CSEP 517, Fall 2015: Assignment 2

Due: Thursday, November 5th at 11pm

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 should provide detailed documentation.

To build the support code, first unzip the files to your local working directory. Then from the source root (i.e. your current directory contains the directory 'edu'), you can compile the provided code with

```
javac -d classes */*/*/*.java */*/*/*/*.java
```

You can then run a simple test file by typing

```
java -cp classes edu.berkeley.nlp.Test
```

You should get a confirmation message back. You may wish to use an IDE such as Eclipse (I recommend it). If so, it is expected that you be able to set it up yourself. 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 (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 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 tasks for this part are as follows:

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

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

   which means the local scorer would have to return counters containing

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

   for each context.

2. Implement at least one of the following 4 methods for handling unknown and/or low frequency words:

   (a) Unknown word classes[1]

   (b) Suffix trees[2]

   (c) Log-linear model for emissions $P(w_i|t_i)$ for unknown words. Note that this approach will be covered in more detail later in the class (see Week 8 notes), and is likely the most difficult.

   (d) Some other reasonable method of your own design. You should still aim to achieve the target accuracies specified below. If you choose this approach, be sure to justify what you did and explain why it provides a solution that is at least as good as the other provided methods.

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).

3. Your final task in this assignment is to upgrade the greedy implementation with a Viterbi decoder, meaning that your implementation should return the optimal sequence of tags under your model. 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.

   Some context on how this works in the starter code:
   Decoders implement the TrellisDecoder interface, which takes a Trellis and produces a path. Trellises

[1]See section 2.7 in `http://www.cs.columbia.edu/~mcollins/hmms-spring2013.pdf`

[2]See section 2.3 in `http://www.coli.uni-saarland.de/~thorsten/publications/Brants-ANLP00.pdf`

are just directed, acyclic graphs, whose nodes are states in a Markov model and whose arcs are transitions in that model, with weights attached.[3] 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).

As a target, once you have completed both the HMM tagger and the decoder, 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. You should discuss how you modeled unknown words, as this will be the key to good performance. 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 not 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

Taking logs and exps is relatively expensive computation. You can often reduce 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 is 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 if you want, it's up to you to find them.

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

[3] An example can be seen in the HMM lecture on slides 17-25