Massachvsetts Institvte of Technology
Department of Electrical Engineering and Computer Science
6.863J/9.611J, Natural Language Processing
Assignment 4: HMM named-entity tagging
Handed out: Friday, October 12 **Due: Monday, October 29**
Last Updated: Wednesday 17th October, 2012 at 12:53
`http://web.mit.edu/6.863/www/fall2012/labs/assignment4.pdf`

Introduction: goals of the laboratory

This assignment will try to convince you that Hidden Markov Model (HMM) taggers can be useful beyond part of speech tagging. In particular, you will build an HMM tagger that can do what is called *named-entity tagging*, or *named entity recognition* (NER). In doing this, you will also gain familiarity with the standard evaluation metrics used in such NLP efforts, based on the concepts of *precision* and *recall.* All the code and data for this assignment can be downloaded from the following URL. (Note that the code is in python. You may also want to review the notes for lecture 4 on HMM tagging, as well as the relevant sections in the J-M textbook, Chapter 6, pp. 174–186, or the (online!) Manning-Schütze book, pp. 345–360. They give detailed descriptions of HMM taggers, as well as worked examples.

`http://web.mit.edu/6.863/fall2012/code/assignment4.zip`

As usual, your writeup and code should be emailed to `6.863-graders@mit.edu`, with the Subject line, "6.863 Assignment 4".

Named-entity tagging is the assignment of labels to words that are not part of speech tags such as nouns (NN) or verbs (VB, VBD), but rather whether the word fits a particular semantic category such as an "organization" or a "person". This turns out to be very useful in tasks like "message understanding." For example, given a sentence such as this: Google bought ITA Software said Larry Page in California
A named-entity tagger might output that *Google* and *ITA-Software* should both be tagged as organizations (ORG), and *Larry Page* as a person (PER); and *California* as a location (LOC); other words could simply remain unlabeled (O). So instead of a part of speech stream as output, it might output the following: Google ORG bought O ITA ORG Software said O Larry PER Page PER in O California LOC

HMM modeling Recall that the joint probability of a sentence consisting of n words $w_1, w_2, \ldots, w_n$ and a sequence of n corresponding tags $t_1, t_2, \ldots, t_n$ for a trigram model may be defined this way:

$$p(w_1, w_2, \ldots, w_n, t_1, t_2, \ldots, t_n)$$
$$= q(t_1|*,*) \cdot q(t_2|t_1,*) \cdot q(\texttt{STOP}|t_n, t_{n-1}) \prod_{i=3}^{n} q(t_i|t_{i-1}, t_{i-2}) \cdot \prod_{i=1}^{n} e(w_i|t_i)$$

Here, the w are the observed output words; the t are the named entity tags; and e are the so-called "emission probabilities" of producing an output w when the corresponding hidden state is t. The symbol $*$ is a special symbol to pad out two symbols before the actual first word that appears in a sentence, and `STOP` is a special HMM state that denotes the end of a sentence.

As usual, if we instead use only a bigram model, then the first product term in the second equation above reduces to just the following:

$$q(t_1|*) \cdot q(t_2|t_1) \cdot q(\texttt{STOP}|t_n) \cdot \prod_{i=3}^{n} q(t_i|t_{i-1},) \cdot \prod_{i=1}^{n} e(w_i|t_i)$$

To develop and then test your HMM NER taggers, you will use a training data set `ner_train.dat` and a development set `ner_dev.key`. Both files are in the following format (one word per line, word and token separated by space, with a single blank line separating sentences):

```
Germany I-LOC
's O
representative O
to O
the O
European I-ORG
Union I-ORG
's O
veterinary O
committee O
Werner I-PER
Zwingmann I-PER
...
```

In this data, there are four types of named entities: person names (PER), organizations (ORG), locations (LOC) and miscellaneous names (MISC). Named entity tags have the format I-TYPE. Only if two phrases of the same type immediately follow each other, the first word of the second phrase will have tag B-TYPE to show that it starts a new entity. Words marked with O are not part of a named entity.

The file `ner.counts` contains contains trigram, bigram, unigram, and emission counts in the following format:

- Lines where the second token is `WORDTAG` contain emission counts $Count(t \rightsquigarrow w)$ (this notation denotes the number of times that word w is observed given that t is the tag). For example, the following part of the output says that in the training data, `DETROIT` was tagged 16 times as an `I-ORG`:

  ```
  16 WORDTAG I-ORG DETROIT
  ```

- Lines where the second token is n-`GRAM` (where n is 1, 2 or 3) contain unigram counts $Count(t)$, bigram counts $Count(t_{n1}, t_n)$, or trigram counts $Count(t_{n2}, t_{n1}, t_n)$. For example, the following says that there were 6917 times that where an `I-LOC` tag was followed by an `O` tag.

  ```
  6917 2-GRAM I-LOC O
  ```

- The following says that there were 607 times where the bigram `I-LOC O` was followed by a second `I-LOC`:

  ```
  607 3-GRAM I-LOC O I-LOC
  ```

Important: note that since the unigram, bigram, and trigram taggers you will implement below are run over a fixed development and test set, we can provide you in advance with *approximately* the results you should get when you write your own programs for tagging this data. This should provide you with some guidance as to whether your own programs are working correctly. (Your mileage may vary due to different rounding effects; however, if your F1-scores are much lower than these, it is likely that you have a bug in your code.) The results are given using so-called *precision*, *recall*, and *F1-score* accuracy results for each tagger in turn. Please refer to page 4 for a brief description of these standard accuracy measures.

	Unigram		
	precision	recall	F1-score
Total:	0.228802	0.530939	0.319793
PER:	0.435367	0.229053	0.300178
ORG:	0.532979	0.374439	0.439860
LOC:	0.149456	0.883860	0.255678
MISC:	0.600000	0.657980	0.627654

Bigram			
	precision	recall	F1-Score
Total:	0.620618	0.504468	0.556548
PER:	0.459140	0.232318	0.308526
ORG:	0.683673	0.400598	0.505184
LOC:	0.612722	0.772083	0.683233
MISC:	0.770101	0.665581	0.714036

Trigram			
	precision	recall	F1-Score
Total:	0.706910	0.593323	0.645155
PER:	0.833916	0.519042	0.639839
ORG:	0.686833	0.432735	0.530949
LOC:	0.631676	0.748092	0.684973
MISC:	0.749695	0.666667	0.705747

Problem 1. (**20 points**) You will first implement a *unigram* tagger, to establish a baseline of accuracy. You can do this by carrying out the following steps:

1.(a) Write a function to compute emission parameters $e(w|t)$ based on the counts in `ner.counts`. We use the notation $Count(t \rightsquigarrow w)$ to denote the number of times that w is observed given that t is the tag. (This will be part of your tagger; you don't need to turn it in seperately)

$$e(w|t) = \frac{Count(t \rightsquigarrow w)}{Count(t)}$$

1.(b) We need to predict emission probabilities for words in the test data that do not occur in the training data. One simple approach is to map infrequent words in the training data to a common class and to treat unseen words as members of this class. Replace infrequent words (those that occur less than 5 times in the training set, according to ner.counts) with a common symbol `_RARE_`, and use the emission parameters $e(RARE|t)$ for predicting emission probabilities for words that didn't occur in the training data. (This will be part of your tagger; you don't need to turn it in separately)

1.(c) As a baseline, implement a simple named entity tagger that always produces the tag $t* = \arg\max_t e(w|t)$ for each word w. Make sure your tagger uses the `_RARE_` word probabilities for rare and unseen words. Your tagger should read in the `ner.counts` file as shown above, and the file `ner_dev.dat` (which is `ner_dev.key` but untagged) and produce output in the same format as the training file, but with an additional column in the end that contains the log probability for each prediction, which looks like this:

```
West I-LOC -4.946599062249232
```

Submit your program as `unigram_tagger.py`, such that it can be run as follows to output the tagged data into a file:

```
python unigram_tagger.py ner.counts ner_dev.dat > ner_dev_unigram_tagger.dat
```

To evaluate the precision, recall, and F1-scores for each named-entity category over the prediction file, run:

```
python eval_ne_tagger.py ner_dev.key ner_dev_unigram_tagger.dat
```

As we indicated above, you should be able to see from this output whether you are coming close to the desired goal on this data. But what are these three scores? Precision and recall are standard evaluation metrics used in classification tasks like this. We can define *precision* as follows. If we have a particular tagging program, then it will assign a certain number of tags that are correct; we call these *true positives*, denoted *tp*. However, the program might also assign a particular tag that is in fact incorrect – e.g., it labels *Larry* as an organization rather than a person. We call these mistakes *false positives*, denoted fp. *Precision* is then defined as the fraction of true positives divided by the sum of the true positives and false positives:

$$precision = \frac{tp}{tp + fp}$$

So if a program has high precision, the tags it does assign are usually correct. If it has a perfect precision score of 1, *all* of the tags the program assigns will be correct. Note however that is possible to make a program have a higher precision simply by making it more and more conservative (more cautious): if it only assigns tags to one or two words where it is very certain about its labeling, then the precision could even be perfect, but the program will have also simply not tagged many words. In this case, the program clearly does not have very good performance, even though its precision is high. To detect this possibility, we use a second notion of performance, *recall.* If a tagger does not classify a particular word with the right tag, then that is a *false negative*, denoted fn. *Recall* is defined as the number of true positives divided by the sum of the true positives plus false negatives:

$$recall = \frac{tp}{tp + fn}$$

Consequently, if we make our system have higher precision by reducing the number of "false alarms" or false positives, then typically this will decrease recall, because whenever the system does not classify a word at all that should have some particular tag, that results in a false negative. In short, one has to balance a system's performance between being conservative (and so precise), and covering more examples (having better recall).

To combine the precision and recall results, we typically take a harmonic mean of the two scores, weighting both equally; this is called the *F1-score.* The harmonic mean is defined as the reciprocal of the average of the reciprocal values of precision and recall; harmonic means are used in such cases because they provide a better summary of the average of two *rates* – and here we are talking about precision and recall rates per some number of words. We can define the harmonic mean in general as follows, for n values:

$$H = \left(\frac{1}{n} \sum_{i=1}^{n} x_i^{-1} \right)^{-1}$$

Given just two values, the precision rate, p, and the recall rate, r, then we can derive the formula for F1 as in the J-M text, pp. 455-456, or C-S, p. 269, as follows by plugging p and r into the formula for the harmonic mean:

$$
\begin{aligned}
&= \frac{1}{\frac{1}{2}\cdot(\frac{1}{p}+\frac{1}{r})} = 2\cdot\frac{1}{(\frac{1}{p}+\frac{1}{r})}\\
&= 2\cdot\frac{1}{\frac{p+r}{p\cdot r}}\\
F1 &= 2\cdot\frac{p\cdot r}{p+r}
\end{aligned}
$$

which written out in a wordy way comes to this:

$$F1 = 2\cdot\frac{precision\cdot recall}{precision + recall}$$

Please report the precision, recall, and F1 scores for your tagger as output by `eval_ne_tagger.py`, along with any observations you made.

Problem 2. (**20 points**) Next, you will implement a bigram named-entity tagger, and compare its performance to the baseline tagger you have implemented.

2.(a) Using the counts in `ner.counts`, write a function that computes the bigram parameters q:

$$q(t_i|t_{i-1}) = \frac{Count(t_{i-1}, t_i)}{Count(t_{i-1})}$$

You should check to make sure that you have covered the start and stop boundary conditions, i.e., $q(t_1|*)$ and $q(\texttt{STOP}|t_n)$. (This will all be part of your tagger; you don't need to turn it in seperately).

2.(b) Using the maximum likelihood estimates for transitions and emissions, implement the Viterbi algorithm to compute a bigram tagger, which does the following:

$$\arg\max_{t_1\ldots t_n} p(w_1\ldots w_n, t_1\ldots t_n)$$

Your tagger should have the same basic functionality as the baseline tagger. Instead of emission probabilities the third column should contain the log-probability of the tagged sequence up to this word.

Submit your program as `bigram_tagger.py`, such that it can be run and evaluated as follows:

```
python bigram_tagger.py ner.counts ner_dev.dat > ner_dev_bigram_tagger.dat
      python eval_ne_tagger.py ner_dev.key ner_dev_bigram_tagger.dat
```

Please report the precision, recall, and F1 scores for your tagger as output by `eval_ne_tagger.py`, along with any observations you made.

Problem 3. (10 points) Now, implement a trigram tagger by using trigram counts and the trigram estimates for the q values, that is:

$$q(t_i|t_{i-2}t_{i-1}) = \frac{Count(t_{i-2}, t_{i-1}, t_i)}{Count(t_{i-2}, t_{i-1})}$$

Submit your program as `trigram_tagger.py`, such that it can be run and evaluated as follows:

```
python trigram_tagger.py ner.counts ner_dev.dat > ner_dev_trigram_tagger.dat
python eval_ne_tagger.py ner_dev.key ner_dev_trigram_tagger.dat
```

Please report the precision, recall, and F1 scores for your tagger as output by `eval_ne_tagger.py`, along with any observations you made.

Problem 4. (**20 points**) You will now improve your trigram tagger (if you were unable to correctly implement the trigram tagger, use your bigram tagger for this problem). Change the way your HMM tagger deals with low-frequency words by grouping them based on informative patterns rather than just into a single class of rare words. For instance, such a class could contain all capitalized words (possibly proper names), all words that contain only capital letters and dots (possibly abbreviations for first names or companies), or all words that consists only of numerals. Use these revised counts to get new estimates for the emission probabilities in your tagger.

Submit the best tagger (the one which achieves the highest F1-score) as `final_tagger.py`, such that it can be run and evaluated as follows:

```
python final_tagger.py ner.counts ner_dev.dat > ner_dev_final_tagger.dat
python eval_ne_tagger.py ner_dev.key ner_dev_final_tagger.dat
```

Please report the precision, recall, and F1 scores for your tagger as output by `eval_ne_tagger.py`, and also discuss the modifications you made that allowed you to achieve these improved scores.