CSE 517, Fall 2013: Assignment 2

Due: Tuesday, October 29th at 11:45pm

In this assignment, you will build the important components of a part-of-speech tagger, including a local scoring model and a decoder. The data and support code are available on the course Dropbox in Catalyst. Please submit two files: a PDF (with your name) containing your writeup and an archive (zip, tar.gz, etc.) including all the code.

The data is taken from the Penn Treebank and includes part-of-speech tagged sentences. The provided support code is in Java and we highly encourage, but do not require, you to use it. In either case, the code submitted must include documentation describing how to run it and javadoc-style comments for classes, methods and critical parts in the code.

The starting class for this assignment is

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

Base Model: To execute the included base model, run 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 (its not great, but its reasonable). The current code then ignores the validation set entirely. On the test set, the baseline tagger 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% accuracy, with a rather pitiful unknown word accuracy of 40%. Your job is to make a real tagger by upgrading this simple tagger's placeholder components.

1 Building a Sequence Model (50%)

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 values 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:

$$\log e(t|w)$$

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

$$\sum_i \log e(t_i|w_i)$$

Your first job is to upgrade the local scorer by building an HMM tagger. Your tagger should maximize the joint probability of the words and tags, in log space

$$\log P(t_1 \dots t_n, w_1 \dots w_n) = \sum_{i=1}^{n} \log(q(t_i|t_{i-1}, t_{i-2})e(w_i|t_i))$$

which means the local scorer would have to return counters containing

$$\text{score}(t_i) = \log q(t_i|t_{i-1}, t_{i-2})e(w_i|t_i)$$

for each context.

You should do something sensible for unknown words using one of the techniques listed below or a different alternative approach. If you pick to implement an approach not listed, please make sure to justify your choice and explain why it provides a solution that is at least as good as the listed ones.

Some common approaches to handle unknown words are:

- Pseudo-word mapping (unknown word classes) - see Michael Collins' notes[1] for details.
- Suffix trees - Brants (2000) includes a nice discussion of using suffixes.
- Log-linear model for the emissions $e(w_i|t_i)$ for unknown words or a complete log-linear tagging model - Michael Collins' notes[2] describe how to build a log-linear model for tagging.[3] **Warning: significantly more work – extra credit 10%**.

2 Building a Sequence Decoder (50%)

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 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 tags and a position in the sentence. The arc weights are scores from your local scorer. In this part of the assignment, it does not 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 the start state and the last is the end state, but yours will instead return the sequence of max 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 gave the gold answer a higher score than the decoders allegedly model-optimal output. As a target, accuracies of 94+ are good, andsim96+ are basically state-of-the-art. Unknown word accuracies of 60+ are reasonable, 80+ are good.

3 Writeup (max. 4 pages)

For the write-up, we want you to describe what you've built and analyze your results. Please describe the details of your approach to handling unknown words (e.g., the pseudo-word classes you use, suffixes

[1] `http://www.cs.columbia.edu/~mcollins/hmms-spring2013.pdf`

[2] `http://www.cs.columbia.edu/~mcollins/loglinear.pdf`

[3] For more about the connection of MaxEnt models and log-linear models see Dan Klein's ACL 2003 tutorial (`http://www.cs.berkeley.edu/~klein/papers/maxent-tutorial-slides.pdf`).

or features). If you decide to build a feature-based model, you should list the 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). You should discuss how you modeled unknown words. Look through the errors and tell us 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 is the unknown word behavior of a tagger that is most important.

4 Advanced 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 horribly slow. If you want, you are free to optimize these to array-based representations. Its not required (or particularly recommended) but if you wanted to do this re-architecting, you might find util.Indexer of use. You can also speed things up by avoiding the construction of the entire trellis. There are several good ways to do this, and I'll leave it to you to find them.

The assignment was adapted from Dan Klein's CS 288 Course at UC Berkeley.

References

Brants, T. (2000). Tnt: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*, pages 224–231. Association for Computational Linguistics.