Lign/CSE 256, Programming Assignment 2: Structured-sequence inference

4 Feb 2009

due 19 Feb 2009

This programming assignment is disjunctive, with two options. Option 1 is the supervised learning task of building the important components of a part-of-speech tagger, including a local scoring model and a decoder.[1] Option 2 is the task of building an unsupervised word-segmentation model.[2]

1 Option 1: Part-of-speech tagging

This assignment is building a supervised part-of-speech tagger. The data for this assignment are available here:

`http://grammar.ucsd.edu/courses/lign256/protected/data2.zip`

It uses the same Penn Treebank data as the first assignment, this time with the part-of-speech labels added in. You can use the code base from assignment 1.

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, `POSTaggerTester`. You will need to run it with the command line option `-path <your_data_path>`, where `your_data_path` is wherever you have unzipped the assignment data. This class by default loads a fully functional, if minimalist, POS tagger. It is somewhat memory-intensive so give at least 100 megabytes to Java when you run it (use the `-mxNm` option, where `N` is the number of megabytes, *before* the class name). Hence:

[1] This option is based on an assignment of Dan Klein's—thanks once more to Dan for permission to use his assignment, code, and prepped data.

[2] Thanks to Sharon Goldwater (and indirectly Michael Brent) for supplying the dataset to be used for this homework assignment.

```
java -cp <your_classes_directory> -mx100m
  edu.berkeley.nlp.assignments.POSTaggerTester -path <your_data_path>
```

The main method first loads the standard Penn Treebank Wall Street Journal (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.

2 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:

$$\log P(t|w) \tag{1}$$

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

$$\sum_i \log P(t_i|w_i) \tag{2}$$

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

$$\sum_i \log \left[P(t_i|t_{i-1}, t_{i-2})P(w_i|t_i)\right] \tag{3}$$

which means the local scorer would have to return

$$score(t_i) = \log \left[P(t_i|t_{i-1}, t_{i-2})P(w_i|t_i)\right] \tag{4}$$

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

$$\sum_i \log P(t_i|t_{i-1}, t_{i-2}, \vec{w}, i) \tag{5}$$

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:

$$score(t_i) = \log P(t_i|t_{i-1}, t_{i-2}, \vec{w}, i) \tag{6}$$

Note that this IS a log distribution over tags. You can build 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 schemas, 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 simple maximum-entropy model of $P(\text{tag}|\text{UNK})$ 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.

3 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. For an HMM model, you should discuss how you modeled unknown words. In either case, you should look through the errors and tell me if you can think of any ways you might fix them (whether you do fix them or not). Pay special attention to unknown words—in practice it's the unknown word behavior of a tagger that's most important.

3.1 Miscellaneous advice

Coding Tips: If you find yourself waiting on your 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) but if you wanted to do this re-architecting, you might find `util.Indexer` of use.

As before, in `edu.berkeley.nlp.util` there are some classes that might be of use – particularly the `Counter` and `CounterMap` classes. These make dealing with word to count and history to word to count maps much easier.

General advice regarding implementing models: it's often said that laziness is a virtue in programming. One way of interpreting this maxim with respect to implementing models is **don't rush to implement a model that you don't yet understand fully.** I personally find that it (a) is ultimately more time-efficient, and (b) leads to better code, to carefully work out simple cases for your model on paper before starting to write code. The small HMM Viterbi example I did on the board and wrote up as a short handout is an example of this. It's useful to go through this type of exercise yourself.

4 Option 2: unsupervised word segmentation

This assignment is to implement the unigram word-segmentation model of Goldwater et al. (2006). The data for this assignment are available here:

`http://grammar.ucsd.edu/courses/lign256/protected/br_data.tgz`

The file `br-phono.txt` is a phonological transcription of part of the CHILDES database, and consists of adult-directed speech to children aged 13 to 23 months (Bernstein-Ratner, 1987; Brent and Cartwright, 1996).

In the unigram word-segmentation model, a posterior distribution over segmented corpora c_{seg} is inferred from an unsegmented corpus c via Bayes' rule:

$$P(c_{seg}|c) = \frac{P(c|c_{seg})P(c_{seg})}{P(c)}$$

We will sample from the posterior distribution using Gibbs Sampling, so we can ignore the denominator:

$$P(c_{seg}|c) \propto P(c|c_{seg})P(c_{seg})$$

The likelihood is very simple: a segmented corpus generates an unsegmented corpus by ignoring all word boundaries except for utterance boundaries (e.g., the segmented corpus `.ba.di.` generates `.badi.`):

$$P(c|c_{seg}) = \begin{cases} 1 & \text{if } c_{seg} \text{ becomes } c \text{ when word boundaries are dropped} \\ 0 & \text{otherwise} \end{cases}$$

The prior is determined by four parameters: $p_{\#_C}$, which governs a geometric distribution over the length N of each utterance i:

$$P(N_i) = p_{\#_C}(1 - p_{\#_C})^{N-1}$$

Next is $p_{\#_C}$, which governs a geometric distribution over the length L of each utterance i:

$$P(L_i) = p_{\#_w}(1 - p_{\#_w})^{L-1}$$

Third is V, the number of phonemes in the language, which governs a uniform distribution over the phonological form of the i-th word type w_i in the lexicon:

$$P_0(w_i|L_i) = \frac{1}{V^{L_i}}$$

Fourth is α, which governs the tendency for the next new word token in the corpus to be novel given existing word types $i = 1, \ldots, k$ with token counts n_i. The probability distribution over the type j of the next word is:

$$P(j=i) = \begin{cases} \frac{n_i}{n+\alpha} & i \leq k \\ \frac{1}{n+\alpha} & i = k+1 \end{cases}$$

where $n = \sum_{i=1}^{k} n_i$. Of these parameters, V is determined by the dataset; you can freely set $\alpha, p_{\#_w}$, and $p_{\#_C}$ as you see fit.

Use Gibbs sampling to sample from the posterior distribution over word segmentations, iterating over each possible inter-word boundary (i.e. each inter-phoneme position). The hypothesis space that you sample from may either explicitly contain hypothesized lexicons (with counts) or you may marginalize over these lexicons. Based on your choice in this regard, present as part of your report the conditional probability distribution from which each Gibbs sample is drawn.

In evaluation, compute the precision, recall, and F1 of word segmentation, word token, and word type inferences. You may either draw a single sample (after a number of "burn-in" iterations) and evaluate that sample, or draw multiple samples and average over them. Be explicit about how you do any averaging of this sort. Also try multiple random initializations and see if the performance of the model varies much across trials.

Once you have implemented the unigram model, you are free to try extending the model. For example, you might try placing priors over $p_{\#_w}$ and/or $p_{\#_C}$ and inferring them from the data, or using a different distribution (e.g., Poisson) over utterance length and/or word length. In each case, describe precisely the changes in the model & the conditional distributions used in Gibbs sampling which result from your changes.

References

Bernstein-Ratner, N. (1987). The phonology of parent-child speech. In Nelson, K. and van Kleeck, A., editors, *Children's Language*, volume 6. Hillsdale, NJ: Erlbaum.

Brent, M. R. and Cartwright, T. A. (1996). Distributional regularity and phonotactic constraints are useful for segmentation. *Cognition*, 61:93–125.

Goldwater, S., Griffiths, T. L., and Johnson, M. (2006). Contextual dependencies in unsupervised word segmentation. In *Proceedings of COLING/ACL*.