CS4740 Project 3: Sequence Tagging

Kang-Li S. Cheng

Contents

1 Introduction

In this sequence tagging project, the task is to identify relevant information in a text and tag it with the appropriate label. In particular, the focus is the **Named Entity Recognition (NER)**, in which the semantic property or category of tokens corresponding to a person, location, organization, etc. is annotated.

For this project, I have chosen to learn and experiment with the Stanford Named Entity Recognizer (Stanford NER) package [1]. The main reasons for this choice are as follows:

- Stanford NER is based on a general implementation of Continuous Random Fields (CRF) sequence models;
- a well-documented API (Javadoc) for the package;

- continuous, active developments of the packages (CoreLP, NER, Parser, POSTagger, etc.) since its first release in 2006;
- the GNU public license (including open source codes) to non-commercial users.

1.1 Continuous Random Fields (CRF)

In a *sequence tagging problem* of natural language texts, the task is to map an input sequence $\mathbf{x} = x_1x_2\dots x_n$ to a tag (label) sequence consisting of $\mathbf{y} = y_1y_2\dots y_n$. In the context of NER, the hidden variables are the tags, and the observed variables are the sequences of words. The hidden Markov Model (HMM) assumes strong independence among the observed variables $\mathbf{x}$ in order to simplify the complexity of the tagging problem. The joint probability $P(\boldsymbol{y}, \boldsymbol{x})$ is modeled as

$$P(\boldsymbol{y}, \boldsymbol{x}) = \prod_{i=1}^{n} P(y_i|y_{i-1})P(x_i|y_i), \quad \text{with initial } P(y_1|y_0) = P(y_1). \tag{1}$$

To find the tag sequence of a given sentence, HMM maximizes the joint probability.

$$\operatorname*{argmax}_{y_i \in \boldsymbol{y}} \prod_{i=1}^{n} P(y_i|y_{i-1})P(x_i|y_i). \tag{2}$$

In reality the strong independence assumption among x_i's impairs the accuracy of HMMs. In contrast, the continuous random fields (CRFs) make no assumptions on the dependencies among the observed variables, but still assumes the the past labels are independent from future labels given the present label. A CRF model is built by specifying feature functions $\boldsymbol{f} = (f_1, f_2, \dots, f_n)$ and their corresponding weights λ_i:

$$P(\boldsymbol{y}|\boldsymbol{x}) = \frac{1}{Z} \exp\left[\sum_{j=1}^{m}\sum_{i=1}^{n} \lambda_j f_j(\boldsymbol{x}, i, y_i, y_{i-1})\right], \tag{3}$$

where Z is a normalizing constant. In training the CRF models, the model parameters are learned by maximizing $\log[P(\boldsymbol{y}|\boldsymbol{x})]$.

Succinctly, HMMs model the joint probability $P(\boldsymbol{y}, \boldsymbol{x})$, whereas CRFs model the conditional probability $P(\boldsymbol{y}|\boldsymbol{x})$. The CRF can incorporate more complex and realistic features of the observed sequence when computing the probability of a label y_i, for example, consideration of the non-local effects for named entity tagging. The superior performance of the CRFs comes at a price: training a CRF model requires significantly more computation then training an HMM.

1.2 Outline of this Report

The Stanford Named Entity Recognizer (NER) (as a part of the natural language processing (NLP) toolkits) is introduced with Java codes. In order to utilize the Stanford NER package, we need to preprocess the training test datasets. The preprocessing of the datasets is described and the codes are listed. Design of the baseline system is described. Comparison of the project NER model with the baseline system is presented. An error analysis of the project NER is included. And finally, results on the test data for Kaggle competition are summarized and attached.

2 Project Implementation Details

2.1 Stanford NER and NLP Toolkits

Stanford NER is a component of the Stanford CoreNLP toolkits, based on a Java annotation pipeline architecture which is is shown in Figure (1).

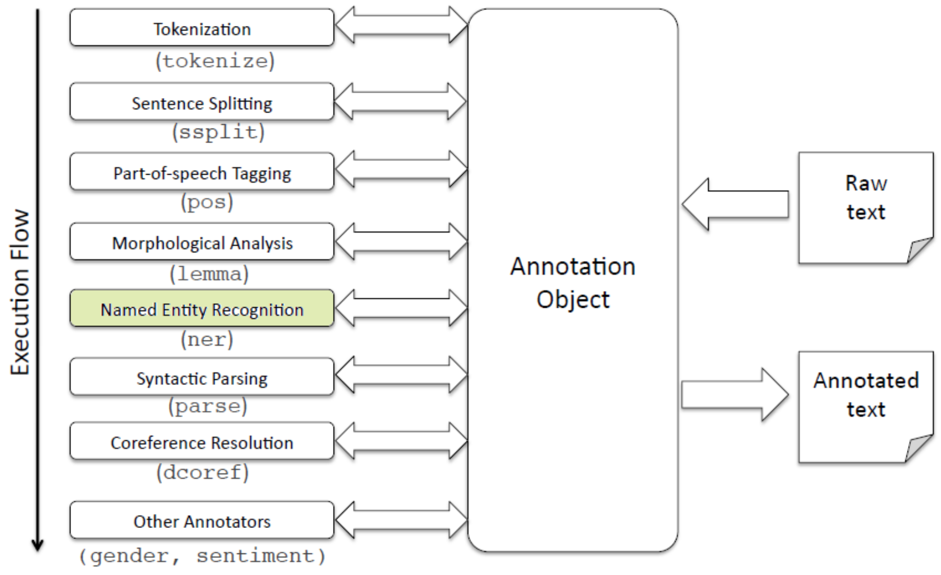

Figure 1: Overall Stanford CoreNLP system architecture. NER is shown as a component in the framework. [2]

The Stanford CoreNLP supports relatively simple Java API and comprehensive command line user interface. Consequently users can get started with package very quickly.

Before we present the command-line and programming interfaces, we need to mention that the current release (version 3.6.0 released on December 9, 2015) requires Java 1.8 on the system. For the Unix/Linux operating systems, after downloading and unzipping the packages, the CLASSPATH environment variable must be set up correctly in order to use the package:

1. Export the package root path such as
   `export nerRoot=/home/klcheng/stanford-ner`

2. Environment variable `CLASSPATH` must be exported as follows:
   `export CLASSPATH=".:$nerRoot/stanford-ner.jar:$nerRoot/lib/*"`

Usually we put such shell commands in the .bashrc file in the user's home directory.

2.2 Stanford NER Command-Line User Interface

New users can start using Stanford right out of the box by testing the command line user interface.

A four-entity class model based on CoNLL2003 training corpus is included in the Stanford NER distribution (*english.conll.4class.distsim.crf.ser.gz*). To generate the named tags with this built-in classifier on an input file called sample.txt, we type:

```
java edu.stanford.nlp.ie.crf.CRFClassifier
-loadClassifier $nerRoot/classifiers/english.conll.4class.distsim.crf.ser.gz
-textFile sample.txt
```

To train a CRF model based on a training corpus called, say, cornell_Train_proj3_2.tsv, we first put the properties of the CRF model in a properties file (say, cornell.prop), then enter the command:

```
java edu.stanford.nlp.ie.crf.CRFClassifier -prop cornell.prop
```

```
-readerAndWriter edu.stanford.nlp.sequences.CoNLLDocumentReaderAndWriter
-entitySubclassification iob1
```

The above command will generate a Stanford NER model based on the options specified in the properties file, *cornell.prop*, which is included in the Appendices. Note the encoding scheme IOB1 (iob1) for `-entitySubclassification` must be correctly specified to match the actual encoding in the training corpus.

2.3 Stanford NER Java Programming API

The complete packages of Stanford NER and CoreNLP are complicated due to the packages' comprehensive capabilities in natural language processing. Here in this project we only focus on how to use Stanford NER for Named Entity Recognition.

The webpage for Stanford NER provides a very good introduction to the package, an FAQ and Javadoc.[1] Many of the functions of the NER can be accessed directly through command line commands. For Java programming access, Stanford NER provides a demonstration Java code called NERDemo.java which is very helpful as a basic introduction.

Java programming for Stanford NER can be very simple. A sample code for generating the final Kaggle competition, called Kaggle5.java, is included in the Appendices. In the main method of the code, we primiarily deal with the `CRFClassifier` class. (But knowing which methods to use takes efforts and time to browse both the Javadoc API and the source codes.) In the main method, we load the classifier by getClassifier(), make a DocumentReaderAndWriter object and call the classifyAndWriteAnswers() method to process the input and write out the text tokens with answers (tags).

The command-line commands for compiling and running NERKaggle5.java with properties file (Kaggle.prop) on the command line are:

```
$javac NERKaggle5.java
$java NERKaggle5 Kaggle.prop
```

The annotated output will be printed on the screen. Kaggle.prop is a properties file in which the test file name and the NER model name are given:

```
#location of the training file in tab-separated two-column format
testFile = cornell_test_token.tsv

#location and name of the NER model (classifier):
loadClassifier=ner-model-cornell-klc.ser.gz
```

In the prop file, any lines starting with # is a comment line. As the comments in the code indicate, the testFile must be a tab-separated file with two columns. The first column contains tokens, the second column usually the tag (but can be something else, in our case, we put the line/token number in the second column). Each line contains only one token and the corresponding tag. The name of the model (the value of `loadClassifier`) is loaded by the Stanford NER, then the test corpus is processed to produce the Kaggle competition results. (As a reminder, the NER model and the test data file in Kaggle.prop must be available.) By default, the classifier writes out the annotated results to the standard output, we can use the Unix shell re-direction to save the output to a text file.

The Stanford NER Java API can be accessed online at `http://nlp.stanford.edu/nlp/javadoc/javanlp`.

2.4 Pre-processing of the Datasets

2.4.1 Training Datasets

The training dataset from CS4740 is organized by sentences. Each sentence is associated with 3 lines, where the first line contains the tokens, the second line the corresponding part-of-speech (POS) tags,

[1]`http://nlp.stanford.edu/software/CRF-NER.shtml`

and the third line the correct labels (in the IOB1 encoding). For instance:

```
Werder Bremen 1 ( Schulz 31st ) Borussia Moenchengladbach 0 .
NNP NNP CD ( NNP NNP ) NNP NNP CD .
B-ORG I-ORG O O B-PER O O B-ORG I-ORG O O
```

The Stanford NER package requires a two-column, tab-separated data file format as follows:

```
Werder B-ORG
Bremen I-ORG
1 O
( O
Schulz B-PER
31st O
) O
Borussia B-ORG
Moenchengladbach I-ORG
0 O
. O
```

I use MATLAB (the code is included in the Appendices) to convert the original training dataset to the format accepted by Stanford NER. (The POS tags are left out because they are not used for this project.)

2.4.2 The Test Dataset for Kaggle Competition

In a very similar way, the *test corpus* provided for CS4740 is also converted to a 2 column format, each line contains only one token in the first column, and the token number (starting from 0) in the second column. The MATLAB code can also be found in the Appendices.

3 Baseline System and Performance Comparison

3.1 Design of the Baseline System

In order to evaluate the performance of the NER model for this project, a baseline system is needed. Since the training and testing files we obtained in CS4740 have many features of news articles (by browsing and sampling the contents of the files), it is appropriate to find similar training corpora for building a baseline system.

The CoNLL-2003 training and development files [3] are freely available from online sources. I use a subset of CoNLL-2003 to build a baseline.

1. CoNLL-2003 training file (`eng.train`) use the same IOB1 encoding scheme. This is very convenient because the project training file uses IOB1 also.

2. `eng.train` contains empty lines and -DOCSTART - - header, we need to remove them and pre-process the file in order that it meets the Stanford NER training file format requirement. The pre-processed dataset (one token per line) is then split into two files of roughly equal number of tokens, they are called `conll2003_train_sep1.tsv` (line 1 to 100,000, one token per line) and `conll2003_train_sep2.tsv` (line 100,001 to 203,621), respectively.

   The baseline NER system is trained on `conll2003_train_sep1.tsv`, whereas the project model is trained on a subset (also of 100,000 tokens) of the class training data.

3. There are a good number of model parameters that can be changed in training a Stanford NER model. The settings used by the baseline system are listed below:

```
noMidNGrams=true
useDisjunctive=true
maxNGramLeng=6       # 6-Gram
usePrev=true
useNext=true
useSequences=true
usePrevSequences=true
maxLeft=1
```

4. The NER model trained on a subset (the first 100,000 tokens, so that the number of tokens is comparable to that in the baseline training file). The project NER model is simply referred to as ProjectNER0. In evaulating the performance, we feed the remaining subset of the CoNLL-2003 training file `conll2003_train_sep2.tsv` to the baseline and to ProjectNER0 model. The results are summarized in the following table for precision, recall and F_1 score evaluated at the entity level:

Model	Total Precision	Total Recall	Toal F_1 score
Baseline	0.8375	0.8136	0.8253
ProjectNER0	0.9065	0.8892	0.8978

The main difference between ProjectNER0 and the Baseline is the the fine tuning of the CRF training parameters. ProjectNER has a gain of about 7% in all three catergories over the baseline, which is quite good. However, computing time required for training ProjectNER0 model is more than twice longer than the Baseline.)

3.2 Results from the Baseline versus the ProjectNER0 Models

The Baseline NER results on `conll2003_train_sep2.tsv`:

```
Entity     P       R       F1    TP    FP    FN
   LOC 0.8716 0.8794 0.8755 3033   447   416
  MISC 0.8613 0.7401 0.7961 1267   204   445
   ORG 0.7569 0.7127 0.7341 1955   628   788
   PER 0.8554 0.8727 0.8640 2591   438   378
Totals 0.8375 0.8136 0.8253 8846  1717  2027
```

The ProjectNER0 results on `conll2003_train_sep2.tsv`:

```
Entity     P        R        F1       TP     FP     FN
   LOC  0.9303   0.9290   0.9296   3204   240    245
  MISC  0.9101   0.8400   0.8736   1438   142    274
   ORG  0.8623   0.8400   0.8510   2304   368    439
   PER  0.9168   0.9168   0.9168   2722   247    247
Totals  0.9065   0.8892   0.8978   9668   997    1205
```

3.3 ProjectNER Model Based on the Full Training Corpus

Finally, we add the second half of the training data (line 100001 to the end of the training corpus) to train the model. We call it ProjectNER: `ner-model-cornell-project-klc.ser.gz`. This is the NER model we use to tag the test data for entering the Kaggle competition.

3.4 Additional Experiments with ProjectNER (the Full Model)

The Stanford Natural Language Processing Group published some NER performance results on their website at `http://nlp.stanford.edu/projects/project-ner.shtml`. Therefore it is interesting for us to compare the performance of ProjectNER model with those published results. (In the table, distributional similarity ("distsim") is an available feature in Stanford NER offering additional gain in performance, but not available in our ProjectNER because it requires additional "lexicon cluster"during the training of the model.)

Model	Precision	Recall	F_1	distsim
CoNLL2003-testa	91.64%	90.93%	91.28%	No
ProjectNER-testa	88.92%	86.50%	87.69%	No
CoNLL2003-testb	88.21%	87.68%	87.94%	Yes
ProjectNER-testb	81.32%	79.20%	80.25%	No

Table 1: Named Entity Recognition Results (at the entity level) by Stanford NER the CoNLL2003 Model and by the ProjectNER model. Testa and Testb stand for the CoNLL2003 English training corpora.

For the CoNLL2003 Testa corpus without distributional similarity, the performance of ProjectNER is quite outstanding in comparison with the state-of-the-art Stanford CRF-NER model trained by a mixture of CoNLL, MUC-6, MUC-7 and ACE named entity corpora [2], whereas ProjectNER model is only trained by the training corpus for the class project. For the CoNLL testb corpus, Stanford NER did not publish the model without distributional similarity, therefore the comparison is not as fair as for Testa.

In summary, the performance of ProjectNER achieves quite an impressive performance as demonstrated by the experimental results.

4 Project Extension: N-Grams and Model Performance

Stanford NER allows us to change the orders of N-Gram when training the model (by changing the value of `maxNGramLeng` in the properties file). In this extension to the main tasks, besides the 6-Gram model we have built, we also build 5-gram, 4-gram, 3-gram and 2-gram models by using the same training corpus (`conll2003_train_sep1.tsv`). Then we evaluate the performance of models by using the same test corpus (`conll2003_train_sep2.tsv`). Each model's training time for the N-Gram experiments takes between 25-40 minutes to complete. Once we have the models, we run the tests to get the measures including precision, recall, and F_1, etc.[3] The results of the F_1 scores are listed in Table (2) for comparisons. As expected, the 6-Gram has the best performance in all four named entities. Since the longest entities occur least likely in personal names (PER), the gain in F_1 scores for PER is the smallest across the N-Gram space, whereas the gains in F_1 scores are strongest in MISC, followed by LOC and ORG.

Another way to look at the relative performance is to compare the false positives (FPs) generated by the NER. Figure (2) shows the total FPs generated by the the N-Grams (the smaller, the better). In our test, 2-Gram produces a total of 1137 FPs, whereas 6-Gram generates only 997 FPs. The reduction

[2] `http://nlp.stanford.edu/software/CRF-NER.shtml`

[3] Stanford NER generates these entity-level performance measures when we run the NER in the test mode.

is around 12% in total FPs from 2-Gram to 6-Gram. (Note that the total number of tokens is 103,267 in the test corpus.)

For general purpose NER, the differences in model performance (based on total F_1 scores) is small (less than 2%). But for more specialized applications like the names of miscellaneous entities, locations and organizations, the NER performance could be more significant due to the orders of N-Gram.

Model	2-Gram	3-Gram	4-Gram	5-Gram	6-Gram
LOC	0.9199	0.9217	0.9234	0.9291	0.9303
MISC	0.8427	0.8485	0.8633	0.8694	0.8736
ORG	0.8395	0.8448	0.8480	0.8508	0.8510
PER	0.9050	0.9069	0.9135	0.9157	0.9168
Total	0.8838	0.8871	0.8930	0.8963	0.8978

Table 2: Comparison of F_1 scores (at the entity level) by varing the orders of N-Gram for the NER model of this Project.

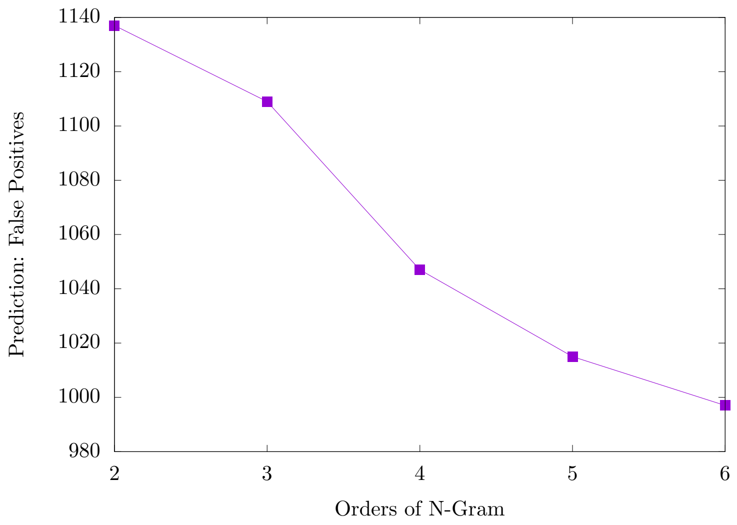

Figure 2: Number of False Positives predicted by the NER model for different orders of N-Gram.

5 Appendices: Listings of the Codes and Files in Project 3

5.1 NERKraggle5.java for Generating the Kaggle Result

The following Java code (with comments) illustrates a useful Java code for tagging the test data:

```
import edu.stanford.nlp.ie.AbstractSequenceClassifier;
import edu.stanford.nlp.ie.crf.*;
import edu.stanford.nlp.io.IOUtils;
import edu.stanford.nlp.ling.CoreLabel;
import edu.stanford.nlp.ling.CoreAnnotations;
import edu.stanford.nlp.sequences.SeqClassifierFlags;
import edu.stanford.nlp.objectbank.*;
//import edu.stanford.nlp.sequences.DocumentReaderAndWriter;
import edu.stanford.nlp.sequences.*;
import edu.stanford.nlp.util.Triple;

import edu.stanford.nlp.util.StringUtils;
import java.util.Properties;

import java.util.List;
import java.util.ArrayList;

/** This is a java code to use the serialized NER classifier
 *  trained by the given data and to produce the output
 *  for Kaggle competition (with the given test dataset).
 *  The output is then processed separately with a Matlab code
 *  to generate a CSV file with the following format:
 *
 *  Type,        Prediction
 *  PER,a-b c-d e-f ...
 *  LOC,a1-b1 c1-d1 ...
 *  ORG,a2-b2 c2-d2 e3-f3 ...
 *  MISC, a3-b3 c3-d3 ...
 *
 *  Kang-Li S. Cheng
 *  For Project3, CS4740 Introduction to NLP,
 *  Cornell University
 *
 * ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 */
/*
     Usage:
     java NERKaggle5 property-file

*/
public class NERKaggle5 {
    public static void main( String[] args ) throws Exception {
        //the NER property file is loaded from the command-line
        // Get the testFile and loadPath from the prop file:

        String testFile = props.getProperty("testFile");
        String loadPath = props.getProperty("loadClassifier");
        //  System.out.println( testFile + '\t'+ loadPath);

        //  Load the NER model by getClassifier:
        CRFClassifier<CoreLabel> classifier =
        CRFClassifier.getClassifier(loadPath);

        //  CRFClassifier<CoreLabel> classifier =
        //  CRFClassifier.getClassifier(loadPath, props);

        // An instance of the DocumentReaderAndWriter is needed:
        DocumentReaderAndWriter<CoreLabel> readerAndWriter =
                                 classifier.makeReaderAndWriter();

        // DocumentReaderAndWriter<CoreLabel> readerAndWriter =
        //                     classifier.defaultReaderAndWriter();
        // For the Classifier to recognize the input file as a
        // tab separated file: first column  the tokens
        // second column the line numbers, we have to call
        // classifyAndWriteAnswers method:
        //
        // *** Calling any other CRFClassifier methods will
        // *** result in classifying
        // *** every single token in the test file, which
        // *** would NOT be what we want.
        //
        classifier.classifyAndWriteAnswers(testFile,
                                           readerAndWriter, false);
    }

    // the third argument of classifyAndWriteAnswers()
        // prints out the score of the test run, so it
        // is not relevant for Kaggle test,
        // but useful in giving the performance scores for
        // training datasets with gold answers.)
}
```

5.2 MATLAB code for Pre-processing the Training Corpus

```
%
% convertTrain.m is used to convert the given train.txt file
% to the Stanford NER training data format: Each line is a single word
% and its label separated by a TAB
%
%  word<TAB>Label
%
% Reading the train.txt data file by textscan. In the file, each "sentence"
% is associated with three lines, as an example below. Each word
% is delimited by a TAB.
%
%  Example sentence in train.txt:
%
% The following bond was announced by lead manager Toronto Dominion     .
% DT  VBG       NN   VBD  VBN       IN  NN   NN      NNP     NNP        .
% O   O         O    O    O         O   O    O       B-PER   I-PER      O
%
% Makes sure numLines (the number of lines) is exactly correct:
numLines=42000;
% every sentence is associated with 3 lines:
numSentence=numLines/3;

midstop=numSentence/2+1;
fileId = fopen('train.txt');
fileOUT = fopen('stanford_2nd_half.txt', 'w');
for i=midstop:numSentence
    % each time textscan reads three lines:
    C = textscan(fileId,'%s',3,'Delimiter', '\n');
    % C is a cell, C{1}{1} is the first line, C{1}{3} the third line
    % for each "sentence".
    for k=1:3
        % actually we won't use the 2nd line tags, but still process it
        S{k} = strsplit(C{1}{k});
    end
    % slen is the length (number of words) of current line (3-line group):
    slen = length(S{1});
    for j=1:slen
        %str = [ S{1}(j) char(9) S{3}(j)]
        % first row is the word
        word=char(S{1}(j));
        % third row is the label (PER LOC ORG MISC)
        label=char(S{3}(j));
        fprintf(fileOUT, '%s\t%s\n', word, label);
    end

end

ST=fclose(fileId);
ST2=fclose(fileOUT);
```

5.3 MATLAB code for Pre-processsing the Test Corpus

```
%
% read the NER test file and extract only the text from the
% file, then write out each token in one line to an output file.
% Makes sure numLines (the number of lines) is exactly correct:
numLines=9000;
% every sentence is associated with 3 lines:
numSentence=numLines/3;
fileId = fopen('cornell_test.txt');
fileOUT = fopen('cornell_token.txt', 'w');
for i=1:numSentence
    % each time textscan reads three lines:
    C = textscan(fileId,'%s',3,'Delimiter', '\n');
    % C is a cell, C{1}{1} is the first line, C{1}{3} the third line
    % for each "sentence".
    %
    % S{1} = strsplit(C{1}{1});
    for k=1:3
        % actually we won't use the 2nd line tags, but still process it
     S{k} = strsplit(C{1}{k});
    end
    % slen is the length (number of words) of current line (3-line group):
    slen = length(S{1});
    for j=1:slen
        %str = [ S{1}(j) char(9) S{3}(j)]
        % first row is the word
        word=char(S{1}(j));
        lineLabel=char(S{3}(j));
        fprintf(fileOUT, '%s\t%s\n', word,lineLabel);
    end

end

ST=fclose(fileId);
ST2=fclose(fileOUT);
```

5.4 MATLAB code for Post-processing the Kaggle Competition Result

```
function C = Kaggle2
% The code post-process the Kaggle5.dat from Stanford NER:
% Kaggle5.dat has the three column format like:
%LONDON                9          I-LOC
%1996-08-25            10         O
%South                 11         I-MISC
%African               12         I-MISC
%
% To generate a CSV file with the following format in result.csv
%
%  Type,Prediction
%  PER,a-b c-d e-f ...
%  LOC,a1-b1 c1-d1 ...
%  ORG,a2-b2 c2-d2 e3-f3 ...
%  MISC,a3-b3 c3-d3 ...

fileID=fopen('Kaggle5.dat');
fileOUT=fopen('result.csv', 'w');
C=textscan(fileID, '%s %f %s');
ST=fclose(fileID);
%
% C{1} is the first column (tokens), C{2} the token number, C{3} the tag.
%
nData=length(C{1});
NER=[C{3}]; %NER is an array

% NER contains one of the tags: 'I-PER', 'I-LOC', 'I-ORG', 'I-MISC' or 'O'

% Use the power of find() function in Matlab: Ind is an array
% containing the line number of the match, then we minus one for the
% token (line no.) starts at 0:

fprintf(fileOUT,'Type,Prediction\n');
%-------------
Ind=find(strcmp(NER,'I-PER'));
Ind=Ind-1; % Test data's tokens starts from 0, so we need to minus 1.
fprintf(fileOUT,'PER,');
disp('PER,')
prediction(Ind, fileOUT);

Ind=find(strcmp(NER,'I-LOC'));
Ind=Ind-1;
fprintf(fileOUT,'LOC,');
disp('LOC,')
prediction(Ind,fileOUT);

Ind=find(strcmp(NER,'I-ORG'));
Ind=Ind-1;
fprintf(fileOUT,'ORG,');
disp('ORG,')
prediction(Ind,fileOUT);

Ind=find(strcmp(NER,'I-MISC'));
Ind=Ind-1;
fprintf(fileOUT,'MISC,');
```

```
disp('MISC, ')
prediction(Ind,fileOUT);

ST2=fclose(fileOUT);
%
% function prediction prints out the ranges of PER, LOC, ORG, MISC
% by using the respective Ind array (which contains the
% line number, i.e., the token position) in the test data file.
%
function prediction(Ind,fileOUT)
consecutive=0;
num=length(Ind);

for i=1:num
   if consecutive==0
       tag_start=Ind(i);
   end

   if (i< num)
       diff=Ind(i+1)-Ind(i);
   end

   if (diff > 1 || i==num)
       tag_end=Ind(i);
       fprintf(fileOUT, '%d-%d_', tag_start,tag_end);
%       X=sprintf('%d-%d ', tag_start, tag_end);
%       disp(X)
       consecutive=0;
       diff=0;
   elseif (diff == 1)
        consecutive=1;
   end
end
fprintf(fileOUT,'\n');
% end of function prediction
```

5.5 Properties File for Training the ProjectNER Model

```
#Content of "cornell-project.prop"
#location of the training file
#trainFile=cornell_Train_proj3_1.tsv
trainFileList = cornell_Train_proj3_1.tsv,cornell_Train_proj3_2.tsv
#location where you would like to save (serialize to) your
#classifier; adding .gz at the end automatically gzips the file,
#making it faster and smaller
serializeTo = ner-model-cornell-project-klc.ser.gz

#structure of your training file; this tells the classifier
#that the word is in column 0 and the correct answer is in
#column 1
map = word=0,answer=1

#these are the features we'd like to train with
#some are discussed below, the rest can be
#understood by looking at NERFeatureFactory
useClassFeature=true
useWord=true
```

```
useNGrams=true
#no ngrams will be included that do not contain either the
#beginning or end of the word
noMidNGrams=true
useDisjunctive=true
maxNGramLeng=6
usePrev=true
useNext=true
useSequences=true
usePrevSequences=true
maxLeft=1
#the next 4 deal with word shape features
useTypeSeqs=true
useTypeSeqs2=true
useTypeySequences=true
useOccurrencePatterns=true
useLastRealWord=true
useNextRealWord=true
normalize=true
wordShape=dan2useLC
disjunctionWidth=4
type=crf

readerAndWriter=edu.stanford.nlp.sequences.ColumnDocumentReaderAndWriter

useObservedSequencesOnly=true

sigma = 20
useQN = true
QNsize = 25

# makes it go faster
featureDiffThresh=0.05
```

References

[1] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 363–370, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[2] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[3] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003 - Volume 4*, CONLL '03, pages 142–147, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.