HW5: Named Entity Recognition

Jeffrey Ling
jling@college.harvard.edu

Rohil Prasad
rprasad@college.harvard.edu

April 16, 2016

1 Introduction

In this assignment, we tackle the problem of Named Entity Recognition (NER). The goal of this task is to construct a model that can pick out and determine the type of important named entities in a given text. For example, in the sentence "I went to Disney World.", the block "Disney World" should be labeled as a special place.

We implement and discuss several approaches to NER in this paper, all trained on a portion of the CONLL 2003 Shared Task dataset. These include a standard Hidden Markov Model (HMM), a Maximum Entropy Markov Model (MEMM), and a structured perceptron. To label entities given our models, we implement the Viterbi algorithm.

Furthermore, unlike previous assignments, feature selection is especially important for this problem (since we don't use dense representations as in neural networks). We implement several of the features described in in [Tjong Kim Sang paper] as well.

In Section 2, we give a formal description of our problem and establish our notation. In Section 3, we give detailed descriptions of the algorithms used for training and evaluation. In Section 4, we present our experimental results. In Section 5, we discuss our findings.

2 Problem Description

We treat this as a tagging problem, similarly to HW2. To capture the named entities, which can span multiple words, we use a system called BIO tagging. A word tagged with an *I* tag We simply the task by only trying to predict the tags, and later extract the mentions from the predicted tag sequences by looking at contiguous blocks of the same tag.

Let $\mathcal{V}$ denote the vocabulary and $\mathcal{T}$ denote the tag set. We use start and end padding words `<s>`, `</s>` along with start and end tags `<t>` and `</t>` to denote the beginning and end of each sequence.

Our raw data consists of sentences $(w_1, \dots, w_l)$ (denoted as $w_{1:l}$) with corresponding tags $(t_1, \dots, t_l)$ (denoted as $t_{1:l}$) for $(w_i, t_i) \in \mathcal{V} \times \mathcal{T}$ and l the length of the sentence. We want to learn a score function

$$f : \mathcal{V}^l \times \mathcal{T}^l \to \mathbb{R}$$

for any sentence length l, and our prediction will be $\arg\max_{t_{1:l}} f(w_{1:l}, t_{1:l})$.

2.1 Markov Models

If we interpet f probabilistically, we can write this as

$$\log p(t_{1:l}|w) = \sum_{i=1}^{l} \log p(t_i|t_{i-1}, \dots, t_1, w)$$

In this problem, we typically make the Markovian assumption so that $p(t_i|t_{i-1}, \dots, t_1, w) = p(t_i|t_{i-1}, w)$ [1].

Denote $\widehat{y}(t_{i-1})_{t_i} = p(t_i|t_{i-1}, w)$. Thus we decompose our score function as

$$f(w_{1:l}, t_{1:l}) = \sum_{i=1}^{l} \widehat{y}(t_{i-1}, w)_{t_i}$$

In the Markov setup, our goal is to learn the transition function that takes as input $(t_{i-1}, w) \in \mathcal{T} \times \mathcal{V}^l$ and outputs a score vector $\mathbb{R}^{|\mathcal{T}|}$ as the score for each next tag. In practice, we don't use the whole sequence $w \in \mathcal{V}^l$ but featurize (t_{i-1}, w) in a local window around w_i.

2.2 Feature Extraction

We describe details of the feature representation of (t_{i-1}, w) around the word w_i.

First, construct the vocabulary $\mathcal{V}$ of the training and testing sets. We only use lower-case words and encode them as sparse one-hot vectors $\boldsymbol{j} \in \{0,1\}^{|\mathcal{V}|}$.

Next, we construct a set $\mathcal{F}$ of features. We implement these based on papers from the CONLL 2003 shared task, but not all of them are used in our final models. The full set includes

- The lemma of a word, as given by the NLTK WordNet Lemmatizer.
- The part of speech of a word as given by the NLTK averaged perceptron POS tagger.
- All possible substrings of a word.

    As a backoff, prefixes and suffixes of a certain length of a word.
- Capitalization information (is all caps, starts with a capitalized letter, or contains a capitalized letter) of a word.

We include the previous tag t_{i-1} in this feature set as well. Also, we make available the features (as given above) of surrounding words in an arbitrary window around w_i (we use a maximum window size of 5 words). Each of these features has an index in $\mathcal{F}$, so we can associate a word w to a sparse vector in $\{0,1\}^{|\mathcal{F}|}$.

We then get a feature function $feat : \mathcal{T} \times \mathcal{V}^l \to \{0,1\}^{|\mathcal{V}|} \times \{0,1\}^{|\mathcal{F}|}$, and we want to learn

$$\widehat{y} : \{0,1\}^{|\mathcal{V}|} \times \{0,1\}^{|\mathcal{F}|} \to \mathbb{R}^{|\mathcal{T}|}$$

that takes in a word and its associated features and outputs a prediction vector $\hat{\boldsymbol{y}}$ where $\hat{y}_k$ is the log probability that the word has tag k.

[1] To handle the boundary case, we use $t_0 = \texttt{<s>}$ to get $p(t_1|t_0, w)$. This will be implicit throughout.

2.2.1 Practice

In practice, we use indices and LookupTable in Torch to handle the sparse vectors. To ensure that our feature representation of our data forms a matrix for Torch to work with, we pad the feature vectors with a special "padding feature" that has index 1 in $\mathcal{F}$ to make them all the same length.

2.3 Evaluation

To evaluate a predicted sequence of tags $\widehat{t_{1:l}}$ for given sentence $w_{1:l}$, we compared the mentions (contiguous blocks of non-O tags) to the mentions from the true sequence of tags $t_{1:l}$. We consider a single mention to be predicted correctly iff it matches up exactly with the mention from the true sequence.

Then by collecting the counts of total true mentions, total predicted mentions, and total correctly predicted mentions over the entire dataset, we can form the precision and recall:

$$prec = \frac{\text{total correctly predicted mentions}}{\text{total predicted mentions}}$$

$$rec = \frac{\text{total correctly predicted mentions}}{\text{total true mentions}}$$

Our metric is then the F_β score for β some parameter:

$$F_\beta = \frac{(\beta^2 + 1) \cdot prec \cdot rec}{\beta^2 \cdot prec + rec}$$

We use $\beta = 1$ as the balanced metric.

Note that the Kaggle metric doesn't use global counts, but computes the F-score per sentence.

3 Model and Algorithms

We use three models to learn the transition function:

- Hidden Markov Model (HMM)
- Maximum Entropy Markov Model (MEMM)
- Structured Perceptron

We first describe the basic Viterbi decoding algorithm that uses the transition model to predict a sequence.

3.1 Viterbi

3.2 Hidden Markov Model

With the HMM, we learn transition probabilities $p(t_i|t_{i-1})$ and emission probabilities $p(w_i|t_i)$ using the MLE estimates with count matrices. We use Laplace smoothing parameter α, and we found $\alpha = 0.001$ worked best.

We then predict our transition function as

$$\widehat{y}(t_{i-1}, w)_{t_i} = \log p(t_i|t_{i-1}) + \log p(w_i|t_i)$$

3.3 Maximum Entropy Markov Model

This is essentially a logistic regression model where we use the features as described in the previous section. We have

$$\widehat{y}(t_{i-1}, w) = \log softmax(feat(t_{i-1}, w)W + b)$$

where W is a $|\mathcal{V}| + |\mathcal{F}|$ by $|\mathcal{T}|$ weight matrix and b is a 1 by $|\mathcal{T}|$ bias term.

We train the MEMM with gradient descent by iterating over each word in the dataset (with its corresponding window of features and tag).

3.3.1 Extension: Neural MEMM

Instead of simple logistic regression, we can use more layers with nonlinearities to model $\widehat{y}$:

$$\widehat{y}(t_{i-1}, w) = \log softmax(\tanh((feat'(t_{i-1}, w)W_0)W_1 + b_1)W_2 + b_2)$$

W_0, a $|\mathcal{V}| + |\mathcal{F}|$ by d_{in} matrix, is an embedding matrix for the words and features. In practice we use a reduced set of features with $feat'$ since dense word vectors should encode most of the information (we only used capitalization features and previous class t_{i-1}, see above section, in this setup). We also have

$$W_1 \in \mathbb{R}^{d_{in} \times d_{hid}}, \quad b_1 \in \mathbb{R}^{d_{hid}},$$
$$W_2 \in \mathbb{R}^{d_{hid} \times |\mathcal{T}|}, \quad b_2 \in \mathbb{R}^{|\mathcal{T}|}$$

We also use SGD to train this model.

3.4 Structured Perceptron

In structured perceptron, we have a linear transition function

$$\widehat{y}(t_{i-1}, w) = feat(t_{i-1}, w)W$$

for W a $|\mathcal{V}| + |\mathcal{F}|$ by $|\mathcal{T}|$ matrix.

Our learning algorithm is different than before, however. For each sentence, we first produce a candidate tag sequence $\widehat{t_{1:l}}$ using Viterbi and our current model. Then, we compare this sequence with the gold sequence $t_{1:l}$, and for each difference in tags $\widehat{t_i}, t_i$, we give $+1$ for the weights in W corresponding to features $feat(t_{i-1}, w)$ and class t_i and -1 for those corresponding to class $\widehat{t_i}$.

We iterate over the dataset using this learning method until convergence in F-score.

3.4.1 Extension: Averaged Weights

4 Experiments

We ran various experiments with feature selection and some hyperparameter tuning for all models. We evaluate with balanced F-score on the validation set.

4.1 Hidden Markov Model

4.2 Maximum Entropy Markov Model

We trained MEMM with batch SGD with batch size 32, learning rate 1, and trained for 20 epochs.

Model	F-score
HMM	100000
MEMM (features???)	100000
Neural MEMM	0.728
Perceptron	100000

Table 1: F-score for mentions on validation set.

4.2.1 Extension: Neural MEMM

We used the same hyperparameters except for learning rate 0.01. We used $d_{in} = 50$ and initialized with the GLOVE embeddings, and $d_{hid} = 100$. Each epoch took about 17 s to train.

We trained for 60 epochs and ended with an F-score of 0.728.

4.3 Structured Perceptron

5 Conclusion

We found that the neural MEMM worked best.

The most time consuming part of this homework for the non-neural MEMM was feature selection, which would significantly change performance depending on features used. This gives us a reason to prefer neural networks as a model easier to use without prior knowledge.