LING/CSCI 5832 Fall 2016 Homework 3 - POS tagging (the Viterbi algorithm)

Due **Tue Oct 11, 2016** before class (**2pm**) on D2L.

In this homework, you will write a bigram (HMM) part-of-speech tagger. The core of the homework will be to produce an implementation of the Viterbi algorithm for decoding the best POS sequence. You will also need to calculate transition and emission probabilities for tokens and tags from a corpus. Don't be intimidated by the length of this description: your code should be much shorter than these instructions! The average effort in lines of Python for the whole homework has hovered around 50-60 lines, including comments, of which the Viterbi algorithm tends to occupy about 30 lines.

The implementation will not be a full-fledged tagger (but once you're done with it, it should be easy to extend to cover all the extra tricks and corner cases). In particular you do not need to:

- Do any smoothing of the emissions or transitions.
- Use log probabilities (the sentences that you need to tag are short enough to not cause underflow with Python multiplication).

Simplifying the task like this helps in putting the focus on understanding and implementing the Viterbi algorithm itself.

The Data

As training data for the tagger you should read in sections 00-18 of the Wall Street Journal portion of the Penn Treebank. This is given to you in the file `wsj00-18.tag`

Recall that this file is organized in a very simple format: each line is a token/tag combination, and empty lines separate sentences.

From this data, you should:

- Calculate (by MLE) the transition probabilities: $p(TAG_i|TAG_j)$ for all occurring tag combinations (no smoothing needed).
- Calculate (by MLE) the emission probabilities: p(word|TAG) for all occurring word/tag combinations (no smoothing needed).

Note that you need to separate the data in `wsj00-18.tag` into sentences in some way. This is because you'll want to add two extra tags to the tagset:

1. A special start tag `<s>`
2. A special end tag `</s>`

These tags/states will never emit anything, but will be included in the transition probabilities, since the start "state" will always be `<s>` and the end state will always be `</s>`. This means that you should calculate the probabilities p(TAG| `<s>`) and p(`</s>` |TAG) also. This is why you need to split the corpus into sentences: without knowing the sentence boundaries, you can't estimate these probabilities.

A concrete example of probabilities

Suppose you only had two sentences in your training data:

```
 DT  VBZ  DT     NN     .
This is   a   sentence  .

PRP  VBP RB   .
 I   am  not  ?
```

Now, you could imagine padding these sentences with start and end tags before you collect the relevant counts:

```
<s>  DT  VBZ  DT     NN     .  </s>
    This is   a   sentence  .

<s> PRP  VBP RB   .  </s>
     I   am  not  ?
```

After this, p(DT| `<s>`) should be 0.5 and P(PRP| `<s>`) also. P(`</s>` |.) should be 1.0. Obviously, there won't be any emission counts for these special tags, and so all emission probabilities for `<s>` and `</s>` will be zero.

As an example of the emission probabilities: P(.|.) and P(?|.) should be 0.5 for each as per MLE.

Code

After extracting and storing either the transition and emission counts (or counts converted into probabilities), you should create a function called `viterbi` that takes as

input a list of tokens (and possibly the counts/probabilities), and returns the best tag sequence as a list.

The last five lines of code in your program should look like this (Python 2, if you're running Python 3, `print` is a function):

```
print viterbi(['This','is','a','sentence','.'], transcounts, emitcounts)
print viterbi(['This','might','produce','a','result','if','the','system','works','v
print viterbi(['Can','a','can','can','a','can','?'], transcounts, emitcounts)
print viterbi(['Can','a','can','move','a','can','?'], transcounts, emitcounts)
print viterbi(['Can','you','walk','the','walk','and','talk','the','talk','?'], trar
```

Note that you don't have to pass exactly the same arguments as here to the `viterbi` -function. Perhaps you pre-calculated the probabilities instead of just counts and may want to pass those instead. Perhaps you even created a class for the POS-tagger. That's fine. Or, you might also want to pass the set of possible parts of speech or some such thing. That's fine, too.

The important thing is that you at least pass the sentence to be tagged as a list of tokens and that your function also returns at least a list of tokens with the best POS-sequence.

When calling `viterbi` with the above sentences, your program should output something like:

```
['DT', 'VBZ', 'DT', 'NN', '.']
['DT', 'MD', 'VB', 'DT', 'NN', 'IN', 'DT', 'NN', 'VBZ', 'RB', '.']
['MD', 'DT', 'NN', 'MD', 'DT', 'NN', '.']
['MD', 'DT', 'MD', 'VB', 'DT', 'NN', '.']
['MD', 'PRP', 'VBP', 'DT', 'NN', 'CC', 'VB', 'DT', 'NN', '.']
```

All the tag sequences are not necessarily correct since we don't do smoothing and since HMM taggers aren't perfect. For example, sentence 3 has the word `can` tagged incorrectly in one of the occurrences. Sentence 4 which replaces that instance of `can` with `move` gets that token right but fails on the previous `can` . (Why?). But notice that the tagger does get the noun/verb distinction right for the last sentence and the tokens `walk` and `talk` .

Since we're not doing any smoothing, the tagger will not work if it encounters any word it hasn't seen in the training data. Keep this in mind when debugging your code. It's a good idea to debug using one of the shorter sentences above, since they are guaranteed to produce some tag sequence, if your code is correct.

What to hand in

You should hand in an archive `firstname.lastname.5832.hw3.tar.gz` or `firstname.lastname.5832.hw3.zip` consisting at least of a Python file `firstname.lastname.5832.hw3.py` that runs as decribed if the `wsj00-18.tag` file is located in the same directory and prints out something like the above. Put a `README` in the archive if you did anything outside a straightforward implementation of the above.

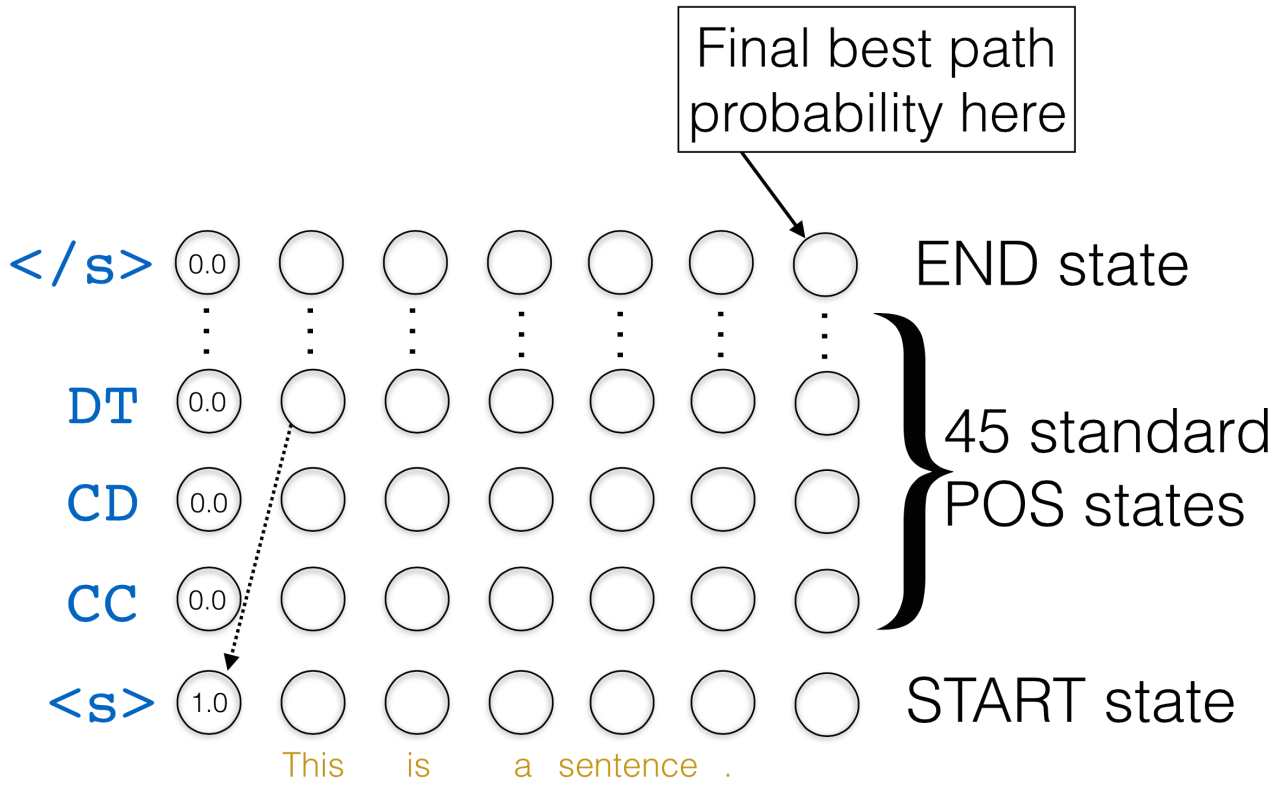
What external resources can I use?

You can use any standard Python module such as `collections` to help with the counting. Additionally, you may use `numpy`, which can be useful for the Viterbi trellis (although using `numpy` in this case doesn't really help with making the code shorter), but if you're used to `numpy`, go ahead and take advantage of it. Crucially, you cannot use any package that implements Viterbi or POS tagging in any way: the core part of the homework is implementing the algorithm.

Tips

The textbook (pages 145-149) has a thorough description of the Viterbi algorithm that you can refer to if you're in trouble.

In general, your trellis data structure (a two-dimensional matrix) when tagging a sentence like `This is a sentence .` could/should look something like this:



That is, you'll have 47x7 two-dimensional matrix for the probabilities. You can pre-fill them all with zeroes, and also prefill the [0,0]-entry with 1.0 (which corresponds to the probability of being in the start state at time 0). Now, you should fill in the columns starting from the one above the first token (`This`) with the probability of the most probable previous state to have arrived from. Also, you should store (perhaps in another matrix of the same shape) which state that was. An example of such a backpointer is shown as the dotted arrow from time step 1 to time step 2. Note that the figure only shows one backpointer, but when filling the trellis, each state at each time should contain a backpointer to some state in the previous time step.

Finally, when you have filled in all the columns and backpointers, you should have the probability of the best path stored in the upper right-hand corner, and it should also have a backpointer. Tracing the backpointers backwards from that position in the matrix, you should be able to recover the best part-of-speech sequence, ending in [0,0]. This is the part-of-speech sequence your `viterbi` -function should return, as a list (minus the start and end tags).