Sequences and Entities

Vincent Nguyen and **Austin Shin**
CS 287: Statistical Natural Language Processing
Harvard University
Cambridge, MA 02138
{vnguyen01, ashin01}@college.harvard.edu

1 Introduction

Named-entity prediction is a thoroughly explored task in Natural Language Processing. We explore the use of sequenced-based learning methods for this task. Beginning with the simple Hidden Markov Model (HMM) to the Maximum Entropy Markov Model (MEMM) to the Structured Perceptron, we track the performances of these classifiers by predicting entity tags of words in a sentence. Sequence-learning approaches demand the use of Dynamic Programming in order to find the sequence tags that maximize probability. To this end, we will discuss and employ Viterbi's algorithm to significantly reduce the time complexity of the search.

2 Data

We use a subset of the 2003 shared task of the Conference on Computational Natural Language Learning (CoNLL) [1]. The Language-Independent Named Entity Recognition (II) task consists of named entities that take the form of persons, organizations, locations, times, and quantities. Each entity tag is assigned to a word in a sentence. The objective of the task is to predict the correct tag for each respective word.

Table 1: CoNLL-2003 Subset

Data	**# Sen**
Train - `train.num.txt`	3345
Valid - `dev_kaggle.txt`	1601
Test - `test.num.txt`	1646

The subset of the CoNLL-2003 shared task contains 5 quantifier tags: `O`, `LOC`, `PER`, `MISC`, and `ORG`. These correspond to other, person, miscellaneous, and organization where the prefix `I-` has been omitted given the nature of the subset. The subset we work with is summarized in Table 1.

[1] http://www.cnts.ua.ac.be/conll2003/ner/

3 Learning Methods

3.1 HMM

The Hidden Markov Model (HMM) can be viewed as an extension of the Naive Bayes Classifier for sequence learning. As such, the HMM models the joint distribution and is a generative classifier:

$$p(Y,X) = \prod_{i=1}^{n} p(x_i|x_{1:i-1}, y_{1:n}) p(y_i|y_1 \dots y_{i-1}) \tag{1}$$

We model the transition distribution between tags, $p(y_i|y_{i-1}, \theta)$, and the emission distribution which are tag-conditional features, $p(x_i|y_i, \theta)$. The log-likelihood is given by

$$L(\theta) = -\sum_{j=1}^{c}\sum_{i=1}^{n} \log p(x_i^j|y_i^j, \theta) + \log p(y_i^j|y_{i-1}^j, \theta) \tag{2}$$

For constructing the distributions, we construct probability tensors. As the tensors closely resemble normalized counts in the Naive Bayes Classifier, we employ Laplace smoothing to account for unseen examples during training time. We discuss this at length in the results and show that the choice of additive α has an enormous effect on performance.

Maximization of the log-likelihood parameters can be computed in closed form which makes this sequence learning very quick to train, not including inference that is. However, the drawbacks are that it operates under the independence assumption much like Naive Bayes. That is, the current features in a sequence are only conditioned on the tag.

3.2 MEMM

The Maximum Entropy Markov Model (MEMM) can be viewed as an extension of Logistic Regression (or a Maximum Entropy Classifier as called in the NLP community) for sequence learning. As such, the MEMM models the conditional distribution and is a discriminative classifier:

$$p(Y|X) = \prod_{i=1}^{n} p(y_i|y_1 \dots y_{i-1}, x_{1:n}) \quad (3)$$

Similarly, we also have a transition and emission distribution. However, for the MEMM, the transition distribution is conditioned also on the input, $p(y_i|y_{i-1}, x_1 \dots x_n)$, and the emission distribution is the same as Equation 3.1. We can imagine this classifier as individual logistic regression classifiers that are connected in sequence. We can condition on each class so that

$$p_{y'}(y_i|x_{1:n}) = \frac{1}{Z(x_{1:n}, y')} \exp \sum_{i=1}^{n} \theta_i \text{feat}_i(y_i, x_{1:n}) \quad (4)$$

where Z is a normalization factor over the entire sequence. We can insert Equation 4 into Equation 3. In order to find the optimal parameters θ, we employ first-order optimization methods, Adam and SGD, that require taking the gradient of the log-likelihood.

By ignoring the independence assumption, the MEMM is a much more rich sequence learner. The MEMM takes longer to train due to the iterative nature of gradient descent methods. We discuss in our results the different tradeoffs between optimization methods and speed-ups.

3.3 Viterbi for Inference

The learning methods introduced above use different representations of inputs and distribution modeling of the data to generate probability scores. However, there still remains the challenge of selecting the best sequence of tags such that the sequence probability is maximized. For a sequence of length n and c different tags, there exists $\mathcal{O}(c^n)$ different tag combinations. This is exponential and not scalable to large sequences. As such, we employ dynamic programming and take advantage of the Markov structure to reduce the time complexity of this inference operation.

Viterbi's algorithm observes that there exists a most likely path for any tag c at any sequence point n. Thus, instead of having to recalculate all the paths up to point n, we can ignore the unlikely paths and keep a running memory of our beset candidate path. This reduces the time complexity to $\mathcal{O}(nc^2)$. This is much more reasonable for variable sequence length. We implement the Viterbi code written by Saketh with slight modifications to generalize for all models [2].

3.4 Structured Perceptron

The Structured Perceptron algorithm works by first computing the most probable tags with the θ initialized to 0:

$$\hat{Y} = \arg\max_{Y} \sum_{i}^{n} \theta_i \text{feat}_i(y_i, x_{1:n}) \quad (5)$$

where Viterbi's algorithm is used to infer the most probable sequence. This sequence is compared to the correct labeled sequence. The differences in the sequence are noted. The weights for the incorrect labels are recomputed via the gradients of the Hinge Loss where the loss function is:

$$L(\theta) = \sum_{i=1}^{N} ReLU(1 - (\hat{y_{i,c}} - \hat{y_{i,c'}}))$$

where $y_{i,c}$ is the score of the correct class at timestep i and $\hat{y}_{i,c}$ is the score of the highest incorrect class at timestep i. So long that the highest possible scoring sequence has a total score greater than 1 + the next best incorrect sequence, there will be no updates made. Thus, the structured perceptron attempts to optimize over sequences, rather than individual ordered pairs of the form (features, previous tag).

3.5 Neural MEMM

As an extension, we added another linear layer on top of a non-linearity to add depth to our MEMM. The input-ouput format is identical to the original MEMM except that our model now exhibits non-linearities.

3.6 CRF

As an added extension, we explore the Conditional Random Field (CRF) which also happened to be a topic of my thesis. Just as we did in the previous classifiers, we want to find the parameters in θ for the feature. First, we can model the conditional probability of a sequenced class label as

$$p(y|x) = \frac{1}{Z(x)} \exp \left\{ \sum_{k=1}^{F} \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \quad (6)$$

where $Z(x)$ is the normalizing factor expressed as

$$Z(x) = \sum_{t=1}^{T} \exp \left\{ \sum_{k=1}^{F} \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \quad (7)$$

We now want to be able to estimate the parameters θ. To do this, we return to the idea of MLE and taking the logarithm of the conditional probability:

$$\log p(Y|X, \theta) = \log \prod_{n=1}^{N} p(y_n|x_n, \theta) \quad (8)$$

$$L(\theta) = \sum_{n=1}^{N} \sum_{t=1}^{T} \sum_{k=1}^{F} \theta_k f_k(y_{nt}, y_{n,t-1}, x_{nt}) - \sum_{n=1}^{N} \log Z(x_n) \quad (9)$$

To optimize the weights, we can take the first or second derivative of the loss function with respect to the weights and use SGD or L-BFGS.

[2] https://cs287.github.io/Lectures/dpnlp.pdf

4 Results

4.1 F-Score Performance Metric

The performance metric for this task is F-Score. We define F-Score as the harmonic mean of precision and recall:

$$F_\beta = \frac{(\beta^2+1)\times P \times R}{\beta^2 \times (P+R)} \tag{10}$$

where P is precision which is defined as for a particular tag, the number of predicted examples that are correct over that number plus the number of incorrectly predicted examples:

$$P = \frac{\text{\# retrieved rel.}}{\text{\# retrieved rel.} + \text{\# retrieved irrel.}} \tag{11}$$

and R is recall which is defined as for a particular tag, the number of predicted examples that are correct over that number plus the number of examples of that tag not predicted:

$$R = \frac{\text{\# retrieved rel.}}{\text{\# retrieved rel.} + \text{\# unretrieved rel.}} \tag{12}$$

We set $\beta = 1$, and to obtain an overall F-Score for each classifier, we average the individual F-Scores for each tag.

4.2 HMM

We train the HMM for word features only. We account for unseen feature and tag counts with additive Laplace smoothing. The precision, recall, and F-score results demonstrate that these values are extremely sensitive to the choice of additive α. The best F-score metric was obtained in Table 2 for $\alpha = 0.5$. A suitable explanation for this phenomenon is that increasing the α drastically shifts probability mass from seen to unseen counts which leads to worse performance metrics.

Table 2: HMM Validation F-Score for $\alpha = 0.5$

Tags	**Precision**	**Recall**	**F-Score**
O	0.9785	0.4946	0.6570
PER	0.5919	0.3718	0.4567
LOC	0.7298	0.4219	0.5347
ORG	0.5191	0.3417	0.4121
MISC	0.5714	0.3636	0.4444
Avg	0.3987	0.6781	0.5021

We also test our HMM on the Kaggle test set. We achieved good scores for the simplicity of the HMM. As previously mentioned, the effect of the additive prior has an extreme sway over the computed F-score. There is a difference between F-scores from the Kaggle test predictions and the validation predictions. We denote the α parameter under the model heading in Table 5.

Table 3: HMM Validation F-Score for $\alpha = 1$

Tags	Precision	Recall	F1-Score
O	0.9928	0.4982	0.6635
PER	0.3461	0.2571	0.2950
LOC	0.6067	0.3776	0.4655
ORG	0.3747	0.2725	0.3156
MISC	0.3307	0.2485	0.2837
Average	0.3308	0.5302	0.4074

Table 4: HMM Validation F-Score for $\alpha = 2$

Tags	**Precision**	**Recall**	**F-Score**
O	0.9984	0.4996	0.6659
PER	0.1070	0.0966	0.1015
LOC	0.3992	0.2853	0.3327
ORG	0.1486	0.1293	0.1383
MISC	0.0626	0.0589	0.0607
Avg	0.2139	0.3431	0.2635

The reason for this discrepancy is the validation metrics were calculated where a predicted sequence `1 3 1` would consider the `3` correct for a given labeled sequence of `1 3 3`. The Kaggle predictions take into consideration the span correctness which results in a lower precision and recall that translates into a decreased harmonic mean of the two.

Table 5: HMM Kaggle Predictions

Model	**F-Score**
HMM+0.01	0.44770
HMM+0.1	0.40760
HMM+0.5	0.32464
HMM+1	0.30534

We wondered if the performance of the HMM would be improved through more training examples. As such, we concatenated the training examples and validation examples. However, comparing the output files showed nearly no substantial difference between the two. We conclude that the HMM model with basic word features and a optimized α cannot be easily improved.

4.3 MEMM

We only use word features for the MEMM. We train the discriminative classifier using the momentum based Adam optimization algorithm. We sampled from batch sizes of 1000 and ran the optimization for 40000 iterations until convergence at a loss of ≈ 0.1. Our loss per iteration is summarized in Figure 1. Usage of Adam greatly improves on the slow convergence times of SGD which takes on the span of 15-20 minutes. This speedup is especially useful for the assignment.

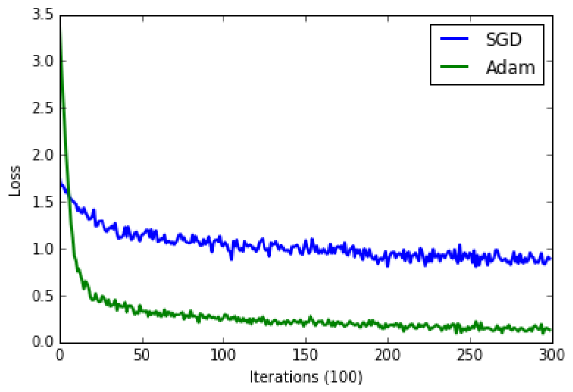

Figure 1: Optimizer Comparison, Batches of 1000

We summarize our validation F-Scores for the MEMM in Table 6 and Table 7 for different loss scores.

Table 6: MEMM Validation F-Score, Loss 0.2

Tags	**Precision**	**Recall**	**F-Score**
O	0.9700	0.4923	0.6532
PER	0.6457	0.3923	0.4881
LOC	0.6971	0.4107	0.5169
ORG	0.5445	0.3525	0.4280
MISC	0.5949	0.3730	0.4585
Avg	0.4042	0.6904	0.5099

Table 7: MEMM Validation F-Score, Loss 0.1

Tags	**Precision**	**Recall**	**F-Score**
O	0.9674	0.4917	0.6520
PER	0.6885	0.4077	0.5122
LOC	0.7328	0.4229	0.5363
ORG	0.5796	0.3669	0.4493
MISC	0.6379	0.3894	0.4836
Avg	0.4157	0.7212	0.5274

As expected, allowing the model to train for a longer period of time to reach a lower loss results in better performance metrics. In addition, the MEMM outperforms the HMM which is expected as the logistic regression outdid the Naive Bayes Classifier in previous tasks, albeit with the optimization time tradeoff.

4.4 Extension: Neural MEMM

The Neural MEMM has exactly the same input-output format as the vanilla MEMM. Performance wise, the neural MEMM achieved slightly higher precision and recall scores than the vanilla MEMM:

The loss using this neural model went down to 0.05 for the same batch size as the MEMM.

Table 8: Neural MEMM Validation F-Score, Loss 0.05

Tags	**Precision**	**Recall**	**F-Score**
O	0.9631	0.4906	0.6501
PER	0.7092	0.4149	0.5235
LOC	0.7755	0.4368	0.5588
ORG	0.5912	0.3715	0.4563
MISC	0.6399	0.3902	0.0.4848
Avg	0.4208	0.7358	0.5354

4.5 Structured Perceptron

While we constructed the Structured Perceptron model, we are still in the process of training it. We find that the most consuming portion of the training lies in the Viterbi algorithm, making training an order of magnitude slower than the logistic regression. Please see code for our model.

Table 9: Structured Perceptron Validation F-Score

Tags	**Precision**	**Recall**	**F-Score**
O	0.9157	0.4780	0.6281
PER	0.0228	0.0223	0.0226
LOC	0.0774	0.0718	0.0745
ORG	0.0424	0.0407	0.0415
MISC	0.0587	0.0554	0.0570
Avg	0.1336	0.2234	0.1672

We note that the precision and recall scores roughly follow the pattern seen in the other tables. We believe there is a bug in our Structured Perceptron code due to a high prevalence of 2 predictions in the tagging.

4.6 Extension: CRF

We used the CRF package written by Terry Peng [3]. For this model, we incorporate a variety of features including lowercase, uppercase, digits, and part of speech. We optimize the CRF using L-BFGS second order approximation. Training time takes under 30 seconds using L-BFGS with line search. The loss no longer decreases at around 200 iterations.

Table 10: CRF Validation F-Score

Tags	**Precision**	**Recall**	**F-Score**
PER	0.82	0.84	0.83
LOC	0.79	0.73	0.76
ORG	0.73	0.67	0.70
MISC	0.82	0.75	0.78
Avg	0.79	0.76	0.77

We also enter our CRF model into the Kaggle competition and achieve moderate scores that put us at 5th place currently.

[3]https://github.com/tpeng/python-crfsuite

Table 11: CRF Kaggle Predictions

Model	**F-Score**
CRF	0.56533
CRF+POS	0.58952
CRF+POS	<u>0.59936</u>

5 Conclusion

The dynamic programming aspect of these models is incredibly important for run times. Our HMM and MEMM trained rather quickly, but the Structured Perceptron took a very long duration. The CRF was run in a package, so everything was optimized to the fullest extent possible. Time complexity was not a problem for the CRF model.

The use of feature functions was also quite necessary to advance far on the Kaggle leaderboard. While we incorporated case-sensitivity and part of speech, we were quite a ways from the top 3 positions. We believe that using lexical features and n-grams will suitably raise the F-score on our CRF model.