CS114 (Spring 2020) Programming Assignment 4
Part-of-speech Tagging with Hidden Markov Models

Due April 3, 2020

You are given `pos_tagger.py`, and `brown.zip`, the Brown corpus (of part-of-speech tagged sentences). Sentences are separated into a training set ($\approx$80% of the data) and a development set ($\approx$10% of the data). A testing set ($\approx$10% of the data) has been held out and is not given to you. You are also given `data_small.zip`, the toy corpus from HW2, in the same format. Each folder (`train`, `dev`, `train_small`, `test_small`) contains a number of documents, each of which contains a number of tokenized, tagged sentences in the following format: `[<word>/<tag>]`. For example:

```
The/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl
said/vbd Friday/nr an/at investigation/nn of/in Atlanta's/np$
recent/jj primary/nn election/nn produced/vbd ``/`` no/at
evidence/nn ''/'' that/cs any/dti irregularities/nns took/vbd
place/nn ./.
```

Assignment

Your task is to implement a supervised hidden Markov model to perform part-of-speech tagging. Specifically, in `pos_tagger.py`, you should fill in the following functions:

- `train(self, train_set)`: This function should, given a folder of training documents, fill in `self.initial`, `self.transition` and `self.emission`, such that:
  - `self.initial[POS]` = log(P(the initial tag is `POS`))
  - `self.transition[POS1][POS2]` =
    log(P(the current tag is `POS2`|the previous tag is `POS1`))
  - `self.emission[POS][word]` =
    log(P(the current word is `word`|the current tag is `POS`))

As in PA2, you should use Numpy arrays for `self.initial`, `self.transition`, and `self.emission`. Again, you should use `self.pos_dict` and `self.word_dict` to translate between indices and words/parts of speech; you will need to fill these in yourself, based on the training data. Be sure to account for `<UNK>` (as a word, but not as a POS tag). You do not need to account for `<S>` or `</S>`. Finally, you should implement add-k smoothing (on all three arrays); as in PA3, you should try to find the value of k that results in the maximum accuracy on the dev set.

Note that although `/` is the separator between words and parts of speech, some words also contain `/` in the middle of the word. In these cases, it is the last `/` that separates the word from the POS.

- `viterbi(self, sentence)`: Implement the Viterbi algorithm!

  Specifically, for each development (or testing) sentence, you should construct and fill in two trellises (Numpy arrays), `v` (for `viterbi`) and `backpointer`. You can look at the pseudo-code given in Figure 8.5 of the Jurafsky and Martin book, reproduced below. Note that π_s are elements of the initial probability distribution, $a_{s',s} \in A$ are elements of the transition matrix, and $b_s(o_t) \in B$ are elements of the emission matrix.

function VITERBI(*observations* of len *T*,*state-graph* of len *N*) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
for each state *s* **from** 1 **to** *N* **do** ; initialization step
    $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
    $backpointer[s,1] \leftarrow 0$
for each time step *t* **from** 2 **to** *T* **do** ; recursion step
  for each state *s* **from** 1 **to** *N* **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$; termination step
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$; termination step
bestpath ← the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 8.5 Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A,B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

Some notes:

- The input `sentence` is a list. A list of what? You will need to translate from words to indices at some point in your code; it is recommended to do this translation outside of the `viterbi` function, so that the input to the function is a list of indices.
- Remember that when working in log-space, you should add the logs, rather than multiply the probabilities.
- Note that operations like `+` are Numpy *universal* functions, meaning that they automatically operate element-wise over arrays. This results in a substantial reduction in running time, compared with looping over each element of an array. As such, your `viterbi` implementation should not contain any for loops that range over states (for loops that range over time steps are fine).
- To avoid unnecessary for loops, you can use *broadcasting* to your advantage. Briefly, broadcasting allows you to operate over arrays with different shapes. For example, to add matrices of shapes $(a, 1)$ and (a, b), the single column of the first matrix is copied b times, to form a matrix of shape (a, b). Similarly, to add matrices of shapes (a, b) and $(1, b)$, the single row of the second matrix is copied a times.
- When performing integer array indexing, the result is an array of lower rank (number of dimensions). For example, if `v` is a matrix of shape (a, b), then `v[:, t-1]` is a vector of shape $(a,)$. Broadcasting to a matrix of rank 2, however, results in a matrix of shape $(1, a)$: our column becomes a row. To get a matrix of shape $(a, 1)$, you can either use slice indexing instead (i.e. `v[:, t-1:t]`), append a new axis of `None` after your integer index (i.e. `v[:, t-1, None]`), or use the `numpy.reshape` function (i.e. `numpy.reshape(v[:, t-1], (a, 1))`).
- In `self.transition`, each row represents a previous tag, while each column represents a current tag. Do not mix them up!
- Finally, you do not have to return the path probability, just the backtrace path.

- `test(self, dev_set)`: This function should, given a folder of development (or testing) documents, return a dictionary of results such that:
  - `results[sentence_id]['correct']` = correct sequence of POS tags
  - `results[sentence_id]['predicted']` = predicted sequence of POS tags (hint: call the `viterbi` function)

  You can assign each sentence a `sentence_id` however you want. Your sequences of POS tags can just be lists, e.g. `['at', 'np', ...]`.

- `evaluate(self, results)`: This function should return the overall accuracy (# of words correctly tagged / total # of words). You don't have to calculate precision, recall, or F1 score.

Write-up

You should also prepare a (very) short write-up that includes at least the following:

- How you set the value of k
- Your evaluation results on the development set (you do not need to include any results on the toy data)

Submission Instructions

Please submit two files: your write-up (in PDF format), and `pos_tagger.py`. Do not include any data.