cs294-5: Statistical Natural Language Processing
Assignment 2: Part-of-Speech Tagging

Due: Oct 8th

Setup: This assignment uses the same code base as assignment 1, but since some people may still be working on assignment 1, this code is in a `src2` directory sister to the old `src`. Similarly, the precompiled classes are in `classes2` instead of `classes`. The data for this assignment is all in `/home/ff/cs294-5/corpora/assignment2`, though you'll notice that some of the data is repeated from assignment 1.

The starting files for this assignment are

```
…/assignments/MaximumEntropyClassifierTester.java
…/assignments/POSTaggerTester.java
```

Make sure you can read the source and data files.

Part 1a. A Maximum Entropy Classifier: Look through the code in `MaximumEntropyClassifierTester.java`. This class contains several subclasses. The most important two are:

`MaximumEntropyClassifier` (implements `classify.ProbabilisticClassifier`)
`MaximumEntropyClassifierFactory` (creates the former)

Look at the main function for the data flow. There are two modes you can run this main method in. If you supply "`-mini`" as the first command line argument, you'll get a miniature classification problem from `miniTest()`. I recommend working with this branch first, since it's easier to debug. We create several training datums (and a test datum), which are either cats or bears, and which have several features each. These training datums are passed to a `MaximumEntropyClassifierFactory` which uses them to learn a `MaximumEntropyClassifer`. This classifier is then applied to the test set, and an accuracy (and distribution over labels) is printed out.

To start out, the whole classification pipeline runs, but there's no maximum entropy involved. You'll have to fill in two chunks of code (marked by "TODO" lines) to turn the placeholder code into a maximum entropy classifier. First, look at

```
MaximumEntropyClassifier.getLogProbabilities()
```

This method takes a datum, and produces the (log) distribution, according to the model, over the various possible labels. There will be some interface shock here, because you're looking at a method buried deeply in my implementation of the rest of the classifier. You are given several arguments, whose classes are defined in this same java file:

```
EncodedDatum datum
Encoding<F,L> encoding
IndexLinearizer indexLinearizer
double[] weights
```

The `EncodedDatum` represents the input datum. It is a sparse encoding, which tells you which features were present in that datum, and with what counts. When you ask an `EncodedDatum` what features are present, it will return feature **indexes** instead of feature objects – for example it might tell you that feature 121 is present with count 1.0 and feature 3317 is present with count 2.0. If you want to recover the original (String) representation of those features, you'll have to go through the Encoding, which maps between features and feature indexes. Encodings also manage maps between labels and label indexes. So while your `miniTest()` labels are "cat" and bear," the Encoding will map these to indexes 0 and 1, and your returned log distribution should be a double array indexed by 0 and 1, rather than a hash on "cat" and "bear".

So, you first have a feature ("fuzzy") and a label ("cat") which map to some feature index (say 2) and some label index (say 0). As outlined above, these are managed by the `Encoding`. The job of the `getLogProbabilities()` method is to return an array of doubles, where the indexes are the label indexes, and the entries are the log probabilities of that label, given the current datum. To do this, you will need to properly combine the voting weights (lambdas from lecture). The double vector `weights` contains these vote weights linearized into a one-dimensional array. To find the voting weight for the feature "fuzzy" and the label "cat," you'll need to take their indexes (2 and 0) and use the IndexLinearizer to find out what index in `weights` to use.

Your job here is to correctly fill in and return an array of log probabilities. Try to do this as efficiently as possible – this is the inner loop of the classifier training. Indeed, the reason for all this primitive-type array machinery is to minimize the amount of time it'll take to train large maxent classifiers.

Run the mini test again. Now that it's actually voting properly, you won't get a 0/1 distribution anymore – you'll get 50/50, because, while it is voting now, the weights are all zero. The next step is to fill in the weight estimation code. Look at

```
Pair<Double, double[]> calculate(double[] x)
```

buried all the way in

```
MaximumEntropyClassifierFactory.MaximumEntropyClassifier.ObjectiveFunction
```

This method takes a vector `x`, which is some proposed weight vector, and calculates the negative data conditional likelihood

$$F(\vec{\lambda}) = -1 \cdot \left[\sum_{(c,d)} \log P(c \mid d, \vec{\lambda}) \right]$$

$$P(c \mid d, \vec{\lambda}) = \frac{e^{\sum_i \lambda_i(c) f_i(d)}}{\sum_{c'} e^{\sum_i \lambda_i(c') f_i(d)}}$$

and a vector of the derivatives of this log likelihood:

$$\frac{\partial F(\vec{\lambda})}{\partial \lambda_i(c)} = -1 \cdot \left[\left[\sum_{(c,d) \in (C,D)} f_i(d) \right] - \left[\sum_{d \in D} P(c \mid d, \vec{\lambda}) f_i(d) \right] \right]$$

Recall that the left sum is the number of times the feature *i* actually occurs in examples with true class *c* in the training, while the right sum is the expectation of the same quantity using the label distributions the model predicts.

The current code just says that the objective if 42 and the derivatives are flat. Note that you don't have to guess at x – that's the job of the optimization code. All you have to do is evaluate proposed x vectors. In scope are the data, the String-to-index encoding, and the linearizer from before:

```
EncodedDatum[] data;
Encoding encoding;
IndexLinearizer indexLinearizer;
```

Write code to calculate the objective and its derivatives, and return the Pair of those two quantities.

Run the `miniTest()` again. This time, the optimization should find a good solution, one that puts all of the mass onto the correct answer "cat."

Almost done! Remember that probability one on cat is probably the wrong behavior here. To smooth, or regularize, our model, we're going to modify the objective function to penalize large weights. In `calculate()`, you should now add code which adds the following to the objective

$$G(\vec{\lambda}) = F(\vec{\lambda}) + \sum_{i,c} \frac{\lambda_i(c)^2}{2\sigma^2}$$

and the derivatives change by the corresponding amounts

$$\frac{\partial G(\vec{\lambda})}{\partial \lambda_i(c)} = \frac{\partial F(\vec{\lambda})}{\partial \lambda_i(c)} + \frac{\lambda_i(c)}{\sigma^2}$$

Run the `miniTest()` one last time. You should now get less than 1.0 on "cat" (0.73 with the default `sigma`).

Part 1b. Using a Maximum Entropy Classifier: Now that your classifier works, goodbye `miniTest()`! Run the main method with a single argument of

```
/home/ff/cs294-5/corpora/assignment2
```

or wherever you may have copied the data. It now loads the proper noun phrase data from Assignment 1, converts each data instance into a list of String features, one for each character unigram in the name. So "Xylex" will become

```
["X", "y", "l", "e", "x"]
```

This should train relatively quickly (should be no more than a few minutes, possibly tens of seconds, and you can reduce the number of iterations for quick tests). It won't work well, though – which is unsurprising. You should get an accuracy of 63.7% using the default amount of smoothing (`sigma` of 1.0) and 40 iterations. This maxent classifier has the same information available as a class-conditional unigram model, though it'll probably work a little better.

Your job here is to flesh out the feature extraction code in

```
MaxiumEntropyClassifierTester.extractFeatures(List<Character> characters)
```

You can take that list of characters and create any String features you want, such as `"BI-Xy"` indicating the presence of the bigram "Xy". Or `"Length<10", "Length=5", "WORD-Xylex."` If you want bigrams (or longer n-grams), you might want to use a `util.BoundedList` to wrap the input list, which lets you ask for list items outside the list's range. Any descriptor of an aspect of the input that seems relevant is fair game (though add feature classes gradually so you can judge how slow you're making your training). Better indicators should raise the accuracy of the classifier. You should easily be able to get your classification accuracy over 70%, and possibly over 90% (a lot harder).

Part 2. The World's Worst POS Tagger: OK, file away your maxent classifier for now. Run the other test harness, `assignments.POSTaggerTester`. This is a fully functional POS tagger, but with some minimalist components. Its main method loads the standard Penn Treebank part-of-speech data set, split in the standard way into training, validation, and test sentences. The current code reads through the training, extracting counts of which tags each word type occurs with. It also extracts a Kneser-Ney style

count over "unknown" words see if you can figure out what it's doing and why. The current code then ignores the validation entirely, and gives each known word in the test data the tag which it occurred most frequently with in training. Unknown words get the tag which had the most types in training. This tagger operates at about 92%, with a rather pitiful unknown word accuracy of 40%. Your job is to make a real tagger out of this one by upgrading several placeholder implementations.

Part 2a. A Better Sequence Model: Look at the main method – the `POSTagger` is constructed out of two components, the first of which is a `LocalTrigramScorer`. This scorer takes `LocalTrigramContexts` and produces a Counter mapping tags to their scores in that context. A `LocalTrigramContext` encodes a sentence, a position in that sentence, and the previous two tags. The dummy scorer ignores the previous tags, looks at the word at the current position, and returns a (log) conditional distribution over tags for that word:

$$\log P(t \mid w)$$

Therefore, the best-scoring tag sequence will be the one which maximizes the quantity:

$$\sum_i \log P(t_i \mid w_i)$$

If you wanted to build a trigram HMM tagger, you would instead want to maximize the quantity

$$\sum_i \log\left[P(t_i \mid t_{i-1}, t_{i-2}) P(w_i \mid t_i)\right]$$

which means the local scorer would have to return

$$\log P(t_i \mid t_{i-1}, t_{i-2}) P(w_i \mid t_i)$$

for each context. (Note that this is NOT a log distribution over tags). If you want to implement an MEMM, you want to maximize the quantity

$$\sum_i \log P(t_i \mid t_{i-1}, t_{i-2}, \vec{w}, i)$$

which means that you will want to build a little maximum entropy model which predicts tags in this context, based on features of this context:

$$\log P(t_i \mid t_{i-1}, t_{i-2}, \vec{w}, i)$$

(Note that this IS a distribution over tags). You can do either. If you choose to build the HMM, you should do something sensible for unknown words, using a technique like suffix trees, maximum-entropy models, or at least a well-smoothed estimate of P(UNK |

tag). Warning: the maxent model may be very slow to train, especially if you add too many feature schemas!

Whichever model you choose to build, your local scorer should use the provided interface for training and validating. The assignment doesn't require that you use the validation data, but it's there if you want it.

Part 2b. A Better Decoder: With your improved scorer, your results should have gone up substantially. However, you may have noticed that the tester is complaining about decoder sub-optimalities. This is because of the second ingredient of the `POSTagger`, the decoder. The supplied implementation is a greedy decoder (equivalent to a beam decoder with beam size 1). Your final task in this assignment is to write a Viterbi decoder. Decoders implement the `TrellisDecoder` interface, which takes a `Trellis` and produces a path. Trellises are really just directed, acyclic graphs, whose nodes are states in a markov model and whose arcs are transitions in that model, with weights attached. In this concrete case, those states are `State` objects, which encode a pair of preceding tags and a position in the sentence. The weights are scores from your local scorer. But in this part of the assignment, it really doesn't matter where the Trellis came from. Take a look at the `GreedyDecoder`. It starts at the `Trellis.getStartState()` state, and walks forward greedily. Your decoder will still return a list of states in which the first state is that start state and the last is the end state, but will return the sequence of least sum weight (weights are log probabilities produced by your scorer). A necessary (but not sufficient) condition for your Viterbi decoder to be correct is that the tester should show no decoder sub-optimalities (these are cases where your model scored the correct answer better than the decoder's choice). As a target, accuracies of 94+ are good, and 96+ is basically state-of-the-art. (Note: if you want to write your decoder before your scorer, you can construct the `MostFrequenctTagScorer` with an argument of true, which will cause it to restrict paths to tag trigrams seen in training – this boosts scores slightly and, more importantly, exposes flaws in the greedy decoder.)

Write-up: For the write-up, I mainly just want you to describe what you've built. For the maxent model, you should mention what feature schemas you used and how well they worked. For the tagger, you should look through the errors and tell me if you can think of any ways you might fix them (whether you do fix them or not). Pay special attention to unknown words – in practice it's the unknown word behavior of a tagger that's most important.

Coding Tip: For your maximum entropy classifier, you will likely find that your code spends all of its time taking logs and exps. You can often avoid a good amount of this work using the `logAdd(x,y)` and `logAdd(x[])` functions in `math.SloppyMath`.