Assignment 3: Sequential CRF for NER

Academic Honesty: Please see the course syllabus for information about collaboration in this course. While you may discuss the assignment with other students, **all work you submit must be your own!**

Goal: In this project you'll 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`. 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:

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

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

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.

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.

Your 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 to use beam search.

Getting Started

Download the data and code from Canvas. You will need Python 3 and numpy. 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 optionally experiment with, 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.

`utils.py`: `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 for Part 3.

`optimizers.py`: Three optimizer classes are provided 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 (25 points)

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. See the provided Viterbi lecture notes for a detailed description of the algorithm.

Note that the code here **does not estimate 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.

The instructors' 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 (50 points)

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**. We do impose constraints in the model to only produce valid BIO sequences (prohibiting a transition to I-X from anything except I-X and B-X), which are implemented as described below.

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. It then loops through each example and applies the gradient updates appropriately.

We also provide a `FeatureBasedSequenceScorer`. This is analogous to `ProbabilisticSequenceScorer`, but implements the CRF potentials on emissions and transitions. Crucially, this model does not rely on transition features, but instead uses hardcoded transitions.

You should take the following steps to implement your CRF:

1. Implement `compute_gradient` by generalizing your Viterbi code to forward-backward and using forward-backward to compute the gradient **for the provided emission features**. You can extend your code to use `FeatureBasedSequenceScorer`, which can be substituted in for `ProbabilisticSequenceScor`

2. Implement `decode` in `CrfNerModel`, which should be able to follow your Viterbi code using the new scorer.

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 The instructors' reference implementation was able to get around 88 F_1 in 3 epochs (taking about 12 minutes) using the given emission features and unregularized Adagrad as the optimizer.

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`.

- The NER tag definition has hard constraints in it (only B-X or I-X can transition to I-X), which are built into the `FeatureBasedSequenceScorer`. **You do not need to learn transition features!**

- If your code is too slow, make sure your forward-backward pass doesn't do unnecessary computation and that you're exploiting appropriate sparsity in the gradients. 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: Beam Search (25 points)

Finally, you should implement **beam search** for your model. Beam search is an approximate inference scheme that only maintains a certain number of hypotheses at each timestep during inference, unlike Viterbi which keeps track of all of them. At each state, the highest-scoring hypotheses are kept in a data structure called a beam.

You should implement this modification in the `decode_beam` method in `CrfNerModel` to implement this. You can use whatever data structures you want to implement the beam, but the `Beam` class provided in `utils.py` provides a natural way to handle a beam of scored hypotheses, although it's not as efficient as it could be.

Your inference (decoding the test set) using a beam size of 2 should be faster (by any margin) than Viterbi while not losing more than 5 F1.

Note that using the `--inference BOTH` flag allows you to run both beam and Viterbi at the same time, to facilitate comparison of these.

Exploration (optional) Try running your model on the German data provided. Does it work well? Take a look at the data and think about why or why not!

Submission and Grading

You will upload your code to Canvas (the Gradescope autograders are unable to handle this assignment).

We will have **two intakes** of the assignment. Any submission received before 11:59pm on Friday, March 5 will be run through the autograder and the grade will be posted to Canvas. You will then be able to resubmit if there are any issues with your code. The second intake will be on the normal assignment due date. This first intake provides a way to verify that you're providing working code to us, so feel free to submit even if your results are still poor to at least verify that everything works correctly.

Any late submissions will be graded as they come in, with some delay.

Your code will be graded on the following criteria:

1. Execution: your code should train and evaluate within the time limits without crashing

2. Accuracy of your models on the development set

3. Accuracy on the blind test set: this is not explicitly reported by the autograder but we may consider it

Make sure that the following commands work before you submit (for Parts 1 and 2+3, respectively):

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

```
python ner.py --model CRF --inference BOTH
```

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)*.