets must first be converted into CRF++ readable format (see Section 3.2.2), and feature templates must be prepared beforehand (see Section 3.3). Commands `crf_learn` and `crf_test` can then be used to train and test datasets, respectively (see Section 3.4).

## 3.2. Pre-processing Techniques

Given the English Hyphenation Dataset, we need to construct the training set and test set. The PA specifications require that the two sets be of equal length and that entries be chosen randomly from the original dataset. The resulting two datasets must then be converted into CRF++ readable format. To do accomplish these tasks, I implement a Python code called `script.py` using Python 2.7.12 with three main methods, namely `parseDataset()`,`splitDataset()`, and `convertToCRFFormat()` .

### 3.2.1. *Splitting the Dataset*

Given the English Hyphenation Dataset, I create function `parseDataset()` that takes as a parameter the file name of the data set (which in this case is `Englishdataset.txt`), stores all the entries into a list called $\texttt{data}_{\texttt{orig}}$, and returns the said list. $\texttt{data}_{\texttt{orig}}$ is then passed into `splitDataset()` wherein the list is shuffled. Afterwards, half of the entries of $\texttt{data}_{\texttt{orig}}$ are removed at random and transferred into another list called $\texttt{data}_{\texttt{train}}$, which will now constitute our training data. Finally the remaining half of the original list will be renamed to $\texttt{data}_{\texttt{test}}$ and will consistute our test set.

Because of the `shuffle()` function and randomizing function `random.choice()`, we are assured of the randomization of both training and test data. This is an important step since failure to randomize can cause high error rates.

### 3.2.2. *Converting to CRF++ Readable Format*

The training and test file must be in a particular format before they can be processed by CRF++ [1]. More specifically, the files must consist of multiple *tokens*, wherein each token consists of multiple columns. The definition of token depends on the problem at hand; in the particular case of hyphenation, each token corresponds to a letter, and a sequence of tokens represents a word. A blank line is used to identify the boundary between words. The last column corresponds to the class label to be predicted by the CRF.

In the hyphenation task, the files consist of only two columns: the first column corresponding to the letters of the word, and the second column corresponding to the class labels. The label of a letter depends on whether or not a hyphen follows it, i.e. if the letter is followed by a hyphen, its label is 1, otherwise 0.

An example output for the hyphenation 'co-ro-nas' would be:

```
c    0
o    1
r    0
o    1
n    0
a    0
s    0
```

followed by a blank space to indicate the end of the word.

In `script.py`, I construct function `convertToCRFFormat()` which takes as input a list, either $\texttt{data}_{\texttt{train}}$ or $\texttt{data}_{\texttt{test}}$ , and converts it into a CRF++ readable output file, `EngHyphen_train.txt`. or `EngHyphen_test.txt`, using the guidelines previously described.

## 3.3. Feature Template

### *3.3.1. Template Basics*

Since CRF++ is a general-purpose tool, feature templates must be constructed beforehand [1]. Each line in a template file denotes one *template* in the format:

*special_macro:%x[row, col]*

wherein *row* corresponds to the relative position from the current focusing token, and *col* corresponds to the absolute or fixed position of the column. Going back to the 'co-ro-nas' example, if the current focusing token were at the letter *n*:

```
c    0
o    1
r    0
o    1
n    0  >> Current focusing token
a    0
s    0
```

The replacements that would take place for a sample template is given in Table 1.

**Table 1. Templates and their corresponding features**

| Template | expanded feature |
|---|---|
| %x[-2,0] | r |
| %x[-2,0] | o |
| %x[0,0] | n |
| %x[1,0] | a |

In CRF++, we can prepend the special macro 'U' to describe unigram features and 'B' to describe bigram features. The bigram template is one that automatically generates a combination of the current output token and previous output token (bigram). In the hyphenation problem, this accounts for the neighboring value $y_{i-1}$. Identifiers can also be appended to in order to distinguish between relative positions.

### *3.3.2. Template Construction*

For the feature template, we need to define the sequence $\bar{x}$ as substrings of lengths 2 to 5, covering up to 4 letters to the left and right of the current letter [2]. We can enumerate all possible substrings $\bar{x}$ as follows: let $\bar{x}$ be some arbitrary sequence, $\bar{x} = x_1x_2x_3x_4x_5x_6x_7x_8x_9$, where the length of $\bar{x}$ is 9 and $x_i$'s represent letters. We can enumerate all possible substrings lengths 2 to 5 given some current focusing token. Suppose our current focusing token is at $x_5$. Then all possible substrings, given $x_5$, are: $\{x_4x_5,\ x_5x_6,\ x_3x_4x_5,\ x_4x_5x_6, ,\ x_5x_6x_7,\ x_2x_3x_4x_5,\ x_3x_4x_5x_6,\ x_4x_5x_6x_7,\ x_5x_6x_7x_8,\ x_1x_2x_3x_4x_5,\ x_2x_3x_4x_5x_6,\ x_3x_4x_5x_6x_7,\ x_4x_5x_6x_7x_8,\ x_5x_6x_7x_8x_9\}$. Thus, we get a maximum of 14 possible substrings. We construct the rules corresponding to each of the 14 possible substrings as follows:

```
#Unigram

U01:%x[-1,0]%x[0,0]

U02:x[0,0]%x[1,0]

U03:%x[-1,0]%x[0,0]%x[1,0]

U04:%x[-2,0]%x[-1,0]%x[0,0]

U05:%x[0,0]%x[1,0]%x[2,0]

U06:%x[-2,0]%x[-1,0]%x[0,0]%x[1,0]

U07:%x[-1,0]%x[0,0]%x[1,0]%x[2,0]

U08:%x[-2,0]%x[-1,0]%x[0,0]%x[1,0]%x[2,0]

U09:%x[-3,0]%x[-2,0]%x[-1,0]%x[0,0]

U10:%x[-3,0]%x[-2,0]%x[-1,0]%x[0,0]%x[1,0]

U11:%x[-1,0]%x[0,0]%x[1,0]%x[2,0]%x[3,0]

U12:%x[0,0]%x[1,0]%x[2,0]%x[3,0]

U13:%x[-4,0]%x[-3,0]%x[-2,0]%x[-1,0]%x[0,0]

U14:%x[0,0]%x[1,0]%x[2,0]%x[3,0]%x[4,0]

#Bigram

B
```

Note that one bigram template ('B') is used, meaning combinations of previous output token and current token are used as bigram features.

Going back to the 'co-ro-nas' example, given that the current focusing token is at the letter *n,* we can easily enumerate in Table 2 the corresponding expanded feature of each template as shown. Note that some templates may not apply to the 'co-ro-nas' due to the length of the possible substrings (e.g. *n* has only 2 letters following it; therefore, U11, U12, and U14 cannot be applied).

**Table 2. Templates and their corresponding features**

| template | feature |
|---|---|
| U01:%x[-1,0]%x[0,0] | on |
| U02:x[0,0]%x[1,0] | na |
| U03:%x[-1,0]%x[0,0]%x[1,0] | ona |
| U04:%x[-2,0]%x[-1,0]%x[0,0] | ron |
| U05:%x[0,0]%x[1,0]%x[2,0] | nas |
| U06:%x[-2,0]%x[-1,0]%x[0,0]%x[1,0] | rona |
| U07:%x[-1,0]%x[0,0]%x[1,0]%x[2,0] | onas |
| U08:%x[-2,0]%x[-1,0]%x[0,0]%x[1,0]%x[2,0] | ronas |
| U09:%x[-3,0]%x[-2,0]%x[-1,0]%x[0,0] | oron |
| U10:%x[-3,0]%x[-2,0]%x[-1,0]%x[0,0]%x[1,0] | orona |
| U13:%x[-4,0]%x[-3,0]%x[-2,0]%x[-1,0]%x[0,0] | coron |

## 3.4. Training and Testing

### 3.4.1. *Training (Encoding)*

For training, I run the following command in command prompt:

```
% crf_learn EngHyphen_template
EngHyphen_train.txt model
```

The `crf_learn` command takes as input the template and training data and produces a model file which will be used later on for testing. For training, I adopted the default parameter setting of CRF++ as was done in [2] by Trogkanis and Elkan. Similar to their study, no development set or tuning set was needed for this project.

### 1.1.1. *Testing (Decoding)*

For testing, I run the following command in the command prompt:

```
% crf_test -m model
EngHyphen_test.txt
>EngHyphen_results.txt
```

The `crf_test` command takes as input the model file created during the training phase as well as the testing data. It then produces results which are saved into the file `EngHyphen_results.txt`.

## 4. Experimental Results

In order to measure accuracy, I write Python script `eval.py` which calculates the confusion matrix based on the results obtained in `EngHyphen_results.txt`, and from this I compute the error rates. As was done in the study by Trogkanis and Elkan, I report both letter-level and word-level error rates. Letter-level error rates are defined as "fraction of letters for which the [CRF] predicts incorrectly whether or not a hyphen is legal after this letter" [2]. Meanwhile, word-level error rates are defined as the "fraction of the words on which the [CRF] makes at least one mistake". Table 3 describes the measures of accuracy based on the works of Trogkanis and Elkan [2] .

**Table 3. Measures of Accuracy [2]**

| **Abbr** | **Name** | **Description** |
|---|---|---|
| TP | true positives | no. of hyphens correctly predicted |
| TN | true negative | no. of hyphens correctly not predicted |
| FP | false positives | no. of hyphens incorrectly predicted |
| FN | false negatives | no. of hyphens failed to be predicted |
| owe | overall word-level errors | No. of words with at least one FP or FN |
| swe | serious word-level errors | No. of words with at least one FP |
| ower | overall word-level error rate | owe/total no. of words |
| swer | serious word-level error rate | swe/total no. of words |
| oler | overall letter-level error rate | (FP+FN) / (TP+TN+FP+FN) |
| sler | serious letter-level error rate | (FP) / (TP+TN+FP+FN) |

Table 4 shows the confusion matrix of the CRF on the English Hyphenation Dataset.

**Table 4. Confusion Matrix (Letter-level)**

| n= 275454 | **Predicted y=1** | **Predicted y=0** |
|---|---|---|
| **Actual class y=1** | TP= 52561 (0.1908) | FN= 3002 (0.0109) |
| **Actual class y=0** | FP= 2774 (0.0101) | TN= 217117 (0.7882) |

**Table 5. Performance of CRF on the English Hyphenation Dataset**

| **Total No. of words:** | 33001 |
|---|---|
| **owe** | 3161 |
| **swe** | 2596 |
| **ower** | 0.096 |
| **swer** | 0.079 |
| **oler** | 0.021 |
| **sler** | 0.010 |

## 5. Analysis and Discussion of Results

We first consider the confusion matrix given by Table 4. We learn that 52561 letters (about 19%) were correctly predicted to have a hyphen following them (we call these the true positives). Meanwhile, 217117 letters (79%) were correctly predicted to *not* have a hyphen following them (true negatives). Another way to interpret the results is based on accuracy where

$$\text{Accuracy} = \frac{(TP+TN)}{(TP+TN+FP+FN)}$$

In which case, we get an overall accuracy of 98%.

We also learn that 2774 letters (1%) were incorrectly predicted to have a hyphen following them (false positives). Meanwhile, 3002 letters (1.1%) should have a hyphen following them but the CRF was unable to predict it (false negatives). We use the FP's and FN's to obtain the overall letter-level error rates and serious letter-level error rates.

Based on the results in Table 5, we learn that CRF yields an overall letter-level error rate of 2% and a serious letter-level error rate of 1%. This means that for at least 2% of the time, a letter will either have a hyphen incorrectly placed after it or fail to have a hyphen placed when one is expected. And for 1% of the time, a hyphen will be incorrectly placed after a letter. Having a low serious letter-level error rate implies that the CRF is unlikely to misplace a hyphen.

Meanwhile, the overall word-level error rate is roughly 10% and the serious word-level error rate is 8%. That is, for at least 10% of the time, the CRF will either incorrectly insert a hyphen in a word or fail to place a hyphen in a word when there should be one. And for at least 8% of the time, the CRF will place hyphen within one of the words where a hyphen is not allowed.

We expect ower and swer to be higher than oler and sler since it only takes one hyphenation error within the word for the word itself to count as erroneous. Nonetheless, we were still able to achieve word-level error rates of less than 10% which are already considerably small. Meanwhile, we achieved very low error rates for oler and sler amounting to 2% and 1%, respectively. Overall, we were able to achieve very low error rates, proving the effectiveness of CRFs for sequential label prediction.

## 6. Conclusion

Conditional random fields are useful for predicting class labels when neighboring class labels must be taken into account. This makes the CRF a popular method for natural language processing and sequential prediction tasks. In this programming assignment, we consider the application of CRF to the problem of hyphenation, or intra-word syllable boundary detection. The results show that we can obtain error rates as low as 1%. Thus, in this programming assignment, I have achieved my objectives (Section 2) and have successfully demonstrated that CRF is indeed a valuable method for the hyphenation problem.

## REFERENCES

[1] Tako, Kudo. 2005. "CRF++: Yet Another CRF toolkit." *Taku910.github.io*. 14 Mar. 2015. Web. 22 Oct. 2016. https://taku910.github.io/crfpp/#templ

[2] Trogkanis, Nikolaos, and Charles Elkan. "Conditional random fields for word hyphenation." *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2010.

[3] Maskey, S. "Statistical Methods for NLP Maximum Entropy Markov Models, Conditional Random Fields" *Cs.columbia.edu*. 27 Mar. 2010. Web. 22 Oct. 2016.http://www.cs.columbia.edu/~smaskey/CS6998/slides/statnlp_week10.pdf

[4] Weld, D. and Kiddon, C. "CSE454 Project Description." *Courses.cs.washington.edu*. n.d. Web. 22 Oct. 2016.https://courses.cs.washington.edu/courses/cse454/09sp/crf.html

[5] Wallach, Hanna M. "Conditional random fields: An introduction." (2004).

[6] Naval, P. “CS284 Programming Assignment 2 Intra-word Syllable Boundary Detection Using Conditional Random Fields.” (2016).