CS 295: Statistical NLP: Winter 2018

Homework 3: Sequence Tagging of Tweets

Sameer Singh
http://sameersingh.org/courses/statnlp/wi18/

A number of tasks in natural language processing can be framed as sequence tagging, i.e. predicting a sequence of labels, one for each token in the sentence. Such tasks include more finer grained tasks such as tokenization and chunking, but also coarse-level part of speech tagging and named entity recognition. In this homework, you will be looking the latter two for a corpus of tweets, and investigating two challenges in sequence modeling, inference and feature engineering. The submissions are due by midnight on **February 27, 2018**.

1 Task: Parts of Speech and Named Entity Recognition on Twitter

The primary tasks of this homework are to perform supervised parts-of-speech tagging and named entity recognition for Twitter data.

1.1 Models for Sequence Tagging

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:

$$\hat{\mathbf{y}} = \operatorname*{argmax}_{\mathbf{y}} s(\mathbf{y}, \mathbf{x}). \tag{1}$$

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.

1.1.1 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:

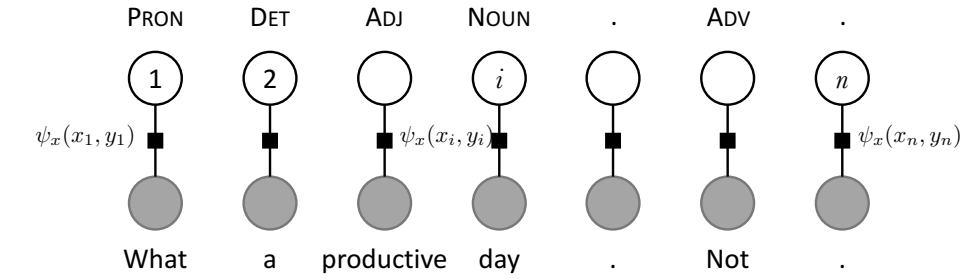
$$\hat{\mathbf{y}} = \operatorname*{argmax}_{\mathbf{y}} s(\mathbf{y}, \mathbf{x}) = \operatorname*{argmax}_{\mathbf{y}} \sum_{i=1}^{n} \psi_x(y_i, i, \mathbf{x}) \tag{2}$$

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. As we describe in class, 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.

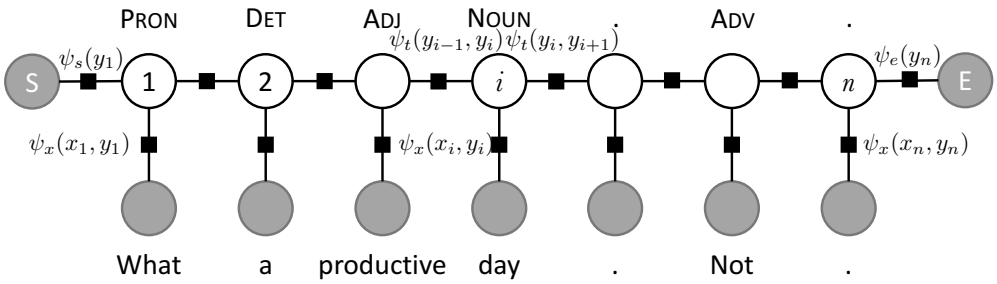
$$\psi_x(y, i, \mathbf{x}) = \theta_y \cdot \phi(i, \mathbf{x}) \tag{3}$$

The dimensions of θ overall is thus $L \times D$, where 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 2.1). 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`, which is the implementation you used in Homework 1 as well. The inference is, of course, not a problem here since for prediction, you just need to find the one of the L labels independently for each token x_i that gives the highest score.



(a) Logistic Regression Model



(b) Conditional Random Field Model

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

1.1.2 Conditional Random Fields

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 2.2. Given labeled data, and a correct implementation of the Viterbi decoding, I have included code that uses the structured Perceptron algorithm to learn the parameters.

1.2 Data

The sequence labeling tasks you will be investigating in this homework are part-of-speech tagging and named entity recognition on Twitter. I have provided the data archive on Canvas which contains the labeled corpus for both of these tasks, with a train and dev split for you. The test data for the assignment will be released to you close to the homework deadline, in order to prevent excessive feature engineering and tuning specific to the test data. The format of the files is pretty straightforward[1], it contains a line for each token (with its label separated by a whitespace), and with sentences separated with empty line. See Figure 2 for an example, and examine the text files yourself (always a good idea).

- *POS Tagging:* Contains tweets annotated with their *universal* parts-of-speech tags, with 379 tweets for training and 112 for dev, and 12 possible part-of-speech labels. The test corpus will contain ~ 300 tweets.
- *Named Entity Recognition:* Contains tweets annotated with their named entities in the BIO format (21 possible classes, for 10 entity types). There are 1804 tweets in training, 590 in dev, and the test set will have 3850 tweets in the test set.

[1] This format, with support for some basic features, is also known as the CONLL format.

```
@paulwalk X
It  PRON
's  VERB
the DET
view  NOUN
from  ADP
where ADV
I PRON
'm  VERB
living  VERB
for ADP
two NUM
weeks NOUN
. .
Empire  NOUN
State NOUN
Building  NOUN
= X
ESB NOUN
. .
Pretty  ADV
bad ADJ
storm NOUN
here  ADV
last  ADJ
evening NOUN
. .
```

(a) Parts-of-speech Tagging

```
@paulwalk O
It  O
's  O
the O
view  O
from  O
where O
I'm O
living  O
for O
two O
weeks O
. O
Empire  B-facility
State I-facility
Building  I-facility
= O
ESB B-facility
. O
Pretty  O
bad O
storm O
here  O
last  O
evening O
. O
```

(b) Named Entity Recognition

Figure 2: Example annotations for a single tweet.

There are a few other files I am including to help you out.

- *Lexicons*, `lexicon`: Each file in this directory is a list of names of a certain type (one per line), for example `people.person` contains a long list of all the people from Wikipedia[2]. Most of these should be self-sufficient from their names, but if not, you can discuss the ones you cannot figure out on Piazza. These should be useful for designing new features, as required in Section 2.1.
- *Evaluation script*, `conlleval.pl`: This is the official evaluation script for the CONLL evaluation. Although it computes the same metrics as Python code I provide, it supports a bunch of features, such as: (a) Latex formatted tables, by using `-l`, (b) BIO annotation by default, turned off using `-r`. In particular, when evaluating the output `pred` files for POS tagging, use `./conlleval.pl -r -d \\t < twitter_dev.pos.pred`, while for named entity recognition, use `./conlleval.pl -d \\t < twitter_dev.ner.pred`.

1.3 Source Code

I have released the initial source code, available at https://github.com/sameersingh/uci-statnlp/tree/master/hw3. This time there are quite a few files, but most of them you do not HAVE to change at all (but might find useful to modify a bit).

- **`data.py`**: The primary entry point that reads the data, and trains and evaluates the tagger implementation.
- **`tagger.py`**: Code for two sequence taggers, logistic regression and CRF. Both of these taggers rely on `feats.py` and `feat_gen.py` to compute the features for each token. The CRF tagger also relies on `viterbi.py` to decode (which is currently incorrect), and on `struct_perceptron.py` for the training algorithm (which also needs Viterbi to be working).
- **`feats.py`** and **`feat_gen.py`**: Code to compute, index, and maintain the token features. The primary purpose of `feats.py` is to map the boolean features computed in `feats_gen.py` to integers, and do the reverse mapping (if you want to know the name of a feature from its index). `feats_gen.py` is used to

[2] Actually, from Freebase, a structured version of Wikipedia

compute the features of a token in a sentence, which you will be extending. The method there returns the computed features for a token as a list of strings (so does not have to worry about indices, etc.).

- **`struct_perceptron.py`**: A direct port (with negligible changes) of the structured perceptron trainer from the `http://pystruct.github.io` project. Only used for the CRF tagger. The description of the various hyper-parameters of the trainer are available here, but you should not modify this file, but instead change them in the constructor in `tagger.py`.
- **`viterbi.py`** and **`viterbi_test.py`**: General purpose interface to a sequence Viterbi decoder in `viterbi.py`, which currently has an incorrect implementation. Once you have implemented the Viterbi implementation, running `python viterbi_test.py` should result in successful execution without any exceptions.

By default, running `python data.py` will run logistic regression with the basic features on POS tagging, and prints the evaluation metrics and write out the prediction files in the `data` directory. I encourage you to run this, followed by `conlleval.pl`, to get an understanding of the output and evaluation. The files that you certainly have to change (and include as part of your submission) are `viterbi.py` and `feat_gen.py`. More details about what you need to implement in the sections below.

2 What to Submit?

Prepare and submit a single write-up (**PDF, maximum 5 pages**) and your modified `viterbi.py` and `feat_gen.py` (compressed in a single `zip` or `tar.gz` file; we will not be compiling or executing it, nor will we be evaluating the quality of the code) to Canvas. Note that Part 1 and Part 2 are completely independent of each other, so you can start with either first (Part 3 builds upon both). The write-up and code should address the following.

2.1 Feature Engineering (35 points)

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. However, you can imagine many other kinds of features that would be useful for both part-of-speech tagging and named entity recognition, and thus 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 writeup, 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 and test set for the logistic regression tagger (you should just need to execute `data.py` after making your changes in `feat_gen.py`).

2.2 Implement Viterbi decoding (35 points)

More important than having a good set of features from a model, we need to be able to make predictions from it. Unfortunately, the conditional random field implementation I have included lacks this feature, and when we try to predict from it, gives a pretty stupid sequence. Thankfully, we have covered the use of dynamic programming multiple times in the class, and thus, here you will implement one of them here, 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 1.1.2). As a

reminder from class, the algorithm contains a data structure $T(i, y)$ that maintains the score of the *best* sequence from $1 \ldots i$ such that $y_i = y$. We saw does 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). The write up should just 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.

2.3 Compare Logistic Regression to CRFs (30 points)

If you have implemented Viterbi correctly, you are now ready to train your CRF tagger using structured Perceptron! Change the tagger that is used in `data.py` currently to the CRF one, and fire it up. Unfortunately, due to the constant calls being made to Viterbi, the training is actually 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 have to play a little bit with the hyper-parameters of the structured Perceptron algorithm.

Your task for this part is to compare (1) the logistic regression and the CRF taggers to each other, (2) compare the basic features with your enhanced set(s) of features, for both logistic regression and CRF. Your comparison should include an aggregate performance evaluation on the dev and test datasets, which methods give the highest accuracy, and by how much? Does it depend on the task, POS or NER? If so, why do you think that matters? Further, can you find/create sentences which highlight your features over the basic ones? 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`.

2.4 Extra Credit: Incorporate features into CRF transitions (20 points)

When are are modeling the transition in the above version of the CRF, we are only using the observed features to score the *emission* factors, and the transition factors independently score the label sequence. However, one can imagine that the features may be also useful in identifying the sequence of labels, and thus we can formulate an alternate CRF model where we do not differentiate between the *emission* and the *transition* factors, i.e.:

$$s(\mathbf{y}, \mathbf{x}) = \sum_{i=1}^{n} \psi_t(y_{i-1}, y_i, i, \mathbf{x}) \tag{6}$$

This function will use $L \times L \times D$ parameters[3]. In order to implement such a model, you will have to: (1) implement a new `Tagger`, (2) ensure `joint_feature` returns the correct feature vector, and finally, (3) implement a Viterbi algorithm that works with such scores. Of course, this requires a little bit more familiarity with my code, so if you are considering doing this, it might be worth meeting with me.

In your write-up, you should include the details about the description of your model and the Viterbi implementation (what's the recursion?), and compare the accuracy results on both the datasets to the logistic regression and the vanilla CRF models.

3 Suggestions/Tips

This is a fairly long description of a homework, and a lot of code for you to look at, so I thought I'll provide some suggestions that might be useful.

- As indicated before, Part 2 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, my implementation, running on a four-year old Macbook Pro, takes $\sim$ 5 seconds to finish the tests (and I believe it'll difficult to make it much faster).

[3] Actually, a bit more, depending on how you incorporate the first and the last transition

- 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. Note, this is a bit of an overhead, so if you're not sure where in my code to do this, talk to me.
- 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.
- For comparing different models, especially if you are using hand-constructed sentences, it will obviously be helpful not to train the models every time. Consider serializing them to disk by using pickle (you should just need to save the weight vectors). Again, if loading the weights is getting confusing, meet me.

4 Statement of Collaboration

It is **mandatory** to include a *Statement of Collaboration* in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using Piazza) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content *before* they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to Piazza, etc.). Especially *after* you have started working on the assignment, try to restrict the discussion to Piazza as much as possible, so that there is no doubt as to the extent of your collaboration.

Since we do not have a leaderboard for this assignment, you are free to discuss the numbers you are getting with others, and again, I encourage you to use Piazza to post your results and comparing them with others.

Acknowledgements

This homework was made possible with the help (and generosity) of Prof. Alan Ritter of the Ohio State University. Thanks, Alan!