**CS-5340/6340, Programming Assignment #3**
**Due: Monday, October 29, 2018 by 11:59pm**

Your task is to implement the Collins & Singer bootstrapping algorithm for Named Entity Recognition (NER). Your NER program should accept three input files: (1) a decision list of seed spelling rules, (2) a training data file, and (3) a test data file. Your program should accept these files as command-line arguments in the following order:

*ner <seed_rules> <training_data> <test_data>*

---

## Training and Test Data Files

The training data file and the test data file will have exactly the same format. Each file will consist of CONTEXT/NP pairs. We will refer to a CONTEXT/NP pair as an **instance**. For named entity recognition, the goal is to assign a label (e.g., LOCATION, ORGANIZATION, or PERSON) to an instance, although some instances may remain unlabeled.

The data file will be formatted like this:

CONTEXT: <words>
NP: <words>

CONTEXT: <words>
NP: <words>

CONTEXT: <words>
NP: <words>

etc.

You can assume that each CONTEXT and each NP will be on a single line. Each CONTEXT: and NP: keyword will be followed by one or more words. There will be one or more blank lines between each instance (CONTEXT/NP pair). Here is what a sample data file might look like:

CONTEXT: subsidiary of
NP: Arsenal Holdings Inc

CONTEXT: shares of
NP: Capital Food Services Ltd

CONTEXT: supplier
NP: Noblee

CONTEXT: interest in
NP: Pancontinental Mining Ltd

CONTEXT: fields in
NP: Canada

Please retain the original case in these data files so that your comparisons *will be* case sensitive. The data has been tokenized for you, so do *not* do any additional tokenization (e.g., do not strip off punctuation marks or do any other modification of the input words).

---

## Rule Representation

Each rule will consist of 3 items:

1. A **type**: SPELLING or CONTEXT
2. The *Contains* **predicate** with an arbitrary string as its argument
3. A **class**: LOCATION, ORGANIZATION, or PERSON.

The characters –> will be used to separate the left-hand side of the rule from the right-hand side. The **type** and **predicate** make up the left-hand side of the rule and the **class** makes up the right-hand side of the rule. If the predicate, given the string as its argument, matches the designated part of an instance (either the CONTEXT for context rules, or the NP for spelling rules), then the instance should be assigned the class label.

Here are two sample rules:

SPELLING *Contains*(Mary) –> PERSON
CONTEXT *Contains*(south) –> LOCATION

The first rule will match an instance if the NP portion contains the word "Mary". These instances will be labeled as a PERSON. For example, NPs such as "Mary", "Mary Smith", and "Dr. Mary Jones" would all be assigned the label PERSON.

The second rule will match an instance if the CONTEXT portion contains the word "south", and these instances will be labeled as a LOCATION. For example, contexts such as "south", "south of", and "near the south end" would all be assigned the label LOCATION.

**Important:** Case should be preserved when matching strings! For example, "City" should not match "city".

### Seed Rules File

The seed rules file will contain spelling rules, one rule per line. For example, a seed rules file might look like this:

SPELLING *Contains*(University) –> ORGANIZATION
SPELLING *Contains*(City) –> LOCATION
SPELLING *Contains*(Mr.) –> PERSON

### The Bootstrapping Algorithm

You should implement your program so that it has two phases:

1. **Training Phase:** Your program should run the Collins & Singer bootstrapping algorithm described on the lecture slides to learn decision lists from the training instances.
2. **Test Phase:** Your program should use the FINAL DECISION LIST (see below) to assign named entity class labels to the test instances. If no rules apply to an instance, then you should leave the instance unlabeled (assign it the class NONE).

You should use the following values and parameters:

- Run the bootstrapping algorithm for 3 iterations. During each iteration you should generate one new set of Context Rules and one new set of Spelling Rules.
- When selecting rules for the decision list, the rule must have a probability $\geq .80$ and a frequency $\geq 5$. Any rules that do not pass those thresholds should be discarded.
- For each rule learning step, you should add the 2 best rules for each class to the decision list. However, in some cases there may not be 2 rules that pass the thresholds listed above, in which case it is ok to add fewer than 2 rules. So for each learning step, you should add anywhere from 0 to 6 rules (2 per class) to the decision list. The rules should be sorted based on the Sorting Criteria described below!
- After 3 iterations, you should append the Context Rules to the end of the Spelling Rules to form the FINAL DECISION LIST. **Do NOT re-sort the rules!** The Seed Spelling Rules should appear at the beginning of the FINAL DECISION LIST (in their original order), followed by the learned Spelling Rules (in the order in which they were generated), followed by the learned Context Rules (in the order in which they were generated).[1]
- In the Test Phase, the rules in the FINAL DECISION LIST should then be applied to the test instances to assign a named entity class to each one.

[1]This is not what Collins & Singer did in their original implementation, but this approach will keep things simple for the assignment.

**Sorting Criteria:** To select the 2 best rules for each class and also to order the rules in the decision lists during bootstrapping, you should sort the rules using these criteria:

1. Sort by probability (higher probabilities are best)
2. If there are ties, then rank them by frequency (higher frequencies are best)
3. If there are still ties, then rank them alphabetically based on the string argument in the *Contains* predicate. As with everything else, this alphabetical sort should be case sensitive.

Please use this exact sorting procedure so that everyone's programs will produce the same output! As an example, here is a properly sorted CONTEXT decision list (this is just an example for illustration purposes, it is NOT the output that your program should generate based on the provided input files):

CONTEXT *Contains*(president) –> PERSON (prob=1.000 ; freq=7)
CONTEXT *Contains*(beach) –> LOCATION (prob=1.000 ; freq=6)
CONTEXT *Contains*(outside) –> LOCATION (prob=1.000 ; freq=6)
CONTEXT *Contains*(east) –> LOCATION (prob=1.000; freq=5)
CONTEXT *Contains*(west) –> LOCATION (prob=1.000; freq=5)
CONTEXT *Contains*(branch) –> ORGANIZATION (prob=0.957 ; freq=20)

---

IMPORTANT: Labeling Instances during Training

The example that I worked through in class has one important difference from this assignment. For the sake of simplicity during the lecture, the labels were assigned to instances using both Spelling and Context rules. BUT FOR THE ASSIGNMENT, you should ONLY use the labels produced by Spelling rules when learning Context rules, and ONLY use the labels produced by Context rules when learning Spelling rules. This is the procedure described on the lecture slides (Collins & Singer algorithm, NER slide #25). Step 3 produces labeled instances by applying the Spelling rules, and these labeled instances are used learn Context rules. Step 6 produces labeled instances by applying the Context rules, and these labeled instances are used to learn Spelling rules.

To implement this, I can imagine two options. (1) In Steps 3 and 6, re-label all of the training instances from scratch in each iteration using the appropriate rules. Or (2) maintain two copies of the training instances. One set would have labels produced by the Spelling Rules, and the other set would have labels produced by the Context Rules. Since new rules are appended to the end of a decision list, you can apply newly learned rules cumulatively.

## The Output

Your program should print the following items as output, with the exact format shown on the next page.

1. The Seed Rules
2. The Context Decision List for iteration #1
3. The Spelling Decision List for iteration #1
4. The Context Decision List for iteration #2
5. The Spelling Decision List for iteration #2
6. The Context Decision List for iteration #3
7. The Spelling Decision List for iteration #3
8. The FINAL DECISION LIST
9. The results of applying the FINAL DECISION LIST to the test data

---

Each rule should be formatted like this:

*type Contains*(<string>) –> *class* (prob=*num* ; freq=*num*)

Please print your probabilities with exactly 3 digits following the decimal point and use rounding. For example, the value 0.366666 should be printed as 0.367 . For the seed rules, use the probability value -1.000 and the frequency value -1. For example, here is how a seed rule should appear in the FINAL DECISION LIST:

SPELLING Contains(Ltd) -> ORGANIZATION (prob=-1.000 ; freq=-1)

A sample output file is available on Canvas – please look it over to make sure you understand how to print your output!

Your output should be formatted as follows:

SEED DECISION LIST
*<rules go here>*

ITERATION #1: NEW CONTEXT RULES>
*<rules go here>*

ITERATION #1: NEW SPELLING RULES
*<rules go here>*

ITERATION #2: NEW CONTEXT RULES
*<rules go here>*

ITERATION #2: NEW SPELLING RULES
*<rules go here>*

ITERATION #3: NEW CONTEXT RULES
*<rules go here>*

ITERATION #3: NEW SPELLING RULES
*<rules go here>*

FINAL DECISION LIST
*<rules go here>*

APPLYING FINAL DECISION LIST TO TEST INSTANCES

CONTEXT: *string*
NP: *string*
CLASS: *class*

CONTEXT: *string*
NP: *string*
CLASS: *class*

etc.

## SUBMISSION INSTRUCTIONS

Please use CANVAS to submit a gzipped tarball file named "ner.tar.gz". (This is an archived file in "tar" format and then compressed with gzip. Instructions appear below if you're not familiar with tar files.) Your tarball should contain the following items:

1. The source code files for your program. Be sure to include all files that we will need to compile and run your program!

   REMINDER: your program **must** be written in Python or Java, and it **must** compile and run on the Linux-based CADE (lab1 or lab2) machines! We will not grade programs that cannot be run on the Linux-based CADE machines.

2. An executable *shell script* named `ner.sh` that contains the exact commands needed to compile and run your NER program on the data files provided on CANVAS. We should be able to execute this file on the command line, and it will compile and run your code. For example, if your code is in Python and does not need to be compiled, then your script file might look like this:

   ```
   python ner.py seedrules.txt training.txt test.txt
   ```

   If your code is in Java, then your script file might look something like this:

   ```
   javac ner/*.java
   java ner/MainClass seedrules.txt training.txt test.txt
   ```

3. A `README.txt` file that includes the following information:

   - Which CADE machine you tested your program on
     (this info may be useful to us if we have trouble running your program)
   - Any known idiosyncracies, problems, or limitations of your program.

4. Submit a trace file called **ner.trace** that shows the output of your program when using the data files available on CANVAS.

---

## GRADING CRITERIA

Your program will be graded based on new input 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 input. **Please** exactly follow the formatting instructions specified in this assignment. We will deduct points if you fail to follow the specifications because it makes our job much more difficult to grade programs that do not conform to the same standards.

**IMPORTANT:** You may not use ANY external software packages or dictionaries to complete this assignment except for *basic* libraries. For example, libraries for general I/O handling, math functions, and regular expression matching are ok to use. But you may not use libraries or code from any NLP-related software packages, or external code that performs any functions specifically related to named entity recognition. All submitted code *must be your own.*

## HELPFUL HINTS

**TAR FILES:** First, put all of the files that you want to submit in a directory called "ner". Then from the parent directory where the "ner" folder resides, issue the following command:

tar cvfz ner.tar.gz ner/

This will put everything underneath the "ner/" directory into a single file called "ner.tar.gz". This file will be "archived" to preserve any structure (e.g., subdirectories) inside the "ner/" directory and then compressed with "gzip".

FYI, to unpack the gzipped tarball, move the ner.tar.gz to a new location and issue the command:

tar xvfz ner.tar.gz

This will create a new directory called "ner" and restore the original structure and contents.

For general information on the "tar" command, this web site may be useful:
https://www.howtogeek.com/248780/how-to-compress-and-extract-files-using-the-tar-command-on-linux/

**TRACE FILES:** You can generate a trace file in (at least) 2 different ways: (1) print your program's output to standard output and then pipe it to a file (e.g., `ner seedrules.txt training.txt test.txt > ner.trace`), or (2) print your output to standard output and use the unix *script* command before running your program on the test files. The sequence of commands to use is:

```
script ner.trace
ner seedrules.txt training.txt test.txt
exit
```

This will save everything that is printed to standard output during the session to a file called `ner.trace`.