CS 272: Statistical NLP: Spring 2021

Homework 3: Sequence Tagging of Tweets

Sameer Singh (and Anthony Chen)
https://canvas.eee.uci.edu/courses/37063

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: context and label dependencies. You will be using the PyTorch library for doing the assignment.

The submissions are due by midnight on **May 21, 2021**.

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

The primary tasks of this homework are to perform supervised parts-of-speech (POS) tagging and named entity recognition (NER) for Twitter data. We will first formalize the task, describe the neural architecture for tagging, and then describe the data and the source code available.

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:

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

1.1 Neural Tagging

The architecture of a typical neural tagger is shown in Figure 1, containing two major components:

- **Embedding:** Converts a one-hot representation of a word into a dense vector. By default, we initialize this mapping randomly, but other options include using word2vec or Glove (two popularly available word embeddings), using subword models, or using contextual embedders like ELMo or BERT.
- **Encoding:** Computes a representation at each position that is based on the context. In the configuration file, this module ignores the context, but can be replaced to various recurrent neural networks.

The representation from the encoder is used to predict the output label at every position. Thus, when using an encoder that takes the context into account, the output predictions are based on looking at the global context, not just at the word itself. However, predictions at each position is still independent of predictions at other positions.

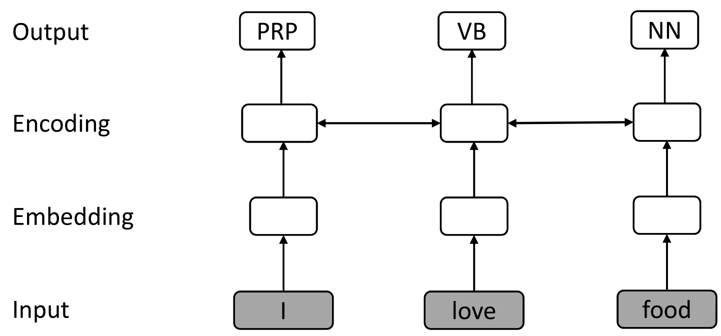

Figure 1: Simple Neural Tagger Architecture

```
@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
. .
```

(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
```

(b) Named Entity Recognition

Figure 2: Example annotations for a single tweet.

Conditional random fields, on the other hand, incorporate sequential information *of 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):

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

A neural CRF model just computes the $\psi_x(y_i, i, \mathbf{x})$ score using the neural tagger architecture described above, i.e. the output of the neural tagger is used as *emission* scores, and transition scores are used to predict the whole label sequence. There are $L \times 1$ parameters for ψ_s and ψ_e each, and ψ_t is captured by $L \times L$ parameters.

Predicting the best sequence from a CRF is, unfortunately, not as straightforward as when predicted independently. You will have to implement this *decoding* step, the details of which are available in Section 2.3.

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. We 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.

1.3 Source Code and Configuration Files

We 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 which you do not need to change at all.

- **`metric.py`** and **`metric_test.py`**: General purpose interface to metrics useful to keep track of when training in `metric.py`. All metrics have been implemented except for the function `__call__` in `AccuracyPerLabel`. The goal of `AccuracyPerLabel` is to keep track of the accuracy per tag so that you can get a more granular understanding of model performance. Once you have finished `AccuracyPerLabel.__call__` implementation, running `python metric_test.py` should result in successful execution without any exceptions.
- **`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.
- **`config/simple_tagger_crf_{pos,ner}.json`**: Configuration files for `simple_tagger` model for POS and NER tasks respectively, i.e. the neural tagger that doesn't use a CRF.
- **`config/neural_crf_{pos,ner}.json`**: Configuration file for `neural_crf` model for POS and NER tasks respectively, which will not work correctly till Viterbi is implemented.
- **`neural_crf.py`**: Neural CRF implementation that uses `viterbi.py` for decoding the labels.

The files that you certainly have to change (and include as part of your submission) are `metric.py`, `viterbi.py`, `config/simple_tagger_{pos,ner}.json` and `config/neural_crf_{pos,ner}.json`. More details about what you need to implement are given in the sections below.

2 What to Submit?

Prepare and submit a single write-up (**PDF, maximum 5 pages**) and source code containing **the config files, `metric.py`, and `viterbi.py`**, along with any other files you modified, (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. The write-up and code should address the following.

2.1 Implementing `AccuracyPerLabel` Metric (15 points)

We have provided a set of `Metrics` in `metric.py` that track useful statistics during training and evaluation. For example, the `Accuracy` metric simply accumulates overall accuracy statistics until it is reset. However, accuracy over all possible labels (*i.e.,* tags) is not that informative. It would be useful to also have accuracy values for each label to see if models to better on some labels.

We have written most of this in the `AccuracyPerLabel` class. However, the `__call__` function is not yet complete. You will have to finish this function so that the counts per label is being tracked. If your implementation is correct, you should be able to run `python metric_test.py` without any errors. If you are having trouble, look at how counts are being accumulated in `Accuracy.__call__`. Also take a look at `metric_test.py` and try to work out by hand how the gold values were computed.

2.2 Improving Simple Tagger (20 points)

We have provided a configuration file for running the simple neural tagger as `config/simple_tagger_ner.json` (and one for POS tagging, omitted henceforth for brevity). To train the model and evaluate it, run:

```
python train.py ./config/simple_tagger_ner.json -s ./model/simple_tagger_ner
```

To understand what the tagger is doing, familiarize yourself with the configuration file. For embedding, the tagger is using a 50-dimensional embedding for each word initialized randomly. The encoding type is *null* to capture no encoding, and thus no context information is used to predict the labels. This means the embeddings themselves are used to predict tags.

The main goal of this part of the assignment is to improve the tagger, and in the process, compare different embedding and encoding techniques. To achieve this, you will only need to modify the configuration files, primarily in the `model` section, but potentially in the `training` section as well. In particular, you should try a few different encoders and embedders, along with the hyper-parameters, and evaluate your changes on the provided dev data. Further, we encourage you to tune and evaluate your models for POS and NER tasks independently. (i.e., your best tuned models will have different parameters/configurations).

In the writeup, you should describe the process by which you reached your best model. As results, include both the dev and the test accuracy as computed by the models, in particular focusing on (1) does using pre-trained embeddings help? (2) does adding an encoder provide improved results?, (3) are you able to get further gains when you change both the embedder and the encoder, and (4) which tags have their accuracy improved the most by improvements to the model, and why do you think this is the case? The quantitative results should be provided in the write-up, possibly as a table. For per-tag accuracy scores, do **not** include them all, **just the per-tag scores which have changed the most**. Since you have two tasks, describe the difference, if any, between what works for one versus the other.

2.3 Implement Viterbi decoding (35 points)

More important than tuning a model, we need to be able to make predictions from it. Unfortunately, the conditional random field implementation we 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 CRF 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). As a reminder from class, the algorithm contains a data structure $T(i,y)$ that maintains the score of the *best* sequence from $1 \dots i$ such that $y_i = y$. We saw this definition is actually recursive, since it depends on the best sequence till $(i-1)$, as follows:

$$T(i,y) = \psi_x(y,i,\mathbf{x}) + \max_{y'} \psi_t(y',y) + T(i-1,y') \tag{3}$$

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.

2.4 Improve and Compare CRFs to Simple Tagger (30 points)

If you have implemented Viterbi correctly, you are now ready to train your neural CRF tagger! Change the command to run a neural CRF tagger as follows, and fire it up (add `_ner` or `_pos`).

```
python train.py ./config/neural_crf_ner.json -s ./model/neural_crf_ner
```

Unfortunately, due to the constant calls being made to Viterbi, the training is actually much slower than simple tagger, so it might be a while before your results come in (**so do not leave this homework till the last day!**). Examine the configuration file to see that it is similar to the simple tagger, i.e. similar embedders and encoders.

Your goal for this part is to improve the neural CRF implementation as before: by varying the embedders, encoders, and hyper-parameters of the training. In particular, focus your evaluation (both quantitative and qualitative) to answer the following questions: (1) Does the default CRF implementation (i.e. original configuration with your Viterbi implementation) outperform the default simple tagger?, (2) Does your best simple tagger outperform the CRF with original configuration?, and (3) Does your best CRF outperform the simple tagger with the same, or best, configuration? Do the answer to these depend on the task, POS or NER? For quantitative evaluation, include the accuracy on both dev and test sets and the per-tag accuracy scores **which are changed most** by using the CRF. For qualitative evaluation, if the CRF changes the performance on certain tags by a great degree or if it improves certain configurations of the simple tagger, provide an explanation as to why this may be.

3 Suggestions/Tips

This is a fairly long description of a homework, and a lot of code for you to look at, here are some suggestions that might be useful.

- Part 1 should be done before doing Part 2 and Part 4 so that you get per-tag accuracies during training. Thus `metric.py` and `metric_test.py` are a good place to start since you only need to add a couple lines.

- Part 3 is also not dependent on anything else, so if you are finding everything overwhelming, you can also start at `viterbi.py` and `viterbi_test.py`, and ignore everything else.
- If you are concerned whether your Viterbi algorithm is horribly inefficient, our implementation, running on a four-year old Macbook Pro, takes ~ 5 seconds to finish the tests (and I believe it'll difficult to make it much faster).
- More complex encoder/embedder is not always better.
- There are many possible changes to make. I suggest making one change at a time and making informed decisions on which keep.
- You will get approximately 64% and 94% dev accuracy with our given `./config/simple_tagger_{pos,ner}.json` for POS and NER tasks respectively.
- If you pass tests with `viterbi_test.py`, you will get approximately 74% and 94% dev accuracy with our given `./config/neural_crf_{pos,ner}.json` for POS and NER tasks respectively.

4 Additional Documentation

4.1 Sample Configuration Changes

You will be able to intuitively change numerical parameters (e.g., **embedding_dim, num_epochs**) and training configurations (e.g., **optimizer**) in a JSON file. However, you may want to refer to the following documents to modify your model configuration. If you start changing size of dimensions, you will need to be careful that they are consistent.

- **Embedding:**

```
"embeddings": {
  "embedding_dim": 50,
  "embedding_path": "glove.6B.50d.txt"
},
```

Figure 3: Example of using pretrained Glove embedding, instead of randomly initialized. You will need to first download this embedding file from https://s3-us-west-2.amazonaws.com/allennlp/datasets/glove/glove.6B.50d.txt.gz and unzip it. Note that we have only tested the code with this embedding file.

- **Encoder:**
  https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

```
"encoder": {
  "type": "torch.nn.LSTM",
  "input_size": 50,
  "hidden_size": 25,
  "num_layers": 1,
  "dropout": 0.1,
  "bidirectional": true
},
```

Figure 4: An example of using a one-layer bidirectional LSTM instead of no encoder

4.2 Train and evaluate your model

To train and evaluate model on train/dev, use the following commands (omitted `_ner` and `_pos` for brevity):

```
python train.py ./config/simple_tagger.json -s ./model/simple_tagger
python train.py ./config/neural_crf.json -s ./model/neural_crf
```

You will use `-s` option to save your model under your specified directory. In this directory, the metrics of the best performing model will be saved in `best_metrics.json`.

4.3 Evaluate Trained Models

If you have already trained your model, and would like to evaluate the trained model again, for example evaluating the simple NER model on the dev data, use the following:

```
python evaluate.py ./model/simple_tagger_ner ./data/twitter_dev.ner
```

5 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 Campuswire) 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, etc.). Especially *after* you have started working on the assignment, try to restrict the discussion to Campuswire 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 Campuswire to post your results and comparing them with others.

Acknowledgements

This homework was made with the help and generosity of Prof. Alan Ritter from Georgia Tech. Thanks, Alan!