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 and De Meulder (2003) 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. The tags are given by

$$\{\text{I-LOC, I-PER, I-ORG, I-MISC, B-MISC, B-LOC, O}\}$$

The suffixes LOC, PER, ORG, and MISC denote whether the named entity in question is a location, person, organization, or miscellaneous. The prefixes I, B, and O are used to split up our sequence of words into "mentions" of named entities and other words.

A word with an I tag indicates that it is in the start or middle of a mention of a named entity of the given tag class. A word with a B tag indicates it is at the end of a mention of a named entity of the given tag class. An O tag indicates that a word is not part of any mention.

In practice, the B tag is only needed to demarcate two consecutive mentions of the same type, and rarely occurs.

We simplfy 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 $(w_1, \ldots, w_l)$ (denoted as $w_{1:l}$) with corresponding tags $(t_1, \ldots, t_l)$ (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

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

In this problem, we typically make the Markovian assumption so that $p(t_i|t_{i-1}, \ldots, t_1, w) = p(t_i|t_{i-1}, w)$ [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 $j \in \{0, 1\}^{|\mathcal{V}|}$.

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.

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

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.

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

Given a transition distribution of tags $p(t_i|t_{i-1})$, we want to predict the sequence of tags $t_1, \ldots, t_n$ with the highest joint probability.

With our Markov assumption, this is equal to

$$p(t_1, \ldots, t_n) = \prod_{i=1}^{n} p(t_i|t_{i-1}, \ldots, t_1) = \prod_{i=1}^{n} p(t_i|t_{i-1})$$

Note that the given product has a term $p(t_1)$, which we cannot calculate using the transition distribution. To avoid this, we pad our sequence of words with a start word $< s >$ and always set t_1 to be the start tag $< t >$. Furthermore, we observe that maximizing this product is the same thing as maximizing the sum of the log transition probabilities $\log p(t_i|t_{i-1})$.

This can be done by the Viterbi algorithm.

The algorithm inductively constructs a $\mathcal{T} \times n$ array π, where $\pi[t][i]$ is the maximum joint probability of a sequence of tags $t_1 =< t >, t_2, \ldots, t_{i-1}, t$.

As a base case, we assume $t_1 =< t >$, setting $\pi[t][1] = 0$ for every $t \in \mathcal{T}$, and then setting $\pi[t][2] = \log p(t| < t >)$.

Then, we can inductively set

$$\pi[t][i] = \max_{t' \in \mathcal{T}} \pi[t'][i-1] + \log p(t|t')$$

.

In practice, we also construct a $\mathcal{T} \times n$ array of *backpointers* $\mathbf{BP}$, where

$$\mathbf{BP}[t][i] = \text{argmax}_{t' \in \mathcal{T}} \pi[t'][i-1] + \log p(t|t')$$

. This allows us to construct our optimal sequence. First, by definition we can set our optimal $t_n = \arg\max_{t \in \mathcal{T}} \pi[t][n]$. Then we can inductively see that the optimal t_{i-1} is equal to $\mathbf{BP}[t_i][i]$, which gives us our full optimal sequence $t_1 =< s >, t_2, \ldots, t_n$.

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 $\alpha = 0.02$ 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

As an extension, we replace the weight matrix W by its average across all training iterations.

In practice, this is done epoch-by-epoch. In a single epoch, the weight matrix W is updated after every sentence of the text is processed. We store the sum of the weight matrices obtained after every sentence, and then divide it by the number of sentences and replace W with this average at the end of the epoch.

This acts as a form of regularization for our structured perceptron and keeps its epoch-by-epoch performance much more consistent.

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, giving the best result for each model below.

Model	F-score
HMM	0.594
MEMM	0.660
Neural MEMM	0.728
Perceptron	0.682

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

4.1 Hidden Markov Model

We optimized the smoothing parameter α in two steps. First, we searched along orders of magnitude from 10^{-4} to 1. Then, for the given order of magnitude 10^{-k} with the highest F-score on the validation set, we searched for value $\alpha = i \cdot 10^{-k}$ with highest F-score across all $i \in \{1, 2, \ldots, 10\}$. The full results of the search are given in the table below.

α	F-score
0.0001	0.590
0.001	0.590
0.01	0.591
0.1	0.557
0.02	0.594
0.03	0.588
0.04	0.573
0.05	0.570
0.06	0.564
0.07	0.561
0.08	0.561
0.09	0.559

Table 2: F-score for various values of α

4.2 Maximum Entropy Markov Model

We trained MEMM with batch SGD with batch size 32, learning rate 1, and trained for 20 epochs. Each epoch took approximately 4 seconds to train. The loss curves for the training and validation sets for our best-performing MEMM are plotted in Figure 1.

We also experimented with different feature sets to optimize the F-score. The features were selected from the following list:

- Prefixes and suffixes of length ≤ 4
- Word lemmas
- Parts of speech of our word and the two adjacent words
- Capitalization information of our word

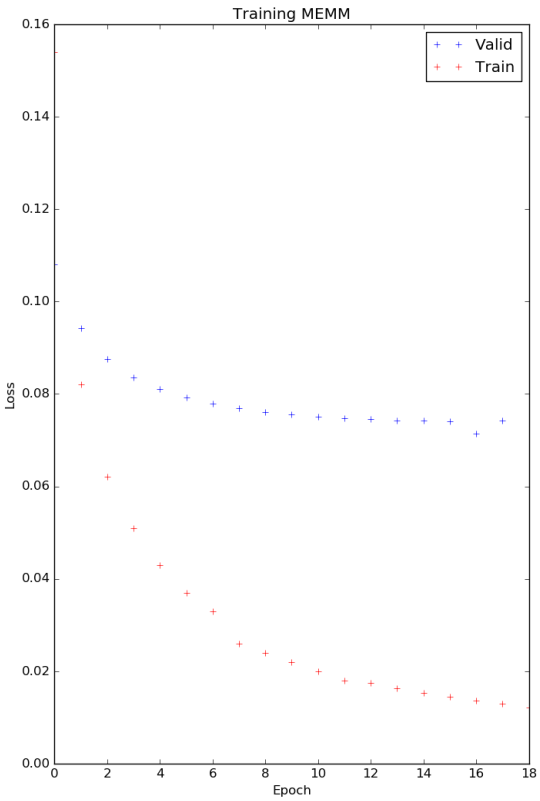

Figure 1: Training curves for MEMM.

We calculated the validation set F-score for the MEMM for five feature sets. One set contained all four of these features, while the other four sets each excluded one of these types of features. The results are given below.

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 seconds to train.

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

4.3 Structured Perceptron

We trained our structured perceptron with averaging for a minimum of 5 epochs and a maximum of 10 epochs, halting the program if the validation F-score is lower than that of the previous epoch. Each epoch took approximately 75 seconds to train.

Due to the structure of this model, the only metrics for performance we have is the epoch-to-epoch precision, recall, and F-score on the validation set. We plot the precision and recall curves

Feature Set	F-score
All	0.6605
No Cap	0.6267
No Lemma	0.6547
No POS	0.6603
No Prefix/Suffix	0.6127

Table 3: F-score of MEMM for various feature sets

for our best-performing perceptron in Figure 2.

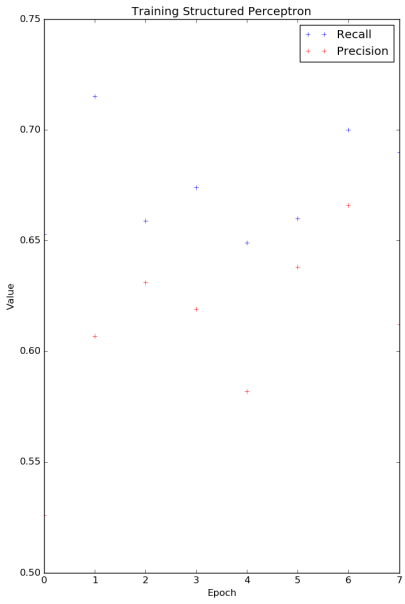

Figure 2: Precision and recall for structured perceptron

We performed the same feature selection experiments for the structured perceptron as we did for the MEMM. We give the maximum epoch-to-epoch F-score for each of the feature sets in the table below.

5 Conclusion

We found that the neural MEMM worked best. This was not an unusual outcome, as many of the top papers in the CONLL 2003 shared task used MEMMs. Our neural MEMM used the GLOVE embeddings, which are a state-of-the-art dense representation for words, which is likely a much better set of features than the ones we used for our logistic regression MEMM. ,

It should also be noted that our perceptron with averaging seemed to perform better than our logistic regression MEMM on the validation set. However, it performed much worse on Kaggle,

Feature Set	F-score
All	0.6827
No Cap	0.6378
No Lemma	0.6713
No POS	0.6629
No Prefix/Suffix	0.6530

Table 4: F-score of MEMM for various feature sets

so it is likely overfitted on the training set and just happened to perform somewhat well on the validation set.

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.

References

Tjong Kim Sang, E. F. and De Meulder, F. (2003). Introduction to the conll-2003 shared task: language-independent named entity recognition. *CONLL*, pages 142–147.