Neural Architectures for Named Entity Recognition

1 Citation

Lampe, Guillaume, et al. "Neural architectures for named entity recognition." arXiv preprint arXiv:1603.01360 (2016).

http://aclweb.org/anthology/N/N16/N16-1030.pdf

2 Abstract

We create two neural network models for named entity recognition. The first uses bidirectional LSTMs (to generate per-word predictions) and a conditional random field (CRF) (to smooth predictions). The second model, which isn't as good as the first, uses a transition-based approach with a Stack-LSTM. We use two kinds of word embeddings. First, we create character-level word embeddings from the supervised corpus and second we use regular word embeddings from the unupervised corpus. It does not use any data from gazetteers (i.e. database of named entities).

3 Introduction

There are very few named entity recognition data sets, so traditional approaches have leveraged gazetteers or unsupervised techniques.

Using a small supervised corpus and large unsupervised corpus, we present an LSTM-CRF model and Stack-LSTM model for named entity recognition. We use character-level word embeddings and regular (i.e. distributional) word embeddings. We get state of the art in many langugages.

4 LSTM-CRF Model

An LSTM takes input sequence $(\mathbf{x}_1, ..., \mathbf{x}_n)$ and produce $(\mathbf{h}_1, ..., \mathbf{h}_n)$. Our LSTM implementation is:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1}\mathbf{W}_{\mathbf{ci}}\mathbf{c}_{\mathbf{t-1}} + \mathbf{b}_{\mathbf{i}}) \tag{1}$$
$$\mathbf{c}_t = (1 - \mathbf{i}_t) \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \tag{2}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{co}\mathbf{c}_t + \mathbf{b}_o) \tag{3}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{4}$$

We actually use a bidirectional LSTM, which means we run one LSTM forward over the input to produce $\overrightarrow{\mathbf{h}}_t$ and one LSTM backward over the input to produce $\overleftarrow{\mathbf{h}}_t$ and then we concatenate them to produce $\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t]$

Now, if we wanted we could use this bidirectional LSTM to predict a tag for each word. We just project the hidden vector for each word down to a vector with one entry per tag and apply a softmax to this vector to get a probability distribution over the tags. However, this approach makes predictions for each word independently, which is suboptimal. To smooth the predictions, we use a conditional random field (CRF). Let $P_{i,j}$ be the score for the j^{th} tag of the i^{th} word in the input. Thus, for the input sequence $\mathbf{X} = (\mathbf{x}_1, ..., \mathbf{x}_n)$ and some output tag sequence $\mathbf{y} = (y_1, ..., y_n)$, we can create a score function (where x_0 and x_{n+1} represent special sentence-start and sentence-end tags - so there are $k + 2$ total possible tags):

$$s(\mathbf{X}, \mathbf{y}) = \sum_{i=1}^{n} A_{y_i, y_{i+1}} + \sum_{i=1}^{n} P_{i,y_i} \tag{5}$$

The $(k + 2) \times (k + 2)$ transition matrix $\mathbf{A}$ is learned. We can turn this score into a probability using a softmax (the denominator must sum over all possible tag sequences). At train time, we use the ground truth tag sequence. At test time, we select the tag sequence that maximizes the score. Since we are only modeling bigram interactions between outputs, the summation over all tag sequences can be computed efficiently with dynamic programming. The $P_{i,j}$ values come from the LSTM after projecting its hidden vectors down to a k dimensional space. You could also add a hidden layer here for a marginal improvement.

We use the IOBES tagging scheme which has the tags inside-named-entity (I), other (O), begins-named-entity (B), ends-named-entity (E), singleton-named-entity (S).

5 Transition Based Chunking Model

There's a paper about a Stack-LSTM that keeps a running embedding of its contents. We use three two-LSTMs: an output and stack (a "scratch pad"). We put the words into a "buffer" indicating they are not yet processed. We have three operations: SHIFT (move next buffer word to stack), OUT (move next buffer word to output), and REDUCE(y) (pop all items from stack, label with y, and push representation, computed from bidirectional LSTM to output). The algorithm terminates when stack and buffer are empty. We parametrize a probability distribution over the actions that depends on the contents of buffer, stack, output and on the history of actions taken so far (i.e. we represent all this state with a stack LSTM for each and concatenate them). At test time, we greedily pick actions until the algorithm terminates.

6 Input Word Embeddings

The input to our models is a sequence of word embeddings. We would like to use popular word embeddings like skip-n-grams. However, we'd also like our model to generalize to new words (e.g. you may have never seen the name "Martensdale, IA", but you can tell it's a named entity because of the capitalization), so we'll use character-level word embeddings. To train character level embeddings, we feed the characters of the word into a bidirectional LSTM and concatenate the final forward and backward hidden state vectors to create the embedding. Since LSTMs and other RNNs tend to bias towards their most recent inputs, so our bidirectional LSTM will effectively encode the prefix and suffix of the word.

We also compute skip-n-gram embeddings. This embedding matrix, along with the character level LSTM embedding model, are fine-tuned during model training. Out input to our LSTM-CRF and Stack-LSTM models is the concatentation of the character level and word level embeddings.

Unknown words are replaced with an "UNK" token. We randomly replace a word with "UNK" sometimes to help our embeddings be robust. We also use dropout (0.5) on the embedding vector input to LSTM-CRF (or Stack-LSTM) to make the model more robust.

7 Experiments

We train with SGD (0.01) with gradient clipping (5.0). Our LSTM-CRF and Stack-LSTM uses 100 dimensional hidden vectors. We use the CoNLL supervised datasets and a variety of unsupervised datasets for embeddings. We train models for English, German, Spanish, and Dutch. Our LSTM-CRF beats all state of the art models (except on Dutch) and Stack-LSTM does pretty well too. Our metric was F1 score. The biggest wins came from pretraining word embeddings on unsupervised data (+7.31), CRF (+1.79), dropout (+1.17), and also using character-level word embeddings (+0.74).

8 Related Work

Other work uses ensembles of different classifier types. Some people have used neural networks. They also use a variety of datasets and sub-models for things like spelling checks.

9 Conclusion

We present the LSTM-CRF and Stack-LSTM models which, when combined with our character-level and word-level word embedding concatentation, get state of the art on named entity recognition in a variety of languages.