

# CSE 517, Fall 2013: Assignment 2

**Due:** Tuesday, Feb 6th at 5pm

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 provide detailed documentation.

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.

The assignment does not necessarily require using a validation set to tune parameters, but you should use it instead of the test set for development. Be sure to only run on the test set once (ok, you can have a second try if you absolutely need it, but remember that running repeatedly invalidates any claims about generalization performance.).

## 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:

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

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 by building an HMM tagger. Your tagger should maximize the joint probability of the words and tags, in log space

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

which means the local scorer would have to return counters containing

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

for each context.

You should do something sensible for unknown words, using a technique like unknown word classes, suffix trees, or a maximum-entropy model of  $P(\text{tag}=\text{UNK})$  used with Bayes rule as part of your emission model.

**Bonus. Building a MaxEnt tagger (15%):** Instead of building an HMM, implement a MEMM (maximum-entropy Markov model) tagger. In that case, your decoder should instead maximize

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

which means that you will want to build a little per-position maximum entropy model which predicts distributions over tags given such contexts, based on whatever features of the contexts that you design. The local score in this case has the form:

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

Note that this is a log distribution over tags. 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. You are also welcome to explore using Perceptron updates to speed up this learning, which we consider an advanced topic. An HMM will train faster (but likely have lower accuracy).

## 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-optimality. 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`s 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 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-optimality - 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, and 96+ 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. 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). 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 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 Ill leave it to you to find them.

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