



Homework #4: Named Entity Recognition

Deadline: 11:59pm Saturday 5 April 2014

Please present a report with all your answers, explanations, and sample images or plots. Submit also a soft copy of the source code and binaries used to generate these results. Please note that copying of any results or source code will result in ZERO credit for the whole homework.

Acknowledgment: This homework is adapted from Chris Manning and Dan Jurafsky's [Coursera](#) NLP class from 2012.

For this assignment, you will be building a maximum entropy Markov model (MEMM) for identifying person names in newswire text. We have provided all of the machinery for training and testing your MEMM, but we have left the feature set woefully inadequate. Your job is to modify the code for generating features so that it produces a much more sensible, complete, and higher-performing set of features.

Although you will be programming in **Python**, you're required to have **Java** working on your system because a significant amount of the starter codes are written in Java. However, you do not need to worry about the Java part because we provide the commands to compile and run the program.

Starter Code

In the starter code we have provided you with three simple starter features, but you should be able to improve substantially on them. We recommend experimenting with orthographic information (uppercase, lowercase, ... etc.) and the surrounding words, and we also encourage you to think beyond these suggestions.

The file you will be modifying is `FeatureFactory.py`.

```
def computeFeatures(self, words, previousLabel, position):  
  
    features = []  
  
    currentWord = words[position]  
  
    """ Baseline Features """  
  
    features.append("word=" + currentWord)  
  
    features.append("prevLabel=" + previousLabel)
```

```
features.append("word=" + currentWord + ", prevLabel=" + previousLabel)

""" TODO: Add your features here """

return features
```

Adding Features to the Code

You will create the features for the word at the given position, with the given previous label. You may condition on any word in the sequence (and its relative position), not just the current word, because they are all observed. You may not condition on any labels other than the previous one.

Each function you build will be a binary function of some kind of feature. The features are stored in a list because we are using a *sparse* representation. Features which have a value of *true* (or `1.0`) will be present in the list and everything not present is assumed to be *false* (or `0.0`).

The argument to `features.append()` is the name of the feature. You need to give a unique name for each feature. The system will use this unique name in training to set the weight for that feature. At testing time the system will use the name of this feature and its weight to make a classification decision.

Consider the following sample feature:

```
features.append("word="+currentWord);
```

and imagine that the current word is "Oakland".

This means "I am a binary function that is true if the current word is 'Oakland'". For every such feature you write, the code will take this function and actually learn two separate weights for this feature: one for "label=PERSON" and one for "label=O".

So just by specifying this feature, the system will learn two functions, call them f_1 and f_2 , where f_1 , which means "I am a binary function that is true if the current word is 'Oakland' and the output label is 'Person'" will get a low weight, and f_2 , which means "I am a binary function that is true if the current word is 'Oakland' and the output label is 'O'" will get a high weight. You can think of the name of both f_1 and f_2 as the string "word=Oakland" and not worry about the fact that we are learning two separate weights. Then, when the system is running on a test sentence that has the word Oakland, it will call your code to create the feature "word=Oakland", and it will automatically create two features, "word=Oakland,label=PERSON" and "word=Oakland,label=O", and it will use those weights in the classifier.

Similarly, this sample feature:

```
features.append("word=Jenny, prevLabel=O");
```

will create two features, each with its own weight, one for "word=Jenny, prevLabel=O, label=PERSON" and one for "word=Jenny, prevLabel=O, label=O".

Types of features to include

Your features should not just be the words themselves. The features can represent any property of the word, context, or additional knowledge.

For example, the case of a word is a good predictor for a person's name, so you might want to add a feature to capture whether a given word was lowercase, Titlecase, CamelCase, ALLCAP, etc.

Imagine you saw the word 'Jenny'. In addition to the feature for the word itself (as above), you could add a feature to indicate it was in Title case, like:

```
if currentWord[0].isupper():  
    features.append("case=Title")
```

It will create two features, each with its own weight, one for "case=Title, label=PERSON" and one for "case=Title, label=O". It is easy to see why this helps. You might encounter an unknown word in the test set, but if you know it begins with a capital letter then this might be evidence that helps with the correct prediction.

Choosing the correct features is an important part of natural language processing. It is as much art as science: some trial and error is inevitable, but you should see your accuracy increasing as you add new types of features.

The name of a feature is not different from an ID number. You can use assign any name for a feature as long as it is unique. For example, you can use "Titlecase" instead of "case=Title".

Running the Program

The data for the homework includes the compiled Java class files needed for running the Python program. If it runs normally, you wouldn't need the Java Development Kit (JDK) installed on your machine, just the Java Runtime Environment (JRE). Otherwise, you will need the JDK and will need to run the two lines below to compile the Java files.

```
$ cd python  
  
$ mkdir classes  
  
$ javac *.java org/json/*.java -d classes
```

```
$ python NER.py ../data/train ../data/dev
```

We have provided you with a training set, called `train` and a development test set called `dev`. Your goal should be to increase F1, which is the harmonic mean of the precision and the recall. If you run the program as-is, you should get the following score:

```
precision = 0.802  
recall = 0.522  
F1 = 0.633
```

When you run the program, you will see it print out a lot of information as it does the optimization, and you can pretty much ignore that. Afterwards, it will print out your score as above. You can give it an additional flag, `-print` and have it print the test set along with the real answers and your guesses. The first column is the word, the second column is the true answer, and the third column is your program's guess. This should help you do error analysis to see what kinds of things you are getting right, and what kind you are getting wrong. This will help you properly target your features.

```
$ python NER.py ../data/train ../data/dev -print
```

If you encounter "Out of Heap Space" problem in the attempt to run or submit the program in Java, increase the heap size by using `"-Xmx2G"` instead of `"-Xmx1G"` option in the `NER.py` file.

Tips

- Start early. This assignment may take longer than the previous assignments if you're aiming for the perfect score.
- Generalize your features. For example, if you're adding the above `"case=Title"` feature, think about whether there is any pattern that is not captured by the feature. Would the `"case=Title"` feature capture `"O'Gorman"`?
- When you add a new feature, think about whether it would have a positive or negative weight for `PERSON` and `O` tags (these are the only tags for this assignment).
- `"-print"` option is useful when you want compare the differences of the predictions before and after adding new features.

Requirements

You are required to add features to the NER system to improve the F-measure on the dev set. Your goal is to get over 85% F-measure.

Please submit your code and report in one zip file, named **CMP462.HW03.firstname.lastname.zip**. For example, if your name is Mohamed Aly, your file should be named **CMP462.HW03.Mohamed.Aly.zip**.

Grading

- 1 pts: report and submission file name
- 3 pts: running code
- 2 pts: F-measure over 70%
- 2 pts: F-measure over 80%
- 2 pts: F-measure over 85%