Statistical NLP - Assignment 3

Due October 11 - at 5pm by email to petrov@cs.nyu.edu

In this assignment, you will build a part-of-speech tagger.

Setup: The code for this assignment is already included in the harness that you have been using. The data for this assignment is available on the web page as usual. The data comes from a recent shared task on "Parsing the Web." The training data is from the Penn Treebank (original text is from the WSJ in the late 1980s). For development you have two datasets: one is in-domain and comes from the WSJ and the other one comes from recent blogs extracted from the web. The (blind) test data also comes from the web:

`http://www.cs.nyu.edu/~petrov/restricted/data3.zip`

The starting class for this assignment is

`nlp.assignments.POSTaggerTester`

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 gives each known word its most frequent training tag. Unknown words all get the same tag (which, and why?). This tagger operates at about 91.3% accuracy, with a rather pitiful unknown word accuracy of 42.7%. 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 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:

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

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

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

Your first job is to upgrade the local scorer. I recommend you build a trigram HMM tagger. Your decoder should maximize the quantity

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

which means the local scorer would have to return counters containing

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

for each context.

The HMM will train relatively fast. To make debugging easier, you might start by first building a bigram (first-order) HMM and then extending your code to the trigram (second-order) case. Without any changes to the unknown word model, your trigram HMM should be able to get the overall accuracy up to around 94% (on the in-domain data). The unknown word accuracy will remain under 50%.

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 for tuning. 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 really 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 also build a structured perceptron model, which has the same linear form and feature locality as a CRF, but training does not require expectation computations.

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. `Trellis`es 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 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 the start state and the last is the end state, but yours 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 gave the gold answer a higher score than the decoder's allegedly model-optimal output.

Without changes to the unknown word model, doing Viterbi decoding (rather than greedy decoding) should bring up the accuracy of your trigram HMM to about 95% and your unknown word accuracy to around 60% (on the in-domain data). Why is the unknown word accuracy improving even if no changes to the unknown word model have been made?

Part 3. Better unknown word model: To further improve the accuracy of your model, you will need to build a better unknown word model. There are many things that one can try for unknown words, using techniques like unknown word classes, suffix trees, etc. Use your intuitions from assignment 2 to come up with useful features. The simplest approach is probably to come up with a set of 50 or so word categories (based on some features) and replace all words appearing 5 times or less in the training set with their word category. At test time, when a previously unseen word is encountered, you can simply map it to its word category.

You can also use the models that you built for assignment 2 as plug-in replacements for the emission model (half-way towards an MEMM – for an MEMM you would also replace the transition probabilities with a locally normalized discriminative model, but that will be likely slow to train). If you are using a discriminative emission model of P(tag|UNK), make sure to apply Bayes' rule to invert that probability estimate. Be careful to keep your model probabilistically sound, otherwise you will most likely see performance degradations.

As a final target, your in-domain accuracy should be above 94; 95+ is good, and 96+ is close to state-of-the-art. Unknown word accuracies of 60+ are reasonable, 80+ are good. The out-of-domain data is significantly more difficult. You should aim to break 91%. Above 91 is good and around 92 is close to state-of-the-art. Why is there such a big difference in performance? I don't expect you to do anything in particular about this, but it would be good to speculate

about potential approaches for alleviating this problem in your write-up.

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.

Leaderboard: If you didn't configure a remote previously, make sure to do so now:

```
git remote add upstream https://github.com/slavpetrov/stat-nlp-nyu.git
```

You can then sync the repository to get the baseline output:

```
git fetch upstream; git checkout master; git merge upstream/master
```

The harness will automatically evaluate your model on the test data and write the output to a text-file in the same directory as the rest of your data. Please submit this file as hw3/output.txt.

Write-up: For the write-up, I mainly just want you to describe what you've built. You should discuss how you modeled unknown words, as this will be the key to good performance. You should also 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 is very important. Overall, do you think the transistion model is more important or the emission model? How would you go about further improving the performance of your tagger?

While no extension in particular is required, you are encouraged to try exploring more than 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 horribly 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. 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.

Errata for TnT: Thorsten Brant's TnT model (see readings) is very close to what you will be building in this assignment. Last year two students found some bugs in the paper:
Equation 7: The subscript on the first l in the second term of the numerator should be $n - i + 2$ instead of $n - i$. Otherwise the recursion will be go in the wrong direction. Equation 11 is also unnecessary, as it is trivially $\frac{1}{s}$.