1 Setup

We've put everything for this assignment (both Java starter code and data files) in the directory /afs/ir/class/cs224n/hw3/. The Java starter code is in hw3/java/, and the data is in hw3/data/. Copy the starter code to your local directory, and make sure you can compile it. Spend some time looking through the principal source files for this assignment:

src/cs224n/assignments/MaximumEntropyClassifierTester.java
src/cs224n/assignments/POSTaggerTester.java

You can do a quick test run for the first one using the command

java cs224n.assignments.MaximumEntropyClassifierTester -mini

A test for the second one will take more memory and time:

java -Xmx250m cs224n.assignments.POSTaggerTester /afs/ir/class/cs224n/hw3/data

Take a few minutes to look through the data directories, too.

2 A Maximum Entropy Classifier

2.1 Building a simple maxent 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 method to see the overall program 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 the miniTest() method. I recommend working with this branch
first, since it’s easier to debug. `miniTest()` creates several training datums and a test datum. Each datum represents either a cat or a bear and has several features (which are just strings). The training data are passed to a `MaximumEntropyClassifierFactory` which uses them to learn a `MaximumEntropyClassifier`. This classifier is then applied to the test data, and an accuracy (and distribution over labels) is printed out.

While the starter code contains a fully-functioning pipeline for training and testing a classifier, the classifier it builds is as dumb as a rock, and does not use maximum entropy. Your job is to turn the placeholder code into a maximum entropy classifier by filling in the two chunks of code marked by “// TODO” lines. First, look at

```
MaximumEntropyClassifier.getLogProbabilities()
```

This method takes a datum, and produces the (log) distribution, according to the model, over the possible labels for the datum. There will be some interface shock here, because you’re looking at a method buried deep inside the 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 feature weights (the \( \lambda s \)). The double vector `weights` contains these feature weights linearized into a one-dimensional array. To find the 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.

(Here’s a tip: you will likely find that your code spends much 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`.)

Run `miniTest()` 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 value of the objective function we’re trying to optimize in training, along with its derivatives. Our objective function is the negative conditional log likelihood of the training data \(\langle C, D \rangle\):

\[
F(\vec{\lambda}) = -1 \cdot \left[ \sum_{\langle c, d \rangle \in \langle C, D \rangle} \log p(c \mid d, \vec{\lambda}) \right]
\]

\[
p(c \mid d, \vec{\lambda}) = \frac{\exp \left[ \sum_{i} \lambda_{i} f_{i}(c, d) \right]}{\sum_{c'} \exp \left[ \sum_{i} \lambda_{i} f_{i}(c', d) \right]}
\]

(Note that the \(i\)s here correspond to the indexes generated by the \texttt{IndexLinearizer}.) The derivatives of the log likelihood are therefore:

\[
\frac{\partial F(\vec{\lambda})}{\partial \lambda_i} = -1 \cdot \left[ \sum_{\langle c, d \rangle \in \langle C, D \rangle} f_{i}(c, d) \right] - \left[ \sum_{\langle c, d \rangle \in \langle C, D \rangle} \sum_{c'} p(c' \mid d, \vec{\lambda}) f_{i}(c', d) \right]
\]

Recall that the left summation 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 is 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 in the \texttt{calculate()} method is evaluate proposed \(x\) vectors. You have available as method arguments the data, the string-to-index encoding, and the linearizer we discussed before:

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

Write code to calculate the objective function and its derivatives, and return the \texttt{Pair} of those two quantities.

Now run \texttt{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 putting probability 1.0 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 the \texttt{calculate()} method, you should now add code which modifies the objective function as follows:

\[
G(\vec{\lambda}) = F(\vec{\lambda}) + \sum_{i} \frac{\lambda_{i}^2}{2\sigma^2}
\]

The derivatives change correspondingly:
\[
\frac{\partial G(\lambda)}{\partial \lambda_i} = \frac{\partial F(\lambda)}{\partial \lambda_i} + \frac{\lambda_i}{\sigma^2}
\]

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

### 2.2 Feature engineering for a maxent classifier

Now that your classifier works, goodbye `miniTest()`! Run the `main` method with a single argument of

```
/afs/ir/class/cs224n/hw3/data/
```

or wherever you may have copied the data. It now loads the proper noun phrase data from the data directory and 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

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

You can take that list of characters and create any `String` features you want, such as "BI-Xy" to indicate 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 as high as 90% (a lot harder).

### 3 Part of Speech Tagging

OK, now let’s turn our attention to the other test harness, `assignments.POSTaggerTester`. This is a fully functional POS tagger, but with some minimalist components. Its principal components are a scorer and a decoder. The scorer is essentially a classifier: it looks at a word and its local context, and assigns probabilities to the possible POS tags for the word. The decoder’s job is to use the probabilities given by the scorer to determine the most likely tag sequence for an entire sentence.

We’ve supplied bare-bones implementations of both the scorer and the decoder. The supplied scorer ignores context altogether, and just gives each known word in the test data the tag with
which it appeared most frequently in the training data. Unknown words get the tag which appeared with the most types in the training data. Meanwhile, the decoder just uses a simple greedy algorithm. While these implementations aren’t very clever, they’re sufficient to get an accuracy of about 92% overall. However, the accuracy on unknown words drops to a rather pitiful 40%. Your job is to extend your maximum entropy classifier to improve the tagging accuracy for unknown words.

Let’s take a closer look at the code in POSTaggerTester.java. Don’t freak out! The code may look overwhelming, but that’s partly because it’s intended to support more than you’re required to do in this assignment. (Specifically, the code is intended to facilitate implementation of an HMM model and the Viterbi algorithm — that’s why you’ll see references to trigrams and lattices in the code.) Fortunately, you don’t need to read or understand every bit of code in this file.

The main method loads the standard Penn Treebank part-of-speech data set, split in the standard way into training, validation, and test sets. The code then reads through the training data, extracting counts of which tags each word type occurs with. It also generates a Kneser-Ney style pseudo-count over “unknown” words — see if you can figure out what it’s doing and why. (Kneser-Ney smoothing is described on pp. 15–17 of the Chen & Goodman smoothing paper, linked from the website.) The current code ignores the validation data entirely — this is something you can change if you like.

In the main() method, you’ll see that the POSTagger is constructed out of two components, a scorer and a decoder. The scorer (here a MostFrequentTagScorer) needs to be an instance of LocalTrigramScorer. (This is a reflection of this harness code’s further purpose as a platform for an HMM implementation.) A LocalTrigramScorer takes a LocalTrigramContext and produces a Counter which maps tags to their scores in that context. A LocalTrigramContext encodes a sentence, a position in that sentence, and the previous two tags. The supplied MostFrequentTagScorer ignores the context (the previous tags). It just looks at the word at the current position, and returns a (log) conditional distribution over tags for that word: \( \log \mathbf{p}(t \mid w) \).

Your job is to build a replacement for MostFrequentTagScorer that does a better job at predicting tags for unknown words by exploiting the maximum entropy classifier code you wrote in the first part. (A good accuracy for unknown words might be something like 75% or 80%.) Your 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.)

4 Write-up

For the write-up, you should describe what you built, what choices you had to make, why you made the choices you did, how well they worked out, and what you might do to improve things further. In particular:

- In your implementation of the maxent model code, we’ll be looking first for correctness, and secondarily for efficiency.
- For both the proper noun semantic class model and the unknown word POS tagging model, an important criterion will be coming up with good and effective features for this task. Describe the feature templates you used, why you choose them, and how well they worked.
- Make sure you describe any data analysis and testing you did as part of finding good features, and show any revealing data analysis on your final results which shows what it is
good and bad at.

• For POS tagging, (unless you’re just bored and want to have fun) you should only be concerned with improving the accuracy of the model on unknown words. We are not expecting you to improve the accuracy of the tagger on known words.

5 Submitting the assignment

5.1 The program: electronic submission

Submission of the program code will be via a script. To submit your program, put all the needed files in one directory on a Sweet Hall machine (or, possibly, another machine with AFS on it and with you being correctly authenticated) and type:

/afs/ir/class/cs224n/bin/submit-hw3

If you need to resubmit it type

/afs/ir/class/cs224n/bin/submit-hw3 -replace

Make sure that you include all the source code for your programs. We will run programs on the Sweet Hall systems, so they should run without problems there. Please don’t include large data files in your submission. Your code doesn’t have to be beautiful but we should be able to scan it and figure out what you did without too much pain.

5.2 The report: turn in a hard copy

You should turn in a write-up of the work you’ve done, as well as the code. Your write-up must be submitted as a hard copy. There is no set length for write-ups, but a ballpark length might be 4 pages, including your evaluation results, a graph or two, and some interesting examples.