# CS-5340/6340, Programming Assignment #3
# Due: Thursday, October 26, 2017 by 11:00pm

Your task is to create a machine learning classifier for Named Entity Recognition (NER). Your classifier should recognize 3 types of named entities: Persons (PER), Locations (LOC), and Organizations (ORG). You should use the BIO labeling scheme to classify each word with one of 7 labels: *B-PER*, *I-PER*, *B-LOC*, *I-LOC*, *B-ORG*, *I-ORG*, or *O*. You will be given a software package to create the machine learning (ML) classifier but you will need to create the feature vectors for the training and test instances.

You should write a program called `ner` that accepts four types of arguments as input: (1) a file of training sentences, (2) a file of test sentences, (3) a file of Locations, and (4) a list of *ftype* arguments indicating which types of features will be generated. There are 7 possible *ftypes*: WORD WORDCON POS POSCON ABBR CAP LOCATION. There will always be one *ftype*, WORD, which will appear first and may be followed by up to 6 additional *ftypes*, in any order! Your program should accept the command-line arguments as follows:

```
ner <train_file> <test_file> <locations_file> WORD [... <ftype_n >]
```

We should be able to run your program with any combination of *ftype* arguments, for example:

```
ner train.txt test.txt locs.txt WORD
ner train.txt test.txt locs.txt WORD CAP POS
ner train.txt test.txt locs.txt WORD POSCON POS WORDCON ABBR CAP LOCATION
```

## 1 Input Files

The training and test files will have the same format. Each file will contain sentences with a class label and a part-of-speech (POS) tag assigned to each word. Each word corresponds to one instance for the ML classifier. The information for each word will be on a separate line, formatted as: `label POS word`. One or more blank lines will separate different sentences. For example, an input file might look like this:

| | | |
|---|---|---|
| B-LOC | NNP | Israel |
| O | NN | television |
| O | VBD | rejected |
| O | DT | the |
| O | NN | skit |
| O | IN | by |
| O | DT | the |
| O | NN | comedian |
| B-PER | NNP | Tuvia |
| I-PER | NNP | Tzafir |

**Important:** *Please use the words exactly as they are provided.* Note that some may be isolated punctuation marks while others may still have punctuation marks attached (e.g., "Mr.").

The Locations file will contain one location per line. Each location will be a single word.

## 2 Feature Types

Your program should be able to produce 7 types of features for a word $w$, as specified by the *ftype* arguments:

**WORD:** the word $w$ itself
**WORDCON:** the previous word $w_{-1}$ and the following word $w_{+1}$
**POS:** the POS tag $p$ of $w$
**POSCON:** the previous POS tag $p_{-1}$ and the following POS tag $p_{+1}$
**ABBR:** a binary feature indicating whether $w$ is an abbreviation. An abbreviation must: (1) end with a period, (2) consist entirely of alphabetic characters [a-z][A-Z] and one or more periods, and (3) have length $\leq 4$
**CAP:** a binary feature indicating whether $w$ is capitalized (i.e., if the first letter is capitalized then this feature should be Yes, including words in all caps such as "IBM").
**LOCATION:** a binary feature indicating whether $w$ occurs in the Locations file

## 3 Output Files

Your `ner` program should do 2 things:

1. Generate human readable features for each training instance and write these features to a new output file. Give this output file the same name as the training file but add the extension `.readable`. Do the same thing on the test file.
2. Generate a feature vector for each training instance and write these feature vectors to a new output file. Give this output file the same name as the training file but add the extension `.vector`. Do the same thing on the test file.

For example, given the command:

```
ner train.txt test.txt locs.txt WORD WORDCON POS
```

Your program should produce four files named:

```
train.txt.readable
test.txt.readable
train.txt.vector
test.txt.vector
```

### 3.1 Readable Output Files

In the Readable files, each feature should be printed as follows:

**WORD:** $w$
**WORDCON:** $w_{-1}$ $w_{+1}$ (with exactly one space between $w_{-1}$ and $w_{+1}$)
**POS:** $p$

**POSCON:** $p_{-1}$ $p_{+1}$ (with exactly one space between $p_{-1}$ and $p_{+1}$)
**ABBR:** yes or no
**CAP:** yes or no
**LOCATION:** yes or no

For example, this output should be produced for "rejected" in the example on Page 1:

**WORD:** rejected
**WORDCON:** television the
**POS:** VBD
**POSCON:** NN DT
**ABBR:** no
**CAP:** no
**LOCATION:** no

You should print these 7 lines for every word in the input file, with a blank line separating the features for different words. ALWAYS print these 7 lines, even if fewer than 7 *ftype* arguments are given. If an *ftype* is not in the arguments, then print "n/a" (for "not applicable") as the feature value (e.g., CAP: n/a).

There are a few special cases where you may need to use pseudo-words or pseudo-POS tags. Please use the following conventions:

| **Pseudo** | **Description** |
|---|---|
| PHI | pseudo-word for the beginning-of-sentence position |
| PHIPOS | pseudo-POS for the beginning-of-sentence position |
| OMEGA | pseudo-word for the end-of-sentence position |
| OMEGAPOS | pseudo-POS for the end-of-sentence position |
| UNK | pseudo-word for words in the test but not the training data |
| UNKPOS | pseudo-POS for POS tags in the test but not the training data |

The UNK pseudo-word should be used for words that appear in the test file but did not appear in the training file. The UNKPOS pseudo-tag should be used for part-of-speech tags that appear in the test file but did not appear in the training file.

For example, consider the word "Israel" from the example on Page 1. Assume that "television" occurs in the test data but not the training data. Given the *ftype* arguments "WORD CAP WORDCON POSCON", you should generate the following output:

**WORD:** Israel
**WORDCON:** PHI UNK
**POS:** n/a
**POSCON:** PHIPOS NN
**ABBR:** n/a
**CAP:** yes
**LOCATION:** n/a

**Important:** Do not cross sentence boundaries when creating the features for a word! The PHI and OMEGA pseudo-words should never be the current word, they are only placeholders for the beginning and end of a sentence for the contextual features.

## 3.2 Vector Output Files

Your program should produce two vector output files: one file containing feature vectors for the training instances, and one file containing feature vectors for the test instances. These files will serve as the input to the ML software, so they need to be formatted exactly as explained below or the ML software will reject it.

Each line should represent a feature vector for one instance (word). Unlike the input files, there should not be any blank lines. The format of each line should be:

```
label feature_id:1 feature_id:1 ...
```

There should be exactly one space between `label` and `feature_id:1` and between each pair of adjacent features. The `label` represents the BIO label. However, the ML classifier requires that labels are integers. So please use the following numeric ids for the BIO labels:

0 for *O*
1 for *B-PER*
2 for *I-PER*
3 for *B-LOC*
4 for *I-LOC*
5 for *B-ORG*
6 for *I-ORG*

The ML tool also requires that each feature ID be an integer value $> 0$, so your program will need to assign a numeric ID to each feature. The number after the colon is the feature value, which will always be 0 or 1, because the software requires binary-valued features. For this ML tool, you only need to list features that have a value of 1. All features that are not listed will be presumed to have a value of 0.

**Important:** The ML software requires that the feature ids appear in ASCENDING order! So you will need to sort the feature ids and print them in ascending order.

**Important:** The same feature ids must be used for both the training and test instances! For example, if the feature ABBR is assigned the ID 22, then ID 22 should be used for ABBR throughout ALL training and test instances.

## 3.3 How to Create Binary Features for String Values in the Vector Files

The ML classifier requires binary features. The ABBR, CAP, and LOCATION features take binary values, so creating features for them is easy: just define a unique feature ID for each one and use the value 1 for “yes” and the value 0 for “no”.

But the WORD, WORDCON, POS, and POSCON features take string values, so you may be wondering how to translate a string feature into a binary feature. The trick is to generate the set of all possible string values from the training data. For the WORD feature, this is the set of all distinct words plus an UNK value. For the POS feature, this is the set of all distinct POS tags plus an UNKPOS value. If a string-valued feature has $k$ possible values, then you should generate $k$ binary features with exactly one of the features having a value of 1 and all other features having a value of 0.

Here is an example. Suppose we have a training corpus containing only two sentences (POS tag appears after the slash). Note that the POS tagger uses a period (.) as the POS tag for end-of-sentence tokens such as periods, exclamation points, and question marks.

**S1:** John/NNP saw/VBD Mary/NNP ./.
**S2:** Mary/NNP waved/VBD !/.

How should we represent the WORD feature? There are 6 distinct words in the corpus, so we would create the 7 binary features shown below:

$x_1$**:** John
$x_2$**:** Mary
$x_3$**:** saw
$x_4$**:** waved
$x_5$**:** .
$x_6$**:** !
$x_7$**:** UNK

For "John", we would produce the following WORD feature values:
$x_1$=1, $x_2$=0, $x_3$=0, $x_4$=0, $x_5$=0, $x_6$=0, $x_7$=0

For "saw", we would produce the following WORD feature values:
$x_1$=0, $x_2$=0, $x_3$=1, $x_4$=0, $x_5$=0, $x_6$=0, $x_7$=0

For the POS feature, there are 3 distinct tags in the corpus, so we should create the 4 binary features shown below:

$y_1$**:** NNP
$y_2$**:** VBD
$y_3$**:** .
$y_4$**:** UNKPOS

For "John", we would produce the following POS feature values:
$y_1$=1, $y_2$=0, $y_3$=0, $y_4$=0

For "saw", we would produce the following POS feature values:
$y_1$=0, $y_2$=1, $y_3$=0, $y_4$=0

**Important:** For the WORDCON and POSCON features, you'll also need to include PHI and OMEGA (or PHIPOS and OMEGAPOS) as possible values, and you'll need to create separate sets of binary features for the previous word and following word. Essentially, you'll be creating 4 sets of binary features (i.e., one set for each of $w_{-1}$, $w_{+1}$, $p_{-1}$, $p_{+1}$).

## 3.4 Feature Vector Examples

Here are illustrations of each step based on the example on the first page (assuming that it is training data).

If the feature type arguments are "WORD CAP POS POSCON", first you would generate all possible word features and assign a unique numeric identifier to each one. There are 9 distinct words in the example, so you must create a feature for each one, and also the UNK pseudo-word mentioned earlier.

Then you should generate all possible POS features. There are 5 distinct part-of-speech tags, so you must create a feature for each one (and the UNKPOS pseudo-tag) in the current position("pos-$p$") associated with word $w$. For the POSCON feature, you will also need to create features for the POS tags in the previous position("prev-pos-$p$") and following position("next-pos-$p$"), including values for the appropriate pseudo-tags. Also, you need one binary capitalization feature.

The full feature set is shown below:

| ID | Feature | ID | Feature | ID | Feature |
|---|---|---|---|---|---|
| 1 | word-Israel | 2 | word-television | 3 | word-rejected |
| 4 | word-the | 5 | word-skit | 6 | word-by |
| 7 | word-comedian | 8 | word-Tuvia | 9 | word-Tzafir |
| 10 | word-UNK | 11 | prev-pos-PHIPOS | 12 | next-pos-OMEGAPOS |
| 13 | pos-NNP | 14 | prev-pos-NNP | 15 | next-pos-NNP |
| 16 | pos-NN | 17 | prev-pos-NN | 18 | next-pos-NN |
| 19 | pos-VBD | 20 | prev-pos-VBD | 21 | next-pos-VBD |
| 22 | pos-DT | 23 | prev-pos-DT | 24 | next-pos-DT |
| 25 | pos-IN | 26 | prev-pos-IN | 27 | next-pos-IN |
| 28 | pos-UNKPOS | 29 | prev-pos-UNKPOS | 30 | next-pos-UNKPOS |
| 31 | capitalized | | | | |

Using the features and identifiers above, you would then generate the following feature vectors. Remember that the feature ids must be sorted in ascending order for the classifier!

| Word | Label & Feature Vector |
|---|---|
| Israel | 3 1:1 11:1 13:1 18:1 31:1 |
| television | 0 2:1 14:1 16:1 21:1 |
| rejected | 0 3:1 17:1 19:1 24:1 |
| the | 0 4:1 18:1 20:1 22:1 |
| skit | 0 5:1 16:1 23:1 27:1 |
| by | 0 6:1 17:1 24:1 25:1 |
| the | 0 4:1 18:1 22:1 26:1 |
| comedian | 0 7:1 15:1 16:1 23:1 |
| Tuvia | 1 8:1 13:1 15:1 17:1 31:1 |
| Tzafir | 2 9:1 12:1 13:1 14:1 31:1 |

## 4 The Machine Learning Tool

**Installation:** The machine learning software is available in the Files folder for Program #3 on Canvas as: `liblinear-1.93.tar.gz`. To install it, first unpackage it (`tar xvfz liblinear-1.93.tar.gz`). In the directory, type: `make`.

This will produce two executable files: `train` and `predict`.

**Training:** to train a classification model (classifier), invoke the command:

```
train -s 0 -e 0.0001 <train_data> <classifier_name>
```

Note: the option "-s 0" indicates that you are using the logistic regression machine learning algorithm. The `train_data` argument is the name of the file containing the feature vectors for the training data. The learned classifier will be saved as a file named `classifier_name`.

**Testing:** to apply a classification model (classifier), invoke the command:

```
predict <test_data> <classifier_name> <predictions_file> > <accuracy_file>
```

The `test_data` file should contain the feature vectors for the test data, and the `classifier_name` should be the file name of the trained classification model. The classifier's predicted labels will be saved as `predictions_file` and the classifier's accuracy results will be saved as `accuracy_file`.

For example, you could use the following two commands to train a classifier with training data and apply that classifier to test data:

```
train -s 0 -e 0.0001 train.txt.vector classifier

predict test.txt.vector classifier predictions.txt > accuracy.txt
```

### Example

When your `ner` program is finished you should be able to perform the following sequence of commands:

1. Generate the feature vectors for the training and test sentences:
   `ner train.txt test.txt locations.txt WORD POSCON ABBR`
2. Train a ML classifier using the training instances:
   `train -s 0 -e 0.0001 train.txt.vector classifier`
3. Use the ML classifier to label the test instances:
   `predict test.txt.vector classifier predictions.txt > accuracy.txt`

**FOR CS-6340 STUDENTS ONLY! (30 pts)**

Your second task is to create a program that evaluates the performance of the NER classifier in terms of entities, instead of BIO labels. You should write a second program called `eval` that accepts two arguments as input: (1) a file of predicted BIO labels for a sentence, and (2) a file of gold BIO labels for the same sentence. Your program should accept the arguments from the command line in the following order:

```
eval <prediction_file> <gold_file>
```

For example, we should be able to run your program like this:

```
eval prediction.txt gold.txt
```

## 1 Input Files

The prediction and gold files will have the same format. Each file will contain exactly one sentence, with each line consisting of a BIO label followed by a word. For example, an input file might look like this:

```
B-LOC   Israel
B-ORG   television
O       rejected
O       the
O       skit
O       by
O       the
O       comedian
B-PER   Tuvia
I-PER   Tzafir
O       who
O       was
O       born
O       in
B-LOC   Haifa
I-LOC   ,
I-LOC   Israel
O       .
```

## 2 Evaluation Metrics

For an NER system, the basic unit is a named entity, not a word. When measuring accuracy, what we truly care about is the number of named entities that were found correctly, not the number of BIO labels that were assigned correctly. For example, an NER tagger could produce many correct

I labels, but if all of the B labels are wrong then it would have 0% accuracy at finding named entities!

For this program, you must group together the BIO labels to identify distinct named entities and assign start and end positions to each one. For example, consider the sentence below:

| Position | Prediction | | Gold | |
|---|---|---|---|---|
| 1 | B-LOC | Israel | B-LOC | Israel |
| 2 | B-ORG | television | O | television |
| 3 | O | rejected | O | rejected |
| 4 | O | the | O | the |
| 5 | B-ORG | skit | O | skit |
| 6 | I-LOC | by | O | by |
| 7 | O | the | O | the |
| 8 | B-PER | comedian | O | comedian |
| 9 | B-PER | Tuvia | B-PER | Tuvia |
| 10 | I-PER | Tzafir | I-PER | Tzafir |
| 11 | I-PER | who | O | who |
| 12 | O | was | O | was |
| 13 | O | born | O | born |
| 14 | O | in | O | in |
| 15 | B-LOC | Haifa | B-LOC | Haifa |
| 16 | I-LOC | , | I-LOC | , |
| 17 | I-LOC | Israel | I-LOC | Israel |
| 18 | O | . | O | . |

A legal named entity must start with a B label. Note that "by" is tagged as I-LOC in the predictions, but it is not a legal LOC entity because it is preceded by a B-ORG label, not a B-LOC label. The legal named entities extracted from the data

above appear below, with the square brackets indicating their beginning and end positions.

| **Prediction** | **Gold** |
|---|---|
| LOCATION: Israel [1-1] | LOCATION: Israel [1-1] |
| ORGANIZATION: television [2-2] | PERSON: Tuvia Tzafir [9-10] |
| ORGANIZATION: skit [5-5] | LOCATION: Haifa , Israel [15-17] |
| PERSON: comedian [8-8] | |
| PERSON: Tuvia Tzafir who [9-11] | |
| LOCATION: Haifa , Israel [15-17] | |

After extracting the named entities, you should (1) compute Recall and Precision for each of the 3 entity types, and (2) compute the Average Recall and Average Precision across all of the entity types. To compute the averages, sum all of the counts and then divide by the total number of entities. A Predicted entity should be considered correct only if it has exactly the same beginning and end positions and the same entity type (PER, ORG, or LOC) as a Gold entity.

## 3 Output File

You program should produce a file named `eval.txt`, formatted as shown below. When printing the entity lists, please list each entity name along with its start and end positions as [i-j], and separate multiple entities with a bar (|). Please sort them based on their start position. If no correct entities are found, then print NONE. Recall and Precision should be printed in fractional form! If the denominator for Recall or Precision is zero, print "n/a".

```
Correct PER = <list correct person entities>
Recall PER = <recall>
Precision PER = <precision>

Correct LOC = <list correct person entities>
Recall LOC = <recall>
Precision LOC = <precision>

Correct ORG = <list correct person entities>
Recall ORG = <recall>
Precision ORG = <precision>

Average Recall = <recall>
Average Precision = <precision>
```

For example, the sentence shown previously should produce this output:

```
Correct PER = NONE
Recall PER = 0/1
Precision PER = 0/2

Correct LOC = Israel [1-1] | Haifa , Israel [15-17]
Recall LOC = 2/2
Precision LOC = 2/2

Correct ORG = NONE
Recall ORG = n/a
Precision ORG = 0/2

Average Recall = 2/3
Average Precision = 2/6
```

## GRADING CRITERIA

Your program will be graded based on new data files!

**So please test your program thoroughly to evaluate the generality and correctness of your code!** Even if your program works perfectly on the examples that we give you, that does not guarantee that it will work perfectly on new files.

---

## ELECTRONIC SUBMISSION INSTRUCTIONS
## (a.k.a. "What to turn in and how to do it")

Please use CANVAS to submit an archived (.tar) or zipped (.zip) file containing the following:

1. The source code files for your `ner` program, and for CS-6340 students, your `eval` program. Be sure to include all files that we will need to compile and run your program!
2. A README file that includes the following information:
   - how to compile and run your code
   - which CADE machine you tested your program on (this info may be useful to us if we have trouble running your program)
   - any known bugs, problems, or limitations of your program
3. Run your code with the following command, using the input files available in the Program #3 folder on CANVAS:

   ```
   ner train.txt test.txt locs.txt WORD POSCON POS WORDCON ABBR CAP LOCATION
   ```

   Then train a classifier with the train.txt.vector file using the ML software, and use the resulting classifier to make predictions on the test.txt.vector file (as explained in Section 4). Then include the following files in your submission:

   (a) train.txt.readable
   (b) test.txt.readable
   (c) train.txt.vector
   (d) test.txt.vector
   (e) predictions.txt
   (f) accuracy.txt
4. **For CS-6340 students:** Please also submit a file called `eval.txt` with the output of your `eval` program for the prediction.txt and gold.txt files in the Program #3 folder on CANVAS.

REMINDER: your program must be written in Python or Java and it must compile and run on the unix-based CADE machines! We will not grade programs that cannot be run on the unix-based CADE machines.