# CS 585 Final Project Report

**Team Member: Xiao Chen, Yiming Peng**

**Date: 12/12/14**

# Abstract

In this project, we implement a CRFSuite-based system to make NER tagging for Twitter, which can recognize spans of input tweets tokens that correspond to a name, and make a BIO representation. The major features used in system includes Lexical/word form, Character affixes, Shape features, Positional offset versions of above and First character upper or lower check. To enrich the features extractor, after exploring different types of features and analyses, three external lexical resources for additional features are finally decided to be added, they are Freebase, POS tagger, and Gazetteer. For evaluation, we use the evaluation script provided to evaluate the labeling accuracy locally. After adding features and learning on initial training set multiple times, we found Freebase improved the accuracy in some degree, but Gazetteer never worked and we got a 0.48 F-score on development set with final modified system.

**Key Words:** NER tagging, Twitter, feature extractor, CRFSuite-based

# 1. Description Implementation

The major classes features composed to be the basic CRFSuite-based system, which include features: Lexical/word form, Character affixes, Shape features, Positional offset versions of above and First character upper or lower check. The detail descriptions for these features and implementation approaches have been listed as below:

Table 1.1

| **Lexical/word form** |
|---|
| ■ This feature mainly expresses the relationship between lexical and word form change of a word. For example, lots of words in sentence head with their first letter capital in fact have the same lexical as the words with all lower cased characters in the body of sentence. The situation that a word with same lexical appears with different word forms in other place should not be ignored.<br>■ Approach to use: Change all tokens to their lower case when adding the feature. |

Table 1.2

| **Character affixes** |
|---|
| ■ This feature mainly expresses the relationships among the first 1, 2, and 3 characters or the last 1, 2, 3 characters in the word.<br>■ Approach to use: First check the length of the aim word, there are three cases: Length = 1, Length = 2, Length >= 3,<br>For each case, add feature like this “character[i] = %s%s%s”% (char 1, char 2, char 3). Rather than adding three characters separately, instead add near substrings containing three/two/one characters. Especially, here for the word whose length >=3, we just consider the first 1, 2, 3 characters and the last 1, 2, 3 characters in the word. |

Table 1.3

| **Shape features** |
|---|
| ■ This feature expresses whether the word's is uppercase followed by lowercase characters; or is all uppercase, or is all lowercase. Many variants are possible. <br> ■ Approach to use: <br> 1. Map characters to a few classes and eliminate repetition. Simply write a method with argument word and return a string indicating the shape feature for the given word. <br> 2. The method can detect the shape in any position of the word, for example, input string "aaBB@!#abcDA" will lead to a return string "aA!aA". <br> 3. The idea always checkes the former character when we go through the word, if encountering a different format we add "A"/"a"/"!" depends on what we see. |

Table 1.4

| **Positional offset versions of above** |
|---|
| ■ These features express the characteristic of the word at position t by using information from surrounding words. <br> ■ Approach to use: According to the CRFSuite tutorial (this part quoted), for example, "He reckons the current account", consider the token 'the'. <br> ● w[t-2], w[t-1], w[t], w[t+1], w[t+2], <br> ● w[t-1]\|w[t], w[t]\|w[t+1], <br> ● pos[t-2], pos[t-1], pos[t], pos[t+1], pos[t+2] <br> ● pos[t-2], pos[t-1], pos[t-1]\|pos[t], pos[t]\|pos[t+1], pos[t+1]\|pos[t+2], <br> ● pos[t-2]\|pos[t-1]\|pos[t], pos[t-1]\|pos[t]\|pos[t+1], pos[t]\|pos[t+1]\|pos[t+2] <br> The attribute "w[0]=the" presents the event where the current token is "the", and the attribute "pos[0]\|pos[1]\|pos[2]=DT\|JJ\|NN" presents the event where the parts-of-speech at the current, next, and two words ahead are DT, JJ, NN. CRFSuite will learn associations between these attributes (e.g, "pos[0]\|pos[1]\|pos[2]=DT\|JJ\|NN") and labels to predict a label sequence for a given text. |

Table 1.5

| **First character upper or lower check** |
|---|
| ■ This feature expresses the property of the first character of each word. ■ Approach to use: Check whether the first character of each word is upper or lower, also some other situations need to be considered, such as the first character of some tokens are not letters, but symbols or similar things. ■ Adding this feature helped a little, because a large portion of entity starts with a capital letter. |

# 2. Major extensions

To make the tagging system more perfect and the labelling accuracy higher, external lexical resources for additional features are used. And that is why we update the feature extractor by adding extensions. In this section, we discuss about the major extensions in project.

## 2.1 Freebase

Freebase contains tens of millions of topics, thousands of types, and tens of thousands of properties. Each of the topics in Freebase is linked to other related topics and annotated with important properties like movie genres and people's dates of birth.

Table 2.1

| **Freebase** |
|---|
| ■ Downloaded about 2GB data (the full version could be over 21GB) deleted Freebase entity list.<br>■ Wrote a separate method to extract entity name in the file, because the file contained many properties for each entity which may be not useful.<br>■ Added features to check if the unchanged token (before change to lowercase) matched an entity name in Freebase entity name list.<br>■ Also tried some Freebase API, the mind is similar to clustering, we wrote a simple .py file to test it. |

## 2.2 POS tagging

In project, we tried two POS taggers, the first one is NLTK (Natural Language Toolkit) tagger, the second one is CMU ARK Tweet Part-of-Speech tagger, which is java based but provided with a python warper.

### 2.2.1 NLTK tagger

Table 2.2

| **NLTK tagger** |
|---|
| ■ Added the tag just as what we did in former part. For one word, we added its tag and its nearby tag denoted by tag[0], tag[-1], tag[1]…<br>More specific example as below:<br>>>> text = word_tokenize ("They refuse to permit us to obtain the refuse permit")<br>>>> nltk.pos_tag(text)<br>[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'), ('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', ‘NN’)] |

### 2.2.2 CMU ARK Tweet NLP Part-of-Speech tagger

The tagger provides a fast and robust Java-based tokenizer and part-of-speech tagger for tweets, its training data of manually labeled POS annotated tweets, a web-based annotation tool, and hierarchical word clusters from unlabeled tweets.

Table 2.3

| **CMU ARK Tweet NLP Part-of-Speech Tagger** |
|---|
| ■ Downloaded and it worked as below.<br>sys.path.append('/Users/chenxiao/Documents/NatureLanguageProcessing/starter_code/ark-tweet-nlp-0.3.2')<br>import CMUTweetTagger<br>sentence = " ".join(['I','want','to','see','how','this','works'])<br>sentenceList = []<br>sentenceList.append(sentence)<br>print CMUTweetTagger.runtagger_parse(sentenceList, run_tagger_cmd="java-XX:ParallelGCThreads=2-Xmx500m-jar /Users/chenxiao/Documents/NatureLanguageProcessing/starter_code/ark-tweet-nlp-0.3.2/ark-tweet-nlp-0.3.2.jar") |

## 2.3 Gazetteer

A gazetteer is a geographical dictionary or directory used in conjunction with a map or atlas. They typically contain information concerning the geographical makeup, social statistics and physical features of a country, region, or continent. Content of a gazetteer can include a subject's location, dimensions of peaks and waterways, population, GDP and literacy rate. This information is generally divided into topics with entries listed in alphabetical order.

Table 2.4

| **Gazetteer** |
|---|
| ■ Downloaded full version of the data as what we did for Freebase, extracted the entity for all the locations.<br>■ To deal with a location containing over one word like "Sulphur Springs Post Office", Two approaches tried:<br>1. Tokenize it and compare the token independently.<br>2. Compare the whole locations. |

# 3. Results

## 3.1 Kaggle competition performance results

The system we implemented for the kaggle competition just included the basic features, which are Lexical/word form, Character affixes, Shape features, Positional offset versions of above and First character upper or lower check. So the labelling accuracy using the base system was not that high. The features added in the base system and the corresponding effects are listed.

Table 3.1

| **Base system** |
|---|
| ■ Feature 1: Lexical/word form<br>Very helpful, it is the first feature we added, it improved F-score a lot.<br>■ Feature 2: Character affixes<br>Helpful<br>■ Feature 3: Shape features<br>Helpful<br>■ Feature 4: Positional offset versions of above<br>Helpful<br>■ Feature 5: First character upper or lower check<br>Helpful |

The performance results for the kaggle competition. Notice the F-score in Starter code temple was based on development set, and the F-score in Base system temple was based on final unlabeled test set. We did not know the specific precision and recall values because the kaggle website only provided the F-score.

Table 3.2

| **System Name** | **Precision** | **Recall** | **F-score** |
|---|---|---|---|
| Starter code | 0.7500 (12/16) | 0.0185 (12/647) | 0.0362 |
| Base system | | | 0.36227 |

## 3.2 Major performance results in final system

After the competition, to enrich the feature extractor, we added three main extensions. External lexical resources for additional features are used, we tried Freebase, POS tagger, and Gazetteer. All the system learning are based on the training set given and major performance results are based on the development set.

Table 3.3

| **System used** | **Precision** | **Recall** | **F-score** |
|---|---|---|---|
| Base | 0.6893 (228/332) | 0.3365 (218/647) | 0.4523 |
| Base + POS tagger | 0.7048 (234/332) | 0.3617 (234/647) | 0.4780 |
| Base + POS tagger + Freebase | 0.7096 (237/334) | 0.3663 (237/647) | 0.4832 |
| Base + POS tagger + Freebase + Gazetteer | 0.6973 (235/337) | 0.3632 (235/647) | 0.4776 |

# 4. Analysis/Exploration

There are two main parts in the tagging system implementation. The first part is the base system building, including adding basic features like Lexical/word form, Character affixes, Shape features, Positional offset versions of above and First character upper or lower check. The F-score improved obviously by adding them because these basic features expressed the association among words and characters, which gave the system a better model to learn.

The second part is about the extensions. In this project, the extensions we made mainly depended on the external lexical resources, such as Freebase, NLTK, CMU ARK Tweet, Gazetteer. This part is also the section we met many problems of F-score improvements. After exploring different types of features and analyses, we summarized the meaningful results, experiments we did to find the reason of results and the analyses below.

Table 4.1

| **Freebase** |
|---|
| ■ After adding this extension, the F-score only improved 0.013, then we tried to make some tests to find why this did not work. By adding "print" to debug program we noticed that there were only 9 matches in our training data. Thus it is reasonable that this extension does not help a lot. <br> ■ After tried some Freebase API, we wrote a simple .py file to test it, it worked well. But when we were going to make the system learn on training set and do the same thing for a large of unlabeled data, it took a very long time. The bad thing was we never knew how long it would take when we ran it in shell. However we thought this might be a good extension and left codes in code folder, but we won't use it for our project. |

Table 4.2

**NLTK tagger**

- By using NLTK tagger, it did not change much about our result although we expected it to. By reading and comparing different data resources, we come up a conclusion that NLTK tagger does not work well for tweets because there are a lot of informal words.

Table 4.3

**CMU ARK Tweet NLP Part-of-Speech Tagger**

- We discovered it to replace NLTK tagger due to the reason we just talked above. So we hoped this tagger worked better than NLTK when dealing with tweets, it works like below:

```
sys.path.append('/Users/chenxiao/Documents/NatureLanguageProcessing/starter_code/ark-tweet-nlp-0.3.2')
import CMUTweetTagger
sentence = " ".join(['I','want','to','see','how','this','works'])
sentenceList = []
sentenceList.append(sentence)
print CMUTweetTagger.runtagger_parse(sentenceList, run_tagger_cmd="java -XX:ParallelGCThreads=2 -Xmx500m -jar /Users/chenxiao/Documents/NatureLanguageProcessing/starter_code/ark-tweet-nlp-0.3.2/ark-tweet-nlp-0.3.2.jar")
```

Result:

```
[[('I', 'O', 0.9983), ('want', 'V', 0.9999), ('to', 'P', 0.9929), ('see', 'V', 0.9967), ('how', 'R', 0.9825), ('this', 'O', 0.6955), ('works', 'V', 0.7838)]]
```

According to the above results, the CMU's tagger provides us a better format of output than NLTK tagger. However, when trying to implement it, we encountered a problem that tagger needed a long time to deal with a large data corpus. So we wrote

| |
|---|
| a "print" to check the speed. Then we decided not to implement this in our project even if it could bring us a better result than using NLTK tagger. |

Table 4.4

| **Gazetteer** |
|---|
| ■ For Gazetteer, basically we make analysis for two approaches,<br>1. Tokenize it and compare the token independently, this is easy to implement but this does not make sense,<br>2. Compare the whole locations, this is harder but also does not make sense for some cases because it tries to overfits and never give a match. |

## 5. Discussion and Future Work

In this project, we studied to implement the CRFSuite-based system for tagging Twitter. The features including lexical/word form, Character affixes, Shape features, Positional offset versions and First character upper or lower check are used in the tag prediction model. Obviously, the features selected can improve the accuracy of tagging correctness effectively. Not satisfied with that, we added external lexical resources for additional features. The results shows that some of them useful, some not. However, expect this way, there are many other ways to polish the tagger system. For example, use unlabeled data for self-training, use separate NLP tools to produce features. There is a far way for us to go for the highest result in latest academic research. In future, we can spend time trying all of them and find a best way to build the system. That will be meaningful and interesting.