Assignment 2

Pavle Marković

December 6, 2020

1 Task

Implement a bigram part-of-speech (POS) tagger based on Hidden Markov Models from scratch. Using NLTK is disallowed, except for the modules explicitly listed below. For this, you will need to develop and/or utilize the following modules:

1. Corpus reader and writer
2. Training procedure, including smoothing
3. Viterbi tagging, including unknown word handling
4. Evaluation

2 Results

Model	Runtime	File size	Accuracy
base	4.507 sec	19.6 MB	0.9095
add one	3.618 sec	29.1 MB	0.8665
add one end token	4.177 sec	31.6 MB	0.8634
base reduced	2.244 sec	2.4 MB	0.0430

Table 1: Models' performances

- **base** - Model without smoothing and with crude unknown words handler.
- **add one** - Model with add-one smoothing and crude unknown words handler.
- **add one end token** - Model with add-one smoothing, with crude unknown words handler and uses additional end token.
- **base reduced** - Model without smoothing, with crude unknown words handler and calculate emission probabilities only for word-tag pair it encountered during training.

3 Discussion

The first thing that caught my attention was the better performance of the base model (without smoothing) then add one model (includes add-one smoothing). If I reason correctly, it seems wiser to put 0 emission probability for a word that was seen in the corpus but not emitted by a particular tag. To me, it kind of makes sense if we assume that our training set is representative, then the model will prefer other tags that actually emitted this word, therefore better chance to choose the right tag. The other thing that was interesting to me for these two models is scores for the ADJ tag, which were the lowest (excluding the X tag). I compared scores of ADJ and DET tags, as they are both modifiers of nouns, and it turns out that both models prefer DET over ADJ (higher Recall, 0.7222 for ADJ and 0.9755 for DET). Therefore, for further improvements, I would try to find a way to balance Recalls between the two.

I have also experimented with adding <END> tag to models, and as it can be seen from the Table 1 for add one model the result is almost identical to the regular add one model. I didn't test for the base model, because it would require further code adapting to handle division with 0, and make the code more complicated. For this task, the additional script, *add_end.py*, is provided. It just add the <END> token at the end of every sentence in the *de-eval.tt* set and creates a new set *de-eval_end.tt*, also provided.

Lastly, I did a small experiment with reducing data structure for emission probabilities, so it only includes emissions for words that a particular tag saw during training. My hypothesis was that words not have been seen during training by a particular tag should be considered as unknown words. As can be seen, the performance for the "crude" unknown words handler was miserable. However, when I tested with a "min" unknown words handler, the performance was slightly better (Accuracy: 0.2795). Note here: *I didn't include "min" unknown words handler in code because it was the naive approach which didn't work as intended for other models (e.g. base model).* Even though the performance for the base reduced model was terrible, I think that difference in file size should not be overlooked, and that with a much more sophisticated data structure and unknown word handler, the performance could be much higher while saving a considerable amount of space. My idea to try next would be to have a separate data structure to store all words encountered during training and an additional handler for such words that are not "real" unknown words.