CS 288: Statistical NLP
Assignment 2: Proper Noun Classification

Due 2/17/10

Setup: Download the code and data zips from the web page (the class code is unchanged from the first assignment if you want to use your old copy). Make sure you can still compile the entirety of the course code without errors. The Java file to start out inspecting for this project is:

```
edu/berkeley/nlp /assignments/ProperNameTester.java
```

Try running it with:

```
java edu.berkeley.nlp.assignments.ProperNameTester
    -path DATA -model baseline -test validation
```

Here, `DATA` is wherever you unzipped the data zip to. If everything's working, you'll get some output about the performance of a baseline proper name classifier being tested.

Proper Name Classification: Proper name classification is the task of taking proper names like *Eastwood Park* and deciding whether they are places, people, etc. In the more general task of named entity recognition (NER), which we do not consider here, one also has to detect the boundaries of the phrases. In general, we might use a variety of cues to make these kinds of decisions, including looking at the syntactic environment that the phrases occur in, whether or not the words inside the phrase occur in lists of known names of various types (gazetteers), and so on. In this assignment, however, you will write classifiers which attempt to classify proper names purely on the basis of their surface strings alone. This approach is more powerful than you might think: for example, there aren't too many people named *Xylex*. Since even the distribution of characters is very distinctive for each of these categories, we will start out trying to make classification decisions on the basis of character n-grams alone (so, for example, the suffix *-x* may indicate drugs while *-wood* may indicate places.

Start out by looking at the main method of `ProperNameTester.java`. It loads training, validation, and test sets, which are lists of `LabeledInstance` objects. These instance objects each represent a proper noun phrase string (such as *Eastwood Park*) gotten using `getInput()` along with a label gotten with `getLabel()`. The labels are one of five label strings: PLACE, MOVIE, DRUG, PERSON, or COMPANY. A classifier is then trained and tested. To start out, we have the default `MostFrequentLabelClassifier`, which always chooses MOVIE since it is the most common label in the training set. In this assignment, you will build better classifiers for this task, first a generative classifier based on an n-gram language model, then a discriminative classifier using a

maximum entropy model.

Part 1: Class-Conditional N-Gram Models: The first approach will be to construct a character-level language model $P(x|y)$ for each class y, placing distributions over strings x. You should be able to do this very rapidly (i.e. in under an hour) using your code from assignment 1. Since there are many fewer characters than words, smoothing won't be as critical and higher order models can be used more easily. Once you have models of $P(x|y)$, you can score each class with the posterior $P(y|x)$ using Bayes' rule. Note that if your language model is a unigram model, this approach is simply multinomial nave-Bayes. The minimal requirement of this part, as far as building models, is to build at least two of the following and compare: a character unigram model, a word unigram model, and a character bigram model (or something better). There is a hook for you to insert your new models in the `main` method.

Your primary goal is not to get the best possible accuracy, but to understand what your classifiers are doing well, what they're doing badly, and why. Augment the provided evaluation code with two kinds of functionality. First, have it produce a confusion matrix, showing which label pairs are most often confused. What are your main errors? Are these errors surprising or reasonable? Why do you think some pairs are easier or harder? Second, have your evaluation code investigate how your classifier's confidence correlates with accuracy. Are more confident (higher posterior likelihood) decisions more likely to be correct? Why or why not? Are the models calibrated, meaning that, for example, guesses which are made with confidence about 70% are correct about 70% of the time? Are there any individual errors that are particularly revealing as to why or how the system makes errors? Cases where you as a human would have trouble? I'm more interested in your observations of what's going wrong than whether you can engineer a better n-gram model (that was assignment 1!). For fun: for each category, sample from your models. Do they produce reasonable names of each kind?

Part 2: A Maximum Entropy Classifier: Next, you will build a feature-driven maximum entropy classifier for the same task. We'll start with building the classifier itself, which is in

```
assignments/MaximumEntropyClassifier.java
```

Forget about proper name identification for the moment, and look at the main method of this class. It runs a miniature problem which you will use to debug your work as you flesh out the classifier, which currently has some major gaps in its code. In the toy problem, we create several training instances (and one test instance), which are either cats or bears, and which have several features each. These training instances are passed to a `MaximumEntropyClassifier.Factory` which uses them to learn a `MaximumEntropyClassifier`. This classifier is then applied to the test set, and a distribution over labels is printed out.

Part 2a: To start out, the whole classification pipeline runs, but there's no maximum entropy classification involved. You'll have to fill in two chunks of code (marked by `TODO` lines) in order to turn the placeholder code into a maximum entropy classifier. First, look at

```
MaximumEntropyClassifier.getLogProbabilities()
```

This method takes an instance (as an `EncodedDatum`), 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. This situation won't be typical for this course, but for here it's necessary to ensure your classifiers are efficient enough for future assignments. In the present method, 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 instance represented as a feature vector. It is a sparse encoding which tells you which features were present for that that instance, 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. `Encoding`s also manage maps between labels and label indexes. So while your test 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*.

Once you've gotten a handle on the encodings, you should flesh out `getLogProbabilities()`. It should 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 instance and weights. To do this, you will need to properly combine the model weights (w from lecture). The double vector weights contains these weights linearized into a one-dimensional array. To find the weight for the feature "fuzzy" on the label *cat*, you'll need to take their individual indexes (2 and 0) and use the `IndexLinearizer` to find out what joint index in `weights` to use for that pair.

Try to do this calculation of log scores 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, which would be very time consuming with a friendlier collection-based implementation.

Part 2b: Once you've gotten the code set to calculate the log scores, run the mini test again. Now that it's actually voting properly according to the model weights, you won't get a 0.0/1.0 distribution anymore - you'll get 0.5/0.5, 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[] w)
```

buried all the way in

```
MaximumEntropyClassifier.ObjectiveFunction
```

This method takes a vector `w`, which is some proposed weight vector, and calculates the (negative) log conditional likelihood of the training labels (y) given the weight vector (w) and examples

(x):

$$L(w) = \sum_i log P(y_i|x_i, w)$$

where

$$P(y|x, w) = \frac{e^{w_y \cdot f(x)}}{\sum_{y'} e^{w_{y'} \cdot f(x)}}$$

Note that here we are using the block formulation where the features are defined over the examples and there is a weight vector for each class; make sure you understand how this is equivalent to the non-block general case!

Your code will have to compute both L and its derivatives:

$$\frac{\partial L}{\partial w_y} = \sum_i I(y_i = y) f(x_i) - \sum_i P(y|x_i, w) f(x_i)$$

Verify these equations for yourself. Recall that the left sum is the total feature count vector over examples with true class y in the training, while the right sum is the expectation of the same quantity, over all examples, using the label distributions the model predicts. To sanity check this expression, you should convince yourself that if the model predictions put mass 1.0 on the true labels, the two quantities will be equal.

The current code just says that the objective is 42 and the derivatives are flat. Note that you don't have to write code that guesses at w - that's the job of the optimization code, which is provided. All you have to do is evaluate proposed weight vectors. In scope are the data, the string-to-index `encoding`, and the `linearizer` from before:

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

You should now write new code to calculate the objective and its derivatives, and return the Pair of those two quantities. Important point: because you wish to maximize L and your optimizer is a minimizer, you will have to negate the computed values and derivatives in the code.

Run the mini test again. This time, after a few iterations, the optimization should find a good solution, one that puts all or nearly all of the mass onto the correct answer, "cat."

Part 2c: Almost done! Remember that predicting 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 a penalty to get a new objective:

$$L'(w) = L(w) - \frac{1}{2\sigma^2} ||w||_2^2$$

The derivatives then change by the corresponding amounts, as well.

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

Part 3: Using the Maximum Entropy Classifier: Now that your classifier works, goodbye mini test! Return to the proper name classification code and invoke it with

```
java edu.berkeley.nlp.assignments.ProperNameTester
    -path DATA -model maxent -test validation
```

It now trains a maxent classifier using your code. The `ProbabilisticClassifier` interface takes a string name and returns a string label. To actually use a feature-based classifier like our maxent classifier, however, we need to write code which turns the name into a vector of feature values. This code is a `FeatureExtractor`, which is defined in

```
ProperNameTester.ProperNameFeatureExtractor
```

This extractor currently converts each name string into a Counter of String features, one for each character unigram in the name. So "Xylex" will become

```
["X" : 1.0, "y" : 1.0, "l" : 1.0, "e" : 1.0, "x" : 1.0]
```

The classifier should train relatively quickly using these unigram features (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 very well, though perhaps better than you might have thought. You should get an accuracy of 63.7% using the default amount of smoothing (sigma of 1.0) and 40 iterations. This classifier has the same information available as a class-conditional unigram model. If you built a class-conditional unigram model in part 1, which worked better? Why?

Your final task is to flesh out the feature extraction code by improving the feature extractor. You can take that input name and create any String features you want, such as `BIGRAM-Xy` indicating the presence of the bigram *Xy*. Or `Length<10`, `Length=5`, `WORD=Xylex`, or `FIRST-LETTER-SAME-AS-LAST-LETTER` – whatever you can think of. (If you want bigrams or longer n-grams, you might find `util.BoundedList` useful - it lets you ask for list items without worrying about a 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 much you're slowing down your training). Better indicators should raise the accuracy of the classifier. You should be able to get your classification accuracy over 70% (very easy), and possibly over 90% (harder). Which features are most useful? Is this classifier calibrated better or worse than the generative ones? What errors remain in your best classifier and why? Do you think this approach would work for language identification?

Random Advice: When you invoke java, you may want to (a) increase the maximum heap size with the `-mx` option (e.g. `-mx512m`) and (b) run the JVM in server model with the `-server` option. You may (or may not) see substantial speed-ups from both.