CS-6390, Programming Assignment #1
Due: Monday, February 6, 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 three arguments as input: (1) a file of training sentences, (2) a file of test sentences, and (3) an *ftype* argument indicating which types of features will be generated. There will be 5 possible feature types: *word*, *wordcap*, *poscon*, *lexcon*, *bothcon*. Your program should accept the arguments from the command line in the following order:

```
ner <train_file> <test_file> <ftype>
```

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

```
ner train.txt test.txt poscon
```

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

Feature Generation

Your `ner` program should do 3 things:

1. Identify all of the distinct words and part-of-speech tags that occur in the training data. You will need to use these lists when creating your feature vectors.
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 input file but add the extension `.<ftype>` (based on the *ftype* argument).
3. Generate a feature vector for each test instance and write these feature vectors to a new output file. Give this output file the same name as the test input file but add the extension `.<ftype>` (based on the *ftype* argument).

For example, given the command:

```
ner train.txt test.txt poscon
```

Your program should produce two files named:

```
train.txt.poscon
test.txt.poscon
```

Important: please use this exact naming convention! Our grading procedure will assume that your output files will have these names.

Part-of-Speech (POS) Tags

Note that some of the POS tags will be punctuation marks! For example, the POS tag for a period is a period (.), and the POS tag for a comma is a comma (,). And some POS tags will contain a punctuation mark (e.g., PP$). When you read the input files, be careful to separate the tokens based on white space. There will always be three tokens per line, separated by white space: its BIO label, its POS tag, and the word.

Feature Types

Your program should be able to produce 5 types of feature vectors for a word w, as specified by the *ftype* argument:

> **word:** use only the word w itself as a feature
>
> **wordcap:** use the word w itself as a feature, and create a binary feature that indicates whether w is capitalized
>
> **poscon:** in addition to the **wordcap** features, create context features for the POS tag of the previous word w_{-1} and the POS tag of the following word w_{+1}
>
> **lexcon:** in addition to the **wordcap** features, create context features for the string of the previous word w_{-1} and the following word w_{+1}
>
> **bothcon:** use ALL of the features above: the **wordcap** features, the POS context features, and the lexical context features.

The ML classifier requires binary features. So you should create sets of word and part-of-speech features, where each feature will have a binary value. For example, for the **word** feature, you will create one feature for every distinct word in the training data. For a given word w, all of the word features will have a zero value except for w, which will have the value 1.

You will also need to create feature sets for each position of a word or part-of-speech. For words, you'll need one set of features for word w, a second set of features for word w_{-1}, and a third set of features for w_{+1}. For example, suppose the word "school" occurs in the training data. You should create three features, one for each position in which "school" may occur: "prev-school", "curr-school", and "next-school". The identifiers that you choose do not matter, as long as you have distinct features for each position of the word.

To illustrate, consider the word "rejected" in the example on Page 1. The features that will have a value of 1 are shown below, for each `ftype` argument. Note that "rejected" is not capitalized, so the capitalization feature would have a value of 0 (so is not shown).

> **word:** curr-rejected
> **wordcap:** curr-rejected
> **poscon:** curr-rejected prev-NN next-DT
> **lexcon:** curr-rejected prev-television next-the
> **bothcon:** curr-rejected prev-NN next-DT prev-television next-the

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

Pseudo word/POS	**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
UNKWORD	pseudo-word for words in the test data that are not in the training data
UNKPOS	pseudo-POS for POS tags in the test data that are not in the training data

For example, suppose that the word "television" occurs in the test data but not the training data. In this case, the features that should be created for the word "Israel" (from the previous example) using the **bothcon** argument are:

curr-Israel capitalized prev-PHIPOS next-NN prev-PHI next-UNKWORD

IMPORTANT: Do not cross sentence boundaries when creating the feature vector 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, respectively, for the contextual features.

Output Files (follow formatting exactly or the ML software will reject the feature vectors!)

Your program must produce two output files: one file containing feature vectors for the training instances, and one file containing feature vectors for the test instances. Each line should represent a feature vector for one instance (word). Unlike the input files, there should <u>not</u> be any blank lines. The format of each line should be:

label feature_id:1 feature_id:1 ...

1. There should be exactly one space between *label* and *feature_id:1* and between each pair of adjacent features.
2. 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:

   Label 0 for *O*
   Label 1 for *B-PER*
   Label 2 for *I-PER*
   Label 3 for *B-LOC*
   Label 4 for *I-LOC*
   Label 5 for *B-ORG*
   Label 6 for *I-ORG*
3. The ML tool also requires that each feature be represented by an integer value > 0. So your program will need to assign a numeric identifier to each type of feature.

   Important: The same feature ids must be used for both the training and test instances! For example, if the feature "prev-NN" is assigned the id 22, then id 22 should be used for "prev-NN" throughout ALL training <u>and</u> test instances.

   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.
4. The number after the colon means that the feature has a value of 1. For this task, we are assuming that all features have a binary value, where 1 indicates the presence of the feature and 0 indicates the absence of the feature. **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.

Debugging Information

To help you, and us, confirm that your **ner** program is working correctly, please have it print the following information to standard output:

```
Found N training instances with X distinct words and Y distinct POS tags.
Found M test instances.
```

For example (this example is just for illustration, the numbers are not real!):

```
Found 12 training instances with 105 distinct words and 9 distinct POS tags.
Found 10 test instances.
```

Feature Vector Examples

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

For the **wordcap** *feature type*, 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 3 pseudo-words mentioned earlier[1]. Also, you need one binary capitalization feature. The full feature set is shown below:

ID	Feature	ID	Feature	ID	Feature
1	curr-Israel	2	curr-television	3	curr-rejected
4	curr-the	5	curr-skit	6	curr-by
7	curr-comedian	8	curr-Tuvia	9	curr-Tsafir
10	curr-PHI	11	curr-OMEGA	12	curr-UNKWORD
13	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	**Wordcap Feature Vector** (Label Features...)
Israel	3 1:1 13:1
television	0 2:1
rejected	0 3:1
the	0 4:1
skit	0 5:1
by	0 6:1
the	0 4:1
comedian	0 7:1
Tuvia	1 8:1 13:1
Tzafir	2 9:1 13:1

For the **bothcon** *feature type*, first you would generate all possible word features, POS context features, lexical context 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 in the current position, previous position, and following position. There are 5 distinct part-of-speech tags, so you must create a feature for each one in the previous position and the following position. And you should include the 3 pseudo-words/tags mentioned earlier[2]. Also, you need one binary capitalization feature. The full feature set is shown below:

[1]Actually, PHI and OMEGA would never be the current word, but we include them here for completeness.

[2]Again, we'll consider all combinations here for completeness, even though some will never be seen. For example, OMEGA will never precede another word.

ID	Feature	ID	Feature	ID	Feature
1	curr-Israel	2	prev-Israel	3	next-Israel
4	curr-television	5	prev-television	6	next-television
7	curr-rejected	8	prev-rejected	9	next-rejected
10	curr-the	11	prev-the	12	next-the
13	curr-skit	14	prev-skit	15	next-skit
16	curr-by	17	prev-by	18	next-by
19	curr-comedian	20	prev-comedian	21	next-comedian
22	curr-Tuvia	23	prev-Tuvia	24	next-Tuvia
25	curr-Tzafir	26	prev-Tzafir	27	next-Tzafir
28	curr-PHI	29	prev-PHI	30	next-PHI
31	curr-OMEGA	32	prev-OMEGA	33	next-OMEGA
34	curr-UNKWORD	35	prev-UNKWORD	36	next-UNKWORD
37	prev-NNP	38	next-NNP	39	prev-NN
40	next-NN	41	prev-VBD	42	next-VBD
43	prev-DT	44	next-DT	45	prev-IN
46	next-IN	47	prev-PHIPOS	48	next-PHIPOS
49	prev-OMEGAPOS	50	next-OMEGAPOS	51	prev-UNKPOS
52	next-UNKPOS	53	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	**Bothcon Feature Vector** (Label Features...)
Israel	3 1:1 6:1 29:1 40:1 47:1 53:1
television	0 2:1 4:1 9:1 37:1 42:1
rejected	0 5:1 7:1 12:1 39:1 44:1
the	0 8:1 10:1 15:1 40:1 41:1
skit	0 11:1 13:1 18:1 43:1 46:1
by	0 12:1 14:1 16:1 39:1 44:1
the	0 10:1 17:1 21:1 40:1 45:1
comedian	0 11:1 19:1 24:1 38:1 43:1
Tuvia	1 20:1 22:1 27:1 38:1 39:1 53:1
Tzafir	2 23:1 25:1 33:1 37:1 50:1 53:1

The Machine Learning Tool

Installation: The machine learning software is available in the Files folder for program #1 on Canvas as: `liblinear-1.93.tar.gz`. To install it, first unpackage it (`tar xvfz liblinear-1.93.tar.gz`). In the directory with the extracted files, you will find a file named "make". Set the executable bit on this file ("`chmod a+x make`"). Then execute it by typing: `make`.

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

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

```
train -s 0 <train_data> <model_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 `model_name`.

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

```
predict <test_data> <model_name> <predictions_file> > <accuracy_file>
```

The `test_data` file should contain the feature vectors for the test data, and the `model_name` is the file name of the trained classifier. 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 train.txt.poscon model.poscon
predict test.txt.poscon model.poscon predictions.txt > accuracy.txt
```

Example

When your 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 poscon
   ```

2. Train a ML classifier using the training instances:

   ```
   train -s 0 train.txt.poscon model.poscon
   ```

3. Use the ML classifier to label the test instances:

   ```
   predict test.txt.poscon model.poscon predictions.txt > accuracy.txt
   ```

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. 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

   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.