Project 1

1 Introduction

In Named Entity Recognition tasks, we are trying to label, in text, the tokens which correspond to entities such as an organization, a person, a place, or some other miscellaneous entity. A common way of labeling this data is the BIO tag scheme, which consists of labeling each token in a sentence with a B, I or O tag. An O tag represents a token that is not part of an entity name, and a B or I represent tokens which are the beginning or continuation of an entity name, respectively. B and I tokens are also appended with a label describing the type of entity the describe, either Person (PER), Location (LOC), Organization (ORG), or some other entity (MISC).

In this project, our task is to build an NER model capable of predicting the correct BIO tags for each token in a given sentence. The model we use is a Conditional Random Field, a discriminative graphical model capable of predicting BIO tags for sequences of tokens of varying length. This model explains the problem as a set of states and transitions, although in our implementation we choose not to use transition data, instead favoring features that attempt to capture relevant information regarding transitions and by enforcing structure in our decoding.

2 Implementation

In my implementation of CRF, I make two variations from the standard CRF. One of them is during training, regarding the computation of forward-backward score, and the other is during decoding, where I enforce some structure in the decoder to prevent illegal sequences from being produced.

2.1 Training

The item of note done differently in my implementation from a standard CRF is the computation of the forward-backward algorithm. This algorithm has complexity of $O(NT^2)$ where N is the number of tokens in the sentence, and T is the number of tags (9 for BIO tags). However, after making the choice not to include transition features into our model, we can simplify the algorithm's complexity to $O(NT)$ by assuming a transition score of 1 for each transition, which prevents us from having to multiply the forward score of the each previous tag by the transition score before multiplying. Instead we can just sum the previous columns forward scores, and for each tag multiply that sum by the emission score of the tag to get that tag's forward score.
This changes the value of a given forward score to be

$$\alpha_n(t) = score(w_n, t) * \sum_{t'}^{T} \alpha_{n-1}(t')$$

where n is the current position in the sequence, and t is the tag whose forward score at position n is being evaluated.

A similar technique can be used to calculate the backward scores.

2.2 Decoding

To predict a sequence of BIO tags for a given sequence, we use the Viterbi algorithm to find the optimal path of tags that gives us the highest score in the model given the unlabeled sequence of tokens. The only thing of note here is that, because we are not using transition features, we instead assume a transition score of 1 unless the transition is considered illegal. An illegal transition is any transition to an I-tag that does not come from the matching category B-tag (including starting with an I tag). When we make this transition in the Viterbi algorithm, we assign a transition score that is very large and negative, to assure that Viterbi will not take this path as the optimal.

3 Extensions

3.1 Training

In addition to using regular Stochastic Gradient Descent, I used the Adagrad Optimizer as well. Additionally, I implemented batching for both of these trainers and tested several different batch sizes.

The Adagrad Optimizer took a significantly longer amount of time to train than SGD, because of the extra complexity involved when updating

Optimizer	Epochs	Batch Size	Time(s)	F1
SGD (1)	5	1	499.591	83.27
SGD (1)	5	5	542.24	79.36
SGD (1)	5	10	484.938	78.69
SGD (1)	5	20	**479.582**	79.36
SGD (1)	10	1	1016.697	83.13
SGD (1)	15	1	1546.944	**84.19**
Adagrad (0.001, 1)	5	1	1412.670	73.22
Adagrad (0.00001,1)	5	1	1409.472	**86.98**
Adagrad (0.0001, 1)	5	10	917.950	78.47
Adagrad (0.0001, 1)	5	50	**746.403**	70.70
Adagrad (0.0001, 1)	15	50	2010.375	77.11

Table 1: Optimizer and Batch Data. **SGD(step size), Adagrad(lambda, step size)**

and accessing the weights. However, with the same step size as SGD and a lambda value of 10^{-5}, the Adagrad optimizer performed significantly better than SGD, achieving a score of 86.98 on the test data after only 5 epochs, compared to SGD's 79.36.

Training in batches however, proved to hurt test set performance significantly. Both SGD and Adagrad saw performance losses for batch sizes greater than one. As batch size got larger, performance got worse and worse, by a significant amount.

My initial thought was that due to the sparsity of the features, there was little overlap between features in a batch, and thus each feature's gradient was getting scaled down by batch size without receiving the size boost of being affected by each item in the batch. However, scaling up step size to match batch size had no effect on test set performance, and additionally the speed improvements seen by batching the Adagrad optimizer would not have occurred if there were little overlap between features in each batch. Running larger batch sizes for longer time periods did improve performance, but with Adagrad, this will eventually taper off to barely moving at all, and this also makes the speed improvements gained from training in batches obsolete.

I decided that training in batches had little benefit for this model and problem, and significantly hurt training.

3.2 Extended Prediction History

A feature that I wanted to add to the data set is a feature referred to as Extended Prediction History in Ratinov (2009). This feature attempts to take advantage of the fact that the ConLL03 data has a lot of sentences that are locally related. For example, in the dev set has an example "Japan then laid siege to the Syrian penalty area for most of the game but rarely breached the Syrian defense". Then, two sentences later the sentence "Japan coach Shu Kamo said 'The Syrian own goal proved lucky for us'" appears. So we should be able to use information from sentences we've recently decoded to help us make our decisions on the current sentence, because of the way the data is shaped. Intuitively, this could hurt us if we make an incorrect prediction early on in the beginning of a sequence of training sentences that are related, because we would be biased towards making that incorrect prediction again.

Used Feature	Time	F1
CRF	1409.472	86.98
CRF with History	1468.346	**88.27**

Table 2: Comparing gain of Extended Prediction History

I implemented this a bit differently than the reference paper, for simplicity. I create a cache of the last 20 sentences that have been decoded by the model thus far. For each sentence in the cache, only named entities and their BIO tags are stored. When we are extracting features for a given example word, the cache is searched in the direction of most recently added to least recently added to see if we've decoded that word recently. Once the sentence is decoded, we add the sentence and it's predicted BIO tags to the cache, and evict the least recently added sentence (assuming we are at the cache's max capacity, which was set to 20). During training feature extraction, a similar method is used, only instead of decoding after feature extraction, we just add to the cache with the golden BIO tag labels. Adding this feature to the CRF model using the Adagrad optimizer and 5 epochs resulted in an increase in score on the F1 test set of about 1.3, moving us from 86.98 to a score of 88.27. Adding this additional layer of feature extraction does add a bit of time to decoding, but for 5 epochs using Adagrad, it only added about 68 seconds to the total time. Inserting and removing items are fast, because we simply push on the top, and pop on the bottom of the cache, so searching the cache in order is what takes most of that time.

3.3 Further Improvements

If I could discern why batches seem to hurt training so poorly and fix it, then there would be a potential for some serious speed improvements to the system. Already, using batches in Adagrad has significant advantages in terms of speed because of how much less overhead you have per epoch simply because you are updating the gradient fewer times. If this was to be parallelized, then Adagrad training would probably get close to, or beat standard SGD in terms of speed. However, because of the hit that test performance takes, it just doesn't seem worth the tradeoff right now.

References

Lev Ratinov and Dan Roth. 2009. *Design Challenges and Misconceptions in Named Entity Recognition*