Chapter 15

Named Entity Recognition

Named entity recognition (NER) is the task of identifying what substrings of a text are names of people, places, organizations, etc. It's an extremely common early step in a lot of text analysis applications.

Usually NER is treated as a sequence labeling problem. Given a text like

Mr. Vinken is chairman of Elsevier N. V. , the Dutch publishing group .

we want to label the words as

Mr.	Vinken	is	chairman	of	Elsevier	N.	V.	,	the	Dutch	publishing	group	.
B-person	I-person	O	O	O	B-org	I-org	I-org	O	O	O	O	O	O

where B stands for "begin," I stands for "inside," and O stands for "outside." Many tasks can be reduced to sequence labeling in a similar way.

15.1 Conditional Random Fields

Now we can use any sequence labeling model we want to; the one we've learned so far is a hidden Markov Model, but now is a good time to introduce a higher-performing model, the *conditional random field* or CRF (Lafferty, McCallum, and Pereira, 2001). CRFs consistently outperform HMMs for sequence labeling tasks, and generally do as well or better than other methods. The downside of CRFs is that they have to be trained using numerical optimization, which is a lot slower than (supervised) HMMs.

15.1.1 Definition

In a HMM, the parameters to be learned are the $p(t' \mid t)$ and $p(w \mid t)$, which are required to sum to one. In a CRF, these parameters are no longer probabilities; they are now just numbers that can be positive or negative, big or small. There are two kinds of parameters: $\lambda(t, t')$ for every tag/tag pair, and $\mu(t, w)$ for every tag/word pair. There's no definition of what the values of these parameters should be; instead, you initialize them to all zeros and optimize them to maximize the likelihood of the training data.

Recall that a HMM is (when written out using log-probabilities, and assuming that t_0 = <s> and w_n = </s>):

$$\log P(\mathbf{t}, \mathbf{w}) = \sum_{i=1}^{n} \left(\log p(t_i \mid t_{i-1}) + \log p(w_i \mid t_i) \right). \tag{15.1}$$

There are two potential problems with this model.

First, it seems like a waste of effort to learn $P(\mathbf{t}, \mathbf{w})$, as if we wanted to be able to generate random tagged word sequences. It seems more economical to learn $P(\mathbf{t} \mid \mathbf{w})$.

So, what if we had a model like

$$\log P(\mathbf{t} \mid \mathbf{w}) = \sum_{i=1}^{n} \log p(t_i \mid t_{i-1}, w_i) \tag{15.2}$$

The generation of each tag now depends on more context, so we have to be more careful about overfitting, but leaving that aside, we have a deeper problem. Consider a sentence like

If I could be a force of nature , I 'd be the van der Waals force . </s>

And suppose that the model incorrectly tags the first 14 words as O:

If	I	could	be	a	force	of	nature	,	I	'd	be	the	van	der	Waals	force	.	</s>
O	O	O	O	O	O	O	O	O	O	O	O	O	O					

Now, what should $P(t_{15} \mid \text{O}, \text{der})$ be? It should not favor O or B-person, because those are not the right labels, and it should not favor I-person, because I-person never comes right after O. We'd like $P(t_{15} \mid \text{O}, \text{der})$ to be able to say "There is no good tag in this context," but it can't, because a probability distribution must sum to one.

The solution is to drop the requirement that the generation of each individual tag be governed by a probability distribution. Instead, we only require that the distribution over tag sequences (**t**) be a probability distribution. That is, we replace all the log-probabilities with parameters that can be positive or negative, big or small:

$$P(\mathbf{t} \mid \mathbf{w}) = \frac{\exp s(\mathbf{t}, \mathbf{w})}{\sum_{\bar{\mathbf{t}}} \exp s(\bar{\mathbf{t}}, \mathbf{w})} \tag{15.3}$$

$$s(\mathbf{t}, \mathbf{w}) = \sum_{i=1}^{n} \lambda(t_{i-1}, t_i, w_i). \tag{15.4}$$

where $\bar{\mathbf{t}}$ ranges over all possible tag sequences of length n.

We've replaced $\log p(t' \mid t, w')$ with $\lambda(t, t', w')$ which is just a number: if it's high, it means the model likes to see t, t', w' together; if it's zero, the model doesn't care; and if it's low, the model doesn't like t, t', w'.

Our job during training is to maximize the conditional likelihood,

$$L = \sum_{(\mathbf{t},\mathbf{w}) \in \text{data}} \log P(\mathbf{t} \mid \mathbf{w}) \tag{15.5}$$

but we'll also look at an easier algorithm, the structured perceptron.

15.1.2 Features

Usually, to make $\lambda(t, t', w')$ easier to estimate, we define it in terms of other parameters,

$$\lambda(t, t', w') = \lambda_1(t, t') + \lambda_2(t', w').$$

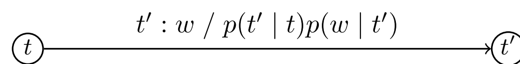
Other possibilities exist as well. For example, since we know that words starting with a capital letter are much more likely to be part of named entities, it would make sense to make the model sensitive to capitalization:

$$\lambda(t, t', w') = \lambda_1(t, t') + \lambda_2(t', w') + \lambda_3(t', \text{cap}(w'))$$

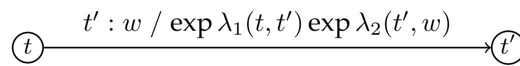
where $\text{cap}(w')$ is 1 iff w' starts with a capital letter. Then the model will hopefully learn a high weight for (say) $\lambda_3(\text{B-person}, 1)$ which works even if w' is unseen in training, like Wolfeschlegelsteinhausenbergerdorff.

15.1.3 As finite transducers

Recall that we wrote an HMM as a finite transducer, where the edges looked like



We can write a CRF as a finite transducer too:



If we run this transducer on tags $\mathbf{t}$ and words $\mathbf{w}$, then the weight of the path is $\exp s(\mathbf{t}, \mathbf{w})$, which is the numerator of (15.3). How do we compute the denominator?

Sometimes, we don't actually care about the denominator. If we have a trained model and we just want to find the most likely tag sequence, all tag sequences have the same denominator so we can just ignore it:

$$\begin{aligned}\arg\max_{\mathbf{t}} P(\mathbf{t} \mid \mathbf{w}) &= \arg\max_{\mathbf{t}} \frac{\exp s(\mathbf{t}, \mathbf{w})}{\sum_{\bar{t}} \exp s(\bar{t}, \mathbf{w})} \\ &= \arg\max_{\mathbf{t}} \exp s(\mathbf{t}, \mathbf{w}),\end{aligned}$$

which we can find using the Viterbi algorithm.

15.1.4 Training

But training is more difficult. As always, we want to maximize the log-likelihood:

$$\log L = \sum_{\mathbf{t}, \mathbf{w} \text{ in data}} \log P(\mathbf{t} \mid \mathbf{w}) \tag{15.6}$$

We do this using stochastic gradient descent, looping over all the sentences, and for each sentence **w** with correct tags **t**, take a step uphill on $\log P(\mathbf{t} \mid \mathbf{w})$. The gradient can be computed by automatic differentiation or manually:

$$\nabla \log L = \sum_{\mathbf{t},\, \mathbf{w} \text{ in data}} \left(\sum_i \lambda(t_{i-1}, t_i, w_i) - E_{\bar{\mathbf{t}}}[\lambda(\bar{t}_{i-1}, \bar{t}_i, w_k)] \right) \tag{15.7}$$

$$E_{\bar{\mathbf{t}}}[\lambda(\bar{t}_{i-1}, \bar{t}_i, w_k)] = \sum_{\bar{\mathbf{t}}} P(\bar{\mathbf{t}}) \lambda(\bar{t}_{i-1}, \bar{t}_i, w_k). \tag{15.8}$$

There's a summation over $\bar{\mathbf{t}}$ again, and this one cannot be ignored.

15.1.5 Summing over all tag sequences

The way that we perform the summation is similar to how the Viterbi algorithm finds the maximum over all possible tag sequences. The only difference is that when a state has multiple incoming edges, we add the weights instead of taking their maximum.

forward$[q_0] \leftarrow 1$
forward$[q] \leftarrow 0$ **for** $q \neq q_0$
for each state q' in topological order **do**
  for each incoming transition $q \rightarrow q'$ with weight p **do**
    forward$[q'] \leftarrow$ forward$[q']$ + forward$[q] \times p$
  end for
end for

15.2 Structured Perceptron

A possibly more intuitive training algorithm is the *structured perceptron*. It can be viewed as an approximation of the above method (in which the expectation of $\lambda(t, t', w')$ is replaced by the most-probable $\lambda(t, t', w')$) or it can be thought of as a learning algorithm in its own right.

for each **t**, **w** in data **do**
  $\hat{\mathbf{t}} \leftarrow \arg\max_{\hat{\mathbf{t}}} P(\hat{\mathbf{t}} \mid \mathbf{w})$
  for $i = 1, \ldots, n$ **do**
    $\lambda(t_{i-1}, t_i, w_i) \leftarrow \lambda(t_{i-1}, t_i, w_i) + 1$
    $\lambda(\hat{t}_{i-1}, \hat{t}_i, w_i) \leftarrow \lambda(\hat{t}_{i-1}, \hat{t}_i, w_i) - 1$
  end for
end for

The intuition is simple: let the model take its best guess $\hat{t}$, then reward the features that would have led to the correct answer **t** and punish the features that led to the guessed answer $\hat{t}$. If $\hat{t} = \mathbf{t}$, then nothing happens. If $\hat{t} \neq \mathbf{t}$, then the model is updated to be more likely to guess **t** in the future.

If $\lambda(t, t', w')$ is broken down into smaller features, the updates are broken down accordingly.

For example, if $\lambda(t, t', w') = \lambda_1(t, t') + \lambda_2(t', w')$:

$$\begin{aligned}
\lambda_1(t_{i-1}, t_i) &\leftarrow \lambda_1(t_{i-1}, t_i) + 1 \\
\lambda_2(t_i, w_i) &\leftarrow \lambda_2(t_i, w_i) + 1 \\
\lambda_1(\hat{t}_{i-1}, \hat{t}_i) &\leftarrow \lambda_1(\hat{t}_{i-1}, \hat{t}_i) - 1 \\
\lambda_2(\hat{t}_i, w_i) &\leftarrow \lambda_2(\hat{t}_i, w_i) - 1
\end{aligned}$$

15.3 RNN+CRFs

The state of the art sequence labeling model is a CRF whose parameters are computed by a RNN (specifically, a bidirectional RNN with LSTM units, but let's just assume a single simple RNN).

Recall that a RNN inputs a sequence of vectors $\mathbf{x}^{(i)}$ and outputs a sequence of vectors $\mathbf{y}^{(i)}$. Here, we let

$$\begin{aligned}
\mathbf{x}^{(i)} &= \text{one-hot vector for } w_i \\
\lambda_2(t_i, w_i) &= \mathbf{y}^{(i)}
\end{aligned}$$

An important caveat is that if a word is used in two different positions, the RNN computes a different $\mathbf{y}$ at each position, so the CRF likewise uses a different $\lambda_w(t_i, w_i)$ at each position. To train, we use gradient ascent to maximize the conditional likelihood, which optimizes the parameters of the RNN and the rest of the parameters of the CRF (λ_1).

Bibliography

Lafferty, John, Andrew McCallum, and Fernando Pereira (2001). "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data". In: *ICML*, pp. 282–289.