CS395T Project 1: Sequential CRF for NER

Due date: Thursday, September 28, 2017 at 5:00pm

Collaboration As stated in the syllabus, you are free to discuss the homework assignments with other students and work towards solutions together. However, all of the code you write must be your own! Your extension should also be distinct from those of your classmates and your writeup should be your own as well. **You should list your collaborators at the top of your written submission.**

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.

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

There are five columns in this data: the word, the POS tag, the chunk bit (a forward of shallow parsing—you can ignore this), a junk column, 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 <num>` (numbers are abstracted away). `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. The only tags present in this dataset are `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 fifth 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 (label, start index, and end index) and 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!

For example, the labeled chunks from the example above are (ORG, 0, 3) and (ORG, 6, 8) using 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 the high 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 somehow.

Getting Started

Download the data and code from Canvas. You will need Python 2.7+ and numpy.[2] Try running

```
python trainer.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 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

`trainer.py`: Contains the implementation pf `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`: A couple of useful utilities: an `Indexer` which tracks a mapping between n objects and the integers 0 through $n-1$; this is useful for mapping features or labels to indices that will be used in feature vectors or dynamic programs. A `Counter` is also provided, which is really just a wrapper around a dictionary from objects to floats with some convenience methods. 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.

[2] If you don't have `numpy`, see `https://www.scipy.org/install.html`.

`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 trainer.py 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 `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.

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

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

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 gets 77.5 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 hard-coding transition potentials or by constraining the model to only produce valid BIO sequences (for example, by structurally prohibiting O-I transitions).

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—perhaps you might write a `FeatureBasedSequenceScorer` with an interface similar to `ProbabilisticSequenceScorer` for this purpose.
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. The instructors' reference implementation was able to get 88.41 F_1 using Adagrad as the optimizer—see if you can beat that with better features!

Vanilla SGD should be sufficient for you to complete this project. However, if you're feeling ambitious on the optimization front, we've provided an `AdagradTrainer` utility in `adagrad_trainer.py`. This object stores a weight vector and applies ℓ_1 regularization lazily: we remember how many updates have been made since the last time each feature was touched, so we can apply the regularization upon access. This object supports two fundamental operations: applying a gradient update (with the gradient represented as a `Counter`), and access, which returns the value of a feature and fast-forwards the regularization.

Implementation Tips

- Make sure that your probabilities from forward-backward normalize correctly! 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 feature set has hard constraints in it. You may want to build these constraints into your inference procedure, or you can attempt to encode them with soft penalties in the transition scores.
- 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 trainer.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; 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 that interests you!

Training Try using structured SVM instead of a CRF. You can use Viterbi decoding instead of forward-backward now, but remember to incorporate loss augmentation! Compare these two algorithms and see if they differ in how quickly they learn and how robust they are to changes in the optimizer.

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 analysis and cite some examples showing what helps and what doesn't.

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?

Submission and Grading

You should submit on Canvas:

1. Your code
2. Your CRF's output on the `testb` blind test set in CoNLL format. Uncomment the appropriate line in the file to produce this.
3. A report of around 2 pages, not including references, though you aren't expected to reference many papers. Your report should list your collaborators, **briefly** restate the core problem and what you are doing, describe relevant details of your implementation, present results from both your basic CRF model as well as with different system variants as part of your extension, and optionally discuss error cases addressed by your extension or describe how the system could be further improved. Your report should be written in the tone and style of an ACL/NIPS conference paper. Any LaTeX format with reasonably small (1" margins) is fine, including the ACL style files[3] or any other one- or two-column format with equivalent density.

[3] Available at `http://acl2017.org/calls/papers/`

Slip Days Per the syllabus, you have seven slip days to use during the semester, so you may choose to use part or all of your budget on this process to turn it in late as needed.

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)*.