CSCI 544: Applied Natural Language Processing

Assignment 6: Named Entity Recognition

Nanyun Violet Peng (Adapted from Sameer Singh)

out: Oct 20, 2017
due: Nov 15, 2017 (Note extra time! This assignment is lengthy so start early!)

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. Named entity recognition (NER) is one of them. In this homework, you will be looking into NER for a corpus of tweets, and investigating two challenges in sequence tagging problems: inference and feature engineering. Before you begin, please download hw6.tar.gz from Blackboard.

You should use the Python code in this archive when you program your own solutions to this assignment, filling in the functions indicated. You may add additional code or functions as long as the basic interface is not altered. All code should be uploaded to Vocareum for documentation purposes; we may test your code to confirm it behaves according to expectation, but we will mainly test the behavior of `viterbi.py` as noted in Question 2.2, and will evaluate your submitted predictions on the test data in Question 3.3. As usual, all work must be your own and may not be copied from or modified from others, including those you know or don't know.

Task: Named Entity Recognition on Twitter

The primary task of this homework is to perform supervised named entity recognition for Twitter data.

Data

The data for this homework comes from Twitter, with named entities labeled. I have provided the data with a train, dev, and dev_test split for you. Additionally, a test set is provided, however, you will not be able to evaluate this set properly. The only way to obtain an evaluation on test data is to submit answers to Vocareum, and the opportunities to do that will be limited. So, *be cautious about how to use the available data: you should train on the train proportion, tune your hyper-parameters based on the performance on the dev data, and occasionally peek at the performance on your dev_test data to make sure things are going as you expected. You want to avoid excessive feature engineering and tuning specific to the dev_test data, because it will not generalize to the unseen real 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 by empty lines. Examining the data for a while before you start doing the work is highly recommended. (Note that for the test data, the labels are all "O". These are not the real labels, which are hidden from you.)

- *Some explanation about the data:* The NER dataset contains tweets annotated with their named entities in the BIO format (**B**eginning of an entity, **I**nside an entity or **O**utside of entities. For 10 entity types, 21 possible classes). There are 1804 tweets in training, 590 in dev, 703 in dev_test, and the test set will have 3147 tweets.

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

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-explanatory 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.

- *Evaluation script*,
  `conlleval.pl`: This is the official evaluation script for the CONLL evaluation. Although I provided some evaluation measurement in the python script for you to verify during training, the final evaluation will be solely based on the CONLL evaluation. To use the CONLL script you can run `data/conlleval.pl -d \\t < twitter_dev.ner.pred` (for the dev set). Substitute the dev_test prediction file to evaluate devtest. Note that the evaluation results you get for test will **not** be accurate because you do not have the true test labels.

Source Code

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).

- `main.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 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 you do 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 hyperparameter settings 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 main.py
```

will run logistic regression with the basic features on NER tagging. It will train on train (`data/twitter_train.ner`), evaluate on dev (`data/twitter_dev.ner`), print the evaluation metrics, and write out the prediction files for each set. I encourage you to run this, followed by `conlleval.pl`, to get an understanding of the output and evaluation. You can evaluate on dev_test by running

```
python main.py -e data/twitter_dev_test.ner
```

You can evaluate on both at the same time by running

```
python main.py -e data/twitter_dev.ner data/twitter_dev_test.ner
```

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 are in the sections below.

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

1 Feature Engineering (30 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 is 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 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 clustering features can be incorporated as cluster memberships (i.e. `is_cluster_k=True/False`). 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.

1.1 Feature design explanation (10 points)

In the writeup, describe what features you have implemented. Briefly explain why you think they are useful, describe them in sufficient details, give examples (if it is helpful to understand).

1.2 Implementation of additional features (20 points)

Implement the features you described and evaluate how much they helped on the dev and dev_test set for the logistic regression tagger (you should just need to execute `main.py` after making your changes in `feat_gen.py`). Briefly write down your improvements.

2 Viterbi decoding (30 points)

More important than having a good set of features for a model, we need to be able to make predictions from it. We have this for the logistic regression classifier. Unfortunately, the conditional random field implementation I have included lacks this part. When you try to make predictions from it now, it only returns a stupid dummy sequence. Thankfully, the algorithm used in CRF to make predictions is a dynamic programming algorithm that everybody loves! Therefore, here you will implement the famous Viterbi algorithm for sequence tagging.

2.1 Pseudocode for Viterbi algorithm (15)

The Viterbi algorithm is a dynamic algorithm that computes the best sequence (and its score) given the various transition and emission scores. The algorithm contains a data structure $T(i, y)$ that maintains the score of the *best* sequence from $1 \dots i$ such that $y_i = y$. You can see that this definition is actually recursive, since it depends on the best sequence till $(i-1)$, as follows:

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

where $\psi_x()$ and $\psi_t()$ are emission and transition scores, respectively. For a correct implementation, you will have to implement the above, while also taking care of the start and the end scores ($\psi_s()$ and $\psi_e()$, respectively), along with keeping track of the back pointers to recreate the best sequence.

In Crowdmark, please describe the pseudocode for Viterbi algorithm. It might be helpful to look at the current `viterbi.py` to get a sense of the expected inputs and outputs. It might also be helpful to look at the brute force algorithm in `viterbi_test.py` to get some inspiration.

2.2 Implement Viterbi algorithm (15)

Since you have the pseudocode in place, let us also go ahead and implement it! The main file you will be modifying is `viterbi.py`, which needs a function to compute the best sequence (and its score) given the start, end, transition and emission scores.

If your implementation is correct, you should be able to run `python viterbi_test.py` without exceptions and pass all the test cases (take a look at this file, it just creates and tests random sequences). Upload `viterbi.py` to Vocareum and your implementation will be confirmed. Note, you may **not** use any library Viterbi function (should one exist) or machine learning libraries such as scikit, but you may use numpy, scipy, and other math libraries.

3 Compare Logistic Regression to CRFs (30 points)

3.1 Analytics (10 points)

From your understanding, which model will work better for the sequence tagging problem? Logistic regression or CRFs? Why? You can give examples if you find them helpful for explanation.

3.2 Experiments (15 points)

If you have implemented Viterbi correctly, you are now ready to train your CRF tagger! You can change the tagger that is used in `main.py` to CRF by invoking as follows:

```
python main.py -T crf
```

You can also include the `-e` options to change the evaluation sets, as needed. Note that the `-T` option can be either `crf` or `logreg`.

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!`). A reasonable estimate of the time to train and evaluate is 30 minutes; if it's taking more than an hour you're probably using too many features, have a suboptimal viterbi implementation, or are doing something else wrong. Please discuss on piazza, come to office hours, etc.! Also, you might have to play a little bit with the hyper-parameters of the structured perceptron algorithm to get desirable results.

Your task for this part is to compare (1) the logistic regression and the CRF taggers to each other with the basic feature, (2) compare the two models again with your enhanced set(s) of features. Your comparison should include an aggregate performance evaluation on the dev and dev_test datasets (average F1 or accuracy).

Please write up the following: in each case, which methods give the highest performance, and by how much? Briefly explain why. Use any graphs, tables, and figures to aid your analysis, including ones generated by `conlleval.pl`.

3.3 Performance on Test (5 points)

Upload your `twitter_test.ner.pred` to Vocareum and your score will be calculated. You can upload `twitter_dev.ner.pred` and `twitter_dev_test.ner.pred` too to make sure you agree with the scorer's numbers. Your score on this question will be based on how well you do relative to others in the class on test, which you can check ahead of time on the leaderboard. You will be limited to 10 submissions per day and will receive limited feedback on your test score, to avoid too much overfitting.

4 Evaluation Metrics (10 points)

When you run the experiments, you will find that the evaluation metrics in the provided python script and the CONLL evaluation script are different. Also, the CONLL script reports both accuracy and F1 scores.

Which metrics do you think are the best? Why? When you explain, compare both metrics, and elaborate on your reasons for choosing one over the other.

What to Submit?

Provide answers in crowdmark to 1.1, 1.2, 2.1, 3.1, 3.2, and 4. Upload `viterbi.py` and `twitter_test.ner.pred` for scores on 2.2 and 3.3, respectively. Upload all the code you used to obtain your final classifier and provide a description of how to run it in a `readme` file; we will select several submissions at random to confirm your implementation. You do not need to upload data, but if you use data beyond that provided, you should describe what was used in your `readme`, and be prepared to supply it upon request.

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.

- If you are concerned whether your Viterbi algorithm is horribly inefficient, my implementation, running on a seven-year old iMac desktop, takes ~ 7 seconds to finish the tests.
- If your features are somewhat expensive to compute, and the code is constantly stuck at the computing features stage even before getting to the 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 plan to tune the hyper-parameters of the Structured Perceptron, given a fixed set of features. Note, this is a bit of an overhead, so if you're not sure where in the code to do this, ask on Piazza.
- 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 everytime. 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, discuss on piazza.

Acknowledgements

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