CS115B (Spring 2024) Homework 3
Part-of-speech Tagging with Structured Perceptrons

Due March 5, 2023

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 two-sentence toy corpus from the Lab 5 Exercise, in the same format. Each folder 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 structured perceptron to perform part-of-speech tagging. Specifically, in `pos_tagger.py`, you should fill in the following functions:

- `make_dicts(self, train_set)`: You should be familiar, from HW1 and HW2, with the use of dictionaries to translate between indices and other entities, such as classes or features. In this assignment, we will be using `self.tag_dict` and `self.word_dict` to translate between indices and either parts of speech or words, respectively. This function should, given the training set, fill in these dictionaries. You do not need to account for the start symbol `<S>`, the stop symbol `</S>`, or the unknown word `<UNK>`. When reading sentences from the file, be sure to skip empty lines.

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 part of speech.

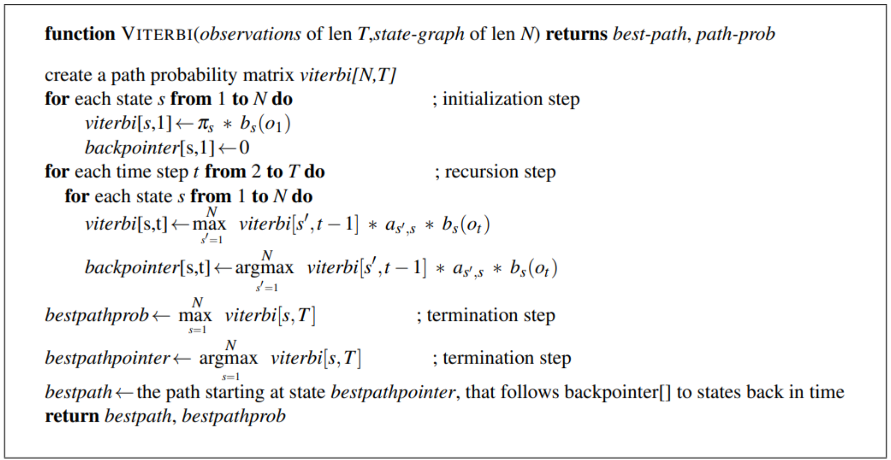
- `load_data(self, data_set)`: This function should, given a folder of documents (training, development, or testing), return a list of `sentence_ids` (noting that a document can contain multiple sentences), and dictionaries of `tag_lists` and `word_lists` such that:

  - `tag_lists[sentence_id]` = list of part-of-speech tags in the sentence
  - `word_lists[sentence_id]` = list of words in the sentence

  You can assign each sentence a `sentence_id` however you want, as long as they are distinct. Again, you may find it helpful to store the tags and words in terms of their indices, using `self.tag_dict` and `self.word_dict` to translate between them. If you come across an unknown word or tag (in the development or testing sets), you may use the `self.unk_index` as the index for that word or tag.

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

  Specifically, for each `sentence`, given as a list of word indices, you should fill in two trellises, `v` (for `viterbi`) and `backpointer`. You can look at the pseudo-code given in Figure 8.10 of the Jurafsky and Martin book, reproduced below. Although the below figure describes the Viterbi algorithm in the context of hidden Markov models, our procedure will be basically the same. Note that π_s are elements of the initial vector, $a_{s',s} \in \mathbf{A}$ are elements of the transition matrix, and $b_s(o_t) \in \mathbf{B}$ are elements of the emission matrix. For simplicity, these will be the only features, and we will ignore the bias term.



```
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] ← $\pi_s * b_s(o_1)$
    backpointer[s,1] ← 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] ← $\max_{s'=1}^{N}$ viterbi[$s'$, $t-1$] $* a_{s',s} * b_s(o_t)$
    backpointer[s,t] ← $\operatorname{argmax}_{s'=1}^{N}$ viterbi[$s'$, $t-1$] $* a_{s',s} * b_s(o_t)$
bestpathprob ← $\max_{s=1}^{N}$ viterbi[$s, T$]                ; termination step
bestpathpointer ← $\operatorname{argmax}_{s=1}^{N}$ viterbi[$s, T$]      ; termination step
bestpath ← the path starting at state bestpathpointer, that follows backpointer[] to states back in time
return bestpath, bestpathprob
```

Figure 8.10 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:

- Remember that when working with structured perceptron scores, you should add the scores, 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))`).

  - If you come across an unknown word, you should treat the emission scores for that word as 0.
  - In the transition matrix, 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.

- `train(self, train_set)`: Given a folder of training documents, this function fills in `self.tag_dict` and `self.word_dict` (using the `make_dicts` function), loads the dataset (using the `load_data` function), shuffles the data, and initializes the three weight arrays `self.initial`, `self.transition`, and `self.emission`; these tasks have already been done for you. Then, for each sentence, this function should:

  - Use the Viterbi algorithm to compute the best tag sequence
  - If the correct sequence and the predicted sequence are not equal, update the weights using the structured perceptron learning algorithm: increment the weights for features in the correct sequence, and decrement the weights for features in the predicted sequence. We will assume a constant learning rate $\eta = 1$. Here, simpler is better—no fancy Numpy tricks needed.

  To give you a sense of how far along your training is, there are 4̃5,000 sentences in total in the train set. We have provided you with pickle.dump() and pickle.load(). This allows you to save your trained POS tagger so that once training is working as you expect, you don't need to rerun training (which takes 1̃0-15 mins) every time you want to run test and evaluate.

- `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 tags

  ◦ `results[sentence_id][‘predicted’]` = predicted sequence of tags

This function should be very short—only a few lines of code.

- `evaluate(self, results)`: This function should return the overall accuracy (number of words correctly tagged / total number of words). You don't have to calculate precision, recall, or F1 score. You should be able to get an accuracy of about 85% on the development set.

As a hint, it took about **9 minutes** (using an AMD Ryzen 7 5800U) to train and test the model on the full dataset. Your mileage may vary, depending on your computer. That being said, if your model takes hours rather than minutes to train, your code is likely not as efficient as it can be. Make sure your code is fully broadcasted!

Grading

Grades will be determined as follows:

- 10%: `make_dicts` correctly creates `self.tag_dict` and `self.word_dict`.
- 15%: `load_data` returns a list of `sentence_ids` and dictionaries of `tag_lists` and `word_lists`, as described above.
- 30%: The Viterbi algorithm is implemented correctly and efficiently.
- 25%: The `train` function gets the best tag sequence, and updates the weights correctly using the structured perceptron learning algorithm.
- 10%: The `test` function gets the correct and predicted sequences of tags, and stores them in the `results` dictionary.
- 10%: Accuracy is computed correctly.
- Within these parameters, partial credit will be assigned where possible.
- Your accuracy on the dev set when training on the full training set should be above at least 80%

Submission Instructions

Please submit one file: `pos_tagger_firstname_lastname.py`. You do not need to write anything up for this assignment.