Assignment 2: Part of Speech Tagging

Niranjan Balsubramanian and Jun Kang

CSE 628 Spring 2018

Acknowledgment

This assignment is based on the work of Professor Sameer Singh of University of California, Irvine. We thank Professor Sameer for helping us to build this assignment.

1 Due Date and Collaboration

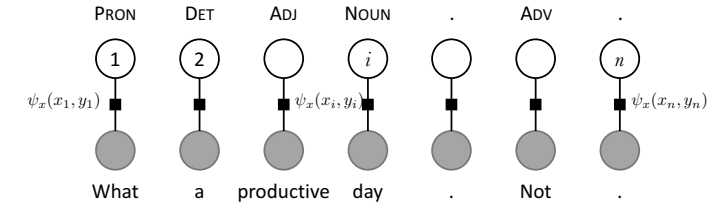
- The assignment is due on Mar 12, 2018 at 11:59 pm. We will accept late submissions until Mar 24 11:59 pm with a penalty of 10 points per day. (Out of 100 points)
- You can collaborate to discuss ideas and to help each other for better understanding of concepts and math.
- You should NOT collaborate on the code level. This includes all implementation activities: design, coding, and debugging.
- You should NOT not use any code that you did not write to complete the assignment.

2 Models

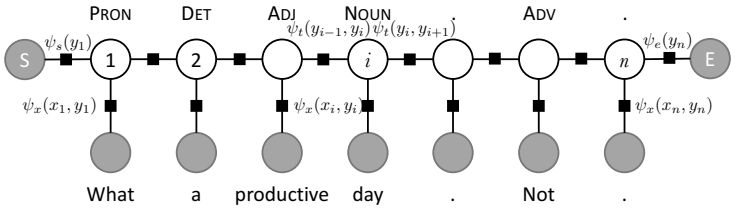
For any given sequence of tokens, $\mathbf{x} = x_1 \dots x_n$, sequence tagging predicts a sequence of labels of the same length, $\mathbf{y} = y_1 \dots y_n$, where $y_i \in \{1 \dots L\}$, the labels of our interest. In discriminative models, we model any sequence of tags $\mathbf{y}$ for input sequence $\mathbf{x}$ with a scoring function $s(\mathbf{y}, \mathbf{x})$, such that the *best prediction* of the model corresponds to the following inference problem:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\text{argmax}}\ s(\mathbf{y}, \mathbf{x}). \tag{1}$$

In the following sections, we will describe the two particular ways of defining s, (a) using an independent classifier, and (b) using a conditional random field.



(a) Logistic Regression Model



(b) Conditional Random Field Model

Figure 1: Comparing Logistic Regression to Conditional Random Fields for Parts-of-Speech Tagging.

2.1 Logistic Regression

When using a logistic regression model, we assume complete independence between the different components y_i in $\mathbf{y}$. Thus the score function factorizes as:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\text{argmax}}\ s(\mathbf{y}, \mathbf{x}) = \underset{\mathbf{y}}{\text{argmax}} \sum_{i=1}^{n} \psi_x(y_i, i, \mathbf{x}) \tag{2}$$

We illustrate this structure in Figure 1a. The classifier, when predicting the label y_i, can use any features of the input sequence $\mathbf{x}$ and the position i. It is sometimes more intuitive to think about the features as being computed for the observed input, but then picking the corresponding parameters according to the y_i label you are trying to predict, i.e.

$$\psi_x(y, i, \mathbf{x}) = \theta_y \cdot \phi(i, \mathbf{x}) \tag{3}$$

The dimensions of θ overall is thus $L \times D$, where L is the vocabulary size of the labels and D is the total number of possible features. The implementation of ϕ, which you will be extending, is the function `token2features` in `feat_gen.py` (more on this in Section 3.1). In particular, for this assignment, we assume each feature is a boolean absence/presence of a particular pattern, i.e. ϕ is binary.

In order to learn parameters from supervised data, we used maximum likelihood training as implemented by `scikit-learn`. For prediction, you just need to find the one of the L labels independently for each token x_i that gives the highest score.

2.2 Conditional Random Field

Conditional random fields are an extension of logistic regression that incorporates sequential information in the labels, while still supporting the use of arbitrary features. The score function for a conditional random field thus combines both the evidence from the observed tokens (ψ_x) and from the neighboring tags (ψ_t, ψ_s, ψ_e):

$$s(\mathbf{y}, \mathbf{x}) = \psi_s(y_1) + \sum_{i=2}^{n} \psi_t(y_{i-1}, y_i) + \psi_e(y_n) + \sum_{i=1}^{n} \psi_x(y_i, i, \mathbf{x}) \tag{4}$$

This structure is shown in Figure 1b. The feature function underlying ψ_x remains the same as in logistic regression as defined in Eq. 3, and thus corresponds to $L \times D$ parameters. There are $L \times 1$ parameters for ψ_s and ψ_e each, and ψ_t is captured by $L \times L$ parameters. Thus the total number of parameters for a CRF are $(2 + L + D) \times L$.

Predicting the best sequence from a CRF is, unfortunately, not as straightforward as in logistic regression. You will have to implement this *decoding* step, the details of which are available in Section 3.2. Given labeled data, and a correct implementation of the Viterbi decoding, the code that uses the structured Perceptron algorithm to learn the parameters is included.

3 What to do

3.1 Feature Engineering

Take a look at `feat_gen.py`. It computes features for a given token (at position i) for a given sentence (a sequence of tokens). The current set of features are pretty basic, they just look at the word, and whether certain default string properties apply to it or not. But note that a *feature* here is just a unique string, such as `WORD=is` or `IS_UPPER`, so you do not have to worry about indexing it as a vector. Further, with the `add_neighs` flag, we also add all the basic features of our neighbors by calling the function recursively, and prefixing the features with a certain keyword. I encourage you to run `python feat_gen.py` to see the features for a simple sentence, and play around with other sentences. Your goal here is to introduce new features and evaluate their utility.

Your code should just extend this function with more features. Feel free to use the provided lexicons or any other external information that you think will be useful for the task. One thing to keep in mind, as you perform feature engineering is that you will have some operations that are expensive and only should be done once, during *preprocessing*. Also keep an eye on how many features you are introducing, since each additional feature can actually increase the number of parameters by quite a bit, which can significantly slow down training. Note that since all the features are boolean, you cannot directly use word embeddings, but of course clustering on top of embeddings can be incorporated as cluster

memberships (i.e. brown clustering style). Finally, by running `feat_gen.py` independent of other files, you can test out whether you are generating the right features, before training a model using them.

In the report, describe what the features that you have implemented are. Try to motivate them (why did you think they'd be useful), describe them in sufficient details, give examples (if it'll be useful to understand), and finally, evaluate how much they helped on the dev set for the logistic regression tagger (Run `data.py` after making your changes in `feat_gen.py`).

3.2 Implement Viterbi decoding

More important than having a good set of features from a model, is that we need to be able to make predictions from it. Unfortunately, the conditional random field implementation included lacks this feature, and when we try to predict from it, gives a pretty stupid sequence. Here you will implement the Viterbi algorithm for sequence tagging.

The main file you will be modifying is `viterbi.py`, which needs a function to compute the best sequence (and its score) given the various transition and emission scores (corresponding to the $\psi_?$s in Section 2.2). The algorithm contains a data structure $T(i, y)$ that maintains the score of the *best* sequence from $1 \dots i$ such that $y_i = y$. This definition is actually recursive, since it depends on the best sequence till $(i-1)$, as follows:

$$T(i, y) = \psi_x(y, i, \mathbf{x}) + \max_{y'} \psi_t(y', y) + T(i-1, y') \tag{5}$$

For a correct implementation, you will have to implement the above, while also taking care of the initial and the final transitions (ψ_s and ψ_e respectively), along with keeping the back pointers to recreate the best sequence.

If your implementation is correct, you should be able to run `python viterbi_test.py` without exceptions and with perfect accuracy (take a look at this file, it just creates and tests random sequences). Your report should include a brief summary (maybe a paragraph) of how you implemented it, and any specific challenges or issues that came up. If you could not get your implementation working, describe where you got stuck.

3.3 Compare Logistic Regression to CRFs

If you have implemented Viterbi correctly, you are now ready to train your CRF tagger using structured Perceptron! You can change the tagger by entering '–model' argument oof `data.py` as below:

```
python data.py --model crf
```

Unfortunately, due to the constant calls being made to Viterbi, the training is quite slow compared to logistic regression, so it might be a while before your results come in (so do not leave this homework till the last day!).

Also, you might want to play a little bit with the hyper-parameters of the structured Perceptron algorithm. (Parameters to struct_perceptron.py.)

Your task for this part is

- (1) To compare the logistic regression and the CRF taggers in terms of their performance,
- (2) To compare the basic features with your enhanced set(s) of features, for both logistic regression and CRF.

Your comparison should include an performance evaluation on the dev dataset: 1) which methods give the highest accuracy, and by how much? 2) Further, can you find/create sentences which highlight your features over the basic ones? 3) Are there sentences for which CRF is much better than logistic regression? Why is it better on these? Use any graphs, tables, and figures to aid your analysis, including ones generated by `conlleval.pl`.

4 Suggestions/Tips

Here are some suggestions that might be useful.

- The provided code is tested with python 2.7.10. It should work with python 2.7.x, but might not be compatible with python 3.X.
- As indicated before, Viterbi part of the assignment is completely independent of anything else, so if you are finding everything overwhelming, just start with `viterbi.py` and `viterbi_test.py`, and ignore everything else.
- If you are concerned whether your Viterbi algorithm is horribly inefficient, the implementation I have, takes ~ 5 seconds to finish the tests with the basic features.
- If your features are somewhat expensive to compute, and the code is constantly stuck at the computing features stage even before getting to the of training, then I suggest you save them to disk. All you need to do is, for each token in each sentence, save the list of strings to file (maybe in the CONLL format, just append tab-separated features to each line). Then, for training, just load this file instead of calling the feature computation code. This'll be particularly useful if you have ML hyper-parameters you want to tune, given a fixed set of features.
- If feature computation finishes fairly quickly, but the CRF is still taking too long to train (compared to without your features), then you have likely introduced too many features.
- Sometimes removing features can be also be helpful in generalization.

5 Data & Files

Refer to the README file.

6 What should you turn in?

Prepare and submit a single zip file with following items:
filename: ⟨SBU_ID⟩.zip (ex. 10111111.zip)

- viterbi.py : completed viterbi implementation
- feat_gen.py : modified to contain your features
- twitter_test.lr.pred : predictions using LR on test dataset
- twitter_test.crf.pred : predictions using CRF on test dataset
- report.pdf : The report should have following contents
  - Description of your viterbi implementation
  - Description of the added features
  - Comparison of the your features against the basic features
  - Comparison of Logistic Regression and CRFs

7 Grading

Here is a coarse grained grading rubric.

- 30 points – Feature Engineering
- 30 points – Viterbi Implementation
- 30 points – Report
- 10 points – Performance on the test dataset.

* 1 point will be deducted for each case of not following the specified file names.