cs294: Statistical Natural Language Processing
Assignment 3: Part-of-Speech Tagging

Due: Feb 28th

In this assignment, you will build the important components of a part-of-speech tagger, including a local scoring model and a decoder.

Setup: The data for this assignment is available on the web page as usual. It uses the same Penn Treebank data as the first assignment, this time with the part-of-speech labels added in.

The starting class for this assignment is

```
edu.berkeley.nlp.assignments.POSTaggerTester
```

Make sure you can access the source and data files.

The World's Worst POS Tagger: Now run the test harness, `assignments.POSTaggerTester`. You will need to run it with the command line option `-path DATA_PATH`, where `DATA_PATH` is wherever you have unzipped the assignment data. This class by default loads a fully functional, if minimalist, POS tagger. The main method first loads the standard Penn Treebank WSJ part-of-speech data, split in the standard way into training, validation, and test sentences. The current code reads through the training data, extracting counts of which tags each word type occurs with. It also extracts a count over "unknown" words – see if you can figure out what its unknown word estimator is (it's not great, but it's reasonable). The current code then ignores the validation set entirely. On the test set, the baseline tagger then gives each known word its most frequent training tag. Unknown words all get the same tag (which, and why?). 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 each of its placeholder components.

Part 1. 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 a setting for two tags preceding that position. 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:

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

Your first job is to upgrade the local scorer. You have a choice between building either an HMM tagger or a maximum-entropy tagger. If you choose to build a trigram HMM tagger, you will 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

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

for each context. (Note that this is NOT a log distribution over tags). If you want to implement an MEMM tagger, you will instead be maximizing 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 these contexts, based on features of the contexts that you design. The local score has the form:

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

Note that this IS a log distribution over tags. You can built either type of tagger. Warning: a full-blown maxent tagger will be *very* slow to train, on the order of hours per run, especially if you add many feature templates, so start early and give yourself plenty of time to run experiments. An HMM will train faster (but likely have lower accuracy) – if you build an HMM, you should do something sensible for unknown words, using a technique like unknown word classes, suffix trees, or a maximum-entropy model of P(tag|UNK) used with Bayes' rule as part of your emission model.

Whichever type of 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. You should also get into the habit of not testing repeatedly on the test set, but rather using the validation set for tuning and preliminary experiments.

Note: if you are feeling very adventurous, you can even build a CRF tagger, but be warned that you'll almost certainly have to rewrite the object-heavy decoding machinery with primitive arrays and indexers, and that you'll essentially have to do parts 1 and 2 of this assignment together. If you are feeling slightly adventurous, you could build a

structured perceptron model, which has the same linear form and feature locality as a CRF, but training does not require expectation computations. If you're feeling that the assignments are easy, try one of these for more of a challenge.

Part 2. A Better Decoder: With your improved scorer, your results should have gone up substantially. However, you may have noticed that the tester is now 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 upgrade the greedy implementation with 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. In this part of the assignment, however, it doesn't really matter where the Trellis came from. Take a look at the `GreedyDecoder`. It starts at the `Trellis.getStartState()` state, and walks forward greedily until it hits the dedicated end state. Your decoder will similarly return a list of states in which the first state is that start state and the last is the end state, but will instead return the sequence of least sum weight (recall that weights are log probabilities produced by your scorer and so should be added). 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+ are basically state-of-the-art. Unknown word accuracies of 60+ are reasonable, 80+ are good.

Note: if you want to write your decoder before your scorer, you can construct the `MostFrequentTagScorer` with an argument of `true`, which will cause it to restrict paths to tag trigrams seen in training – this boosts scores slightly and exposes the greedy decoder as suboptimal.

Write-up: For the write-up, I mainly just want you to describe what you've built. For a maxent model, you should mention what feature schemas you used and how well they worked. A good tool for this kind of analysis is a table showing how well each feature class does on its own (when added to a core set of features) or how much loss in performance your best model suffers when that feature class is removed (an ablation study). For an HMM model, you should discuss how you modeled unknown words, as this will be the key to good performance. In either case, you should look through the errors and tell me if you can think of any ways you might fix them, be it with features, model changes, or something else (whether you do fix them or not does no matter here). Pay special attention to unknown words – in practice it's the unknown word behavior of a tagger that's most important. While no extension in particular is required, a top-scoring submission should have explored something more than simply writing a vanilla HMM. That could be extensive work on unknown words, features, a tricker model like a CRF or

structured perceptron, a comparison of two model types, a faster decoder of some kind, etc.

Coding Tips: If you find yourself waiting on a local maxent classifier, and want to optimize it, you will likely find that your code spends all of its time taking logs and exps in its inner loop. You can often avoid a good amount of this work using the `logAdd(x,y)` and `logAdd(x[])` functions in `math.SloppyMath`. Also, you'll notice that the object-heavy trellis and state representations are slow. If you want, you are free to optimize these to array-based representations. It's not required (or particularly recommended, really, unless you build a CRF) but if you wanted to do this re-architecting, you might find `util.Indexer` of use.