COSI 134: POS tagging with Perceptron

Elena Álvarez Mellado

November 20, 2019

This report describes the implementation of the structured perceptron and averaged perceptron algorithms for an English language POS tagger.

1 Code structure

1.1 `train()`

The main point of entry to the code is the `train()` method. It passes as parameter:

- training set
- test set
- whether the model is averaged or not (`True/False`. Default: `False`)
- features to be ablated (`curr, prevtag`.... Default: `[]`)

If the model is averaged, the weights will be accumulated and updated with an averaged value at the end of each iteration. The variable `iterations` establishes the number of iterations through the training set. I chose to set the number of iterations beforehand (instead of running the model until it converged) because each training iteration took a few hours to run (from 2h to 6h, depending on the model), which made the convergence process very impractical.

Every instance on the training set is sent to the `viterbi()` method (see next subsection), which outputs the most likely sequence of tags and the global feature vector for that sequence. Then, we get the global feature vector for the correct sequence of tags. If we are on the averaged model, both the predicted and the correct sequence of tags are accumulated (weights will be updated every 100 instances). If we are on the not-averaged model, weights will be updated immediately (we sum the corrected feature vector and substract the predicted one).

1.2 `viterbi()`

The `viterbi()` method inputs a sentence and outputs the most likely sequence of tags and the global feature vector for that sequence of tags.

This method creates a `Trellis` object (number of `Columns` = words in the sentence) and populates each `Column` on the `Trellis` with `States`, each `State` representing the possible state of coming from the previous tag (previous `State`) into the current State. Each `State` object stores the feature vector that represents going into that state from the previous `State()`, the score, the current tag at that `State` and the previous `State`. This way, once we reach the end of `viterbi()`, we will be able to retrieve what final `State` produced the highest score and backtrack to get the previous `State` for every `State` for that sequence.

1.3 `create_feature_vector()`

This method creates a feature representation of a given state using the `Vector` object. The feature representation will be encoded as a Python dictionary where the keys are tuples representing the feature and the current tag (for example: `(prev=dog, VBZ)`) and a value (1) that represents that the feature is activated. This method depends on the `features()` method of the `Sentence` class, that returns the set of features for a given position on a given `Sentence`. If a feature was set to be ablated on the `train()` method, it will be deleted.

1.4 `tag()`

This method takes the unannotated data to be tagged (test data) and sends every instance to the `viterbi()` algorithm. The result is a list of `Sentence` objects with the predicted tags.

2 Experiments and results

Two types of experiments were performed: accuracy scores obtained with different number of iterations and feature ablation.

2.1 Accuracy and number of iterations

Different number of iterations were tried for both the averaged and not-averaged model. Although the values were similar for both models, the averaged perceptron produced better results than the not-averaged model (see Table 1). Both models were quite slow to train. Each iteration on the structured perceptron took more than 2h. While one iteration on the averaged perceptron could take more than 5h to run.

Number of iterations	**Perceptron accuracy**	**Averaged perceptron accuracy**
1	0.9368	0.9581
2	0.9425	0.9608
3	0.9455	**0.9622**

Table 1: Accuracy on dev set for perceptron and averaged perceptron models.

Ablated feature	**Accuracy on dev set**
+ all features	0.9368
- current word	0.9331
- previous tag	0.9340
- previous word	0.9361
- previous previous word	0.9378
- next word	0.9363
- next next word	0.9396
- prefix	0.9320
- suffix	0.9182

Table 2: Feature ablation for structured perceptron model (1 iteration).

2.2 Feature ablation

One feature at a time was ablated to see the influence of every feature on the results. The model that was chosen for the ablation experiment was the not-averaged perceptron model (with iterations = 1). The reason to choose this setting (instead of the averaged model, that produced highest scores) is that each iteration of the averaged model took more than 5 hours of training, while the non-averaged model took 3. Therefore, in order to speed up the process, the model that was chosen for the feature ablation experiment was the structured perceptron model with one iteration only.

The accuracy obtained when ablating each feature was slightly lower than the accuracy obtained when all features were used, except for the +2 word and the -2 word. The ablated feature that produced the lowest score was (surprisingly) the suffix feature. This seems to mean that, although all features are relevant, the morphological information of the word (encoded as the last 4 characters) are more relevant in English POS-tagging than the previous tag (in the dataset, at least).