Machine Learning with Linear Chain Conditional Random Fields

James Diotte
CUNY Graduate Center
(Dated: May 27, 2017)

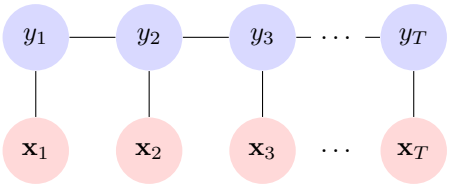
Abstract:

I. INTRODUCTION

A conditional random field (CRF) is a special type of undirected probabilistic graphical model (PGM) that aims to model the conditional dependence $p(\mathbf{y}|\mathbf{x})$ of a multi-dimensional output $\mathbf{y} \in \{1, \dots, L\}^T = \mathbb{L}^T$ on input features $\mathbf{x}$ assuming that there are stochastic dependencies between some of the components of $\mathbf{y}$. We think of the components y_i of $\mathbf{y}$ as nodes in a graph and then vertices in this graph encode our stochastic dependence. In this project we focus only on linear chain CRFs which are modeled as a straight chain graph as follows:

$$\mathbf{y} = (y_1.y_2, \dots, y_T)$$

$$\mathbf{x} = (x_1, x_2 \dots, x_T)$$



And the model takes the form:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \exp(\sum_{t=1}^{T} \phi(y_t, y_{t-1}, \mathbf{x}, t))$$

Where:

$$Z(\mathbf{x}) = \sum_{\mathbf{y}'} \exp(\sum_{t=1}^{T} \phi(y'_t, y'_{t-1}, \mathbf{x}, t))$$

ϕ is called the potential function and it splits as the sum of unary and binary potential functions as:

$$\phi(y_t, y_{t-1}, \mathbf{x}, t) = \mathbf{w}_u^T \mathbf{f}_u(y_t, \mathbf{x}, t) + \mathbf{w}_b^T \mathbf{f}_b(y_t, y_{t-1}, \mathbf{x}, t)$$

$$= \mathbf{w}^T \mathbf{f}(y_t, y_{t-1}, \mathbf{x}, t)$$

Here $\mathbf{w}_u \in \mathbb{R}^{K_u}$, $\mathbf{w}_b \in \mathbb{R}^{K_b}$ are the learnable parameters of the model and the vector-valued functions $\mathbf{f}_u, \mathbf{f}_b$ are called the unary and binary feature functions (resp.) and are usually chosen based on domain knowledge. Intuitively, the unary feature function makes local predictions while the binary feature function makes corrections based on the graph structure. This particular form of chain CRF model is not the most general but rather is a parameter tied model, what this means is that we use the same unary weights for each node and the same binary weights for each edge, in general we can have a different weight vector for each.

II. MODEL INFERENCE

Once we have a trained model (see Section III) we would like to be able to do inference tasks with it. First is simply finding the conditional probability $p(\mathbf{y}|\mathbf{x})$ of a pair $(\mathbf{y}|\mathbf{x})$, this requires computing the partition function $Z(\mathbf{x})$ which-as can be seen from its definition- requires a summation over exponentially many terms (L^T). To make the computation of the partition function tractable we use the dynamic programming algorithm known as the *forward-backward algorithm*. The forward-backward algorithm creates two tables $\alpha[t, j]$ and $\beta[t, j]$ as follows:

- $\alpha[1, j] = \phi(j, y_0, \mathbf{x}, 1)$ for all $j \in \mathbb{L}$
- $\beta[T, j] = 1$ for all $j \in \mathbb{L}$
- $\alpha[t, j] = \log \sum_{i=1}^{L} \exp(\phi(j, i, \mathbf{x}, t) + \alpha[t-1, i])$
- $\beta[t, j] = \log \sum_{i=1}^{L} \exp(\phi(i, j, \mathbf{x}, t+1) + \beta[t+1, i])$

Here y_0 is a fixed dummy state (as first node doesn't have a predecessor in a linear chain CRF) and the log-sum-exponentials are computed in the numerically stable way by first subtracting the maximum of the summands: $\log \sum \exp(a_i) = m + \log \sum \exp(a_i - m)$. With the alpha and beta tables we can then compute both $\log Z(\mathbf{x})$ and $\log p(y_t = i, y_{t-1} = j|\mathbf{x})$, where the last term are the pairwise marginals that will be used later on in gradient calculations.

$$\log Z(\mathbf{x}) = \log \sum_{j=1}^{L} \exp(\alpha[T, j])$$

$$\log p(y_t = i, y_{t-1} = j|\mathbf{x}) = \alpha[t-1, i] +$$

$$\phi(j, i, \mathbf{x}, t) + \beta[t, j] - \log \sum_{l=1}^{L} \exp(\phi(l, y_0, \mathbf{x}, 1) + \beta[1, l])$$

Another important inference task is making predictions on previously unseen data. The method used in this project is the maximum a posteriori probability estimation (MAP), given a data vector $\mathbf{x}$ we predict the most likely $\hat{\mathbf{y}}$ as:

$$\begin{aligned}\hat{\mathbf{y}} &= \underset{\mathbf{y}}{\operatorname{argmax}}\, p(\mathbf{y}|\mathbf{x}) \\ &= \underset{\mathbf{y}}{\operatorname{argmax}}\, \log(p(\mathbf{y}|\mathbf{x})) \\ &= \underset{\mathbf{y}}{\operatorname{argmax}} \sum_{t=1}^{T} \phi(y_t, y_{t-1}, \mathbf{x}, t) - \log(Z(\mathbf{x}) \\ &= \underset{\mathbf{y}}{\operatorname{argmax}} \sum_{t=1}^{T} \phi(y_t, y_{t-1}, \mathbf{x}, t)\end{aligned}$$

Just like the partition function this involves taking the argmax over exponentially many terms and also like the partition function it can be done with a dynamic programming algorithm as follows:

- $\delta[1, j] = 0$ for all $j \in \mathbb{L}$
- $\delta_a[1, j] = 0$ for all $j \in \mathbb{L}$
- $\delta[t, j] = \max_{1 \le i \le L} \phi(j, i, \mathbf{x}, t) + \delta[t-1, i]$
- $\delta_a[t, j] = \underset{1 \le i \le L}{\operatorname{argmax}} \phi(j, i, \mathbf{x}, t) + \delta[t-1, i]$
- $\hat{y}_T = \underset{1 \le i \le L}{\operatorname{argmax}} \delta[T, i]$
- $\hat{y}_t = \delta_a[t+1, \hat{y}_{t+1}]$ for $1 \le t < T$

Other inference tasks that can be done but I did not have time to implement in this project include sampling from the modeled distribution via Markov Chain Monte Carlo (MCMC) as well as an alternative prediction framework called maximum posterior marginal (MPM).

III. PARAMETER LEARNING

In this project the set up for all our experiments were supervised classification tasks and as such we have a collection of observed data $\mathcal{D} = \{(\mathbf{x}^n, \mathbf{y}^n)\}_{n=1}^N$ from which to learn the ideal weights $\mathbf{w} = (\mathbf{w}_u, \mathbf{w}_b)$ for our model. As a generative model the loss function chosen is the negative-log-likelihood:

$$nll(\mathbf{w}) = -\sum_{n=1}^{N} \log p(\mathbf{y}^n|\mathbf{x}^n, \mathbf{w})$$

Which becomes:

$$nll(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{t=1}^{T} [\mathbf{w}^T \mathbf{f}(y_t^n, y_{t-1}^n, \mathbf{x}^n, t) - \log Z(\mathbf{x}^n)]$$

The computational goal is to find the weights that minimize the negative-log-likelihood and to accomplish this we will use gradient descent. The gradient is given by the following expression which can be easily derived with a little bit of calculus:

$$\begin{aligned}\nabla nnl(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{t=1}^{T} [&\mathbf{f}(y_t^n, y_{t-1}^n, \mathbf{x}^n, t) \\ &- \mathbb{E}_{\mathbf{y} \sim p(\mathbf{y}|\mathbf{x}^n, \mathbf{w})}[\mathbf{f}(y_t, y_{t-1}, \mathbf{x}^n, t)]]\end{aligned}$$

Using the independence structure in the graph the expected value term is equivalent to:

$$\sum_{y_t, y_{t-1} \in \mathbb{L}} p(y_t = i, y_{t-1} = j|\mathbf{x}^n, \mathbf{w}) \mathbf{f}(y_t, y_{t-1}, \mathbf{x}^n, t)$$

This term is much more tractable as the expected value would require exponentially (L^T) many operations where this formulation only requires L^2 and we have our dynamic implementations of the marginals. However if there are a lot data points and or lots of features this could still be prohibitively expensive in which case approximate methods must be used. One such method, not used in this project due to time constraints, is stochastic MLE which uses MCMC sampling from our distribution to estimate the expectation in the gradient. The second method and the one employed in this project is to replace the negative-log-likelihood by the negative-log-pseduo-likelihood (pll):

$$pll(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{t=1}^{T} \log p(y_t^n|\mathbf{y}_{\mathcal{N}(t)}^n, \mathbf{x}^n, \mathbf{w})$$

Where in our simple linear chain case the neighbors $\mathcal{N}(t)$ are just the predecessor and successor, $t-1$ abd $t+1$ (or just one or the other if at the beginning or end of chain). The pseudo-marginal term is given by:

$$p(y_t|\mathbf{y}_{\mathcal{N}(t)}, \mathbf{x}) = \frac{1}{Z_t} \exp(\phi(y_t, y_{t-1}, \mathbf{x}, t) + \phi(y_{t+1}, y_t, \mathbf{x}, t+1))$$

And the pseudo partition function is:

$$Z_t(\mathbf{x}) = \sum_{l=1}^{L} \exp(\phi(l, y_{t-1}, \mathbf{x}, t) + \phi(y_{t+1}, l, \mathbf{x}, t+1))$$

Unlike the full partition function this requires only a linear pass over the label set making the pseudo-marginal term very quick to compute. The corresponding gradient we get is:

$$\begin{aligned}\nabla pll(\mathbf{w}) \quad = \quad -\sum_{n=1}^{N}\sum_{t=1}^{T} [&\phi(y_t^n, y_{t-1}^n, \mathbf{x}, t) \; + \\ \phi(y_{t+1}^n, y_t^n, \mathbf{x}, t+1) - &\sum_{l=1}^{L} p(l|\mathbf{y}_{\mathcal{N}(t)}^n, \mathbf{x}^n, \mathbf{w})(\phi(l, y_{t-1}^n, \mathbf{x}, t) + \\ \phi(y_{t+1}^n, l, \mathbf{x}, t+1))]\end{aligned}$$

While this looks a bit more complicated it is in fact much cheaper to compute than the full gradient. Some other practical issues arise when our data set $\mathcal{D}$ is large or when the data vectors $\mathbf{x}$ are in a high dimensional space. To deal with the first case we employ mini-batch

stochastic gradient descent algorithms in particular this project uses the ADAM algorithm [1]. Also employed in this project to deal with the second issue is the technique of two stage training. If we set our binary weights to zero our graphical model essentially degenerates to a bunch of isolated nodes in which case the mathematical model is essentially just logistic regression at each node. To speed up training we can can pre-train our unary features as a logistic model on each individual node of the training data and then learn the binary feature weights with the above gradient, which is now much lower dimensional. Finally for both *nll* and *pll* we can add an L_2 regularization factor, however during my experiments I found that the regularizer decreased performance and so it was discarded.

IV. EXPERIMENTS

For this project linear chain CRF models were tested on 3 different data sets:

- Synthetic Chain Data
- OCR Handwritten Character Data set
- Gesture Recognition Data sets

In each case the model were evaluated based on two different metrics: the average Hamming accuracy which is the average percentage of components in a chain predicted correctly, and $0-1$ accuracy score which considers a prediction correct if and only if every component is predicted correctly. In each experiment the chain model is compared to two local (unary) models: linear kernel support vector classifier, and logistic regression (or unary only chain model).

A. Synthetic Data

The synthetic data set used was generated in Matlab and consists of 1000 data points which are each sequences of length 10 with 3 possible labels at each node. The data vector $\mathbf{x}$ consists of a 3-dimensional vector for each node in the chain which was generated as random normal noise added to component correlated to the label. The unary feature map chosen as:

$$\mathbf{f}_u(y_t, \mathbf{x}, t) = \mathbf{x}_t \otimes \mathbf{e}_{y_t} \in \mathbb{R}^3 \otimes \mathbb{R}^3 \cong \mathbb{R}^9$$

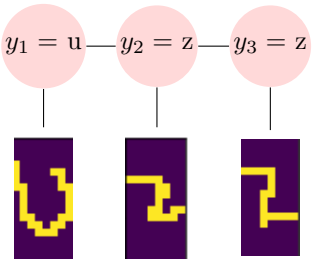
Where the tensor is flattened to be a 9-d vector. The binary feature vector was chosen as:

$$\mathbf{f}_b(y_t, y_{t-1}, \mathbf{x}, t) = (\mathbf{x}_t + \mathbf{x}_{t-1}) \otimes \mathbf{e}_{y_t} \otimes \mathbf{e}_{y_{t-1}} \in \bigotimes_{k=1}^{3} \mathbb{R}^3 \cong \mathbb{R}^{27}$$

Again the tensor is represented as a flattened 27-d vector, and here we set the intial dummy states as $y_0 = 1$ and $\mathbf{x}_0 = (0, 0, 0)$. The experimental setup was a random independent test set of 100 examples was set aside, the CRF model was then trained with ADAM stochastic gradient descent and maximum likelihood on the remaining 900 examples. Using Python's Scikit-learn package a linear kernel support vector classifier and logistic regression model were then both trained and tested on the same data.

B. OCR

The OCR data set consists of 52152 16×8 binary pixel images of handwritten lowercase characters representing 6876 individual words (potentially with first upper-case letter removed). Our task is the predict the written word given these images. We do this by modeling a word as a linear chain with each node representing a letter and then the image is flattened to a 128-d vector. Example:



In this experment the unary feature function was pre-trained on all the individual letter vectors using a multilayer perceptron with the following architecture:

$$MLP(\mathbf{x}) = \sigma_2(\mathbf{W}_2^T \sigma_1(\mathbf{W}_1^T \mathbf{x} + \mathbf{b}))$$

Where σ_2 is the softmax function and σ_1 is the rectified linear unit, the first weight matrix has 256 hidden units. The output of this model is a 26-d vector where the i-th component is the probability of $\mathbf{x}$ being the i-th letter. Now for our model this makes the unary potential function has no weights, it is simply:

$$\phi_u(y_t, \mathbf{x}, t) = (\mathbf{W}_2^T \sigma_1(\mathbf{W}_1^T \mathbf{x_t} + \mathbf{b}))_{y_t} \in \mathbb{R}$$

In other words its the y_t-th component of the MLP output before going into softmax. Our binary feature map is simple, it is just:

$$\mathbf{f}_b(y_t, y_{t-1}, \mathbf{x}, t) = \mathbf{e}_{y_t} \otimes \mathbf{e}_{y_{t-1}} \in \mathbb{R}^{26} \otimes \mathbb{R}^{26}$$

So the job of the binary potential is to learn the most and least probable letter bigrams. Here the initial state $y_0 = 0$ so the map outputs the zero vector for initial state. Since the unary potential was pre-trained on all letters the goal for this experiment was to use a training test split to see if the learned binary weights could improve

upon the unary only model. So words were separated into training set of 6200 words and a test set of 686 words and the binary weights were trained on the training set. The baseline linear SVC was trained like the unary potential on all the individual letters and was then tested on the test set so they were not independent, the goal was just to see if the binary weights could improve the unary only model. The gradient of *nll* was too expensive to compute and thus pseudo-likelihood and ADAM stochastic gradient descent were used to fit the model.

C. Gesture Data sets

This experiment is on pre-extracted and processed data coming from videos of people making hand gestures. Each video is condensed down and a node represents the gesture made during a 30 frame length. For each node there is a 60-dimensional bag-of-words feature. There are 20 distinct sets of videos and each one is split into 30 training videos and 17 testing videos (except the second set which only has 16). Each of the 20 sets is treated separately hence we get 20 slightly different models in each case. The model used was very similar to the one used in the OCR experiment except the unary potential MLPs had varying structure with either 256 or 512 hidden units and a dropout layer was added between hidden and output layer (see source code for exact parameters in each case), and for the binary features the label class ranged from 9 to 14 so get binary feature vectors of dimensions ranging from 9^2 to 14^2. Pseudo-likelihood and ADAM stochastic gradient descent were used to fit each model.

V. RESULTS AND DISCUSSION

A. Synthetic

Model:	Average Hamming:	Zero-One:
SVC	0.923	0.48
LogReg	0.923	0.49
CRF	0.954	0.62

FIG. 1. Synthetic Results

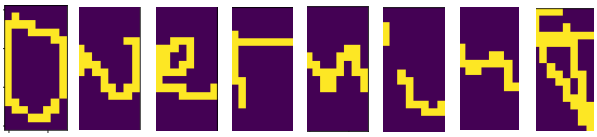
In FIG 1 we can see the results of the experiment on the synthetic data: the Average Hamming column is the average Hamming score (percent of sequence predicted correctly0 of the model across all 100 test samples, the Zero-One column is the percentage of predictions which were completely accurate. We can see that the CRF model outperformed both linear kernel support vector classifier and basic logistic regression in both metrics.

B. OCR

Model:	Average Hamming:	Zero-One:
SVC	0.874	0.46
Unary Only	0.955	0.738
Unary+Binary	0.979	0.91

FIG. 2. OCR Results

The OCR experiment showed an very large increase in zero-one loss from SVC and unary only to the full binary CRF model, in fact it improved nearly 20% and brings the overall accuracy to near 98%. We can visualize examples where the binary potentials improve the unary. Consider the following word:



The full CRF model is able to accurately predict the entire word 'overning', whereas the unary only CRF model predicts 'ouerninp' and SVC predicts 'overminq'.

C. Gestures

Models: 0 = SVC, 1 = Unary only CRF, 2 = full CRF (Binary and Unary).

Model:	Average Hamming:	Zero-One:
0	0.392	0
1	0.776	0
2	0.872	0

FIG. 3. Gesture Set 1 Results

Model:	Average Hamming:	Zero-One:
0	0.464	0
1	0.739	0
2	0.782	0.059

FIG. 4. Gesture Set 2 Results

Model:	Average Hamming:	Zero-One:
0	0.346	0
1	0.552	0
2	0.623	0.059

FIG. 5. Gesture Set 3 Results

Model:	Average Hamming:	Zero-One:
0	0.342	0
1	0.801	0
2	0.856	0

FIG. 6. Gesture Set 4 Results

Model:	Average Hamming:	Zero-One:
0	0.237	0
1	0.731	0
2	0.773	0

FIG. 7. Gesture Set 5 Results

Model:	Average Hamming:	Zero-One:
0	0.289	0
1	0.649	0
2	0.68	0

FIG. 8. Gesture Set 6 Results

Model:	Average Hamming:	Zero-One:
0	0.258	0
1	0.731	0
2	0.779	0

FIG. 9. Gesture Set 7 Results

Model:	Average Hamming:	Zero-One:
0	0.289	0
1	0.633	0
2	0.648	0

FIG. 10. Gesture Set 8 Results

Model:	Average Hamming:	Zero-One:
0	0.45	0
1	0.813	0
2	0.879	0

FIG. 11. Gesture Set 9 Results

Model:	Average Hamming:	Zero-One:
0	0.299	0
1	0.573	0
2	0.686	0

FIG. 12. Gesture Set 10 Results

Model:	Average Hamming:	Zero-One:
0	0.441	0
1	0.662	0
2	0.719	0

FIG. 13. Gesture Set 11 Results

Model:	Average Hamming:	Zero-One:
0	0.511	0
1	0.805	0
2	0.847	0.294

FIG. 14. Gesture Set 12 Results

Model:	Average Hamming:	Zero-One:
0	0.49	0
1	0.769	0.059
2	0.816	0

FIG. 15. Gesture Set 13 Results

Model:	Average Hamming:	Zero-One:
0	0.49	0
1	0.797	0
2	0.789	0.118

FIG. 16. Gesture Set 14 Results

Model:	Average Hamming:	Zero-One:
0	0.346	0
1	0.753	0
2	0.828	0

FIG. 17. Gesture Set 15 Results

Model:	Average Hamming:	Zero-One:
0	0.372	0
1	0.682	0
2	0.736	0

FIG. 18. Gesture Set 16 Results

Model:	Average Hamming:	Zero-One:
0	0.44	0
1	0.658	0
2	0.706	0

FIG. 19. Gesture Set 18 Results

Model:	Average Hamming:	Zero-One:
0	0.476	0
1	0.781	0.059
2	0.84	0.176

FIG. 20. Gesture Set 19 Results

Model:	Average Hamming:	Zero-One:
0	0.478	0
1	0.716	0
2	0.767	0

FIG. 21. Gesture Set 20 Results

Model:	Average Hamming:	Zero-One:
0	0.395	0
1	0.718	0.006
2	0.77	0.035

FIG. 22. Gesture Set Overall Average Results

There are many figures to look over here but in all but one the full CRF model outperforms unary only CRF, both of which always beat out the SVC model. All of the models have very poor zero-one loss only beating 0 % in a few instances. Overall the full CRF model has significantly higher performance.

D. Conclusion

In all three experiments the full CRF (using both unary and binary potentials) was the top performer with respect to both metrics used. In the OCR experiment in particular the results were pretty drastic, the full CRF model scored nearly 20% higher in the zero-one loss metric and had a nearly perfect 97.8% average Hamming accuracy score. Overall this project showed that CRF models are potentially useful in other tasks where computer vision and natural language processing intersect. It would be interesting to test other local classifiers to see if any could potentially beat the power of the full CRF model in the these three experiments.

VI. SOURCE CODE

All source code containing all the parameters of the models used can be found here: https://github.com/jad2192/main/tree/master/machine-learning/CRF

[1] Kingma, Diederik P. and Ba, Jimmy. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG], December 2014.