CS542 (Fall 2021) Programming Assignment 3
Part-of-speech Tagging with Structured Perceptrons

Due October 30, 2021

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 (≈80% of the data) and a development set (≈10% of the data). A testing set (≈10% of the data) has been held out and is not given to you. You are also given `data_small.zip`, a toy corpus very similar to the data from the worked example (in Lecture 16, reproduced for you in PDF form in `PA3_example.pdf`, in the same format). Finally, you also have `brown_news.zip`, which is the news data only from the Brown corpus, which is intended to let you check that your algorithms scale up without having to test it on the full Brown corpus. 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 PA1 and PA2, 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>`.

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

  The recommended format for `sentence_id`, that we will be using in testing your code, is the document name followed immediately by the ordinal number of the sentence in that document. That is, in a document called `ca01`, the `sentence_id` of the first sentence would be `ca010`, the second sentence `ca011`, etc. Again, you will 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), 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.

  Some notes:

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 8.10 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.

- 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))`). Please be careful using `numpy.reshape` and make sure at all steps that your code can run with the provided automatic grader!

- 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. As in the worked toy example, we will assume a constant learning rate $\eta = 1$. Here, simpler is better—no fancy Numpy tricks needed.

- `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, sentences, 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.

Hints

- Do not change code outside of the `# BEGIN STUDENT CODE` and `# END STUDENT CODE` comments. The exception to this is commenting/uncommenting directories in `main` to switch between datasets. If some line or section says `DO NOT CHANGE!!`, DO NOT CHANGE!! it under *any* circumstances.

- There are code comments in each section that lay out, with some specificity, what needs to happen at each step. These, `PA3_example.pdf` and Lecture 16 should contain everything you need to know to complete this assignment.

- You have a sample grading script available. It uses the `data_small` dataset for some parts and specific hard-coded inputs at other points. Use it at each step. It tests each of the 6 functions you are tasked with implementing. Look in the autograder source code to see what the expected format of the output of each function should be. If your numbers are correct but the format is wrong, you will not get credit for that part. **This is your only warning!**

- Write code using the `data_small` dataset first, testing against the autograder. When that works, verify that your code can scale up using the `brown_news` dataset. With a fully-broadcasted implementation, this dataset takes about a minute (using a 2.3 GHz 8-core processor) to train and test. Fix any bugs. Then conduct your final test on the `brown` dataset. Turn in code that runs out of the box with the `brown` dataset. Any other dataset left in your submission may result in a deduction!

- As a hint, it took about **18 minutes** (using a 2.3 GHz 8-core processor) 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!

- A correct implementation should be able to get an accuracy larger than 85% on the `brown/dev` development set. On the smaller/faster `brown_news/dev` development set, you should be seeing numbers above 70% accuracy. Due to shuffling the training set and use of unordered dictionaries when creating the word and tag lists, you may expect to see exact numbers change slightly over multiple runs.

Grade Breakdown

- `make_dicts`: 10 points;
- `load_data`: 10 points;
- `viterbi`: 20 points;
- `train`: 20 points;
- `test`: 10 points;
- `evaluate`: 10 points;
- The remaining 20 points come from your code running cleanly in standalone mode, without the autograder, and producing sufficiently accurate results.

Submission Instructions

Please submit one file: `pos_tagger.py`. You do not need to include any data or a write-up.