CS 388 Assignment 2: Sequential CRF for NER

Due date: September 30th, 2021 at 11:59pm CST

Collaboration Policy This assignment can be done in a group of two. If your partner has already created a team, join their team. Otherwise, enter a name for your team and continue. GitHub Classroom will provide a link to your team's private repository for you to clone to your computer. All team code should exist in the repository. Only the code that is present in the master branch of your repository by the due date will be evaluated as part of your submission! Please feel free to discuss the homework assignments with other students and work towards solutions together. However, all of the code you write must be your (and your teammate's) own! Your writeup must be your (and your teammate's) own as well.

==========

- **Github Classroom invite link**: `https://classroom.github.com/g/UObvdEYP`

==========

Overview: You will implement a CRF sequence tagger for NER. You'll implement the Viterbi algorithm on a fixed model first (an HMM), then generalize that to forward-backward and implement learning and decoding for a feature-based CRF as well. The primary goal of this assignment is to expose you to inference and learning for a simple structured model where exact inference is possible. Secondarily, you will learn some of the engineering factors that need to be considered when implementing a model like this.

The expectation in this project is that you understand CRFs and HMMs from scratch using the code we give you. **You should not call existing CRF libraries in your solution to this project.**

Background Named entity recognition is the task of identifying references to named entities of certain types in text. We use data presented in the CoNLL 2003 Shared Task (Tjong Kim Sang and De Meulder, 2003). An example of the data is given below:

```
Singapore NNP I-NP B-ORG
Refining NNP I-NP I-ORG
Company NNP I-NP I-ORG
expected VBD I-VP O
to TO I-VP O
shut VB I-VP O
CDU NNP I-NP B-ORG
3 CD I-NP I-ORG
. . O NONE O
```

There are four columns here: the word, the POS tag, the chunk bit (a form of shallow parsing—you can ignore this), and the column containing the NER tag. NER labels are given in a BIO tag scheme: beginning, inside, outside. In the example above, two named entities are present: `Singapore Refining Company` and `CDU 3`. `O` tags denote text not part of a named entity. `B` tags indicate the start of a named entity, and `I` tags indicate the continuation of the previous named entity. Both `B` and `I` tags are hyphenated and contain a type after the hyphen, which in this dataset is one of `PER`, `ORG`, `LOC`, or `MISC`. In HW1, we considered a simplified tag set which did not distinguish `B` and I. A `B` tag can immediately follow another `B` tag in the case where a one-word entity is followed immediately by another entity. However, note that an `I` tag can only follow an `I` tag or `B` tag of the same type.

An NER system's job is to predict the NER chunks of an unseen sentence, i.e., predict the last column given the others. Output is typically evaluated according to chunk-level F-measure.[1] To evaluate a single sentence, let C denote the predicted set of labeled chunks represented by a tuple of (label, start index, end index) and let C^* denote the gold set of chunks. We compute precision, recall, and F_1 as follows:

$$\text{Precision} = \frac{|C \cap C^*|}{|C|}; \quad \text{Recall} = \frac{|C \cap C^*|}{|C^*|}; \quad F_1 = \frac{1}{\frac{1}{2}\left(\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}\right)}$$

The gold labeled chunks from the example above are (ORG, 0, 3) and (ORG, 6, 8) using 0-based indexing and semi-inclusive notation for intervals. This differs from HW1 which computed token level F1 score.

To generalize to corpus-level evaluation, the numerators and denominators of precision and recall are aggregated across the corpus. State-of-the-art systems can get above 90 F_1 on this dataset; we'll be aiming to get close to this and build systems that can get in at least the mid-80s.

==========

Task

You will be building a CRF NER system. This is broken down for you into a few steps:

1. Implementing Viterbi decoding for a generative HMM.
2. Generalizing your Viterbi decoding to forward-backward to compute marginals, then using those to train a simple feature-based CRF for this task.
3. Extending the CRF for NER.

Getting Started Try running

```
$ python ner.py
```

This will run a very bad NER system, one which simply outputs the most common tag for every token, or `O` if it hasn't been seen before. The system will print a bunch of warnings on the dev set—this baseline doesn't even produce consistent tag sequences.

Data `eng.train` is the training set, `eng.testa` is the standard NER development set, and `eng.testb.blind` is a blind test set you will submit results on. The `deu*` files are German data that you can experiment with in your extension, but you're not required to do anything with these.

Code We provide

`ner.py`: Contains the implementation of `BadNerModel` and the main function which reads the data, trains the appropriate model, and evaluates it on the test set.

`nerdata.py`: Utilities for reading NER data, evaluation code, and functions for converting from BIO to chunk representation and back. The main abstraction to pay attention to here is `LabeledSentence`, which contains a sequence of `Token` objects (wrappers around words, POS tags, and chunk information) and a set of `Chunk` objects representing a labeling. Gold examples are `LabeledSentence` and this is also what your system will return as predictions.

[1] Tag-level accuracy isn't used because of the prevalence of the `O` class—always predicting `O` would give extremely high accuracies!

`utils.py`: Simliar as in HW 1: `Indexer` is as before, with `Indexer` additionally being useful for mapping between labels and indices in dynamic programming. A `Beam` data structure is also provided, but you won't need that in this project unless you choose to do beam search as your extension.

`optimizers.py`: Simliar as in HW 1: three optimizer classes implementing SGD, unregularized Adagrad, and L1-regularized Adagrad. These wrap the weight vector, exposing `access` and `score` methods to use it, and are updated by `apply_gradient_update`, which takes as input a `Counter` of feature values for this gradient as well as the size of the batch the gradient was computed on.

`models.py`: You should feel free to modify anything in this file as you need, but the scaffolding will likely serve you well. We will describe the code here in more detail in the following sections.

Next, try running

```
$ python ner.py --model HMM
```

This will crash with an error message. You have to implement Viterbi decoding as the first step to make the HMM work.

Part 1: Viterbi Decoding

Look in `models.py`. `train_hmm_model` estimates initial state, transition, and emission probabilities from the labeled data and returns a new `HmmNerModel` with these probabilities. Your task is to implement Viterbi decoding in this model so it can return `LabeledSentence` predictions in `decode`.

We've provided an abstraction for you in `ProbabilisticSequenceScorer`. This abstraction is meant to help you build inference code that can work for both generative and probabilistic scoring as well as feature-based scoring. `score_init` scores the initial HMM state, `score_transition` scores an HMM state transition, and `score_emission` scores the HMM emission. All of these are implemented as log probabilities. Note that this abstraction operates in terms of indexed tags, where the indices have been produced by `tag_indexer`. This allows you to score tags directly in the dynamic programming state space without converting back and forth to strings all the time.

You should implement the Viterbi algorithm with scores that come from log probabilities to find the highest log-probability path.

$$P(y_1, ..., y_n | x_1, ..., x_n) \propto P(y_1, ..., y_n, x_1, ..., x_n) = P(y_1|\text{start}) \left[\prod_{i=2}^{n} P(y_i|y_{i-1})\right] \left[\prod_{i=1}^{n} P(x_i|y_i)\right]$$

$$\arg\max_{y_1,...,y_n} P(y_1, ..., y_n | x_1, ..., x_n) = \arg\max_{y_1,...,y_n} \log P(y_1|\text{start}) + \left[\sum_{i=2}^{n} \log P(y_i|y_{i-1})\right] + \left[\sum_{i=1}^{n} \log P(x_i|y_i)\right]$$

Note that this formulation **does not incorporate STOP probabilities!** These aren't too important in practice when using the tagger for posterior inference (as opposed to sampling from it), and it simplifies the code and keeps it more similar to Part 2. So you should simply disregard the transition to STOP in any pseudocode you reference.

If you're not sure about the interface to the code, take a look at `BadNerModel` decoding and use that as a template. FYI, a reference implementation of Viterbi decoding for the HMM gets 76.89 F_1 on the development set, though yours may differ slightly.

Implementation tips

- Python data structures like lists and dictionaries can be pretty inefficient. Consider using numpy arrays in dynamic programs.
- Once you run your dynamic program, you still need to extract the best answer. Typically this is done by either storing a backpointer for each cell to know how that cell's value was derived or by using a backward pass over the chart to reconstruct the sequence.

Part 2: CRF Training

In the CRF training phase, you will implement learning and inference for a CRF sequence tagger with a fixed feature set. We provide a simple CRF feature set with emission features only. While you'll need to impose some kind of sequential constraints in the model, transition features are often slow to learn: you should be able to get good performance by constraining the model to only produce valid BIO sequences (prohibiting a transition to I-X from anything except I-X and B-X).

We provide a code skeleton in `CrfNerModel` and `train_crf_model`. The latter calls feature extraction in `extract_emission_features` and builds a cache of features for each example—you can feel free to use this cache or not.

You should take the following steps to implement your CRF:

1. Generalize your Viterbi code to forward-backward.
2. Extend your forward-backward code to use a scorer based on features. Feel free to use the `FeatureBasedSequenceScorer` included in `models.py` for this purpose. At present, only the most basic features (word-indicator and part-of-speech) are provided. You must select and implement additional features to improve the model's performance and receive full credit.
3. Compute the stochastic gradient of the feature vector for a sentence.
4. Use the stochastic gradient in a learning loop to learn feature weights for the NER system.

To get full credit on the assignment, you should get a score of at least 85 F_1 on the development set. Assignments falling short of this will be judged based on completeness and awarded partial credit accordingly: if you at least get 75-80, for example, we can see that your implementation may be mostly working and you will receive substantial credit. The instructors' reference implementation was able to get 88.2 F_1 using the given emission features and unregularized Adagrad as the optimizer—see if you can beat that!

Implementation Tips

- Make sure that your probabilities from forward-backward normalize correctly! You should be able to sum the forward x backward chart values at each sentence index and get the same value (the normalizing constant). You can check your forward-backward implementation in the HMM model if that's useful.
- When implementing forward-backward, you'll probably want to do so in log space rather than real space. (+, x) in real space translates to (log-sum-exp, +) in log space. Use `numpy.logaddexp`.
- Remember that the NER tag definition has hard constraints in it (only B-X or I-X can transition to I-X), so be sure to build these into your inference procedure. You can also incorporate features on tag pairs, but this is substantially more challenging and not necessary to get good performance.

- If your code is too slow, try (a) making use of the feature cache to reduce computation and (b) exploiting sparsity in the gradients (`Counter` is a good class for maintaining sparse maps). Run your code with `python -m cProfile ner.py` to print a profile and help identify bottlenecks.
- Implement things in baby steps! First make sure that your marginal probabilities look correct on a single example. Then make sure that your optimizer can fit a very small training set (10 examples); you might want to write a small amount of extra code to compute log-likelihood and check that this goes up, along with train accuracy. Then work on scaling things up to more data and optimizing for development performance.

Part 3: Extensions

Here are some possibilities for extensions, but feel free to investigate anything else that you might find interesting. Grading of the extension will be assessed on the basis of how sophisticated your extension is, how thorough your analysis is, and how much evidence there is of substantial effort. A simple extension with fleshed-out analysis can receive full credit:

Features Try features on POS tags and other improvements to the feature set. If you do this, you should do more than just add some a few new features—add detailed quantitative analysis and cite some examples showing what helps and what doesn't. How would feature set impact performance in terms of accuracy and efficiency?

Speed Try making the system faster. This might consist of speeding up feature extraction (is there a way to handle parallel features across tags more effectively?), inference (can we use beam search and structured perceptron training to train the system faster?), or training (are there better optimizers that converge more quickly?).

German You also have access to German NER data—does the system perform well on this data? Can you add features or change it to get better performance? Note that simply running your model on this dataset and reporting results does not constitute a substantial extension.

==========

Submission / Grading

Code / Prediction Performance (45pts) Your code should be in master branch of your repository. In addition, you should have your CRF's output on the `testb` blind test set in CoNLL format and submit to Gradescope. Uncomment the appropriate line in the file to produce this.

Written Report (55pts) A report of around 2-3 pages and should be submitted to Gradescope. The written report should include the followings:

- Model: Description of your approaches and choices you have made. You may use the Analysis section to provide more insight into how you arrived at these decisions, but be sure that the final model is sufficiently summarized here. Also, be sure to describe in this section all of those extensions (Part 3) that you investigated.

- Results: Tables and plots to report performance of your approach. Specify exactly what the numbers and axes mean (e.g., F1, precision, etc).
- Analysis: Perform ablation study of your extensions and key decisions, and discuss the results.
- Implementation Details: Major hyper parameters should be clearly reported and ablated.

Slip Days You may apply slip days to this project as described in the syllabus. If you are working as a pair, it will deduct slip days from **both** members.

References

Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of the Conference on Natural Language Learning (CoNLL)*.

This homework is originally developed by Greg Durrett.