Conditional Random Fields

Rahul Gupta*
(under the guidance of Prof. Sunita Sarawagi, KReSIT, IIT Bombay)

Abstract

In this report, we investigate Conditional Random Fields (CRFs), a family of conditionally trained undirected graphical models. We give an overview of linear CRFs that correspond to chain-shaped models and show how the marginals, partition function and MAP-labelings can be computed. Then, we discuss various approaches for training such models - ranging from the traditional method of maximizing the conditional likelihood or its variants like the pseudo likelihood to margin maximization. For the margin-based formulation, we look at two approaches - the SMO algorithm and the exponentiated gradient algorithm. We also discuss two other training approaches - one that attempts at removing the regularization term and other that uses a kind of boosting to train the model.

Apart from training, we look at topics like the extension to segment level CRFs, inducing features for CRFs, scaling them to large label sets, and performing MAP inferencing in the presence of constraints.

From linear CRFs, we move on to arbitrary CRFs and discuss exact algorithms for performing inferencing and the hardness of the problem. We go over a special class of models - Associative Markov Networks, which are applicable in some real-life scenarios and which permit efficient inferencing. We then look at collective classification as an application of general undirected models.

Finally, we very briefly summarize the work that could not be covered in this report and look at possible future directions.

1 Undirected Graphical Models

Let $X = X_1, \ldots, X_n$ be a set of n random variables. Assume that $p(X)$ is a joint probability distribution over these random variables. Let X_A and X_B be two subsets of X which are known to be conditionally independent, given X_C. Then, $p(.)$ respects this conditional independence statement if

$$p(X_A|X_B, X_C) = p(X_A|X_C) \tag{1}$$

or alternatively,

$$\begin{aligned} p(X_A, X_B|X_C) &= \frac{p(X_A, X_B, X_C)}{p(X_C)} \\ &= \frac{p(X_A|X_B, X_C)p(X_B, X_C)}{p(X_C)} \\ &= p(X_A|X_C)p(X_B|X_C) \end{aligned} \tag{2}$$

The shorthand notation for such a statement is : $X_A \perp X_B|X_C$.

Given X and a list of such conditional independence statements, we would like to characterize the family of joint probability distributions over X that satisfy all these statements. To achieve this, consider an undirected graph $G = (X, E)$ whose vertices correspond to our set of random variables. We would construct the edge set E in such a manner that the following property holds: If the deletion of all vertices in X_C from the graph results in the removal of all paths from X_A to X_B, then $X_A \perp X_B|X_C$. Conversely, given an undirected graph $G = (X, E)$, we can exhaustively enumerate all conditional independence

*grahul@it.iitb.ac.in

statements represented by it. However, note that the number of such statements can be exponential in the number of vertices.

Let us restrict our attention to 'Markovian' probability distributions. A probability distribution $p(.)$ is said to be Markovian w.r.t G and a set of vertices S if

$$p(S|\bar{S}) = p(S|N(S)) \tag{3}$$

where $N(S)$ is the set of those neighbours of vertices in S which lie outside S. $N(S)$ is often called the Markovian blanket of S.

If $p(.)$ is Markovian for all singleton sets $S = \{X_i\}$, then $p(.)$ is said to be locally Markovian. If $p(.)$ is Markovian for all sets $S \in 2^X$, then $p(.)$ is globally Markovian. Trivially, a globally Markovian distribution is also locally Markovian.

Hammersley and Clifford proved the following two theorems regarding Markovian distributions. The proofs are available in [Cli90]. Here $\mathcal{C}$ is the set of all cliques in the graph.

Theorem 1. *A locally Markovian distribution is also globally Markovian.*

Theorem 2. *P is Markovian iff it can be written in the form*

$$P(X) \propto \exp(\sum_{C \in \mathcal{C}} Q(C, X))$$

In Theorem 2, $Q(.)$ is an arbitrary real valued function that judges how likely is an assignment of values to the random variables that form the clique vertices.

By summing over all possible assignments, we can remove the proportionality sign and write $P(X)$ as

$$P(X) = \frac{\exp(\sum_{C \in \mathcal{C}} Q(C, X))}{\sum_X \exp(\sum_{C \in \mathcal{C}} Q(C, X))} \tag{4}$$

The denominator in Equation 4 is denoted as Z and is called the partition function.

The exponential form in Equation 4 allows us to write $P(X)$ as a product :

$$P(X) = \frac{\prod_C \psi_C(X)}{Z} \tag{5}$$

where $\psi_C(X) = \exp(Q(C, X))$ is called the potential function for clique C.

Note: There is a slight abuse of notation here. Both Q and ψ_C do not take the entire assignment X as input, but only the assignment restricted to the vertices in C.

The potential functions can be intuitively seen as preference functions over assignments to clique vertices. A more probable assignment $X = (x_1, \ldots, x_n)$ is likely to have better contributions from most of the constituent potential functions than a less probable assignment. However, the potential function of a clique should not be confused with its marginal distribution. Infact, as we will see in Section 5.1, potential function is just one of the terms that the marginal is proportional to.

This is one of the areas where undirected models score over directed models like MEMMs and HMMs. Directed models have a 'probability mass conservation constraint' that forces the local distributions to be normalized to 1. Hence, they suffer from the the label bias problem ([LMP01]). In undirected models, the local potential functions are unnormalized, and instead, global normalization is done using Z.

1.1 Conditional Random Fields

Consider a scenario where a hidden process is generating observables. Assume that the structure of the hidden process is known. For example, in NER and POS tagging tasks, we make the assumption that a particular POS tag (or named entity tag) depends only on the current word and the immediately previous and the immediately next tags. This corresponds to an undirected graphical model in the shape of a linear chain. Another example is the classification of a set of hyperlinked documents. The label of a

document can be assumed to be dependent upon the document itself and the labels of the documents that link into it or out of it.

Two tasks arise in these scenarios:

1. **Learning:** Given a sample set of the observables $\{\mathbf{x_1}, \ldots, \mathbf{x_N}\}$ along with the values of the hidden labels $\{\mathbf{y_1}, \ldots, \mathbf{y_N}\}$, learn the best possible potential functions such that some criteria is maximized.

2. **Inference:** Given a new observable $\mathbf{x}$, find the most likely set of hidden labels $\mathbf{y}^*$ for $\mathbf{x}$, i.e. compute (exactly or approximately):
$$\mathbf{y}^* = \arg\max_{\mathbf{y}} P(\mathbf{y}|\mathbf{x}) \tag{6}$$

Here, the graphical model would have some nodes (say Y_i's) and edges corresponding to the labels and the dependencies between them and atleast one more node (say X) corresponding to the observable $\mathbf{x}$, along with some edges of the kind (X, Y_i). The joint probability distribution can thus be written as

$$P(\mathbf{x}, y_1, \ldots, y_M) = \frac{1}{Z}\psi_{\{X\}}(\mathbf{x}) \prod_{C \in \mathcal{C}, C \neq \{X\}} \psi_C(\mathbf{x}, \mathbf{y}) \tag{7}$$

Learning this joint distribution is both intractable (because the $\psi_{\{X\}}(.)$ function is hard to approximate without making naive assumptions) as well as useless (because $\mathbf{x}$ is already provided to us). Thus, it makes sense to learn the following conditional distribution:

$$P(y_1, \ldots, y_M|\mathbf{x}) = \frac{1}{Z_{\mathbf{x}}} \prod_{C \in \mathcal{C}, C \neq \{X\}} \psi_C(\mathbf{x}, \mathbf{y}) \tag{8}$$

Note that the normalizer is now observable-specific.

The undirected graph with the set of nodes $\{X\} \cup Y$ and the relevant Markovian properties is called a conditional random field (CRF). From now on, we will assume that $\mathcal{C}$ excludes the singleton clique $\{X\}$.

1.2 CRFs for sequence labeling

Before we move further, let us look at a special kind of CRFs, one where all the nodes in the graph form a linear chain. Such models are extensively used in POS tagging, NER tasks and shallow parsing ([LMP01], [SP03]). For these models, the set of cliques, $\mathcal{C}$, is just the set of all cliques of size 1 (viz. the nodes) and the set of all cliques of size 2 (the edges). Thus, the conditional probability distribution can be written as:

$$P(Y_1, \ldots, Y_M|X) = \frac{1}{Z_{\mathbf{x}}} \prod_i (\psi_i(Y_i, X)\psi'_i(Y_i, Y_{i-1}, X)) \tag{9}$$

where $\psi_\_(.)$ acts over single labels and $\psi'_\_(.)$ acts over edges. Most sequence labeling applications parameterize $\psi_i(.)$ and $\psi'_i(.)$ in a log-linear fashion.

$$\psi_i(.) = \exp(\sum_k \theta_k s_k(y_i, \mathbf{x}, i)) \tag{10}$$

$$\psi'_i(.) = \exp(\sum_j \lambda_j t_j(y_{i-1}, y_i, \mathbf{x}, i)) \tag{11}$$

where s_k is a state feature function that uses only the label at a particular position, and t_j is a transition feature function that depends on the current and the previous label. Examples of some such functions are: "is the label NOUN and the current word capitalized?" and "was the previous label SALUTATION, current label PERSON and the current word in the dictionary of proper nouns?". The parameters $(\mathbf{\Theta}, \mathbf{\Lambda})$ denote the importance of each of the features and are learnt during the learning phase by maximing some criteria like conditional log likelihood.

For ease of notation, we will merge the node features with the edge features and use f_j to denote the j^{th} feature function. Assume that there are a total of k feature functions. All the learnt parameters will

be merged into a single $\mathbf{\Lambda}$ vector $(k \times 1)$. Now consider the $k \times n$ matrix $\mathcal{F}$ where $\mathcal{F}_{ji} = f_j(y_i, y_{i-1}, \mathbf{x}, i)$. Thus, the conditional probability of a given label sequence can be succintly written as

$$P(y_1, \ldots, y_n|\mathbf{x}) = \frac{\exp(\mathbf{\Lambda}^T \mathcal{F}\mathbf{1}_{n\times 1})}{Z_{\mathbf{x}}} \tag{12}$$

The vector $\mathcal{F}\mathbf{1}_{n\times 1}$ is called the global feature vector and is denoted as $\mathbf{F}(\mathbf{y}, \mathbf{x})$. $\mathbf{f}(y_i, y_{i-1}, \mathbf{x}, i)$ will denote the local feature vector at the i^{th} position. The quantities $\exp(\mathbf{\Lambda}^T \mathbf{f}(y, y', \mathbf{x}, i))$ are often represented using matrices M_i's whose rows and columns are indexed by labels.

Note that the normalizer of the conditional probability is independent of $\mathbf{y}$, so during inferencing, we have to compute $\mathbf{y}^*$ such that :

$$\mathbf{y}^* = \arg\max_{\mathbf{y}} \mathbf{\Lambda}^T . \mathbf{F}(\mathbf{y}, \mathbf{x}) \tag{13}$$

1.2.1 Forward and backward vectors

Since the space of possible label sequences is exponentially large in the size of the input, techniques like dynamic programming are used, both in training as well as inferencing. Suppose that we are interested in tagging a sequence only partially, say till the position i. Also, lets assume that the last label in this partial labeling is some arbitrary but fixed y. Denote the unnormalized probability of a partial labeling ending at position i with label y by $\alpha(y, i)$. Similarly, denote the unnormalized probability of a partial segmentation starting at position $i + 1$ assuming a label y at position i by $\beta(y, i)$.

α and β can be computed via the following recurrences:

$$\alpha(y, i) = \sum_{y'} \alpha(y', i-1). \exp(\mathbf{\Lambda}^T \mathbf{f}(y, y', \mathbf{x}, i)) \tag{14}$$

$$\beta(y, i) = \sum_{y'} \beta(y', i+1). \exp(\mathbf{\Lambda}^T \mathbf{f}(y', y, \mathbf{x}, i+1)) \tag{15}$$

where $\mathbf{f}(.,.,.,i)$ is the feature vector evaluated at the i^{th} sequence position. The base cases are:

$$\alpha(y, 0) = [\![y = \text{`}start']\!] \tag{16}$$

$$\beta(y, n+1) = [\![y = \text{`}stop']\!] \tag{17}$$

α and β are called the forward and backward vectors respectively. We can now write the marginals and partition function in terms of these vectors.

$$P(Y_i = y|\mathbf{x}) = \alpha(y, i)\beta(y, i)/Z_{\mathbf{x}} \tag{18}$$

$$P(Y_i = y, Y_{i+1} = y'|\mathbf{x}) = \alpha(y, i) \exp(\mathbf{\Lambda}^T \mathbf{f}(y', y, \mathbf{x}, i+1))\beta(y', i+1)/Z_{\mathbf{x}} \tag{19}$$

$$Z_{\mathbf{x}} = \sum_y \alpha(y, |\mathbf{x}|) = \sum_y \beta(y, 1) \tag{20}$$

1.2.2 Inference in linear CRFs using the Viterbi algorithm

In CRFs, training and inference are often interleaved. At each iteration during training, the system computes its best estimate for labeling the training data and updates the model based on the error in that estimate. Given the parameter vector $\mathbf{\Lambda}$, the best labeling for a sequence can be found exactly using the Viterbi algorithm.

For each tuple of the form (i, y), the Viterbi algorithm maintains the unnormalized probability of the best labeling ending at position i with the label y. The labeling itself is also stored along with the probability. Denoting the best unnormalized probability for (i, y) by $V(i, y)$, the recurrence is:

$$V(i, y) = \begin{cases} \max_{y'}(V(i-1, y'). \exp(\mathbf{\Lambda}^T \mathbf{f}(y, y', \mathbf{x}, i))) & (i > 0) \\ [\![y = start]\!] & (i = 0) \end{cases} \tag{21}$$

The normalized probability of the best labeling is given by $\frac{\max_y V(n,y)}{Z_{\mathbf{x}}}$ and the labeling itself is given by $\arg\max_y V(n, y)$. Thus, if y can range over a set of m labels, then the runtime of the Viterbi algorithm is $O(nm^2)$.

2 Training

The various methods used to train CRFs differ mainly in the objective function they try to optimize. We look at the following methods to train a CRF.

1. The penalized log-likelihood criteria.
2. Pseudo log-likelihood.
3. Voted perceptron.
4. Margin maximization.
5. Gradient tree boosting.
6. Logarithmic pooling.

2.1 Penalized log-likelihood

The conditional log-likelihood of a set of training instances $(\mathbf{x^k}, \mathbf{y^k})$ using parameters $\mathbf{\Lambda}$ is given by:

$$\mathcal{L}_{\mathbf{\Lambda}} = \sum_k \mathbf{\Lambda}^T.\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - \log Z_{\mathbf{\Lambda}}(\mathbf{x^k}) \tag{22}$$

The gradient of the log-likelihood is given by

$$\begin{aligned}
\nabla\mathcal{L}_{\mathbf{\Lambda}} &= \sum_k (\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - \frac{\sum_{\mathbf{y}} \mathbf{F}(\mathbf{y}, \mathbf{x^k}) \exp(\mathbf{\Lambda}^T \mathbf{F}(\mathbf{y}, \mathbf{x^k}))}{Z_{\mathbf{\Lambda}}(\mathbf{x^k})}) \\
&= \sum_k (\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - \sum_{\mathbf{y}} \mathbf{F}(\mathbf{y}, \mathbf{x^k}) P(\mathbf{y}|\mathbf{x^k})) \\
&= \sum_k (\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - E_{P(\mathbf{y}|\mathbf{x^k})}[\mathbf{F}(\mathbf{y}, \mathbf{x^k})])
\end{aligned} \tag{23}$$

where $E_{\_}[.]$ is the expected value of the global feature vector under the conditional probability distribution.

Note that putting the gradient equal to zero corresponds to the maximum entropy constraint. This is expected because CRFs can be seen as a generalization of logistic regression. Recall that for logistic regression, the conditional distribution that maximizes the log-likelihood also has the maximum entropy, assuming that the statistics in the training data are preserved. In both cases, this is made possible because of the exponential form of the distribution, which is the only family of distributions to posess such characteristics ([Ber]).

Like logistic regression, CRFs too suffer from the bane of overfitting. Thus, we impose a penalty on large parameter values. The most popular technique imposes a zero prior on all the parameter values. The penalized log-likelihood is given by (upto a constant):

$$\mathcal{L}_{\mathbf{\Lambda}} = \sum_k (\mathbf{\Lambda}^T.\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - \log Z_{\mathbf{\Lambda}}(\mathbf{x^k})) - \frac{\|\mathbf{\Lambda}\|^2}{2\sigma^2} \tag{24}$$

and the gradient is given by

$$\nabla\mathcal{L}_{\mathbf{\Lambda}} = \sum_k (\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - E_{P(\mathbf{y}|\mathbf{x^k})}[\mathbf{F}(\mathbf{y}, \mathbf{x^k})]) - \frac{\mathbf{\Lambda}}{\sigma^2} \tag{25}$$

The tricky term in the gradient is the expectation $E_{P(\mathbf{y}|\mathbf{x^k})}[\mathbf{F}(\mathbf{y}, \mathbf{x^k})]$ those computation requires the enumeration of all the $\mathbf{y}$ sequences. Let us look at the j^{th} entry in this vector, viz. $F_j(.)$. $F_j(\mathbf{y}, \mathbf{x^k})$ is

equal to $\sum_i f_j(y_i, y_{i-1}, \mathbf{x^k}, i)$. Therefore, we can rewrite $E_{P(\mathbf{y}|\mathbf{x^k})}[F_j(\mathbf{y}, \mathbf{x^k})]$ as

$$\begin{aligned} E_{P(\mathbf{y}|\mathbf{x^k})}[F_j(\mathbf{y}, \mathbf{x^k})] &= E_{P(\mathbf{y}|\mathbf{x^k})}[\sum_i f_j(y_i, y_{i-1}, \mathbf{x^k}, i)] \\ &= \sum_i E_{P(\mathbf{y}|\mathbf{x^k})}[f_j(y_i, y_{i-1}, \mathbf{x^k}, i)] \\ &= \sum_i \sum_{y',y} \alpha(i-1, y').f_j(y, y', \mathbf{x^k}, i).e^{\mathbf{\Lambda}^T \mathbf{f}(y,y',\mathbf{x^k},i)}.\beta(i, y) \\ &= \sum_i \alpha_{i-1}^T Q_i \beta_i \end{aligned} \tag{26}$$

where α_i, β_i are the forward and backward vectors at position i, indexed by labels and Q_i is a matrix s.t. $Q_i(y', y) = f_j(y, y', \mathbf{x^k}, i).e^{\mathbf{\Lambda}^T \mathbf{f}(y,y',\mathbf{x^k},i)}$.

Thus, after all the α, β vectors and Q matrices have been computed (only $O(mn + km^2)$ values), the gradient can be easily obtained. Various iterative methods have been used to maximize the log-likelihood. Some of them are :

1. Iterative Scaling and its variants like Improved Iterative Scaling, Generalized Iterative Scaling etc.
2. Conjugate Gradient Descent and its variants like Preconditioned Conjugate Gradient Descent and Mixed Conjugate Gradient Descent.
3. Limited Memory Quasi Newton method (L-BFGS).

L-BFGS is a scalable second order method and has thus become the tool of choice in the past few years. We briefly go over the basic algorithm. An outline of the other methods, as applied to CRFs, can be seen in [LMP01], [Wal02] and [SP03].

2.1.1 L-BGFS

The standard Newton method uses second order derivatives to update the current guess of the optimum. Using Taylor's expansion, a function f can be approximated in a local neighbourhood of $\mathbf{x}$ as :

$$f(\mathbf{x} + \mathbf{\Delta}) \approx f(\mathbf{x}) + \mathbf{\Delta}^T \nabla|_{\mathbf{x}} + \frac{1}{2} \mathbf{\Delta}^T \mathbf{H}|_{\mathbf{x}} \mathbf{\Delta} \tag{27}$$

where $\nabla|_{\mathbf{x}}$ and $\mathbf{H}|_{\mathbf{x}}$ are the gradient and Hessian at $\mathbf{x}$. Optimizing w.r.t. $\mathbf{\Delta}$, we get the Newton update rule:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \mathbf{H}_k^{-1} \nabla_k \tag{28}$$

The step-size η is computed via line-search methods or taken to be 1 for quadratic optimization problems. However, when the dimensionality is large, computing the inverse of the Hessian is not feasible. So we need methods to approximate the inverse and update this approximation at each iteration. Denoting $\mathbf{H}_k^{-1}$ by $\mathbf{B}_k$, the BFGS update step gives such an approximation :

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{s_k s_k^T}{y_k^T s_k}(\frac{y_k^T \mathbf{B}_k y_k}{y_k^T s_k} + 1) - \frac{1}{y_k^T s_k}(s_k y_k^T \mathbf{B}_k + \mathbf{B}_k y_k s_k^T) \tag{29}$$

where $y_k = \nabla_k - \nabla_{k-1}$ and $s_k = \mathbf{x}_k - \mathbf{x}_{k-1}$. $\mathbf{B}_0$ is usually taken to be a positive-definite diagonal matrix. The BFGS update does away with the inverse computation, but we still have to store all the s_k and y_k vectors of the previous iterations. The L-BFGS algorithm solves this problem by storing only $\theta(m)$ such vectors, corresponding to the last m iterations. At the $(m+i)^{th}$ iteration, the vectors corresponding to i^{th} iteration are thrown away. To see this, note that the BFGS update step can be re-written as :

$$\begin{aligned} \mathbf{B}_{k+1} &= (I - \rho_k s_k y_k^T)\mathbf{B}_k(I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \quad (\text{where } \rho_k = \frac{1}{y_k^T s}) \\ &= v_k^T \mathbf{B}_k v_k + \rho_k s_k s_k^T \end{aligned} \tag{30}$$

Algorithm 1 *ComputeDirection*$(k,\ \{(s_i, y_i)|\ k-1 \leq i \leq k-m\})$

$\mathbf{d}_k \leftarrow \nabla_k$
for $k-1 \leq i \leq k-m$ **do**
  $\beta_i \leftarrow \rho_i \mathbf{d}_k^T s_i$
  $\mathbf{d}_k \leftarrow \mathbf{d}_k - \beta_i y_i$
end for
$\mathbf{d}_k \leftarrow \mathbf{B}_0 \mathbf{d}_k$
for $k-m \leq i \leq k-1$ **do**
  $\mathbf{d}_k \leftarrow \mathbf{d}_k + (\beta_i - \rho_i \mathbf{d}_k^T y_i) s_i$
end for
return $\mathbf{d}_k$

Discarding the old vectors at the $(m+i)^{th}$ iteration is equivalent to making $v_i = I$ and $\rho_i s_i s_i^T = 0_{n \times n}$. But we are not interested in explictly approximating $\mathbf{B}_{k+1}$. Rather, we just need to compute the direction in which to update $\mathbf{x}_k$, viz. $\mathbf{B}_k \nabla_k$ ($= \mathbf{d}_k$ say). Algorithm 1 shows how $\mathbf{d}_k$ can be computed using the stored values of s_i and y_i.

L-BFGS has been experimentally shown to be a very practical second-order optimization algorithm on real life problems. It has been shown to be considerably faster than conjugate gradient methods, which are first order. In addition to the basic L-BFGS algorithm, a host of improvements have been suggested to make it converge even faster. Some of them are :

1. After the direction $\mathbf{d}_k$ is computed, the step-length η is computed using Wolfe conditions :

$$\begin{aligned} f(\mathbf{x_k} + \eta \mathbf{d}_k) &\leq f(\mathbf{x}_k) + \mu \eta \nabla_k^T \mathbf{d}_k \text{ (Objective decreases a lot)} \\ |\nabla_{\mathbf{x_k} + \eta \mathbf{d_k}}| &\geq \nu |\nabla_k \mathbf{d}_k| \qquad \text{(Curvature Condition)} \end{aligned}$$

   Here μ and ν are pre-specified constants such that $0 \leq \mu \leq 1$ and $\mu \leq \nu \leq 1$. Usually a value of $\eta = 1$ is checked for compliancy with Wolfe conditions before proceeding with line-search.

2. In Algorithm 1, instead of $\mathbf{B}_0$, a scaled version $\mathbf{B}_0^k = \frac{\mathbf{y}_k^T s_k}{\|\mathbf{y}_k\|^2} \mathbf{B}_0$ is used.

2.2 Voted Perceptron Method

Perceptron uses an approximation of the gradient of the unregularized log-likelihood function. Recall that the gradient is given by :

$$\nabla \mathcal{L}_{\mathbf{\Lambda}} = \sum_k (\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - E_{P(\mathbf{y}|\mathbf{x^k})}[\mathbf{F}(\mathbf{y}, \mathbf{x^k})]) \tag{31}$$

Perceptron-based training considers one misclassified instance at a time, along with its contribution to the gradient viz. $(\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - E_{P(\mathbf{y}|\mathbf{x^k})}[\mathbf{F}(\mathbf{y}, \mathbf{x^k})])$. The feature expectation is further approximated by a point estimate of the feature vector at the best possible labeling. The approximation for the k^{th} instance can be written as :

$$\nabla \mathcal{L}_{\mathbf{\Lambda}} \approx (\mathbf{F}(\mathbf{y^k}, \mathbf{x^k}) - \mathbf{F}(\mathbf{y_k^*}, \mathbf{x^k})) \ \ (\mathbf{y_k^*} = \arg\max_{\mathbf{y}} \mathbf{\Lambda^T F}(\mathbf{y}, \mathbf{x^k})) \tag{32}$$

Note that this approximation is analogous to approximating a Bayes-optimal classifier with a MAP-hypothesis based classifier. Using this approximate gradient, the following first order update rule can be used for maximization :

$$\mathbf{\Lambda}_{t+1} = \mathbf{\Lambda}_t + \mathbf{F}(\mathbf{y_k}, \mathbf{x^k}) - \mathbf{F}(\mathbf{y_k^*}, \mathbf{x^k}) \tag{33}$$

This update step is applied once for each misclassified instance $\mathbf{x^k}$ in the training set and multiple passes are made over the training corpus. However, it has been reported that the final set of parameters

obtained suffer from overfitting ([Col02]). To solve this, [Col02] suggests a voting scheme, where, in a particular pass of the training data, all the updates are collected and their unweighted average is applied as an update to the current set of parameters. The voted perceptron scheme has been shown to achieve much lower errors in a much less number of iterations than the non-voted perceptron.

2.3 Pseudo Likelihood

So far we have been interested in maximizing the conditional probability of joint labelings. For a training instance $\mathbf{x^i}, \mathbf{y^i}$, if the trained model predicts a labeling $\mathbf{y}$ other than $\mathbf{y^i}$ then an error is said to have occured. However, in many scenarios, we are willing to assign different error values to different labelings $\mathbf{y}$. For example, in case of POS tagging, a labeling which matches the training data labeling in all positions except one is better than a labeling which matches in only a few positions.

Thus, for these scenarios, it makes sense to maximize the marginal distributions $P(y_t^i|\mathbf{x})$ instead of $P(\mathbf{y^i}|\mathbf{x})$. This objective is called the pseudo-likelihood and for the case of linear CRFs, it is given by :

$$\mathcal{L}'(\mathbf{\Lambda}) = \sum_i \sum_{t=1}^{t=|\mathbf{x^i}|} \log P(y_t^i|\mathbf{x^i}, \mathbf{\Lambda}) \tag{34}$$

The marginal distribution $P(y_t^i|\mathbf{x^i}, \mathbf{\Lambda})$ is given by :

$$P(y_t^i|\mathbf{x^i}, \mathbf{\Lambda}) = \sum_{\mathbf{y}:y_t=y_t^i} \frac{\exp(\mathbf{\Lambda^T F(y, x^i)})}{Z_{\mathbf{\Lambda}}(\mathbf{x^i})} \tag{35}$$

and the gradient of $\mathcal{L}'$ is

$$\begin{aligned} \nabla &= \sum_i \sum_t \left(\frac{\sum_{\mathbf{y}:y_t=y_t^i} \mathbf{F(y, x^i)} e^{\mathbf{\Lambda^T F(y, x^i)}}}{\sum_{\mathbf{y}:y_t=y_t^i} e^{\mathbf{\Lambda^T F(y, x^i)}}} - \sum_{\mathbf{y}} \frac{e^{\mathbf{\Lambda^T F(y, x^i)}}}{Z_{\mathbf{\Lambda}}(\mathbf{x^i})}\right) \\ &= \sum_i \sum_t (E_{P(\mathbf{y}|\mathbf{x}, \mathbf{\Lambda}, y_t^i)}[\mathbf{F(y, x^i)}] - E_{P(\mathbf{y}|\mathbf{x}, \mathbf{\Lambda})}[\mathbf{F(y, x^i)}]) \end{aligned} \tag{36}$$

The second expectation, which arises from the gradient of $\log Z_{\mathbf{\Lambda}}(\mathbf{x^i})$, can be computed as in the case of log-likelihood, using forward and backward vectors. The k^{th} component of the first expectation can be rewritten as :

$$\begin{aligned} E_{P(\mathbf{y}|\mathbf{x}, \mathbf{\Lambda}, y_t^i)}[\sum_j f_k(y_j, y_{j-1}, x^i)] &= \sum_j E_{P(\mathbf{y}|\mathbf{x}, \mathbf{\Lambda}, y_t^i)}[f_k(y_j, y_{j-1}, x^i)] \\ &= \sum_j E_{P(y_j, y_{j-1}|\mathbf{x}, \mathbf{\Lambda}, y_t^i)}[f_k(y_j, y_{j-1}, x^i)] \end{aligned}$$

The second identity holds because of the fact that $E_{p(A,B)}[g(A)] = E_{p(A)}[g(A)]$. Now, $P(y_j, y_{j-1}|\mathbf{x}, \mathbf{\Lambda}, y_t^i)$ can be computed directly using three recursively computed vectors viz. the α, β vectors and a new vector, say γ. γ is defined as the partial unnormalized probability of starting at state i with label y and ending at state j with label y'. Thus, γ can be computed as :

$$\gamma(i, j, y, y') = \begin{cases} \sum_{y''} \gamma(i, j-1, y, y'') e^{\sum_k \lambda_k f_k(y', y'', j-1, \mathbf{x})} & (i < j) \\ [\![y = y']\!] & (i = j) \end{cases} \tag{37}$$

Note that γ can also be computed in a backward fashion. Using α, β and γ, we can obtain $P(y_j, y_{j-1}|\mathbf{x}, \mathbf{\Lambda}, y_t^i)$ as

$$P(y_j, y_{j-1}|\mathbf{x^i}, \mathbf{\Lambda}, y_t^i) = \begin{cases} \frac{\alpha(t, y_t^i)\gamma(t, j-1, y_t^i, y_{j-1}) e^{\sum_k \lambda_k f_k(y_j, y_{j-1}, j, \mathbf{x^i})} \beta(j, y_j)}{Z_{\mathbf{\Lambda}}(\mathbf{x^i})} & (t \leq j-1) \\ \frac{\alpha(j-1, y_{j-1}) e^{\sum_k \lambda_k f_k(y_j, y_{j-1}, j, \mathbf{x^i})} \gamma(j, t, y_j, y_t^i) \beta(t, y_t^i)}{Z_{\mathbf{\Lambda}}(\mathbf{x})} & (t \geq j) \end{cases} \tag{38}$$

However, computing these probabilities for all instances in a training corpus can require anywhere from $O(mn)$ to $O(m^2n^2)$ γ values. For a large or varied corpus, this is prohibitive and an alternate mechanism, as outlined in [KTR02], is used to directly compute $\sum_{t=1}^{t=|\mathbf{x^i}|} P(y_j, y_{j-1}|\mathbf{x^i}, \mathbf{\Lambda}, y_t^i)$.

2.4 Max Margin Method

In this section, we look at an approach to train CRFs in a max-margin sense. Recall that the margin is a measure of a classifier's ability to contain any loss that it incurs while labeling data with a wrong label. A classifier that achieves a larger margin while training is less likely to make errors than one with a smaller margin.

In CRFs, we are dealing with structured classification, so it doesn't make much sense to use a $0-1$ loss function that penalizes all wrong labelings alike. Instead, a Hamming loss function that counts the number of mislabelings is more intuitive. This loss function has the added advantage of being decomposable. Now, let us define the margin criteria as follows:

$$\mathbf{\Lambda}^T(\mathbf{F}(\mathbf{x^i}, \mathbf{y^i}) - \mathbf{F}(\mathbf{x^i}, \mathbf{y})) \geq \gamma L(i, \mathbf{y}) \quad \forall i, \mathbf{y} \neq \mathbf{y^i} \tag{39}$$

Here, γ is the margin that we want to be as high as possible and $L(i, y)$ is the loss incurred when we mislabel $\mathbf{x^i}$ with $\mathbf{y}$. As a shorthand, we will denote the differnce in global feature vector by $\Delta\mathbf{F}_{i,\mathbf{y}}$. Thus, we can write our optimization program as:

$$\max \gamma \text{ s.t. } \mathbf{\Lambda}^T \Delta\mathbf{F}_{i,\mathbf{y}} \geq \gamma L(i, \mathbf{y}) \quad \forall i, \mathbf{y} \neq \mathbf{y^i} \tag{40}$$

or equivalently,

$$\min \frac{\mathbf{\Lambda}^T\mathbf{\Lambda}}{2} \text{ s.t. } \mathbf{\Lambda}^T \Delta\mathbf{F}(i, \mathbf{y}) \geq L(i, \mathbf{y}) \quad \forall i, \mathbf{y} \tag{41}$$

This is similar to the problem formulation in the case of SVMs for separable data. Carrying this analogy forward to inseparable data, the quadratic program (QP) can be written as:

$$\begin{aligned} &\min \frac{\mathbf{\Lambda}^T\mathbf{\Lambda}}{2} + C\sum_{\mathbf{i}} \xi_i \\ &\text{s.t. } \mathbf{\Lambda}^T \Delta\mathbf{F}(i, \mathbf{y}) \geq L(i, \mathbf{y}) - \xi_i \quad \forall i, \mathbf{y} \end{aligned} \tag{42}$$

ξ_i is the slack associated with the i^{th} data instance. The correspond dual is given by:

$$\begin{aligned} &\max \sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} L(i, \mathbf{y}) - \frac{1}{2}|\sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} \Delta\mathbf{F}_{i,\mathbf{y}}|^2 \\ &\text{s.t. } \sum_{\mathbf{y}} \alpha_{i,\mathbf{y}} = C \quad \forall i \\ &\alpha_{i,\mathbf{y}} \geq 0 \quad \forall i, \mathbf{y} \end{aligned} \tag{43}$$

The primal and dual optima are related to each other by:

$$\begin{aligned} \mathbf{\Lambda}^* &= \sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} \Delta\mathbf{F}_{i,\mathbf{y}} \\ &= \sum_{i} \mathbf{F}(\mathbf{x^i}, \mathbf{y^i}) - \sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} \mathbf{F}(\mathbf{x^i}, \mathbf{y}) \end{aligned} \tag{44}$$

Now because $\mathbf{y}$ has a structure, the number of primal constraints (and the dual variables) can be exponentially large. So, we cannot directly apply any optimization techniques to the primal or the dual program. It is here that the decomposability of the loss function and $\Delta\mathbf{F}$ comes to our rescue. Recall that the global feature vector is just a sum over local feature vectors. Note that the first term in the dual objective can be written as:

$$\sum_{\mathbf{y}} \alpha_{i,\mathbf{y}} L(i, \mathbf{y}) = \sum_{\mathbf{y}} \sum_{j} \alpha_{i,\mathbf{y}} L(i, y_j) = \sum_{j, y_j} L(i, y_j) \sum_{\mathbf{y} \sim [y_j]} \alpha_{i,\mathbf{y}} \tag{45}$$

Here, $\mathbf{y} \sim [y_j]$ means all those labelings $\mathbf{y}$ which assign a label y_j to the j^{th} position. Now note that because of the dual program constraints, the α values behave like probabilities (that sum to C instead of 1). So, the quantity $\sum_{y \sim [y_j]} \alpha_{i,\mathbf{y}}$ can be seen as the marginal probability of having the label y_j at the

j^{th} position. We will denote this marginal by $\mu_i(y_j)$. Similarly, the second term in the dual objective can be rewritten because of the decomposability of the global feature vector ($\Delta\mathbf{F}_{i,\mathbf{y}} = \sum_{j,k} \Delta\mathbf{F}_{i,y_j,y_k}$). In this case, we have the pairwise marginals: $\mu_i(y_j, y_k) = \sum_{\mathbf{y} \sim [y_j, y_k]} \alpha_{i,\mathbf{y}}$. The original dual can thus be rewritten as:

$$\max \sum_{i,j,y_j} \mu_i(y_j) L(i, y_j) - \frac{1}{2} \sum_{i,i'} \sum_{\substack{(j,k) \\ y_j, y_k}} \sum_{\substack{(j',k') \\ y_{j'}, y_{k'}}} \mu_i(y_j, y_k) \mu_{i'}(y_{j'}, y_{k'}) \mathbf{f}(\mathbf{x^i}, y_j, y_k)^T \mathbf{f}(\mathbf{x^{i'}}, y_{j'}, y_{k'})$$

$$\text{s.t.} \sum_{y_j} \mu_i(y_j, y_k) = \mu_i(y_k), \quad \sum_{y_j} \mu_i(y_j) = C, \quad \mu_i(y_j, y_k) \geq 0 \tag{46}$$

$\mathbf{f}(.)$ is the local feature vector that arises because of the decomposition of $\mathbf{F}(.)$. Hence, if there were N training instances of length M each and $|\mathcal{Y}|$ possible labels for a particular word, then the original dual with $N|\mathcal{Y}|^M$ variables has been reduced to an equivalent form with just $NM|\mathcal{Y}|^2$ variables. Further, the optimal solution for the primal can be computed from the optimal dual solution via:

$$\mathbf{\Lambda}^* = \sum_{i,(j,k),y_j,y_k} \mu_i(y_j, y_k) \Delta\mathbf{f}(\mathbf{x^i}, y_j, y_k) \tag{47}$$

Looking at Equation 46, it is clear that we can use the standard kernel trick as in SVMs, to compute the dot product of the feature vectors as projected in a very high (possibly infinite) dimensional space. We now briefly discuss two approaches to solve the max-margin formulation.

2.4.1 SMO Algorithm

The SMO algorithm for SVMs considers two α variables at a time, keeping their sum constant, so as to obey the dual constraints. At each iteration, the algorithm optimally redistributes the mass between the two chosen dual variables, keeping the other dual variables fixed. The next pair of dual variables are chosen through a heuristic.

In our case, we cannot afford to materialize an exponential number of dual variables. So, we run a variant of SMO as follows: we choose two μ variables based on some criteria. Then, using these two, we generate two α variables. Due to the many-one dependence between α and μ, there are multiple choices for the α vector. We choose a vector α which is consistent with the μ variables and has the maximum entropy. The SMO algorithm modifies the generated pair of α's and updates the corresponding μ variables.

If we choose to generate $\alpha_{i,\mathbf{y^1}}$ and $\alpha_{i,\mathbf{y^2}}$ and shift a mass ϵ to the first variable, then the effect on an explicit dual variable $\mu_i(y_j, y_k)$ is:

$$\mu_i^{new}(y_j, y_k) = \mu_i^{old}(y_j, y_k) + \epsilon[\![y_j = y_j^1, y_k = y_k^2]\!] - \epsilon[\![y_j = y_j^2, y_k = y_k^2]\!] \tag{48}$$

The optimal value of ϵ can be found in closed form and used to update the μ dual variables. The next pair of variables can be chosen using any heuristic.

2.4.2 Exponentiated Gradient Algorithm

The generic exponentiated gradient algorithm is used to solve QPs with a positive-semidefinite coefficient matrix. It applies positive multiplicative updates to the variables, thus ensuring their non-negativity all the way. Consider the following QP ($\alpha = \{\alpha_{1,\mathbf{y_1}}, \ldots, \alpha_{2,\mathbf{y_1}}, \ldots, \alpha_{n,\mathbf{y_1}}, \ldots\}$):

$$\min J(\alpha) = \frac{1}{2} \alpha^T A \alpha + b^T \alpha$$

$$\text{s.t.} \sum_{\mathbf{y}} \alpha_{i,\mathbf{y}} = 1 \quad \forall i, \quad \alpha_{i,\mathbf{y}} \geq 0 \quad \forall i, \mathbf{y} \tag{49}$$

Algorithm 2 outlines the exponentiated gradient approach to solve this QP.

Note that this is a slightly different formulation from the one we saw earlier. Here, the α_i variables sum upto 1 rather than C. It is easy to outline the one-one correspondence between the formulation and

Algorithm 2 *ExponentiatedGradient*(A, b)

Choose any learning rate $\eta > 0$.
$\alpha^1 \leftarrow$ Any feasible solution
for $1 \leq t \leq T$ **do**
  $\nabla^t \leftarrow A\alpha^t + b$ (gradient)
  $\forall i, \mathbf{y} \;\; \alpha^{t+1}_{i,\mathbf{y}} = \frac{\alpha^t_{i,\mathbf{y}} \exp(-\eta \nabla^t_{i,\mathbf{y}})}{\sum_{\mathbf{y}'} \alpha^t_{i,\mathbf{y}'} \exp(-\eta \nabla^t_{i,\mathbf{y}'})}$
end for
return α^{T+1}

the quantities required by Algorithm 2.

$$\begin{aligned}
J(\alpha) &= -C(\sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} L_{i,\mathbf{y}} - \frac{C}{2} |\sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} \Delta\mathbf{F}_{i,\mathbf{y}}|^2) \\
&= -C(\sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} L_{i,\mathbf{y}} - \frac{C}{2} |\sum_{i} \mathbf{F}(\mathbf{x^i}, \mathbf{y^i}) - \sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} \mathbf{F}(\mathbf{x^i}, \mathbf{y})|^2) \\
\nabla_{i,\mathbf{y}} &= -CL_{i,\mathbf{y}} - C^2 \mathbf{F}(\mathbf{x^i}, \mathbf{y})^T (\mathbf{F}(\mathbf{x^i}, \mathbf{y^i}) - \sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} \mathbf{F}(\mathbf{x^i}, \mathbf{y})) \\
&= -C(L_{i,\mathbf{y}} + \mathbf{\Lambda}^T \mathbf{F}(\mathbf{x^i}, \mathbf{y})) \;\; \text{(See Equation 44)}
\end{aligned}$$

In the last identity we used the fact that $\mathbf{\Lambda}$ is our current estimate of the optima and we absorbed C because the α values have been scaled down.

Now we still have the old problem of facing an exponential number of α variables, and once again, the decomposability of the global feature vector and the loss function saves us. Also, note that because of the exponential updates, it helps if we parameterize $\alpha_{i,\cdot}$ themselves in an exponential form.

$$\alpha_{i,\mathbf{y}} = \frac{\exp(\sum_{r \in R(\mathbf{x^i}, y)} \theta_{i,r})}{\sum_{\mathbf{y}'} \exp(\sum_{r \in R(\mathbf{x^i}, y)} \theta_{i,r})} \tag{50}$$

Here $R(\mathbf{x^i}, \mathbf{y})$ is the set of parts the loss function and the global feature vector decompose over. In the case of linear CRFs, $R(\mathbf{x^i}, \mathbf{y})$ is the set of nodes and edges of the chain whose labelings and local features are consistent with $\mathbf{y}$ and $\mathbf{F}(\mathbf{x^i}, \mathbf{y})$ respectively. The number of θ variables is much less than those of the α variables (the dominant term is governed by the size of the biggest part).

Instead of multiplicatively updating the α variables, we can additively update (a potentially much less number of) θ variables at each iteration of Algorithm 2. The only hitch is computing the gradient, or rather, computing $\mathbf{\Lambda^t} = C(\sum_i \mathbf{F}(\mathbf{x^i}, \mathbf{y^i}) - \sum_{i,\mathbf{y}} \alpha^t_{i,\mathbf{y}} \mathbf{F}(\mathbf{x^i}, \mathbf{y}))$. The second term can be rewritten as $\sum_{i,\mathbf{y}} \alpha_{i,\mathbf{y}} \sum_{r \in R(\mathbf{x^i}, \mathbf{y})} \mathbf{f}(\mathbf{x^i}, r) = \sum_{i, r \in R(\mathbf{x^i}, \cdot)} \mu_{i,r} \mathbf{f}(\mathbf{x^i}, r)$. If we can calculate $\mu_{i,r} = \sum_{i,\mathbf{y}: r \in R(\mathbf{x^i}, \mathbf{y})} \alpha_{i,\mathbf{y}}$ easily, then the gradient can be efficiently computed. For the case of linear CRFs, $\mu_{i,r}$ is the marginal probability of observing a particular label (or label pair) at a node (or an edge), using the current weight vector.

Experimental evidence [BCTM04] shows that the exponentiated gradient algorithm ends up with a better objective and doesn't plateau out as much as the SMO algorithm.

2.5 Gradient Tree Boosting

The potential functions of CRFs belong to the exponential family of functions:

$$\psi(y, y', \mathbf{x}, i) = \exp(\phi(y, y', \mathbf{x}, i))$$

Gradient tree boosting learns $\phi(.)$'s using functional gradient ascent. Functional gradient ascent allows us to see how the objective function behaves as a function of ϕ. We begin with an initial guess of the $\phi()$ functions (and thus, the feature weights). At each step of functional gradient ascent, we add a 'delta' function to the current approximation of $\phi()$.

This 'delta' function has no closed form, instead it is represented using regression trees. At the end of M iterations, the functional approximation of a particular $\phi()$ is given by:

$$\phi_M(y, y', \mathbf{x}, i) = \phi_0(y, y', \mathbf{x}, i) + \Delta_1 + \ldots + \Delta_M \tag{51}$$

A big advantage of this approach is that it allows efficient induction of conjunctive features. In Section 4.2, we will look at a greedy feature induction mechanism. However, functional gradient ascent learns one regression tree per iteration, and thus induces numerous simultaneous features per iteration.

The core issue in gradient tree boosting is estimating the delta function in each iteration. For a fixed training sample, $(\mathbf{x}, \mathbf{y})$, the delta function's value at $(\mathbf{x}, \mathbf{y})$ is the functional gradient of the conditional likelihood of the sample.

$$\Delta_m(\mathbf{x}, \mathbf{y}) = \frac{\partial}{\partial \phi_{m-1}} \log P(\mathbf{y}|\mathbf{x}) \tag{52}$$

Given the value of Δ_m at many such points, we can arrive at a representation of Δ_m by learning a regression tree h_m that minimizes the square error $\sum_i (h_m(\mathbf{x^i}, \mathbf{y^i}) - \Delta_m(\mathbf{x^i}, \mathbf{y^i}))^2$. One way to learn such a regression tree is to use a variant of the CART algorithm. Overfitting can be avoided by stopping the procedure at L leaves, where L is a preset parameter ([Fri01]).

In our scenario, the functional gradient of the conditional likelihood can easily be simplified ([DAB04]):

$$\frac{\partial}{\partial \phi(y, y', \mathbf{x}, i)} \sum_t (\phi(y_t, y_{t-1}, \mathbf{x}, t) - \log Z(\mathbf{x})) \quad = \quad [\![y_{i-1} = y' \ \& \ y_i = y]\!] - P(y_{i-1} = y', y_i = y|\mathbf{x}) \tag{53}$$

where the probability term is equal to $\alpha(i-1, y') \exp(\phi(y, y', \mathbf{x}, i)) \beta(i, y) / Z(\mathbf{x})$. Note that the gradient's value is simply the error in our current estimation of the pairwise marginal.

Computationally, if we have N training samples of size n each, then we generate $N|\mathcal{Y}|^2 n$ samples to learn the delta functions. To scale the algorithm, [Fri01] suggests using sampling and discarding small-valued delta-samples to cut down on the computational costs.

After learning the regression trees for all the ϕ's, at testing time, given a sample $\mathbf{x}$, we can compute its best labeling by running a modified version of the Viterbi algorithm.

2.6 Logarithmic Pooling

Strictly speaking, logarithmic pooling is not an alternate way of training, rather an alternate way to regularize CRFs. The standard way to avoid overfitting in CRFs is to impose a prior on the feature weights (usually $\mathbf{0}_{1\times k}$) , and to penalize any deviation from this prior according to the Euclidean distance. The intuition is that CRFs, like logistic regression, have a tendency to assign arbitrary large feature weights when unregularized. Hence, like SVMs and logistic regression, a penalty term to counter this is included in the objective function.

However, there are two issues while dealing with this kind of regularization. The penalty term is usually of the form $\frac{(\mathbf{\Lambda} - \mathbf{\Lambda_0})^2}{\sigma^2}$, thus forcing the user to select $k+1$ parameters before starting the training. Usually k, the number of features, is very large, and so searching through the hyperparameter space is very difficult, even with cross-validation.

Logarithmic pooling ([SCO05]) tackles this problem by training multiple unregularized CRFs (in the conventional manner) on the training data. At inference time, the predictions of these individual 'experts' are combined using previously learnt weights. The combination is done by taking a weighted geometric mean of the individual distributions:

$$p(\mathbf{y}|\mathbf{x}) = \frac{\prod_i (p_i(\mathbf{y}|\mathbf{x}))^{w_i}}{Z_{LOP}(\mathbf{x})} \tag{54}$$

where $p_i(.)$ is the conditional probability provided by the i^{th} expert and w_i is the expert's weight. As before, $Z_{LOP}(x)$ is the partition function obtained by summing the probabilities to 1.

When $p(.)$ is pooled using the above form, it can be shown that its KL distance from the true distribution $p^*(.)$ is given by :

$$KL(p^*, p) = \sum_i w_i KL(p^*, p_i) - \sum_i w_i KL(p, p_i) \tag{55}$$

Thus, to obtain a small distance between p and p^*, we need individual component distributions that are close to the true distribution but are diverse amongst themselves (and thus to $p(.)$). This is analogous to bagging, where we need diverse classifiers in order for the combination distribution to be good.

Also, note that because the individual experts are combined using a product form, the resultant distribution can be seen as one coming from a single CRF.

$$\begin{aligned} p(\mathbf{y}|\mathbf{x}) &= \frac{\prod_i (p_i(\mathbf{y}|\mathbf{x}))^{w_i}}{Z_{LOP}(\mathbf{x})} \\ &= \frac{\prod_i \exp(w_i {\mathbf{\Lambda_i}}^T \mathbf{F_i}(\mathbf{y}, \mathbf{x}))}{Z_{LOP}(\mathbf{x}) \prod_i Z_{\mathbf{\Lambda_i}}(\mathbf{x})^{w_i}} \\ &= \frac{\exp(\sum_i w_i {\mathbf{\Lambda_i}}^T \mathbf{F_i}(\mathbf{y}, \mathbf{x}))}{Z(\mathbf{x})} \end{aligned} \tag{56}$$

where $Z(\mathbf{x}) = Z_{LOP}(\mathbf{x}) \prod_i Z_i(\mathbf{x})^{w_i} = \sum_{\mathbf{y}} \exp(\sum_i w_i {\mathbf{\Lambda_i}}^T \mathbf{F_i}(\mathbf{y}, \mathbf{x}))$ and $\mathbf{F_i}$ is the global feature vector of the i^{th} expert. Thus, the combined distribution has a feature vector of length M, where M is the number of experts. The i^{th} feature value is ${\mathbf{\Lambda_i}}^T \mathbf{F_i}(\mathbf{y}, \mathbf{x})$. All the useful statistics, like feature value expectations, can be computed using the usual dynamic programming based techniques that are used for normal CRFs. Similarly, a minor variant of Viterbi can be used at inference time. The details for these are omitted, and instead we focus on learning the weight vectors.

2.6.1 Learning component weights

The weights w_i are learnt by maximing the log-likelihood of the training data. Note that before learning these weights, we have already learnt the feature weights for the individual experts. The log-likelihood of the training data is given by :

$$LL(\mathbf{w}) = \sum_j \sum_i w_i {\mathbf{\Lambda_i}}^T \mathbf{F_i}(\mathbf{y}^j, \mathbf{x}^j)) - \sum_j \log Z(\mathbf{x}^j) \tag{57}$$

The i^{th} component of the gradient is :

$$\nabla_i = \sum_j \mathbf{\Lambda}_i^T \mathbf{F}_i(\mathbf{y}^j, \mathbf{x}^j) - E_p[\mathbf{\Lambda}_i^T \mathbf{F}_i(\mathbf{y}^j, \mathbf{x}^j] \tag{58}$$

Again, the gradient can be computed using the dynamic programming techniques mentioned before. The standard gradient based methods such as conjugate gradient or LBFGS can now be used.

One thing to note here is that there is no regularization term in the likelihood function. Experiments with the regularization of the weights using a Dirichlet prior suggest that very little is gained by imposing any prior onto the weights. One reason why overfitting may not occur here inspite of the absence of regularization is that the number of experts is typically low as compared to the number of features in the original CRFs. Therefore the extent of overfitting seen during the training of feature weights may not happen here.

Logarithmically pooled CRFs have been experimentally shown to be comparable or slightly better than their regularized counterparts in NER and POS tagging tasks.

2.6.2 Choice of experts

As in bagging, the experts can be chosen in a variety of ways :

1. By exposing varying random subsets of the feature set available to the trainer.
2. By partitioning the features such that each set deals with either events only behind the current position or only in the present or only in the future.
3. Partitioning the features by label instead of by sequence position.

3 Semi-Markov CRFs

Till now we have been dealing with CRFs that use features that depend only the previous label y_{i-1} (Markovian assumption), the current label y_i, input string $\mathbf{x}$ and the position i. However for typical NER and POS tasks, it often turns out that we wish to simultaneously assign the same label to a contiguous chunk of words. Note that if we use the traditional word-based CRFs, then we cannot encode long-range label dependencies without violating the Markovian assumption. A pragmatic solution is to mark an entire segment of words together with the same label.

For that to happen, the trained model has to decide how long the chunk has to be and what label will it get. As we are still in the realm of first-order Markovian dependence, the label of a segment can depend only on the segment features and the label of the previous segment. Note that the family of segment-level features is much more powerful than that of word-level features. Binary functions such as $[\![3 \leq SegmentLength \leq 5]\!]$ cannot be encoded using word-level features.

Some notation before we proceed further. A segment s_i is denoted by a tuple (t_i, u_i, y_i) where t_i and u_i are the start and end offsets and y_i is the segment's label. A segmentation comprises of consecutively labeled segments and is denoted by $\mathbf{s}$. Thus a feature function evaluation at the j^{th} segment is a function of $(y_j, y_{j-1}, \mathbf{x}, t_j, u_j)$. All partial segmentations that end at position i in the string and label the last segment with y will be denoted by $\mathbf{s}_{i:y}$. Similarly, all partial segmentations that start at position $i+1$, given that the label of the segment ending at i is y will be denoted by $\mathbf{s}^{i:y}$. Since we are segmenting and labeling simultaneously, we will use $\mathbf{s}$ to denote model output rather than $\mathbf{y}$.

From here on we assume that the model rejects any segmentation whose maximum segment size is more than L. The training time and Viterbi run-time will clearly be a function of L.

3.1 Forward and backward vectors

The forward and backward vectors can be naturally extended to work with segments.

$$\alpha(y,i) = \sum_{\mathbf{s}' \in \mathbf{s_{i:y}}} e^{\mathbf{\Lambda}^T \mathbf{F}(\mathbf{s}',\mathbf{x})}$$
$$\beta(y,i) = \sum_{\mathbf{s}' \in \mathbf{s^{i:y}}} e^{\mathbf{\Lambda}^T \mathbf{F}(\mathbf{s}',\mathbf{x})}$$

Note that since $\mathbf{s}'$ is a partial segmentatation, the global feature vector $\mathbf{F}(.)$ is computed only till the i^{th} position for α's and only beyond i for β's. The vectors can be calculated using a small variant of the recursion discussed in Section 1.2.1.

$$\alpha(y,i) = \sum_{d=1}^{\min(L,i)} \sum_{y'} \alpha(y', i-d) e^{\mathbf{\Lambda}^T \mathbf{f}(y,y',\mathbf{x},i-d+1,i)}$$
$$\beta(y,i) = \sum_{d=1}^{\min(L,|\mathbf{x}|-i)} \sum_{y'} e^{\mathbf{\Lambda}^T \mathbf{f}(y',y,\mathbf{x},i+1,i+d)} \beta(y', i+d)$$

The base cases are :

$$\alpha(y,0) = 1$$
$$\beta(y,|\mathbf{x}|+1) = 1$$

As before, the value of the partition function equals $\sum_y \alpha(y,|\mathbf{x}|)$ and the marginal $P((i,j,y)|\mathbf{x})$ is equal to $\sum_{y'} \alpha(i-1,y') \exp(\mathbf{\Lambda}^T \mathbf{f}(y,y',i,j,\mathbf{x}))\beta(j,y)$.

3.2 Inferencing

The Viterbi algorithm can be modified to look at all candidate segments and labels at each step. Denoting the unnormalized probability of the best partial segmentation ending at position i with label y by $V(i,y)$,

the recursion can be written as :

$$V(i,s) = \begin{cases} \max_{y',d=1\cdots L} V(i-d,y').e^{\mathbf{\Lambda}^T \mathbf{f}(y,y',\mathbf{x},i-d+1,i)} & (i \geq 1) \\ 0 & (i=0) \\ -\infty & (i<0) \end{cases} \tag{59}$$

Ofcourse, we can work with $\log V(.)$ instead of $V(.)$, in which case, we can replace the product with sum and do away with the exponentation. The best overall segmentation is the one with score $\max_y V(|\mathbf{x}|, y)$.

From the recursion, it is obvious that at each step we are doing L times more work as compared to word-level CRFs. Since $L \leq |\mathbf{x}|$, we are still performing polynomial time inferencing.

3.3 Training

Using the penalized log-likelihood criteria, the gradient can be written as :

$$\nabla_{\mathbf{\Lambda}} = \sum_j (\mathbf{F}(\mathbf{x}^j, \mathbf{s}^j) - \frac{\sum_{\mathbf{s}'} \mathbf{F}(\mathbf{x}^j, \mathbf{s}^j) e^{\mathbf{\Lambda}^T \mathbf{F}(\mathbf{x}^j, \mathbf{s}^j)}}{Z_{\mathbf{\Lambda}}(\mathbf{x}^j)}) - \frac{\mathbf{\Lambda}}{\sigma^2} \tag{60}$$

The k^{th} component of the second term is the expectation of the k^{th} function's global value. Consider the partial unnormalized expectation $\eta_k(i,y) = \sum_{\mathbf{s}'} f_k(\mathbf{s}', \mathbf{x}) \exp(\mathbf{\Lambda}^T \mathbf{F}(\mathbf{s}', \mathbf{x}))$, where the segmentations $\mathbf{s}'$ are restricted to the first i words. The required expectation is $\sum_y \eta_k(|\mathbf{x}|, y)/Z_{\mathbf{\Lambda}}(\mathbf{x})$. We can recursively compute η_k as :

$$\begin{aligned} \eta_k(i,y) &= \sum_{d,y'} (\eta_k(i-d,y') e^{\mathbf{\Lambda}^T \mathbf{f}(y,y',\mathbf{x},i-d+1,i)} \\ &+ \sum_{\mathbf{s}' \in \mathbf{s}_{(\mathbf{i-d}),\mathbf{y}'}} e^{\mathbf{\Lambda}^T \mathbf{F}(\mathbf{s}',\mathbf{x}) + \mathbf{\Lambda}^T \mathbf{f}(y,y',\mathbf{x},i-d+1,i)} f_k(y,y',\mathbf{x},i-d+1,i)) \\ &= \sum_{d,y'} (\eta_k(i-d,y') + \alpha(i-d,y') f_k(y,y',\mathbf{x},i-d+1,i)) e^{\mathbf{\Lambda}^T \mathbf{f}(y,y',\mathbf{x},i-d+1,i)} \end{aligned}$$

4 Miscellaneous topics on linear CRFs

4.1 Scaling CRFs for medium-sized label sets

In the previous sections we had seen that computing any important statistic like the forward-backward vectors, probability of the best labeling, or any marginal probability requires time proportional to $O(|\mathcal{Y}|^2)$. When the label set is large, this can significantly slow everything down, including training. [CSO05] presents an approach to remove the $|\mathcal{Y}|^2$ term by training multiple CRFs over binary label sets. The label set is represented by a group of possibly overlapping subsets. Let $T_1, \ldots, T_s$ be these subsets. A label y is assigned the code $b_1 \cdots b_s$ where b_i is one if $y \in T_i$. Now s different CRFs are trained to output binary labels. For this, the training data is transformed into s different versions. In the i^{th} version, a label y is replaced by 1 if $y \in T_i$ and 0 otherwise. Note that the runtimes of all these CRFs is independent of $|\mathcal{Y}|^2$.

The value of s is still set through engineering. If s is low then the number of binary CRFs is too low to distinguish between $|\mathcal{Y}|$ labels. If s is too large, then many pairs of binary CRFs will be highly correlated.

Consider the case where all strings are of length one. Then the problem reduces to multi-class classification. In that case, the outputs of all the binary CRFs can be used to construct an s-bit string. The final output is the label whose code has the least Hamming distance to this bit string. If the code of the correct label is atleast a Hamming distance l from all other codes, then this allows upto $l/2$ individual binary CRFs to be wrong.

We can extend this to the sequence labeling scenario in multiple ways:

1. The Viterbi labeling is obtained from each of the CRFs. Each such labeling is a binary string. At each position j of the input string $\mathbf{x}$, we construct an s-bit string from the corresponding j^{th} bits of the Viterbi strings. The j^{th} position is assigned the label whose code is nearest to this bit string. This approach is very efficient but doesn't incorporate the confidence scores of the individual CRFs.

2. At each position j, all the CRFs output the marginal $P(y_j = 1|\mathbf{x})$, $\forall j$. This vector of marginal is compared with all the codes (e.g. using the L_1-metric) and the label with the closest code is outputted for the position j.

3. For each labeling $\mathbf{y}$, each CRF C_i outputs its confidence $P(b_i(\mathbf{y})|\mathbf{x})$, $b_i(\mathbf{y})$ being the bit string conversion of $\mathbf{y}$, specific to C_i. The overall best labeling is the one that maximizes $\prod_i P(b_i(\mathbf{y})|\mathbf{x})$. The maxima can be found using a minor variant of Viterbi. Note that this is a special form of logarithmic pooling (ref. Section 2.6), where all the weights are set to one.

Experimental evidence ([CSO05]) suggests that the last approach gives the lowest test error. However, one open question is to efficiently select a good set of subsets. A greedy approach to select subsets in a forward manner is infeasible because of the exponential size of the problem. [CSO05] discusses a code selection heuristic that picks a code length which minimizes an upper bound on the error of the overall classifier.

4.2 Efficient feature induction

So far we had assumed that the feature set has been fixed apriori for us. However, for many applications, selecting a good subset of features is a highly non-trivial problem. Consider the case of entity extraction or POS tagging, where linear CRFs can be used. Any feature depends on a potential label of a token, the input string and optionally, the labels of the two neighbouring tokens. Typically, the features are boolean which fire only when the current word and the corresponding labels fulfill some criteria, e.g. capitalization, dictionary presence etc. .

More complex features can be constructed using the conjuncts of simpler features. However, the space of even the simple features is very large (e.g. dictionary features) and so, the number of conjuncts is larger than we can handle. One possible way is to use forward feature selection ([McC03]) that greedily learns a good subset of features (atomic as well as conjuncts) for a given training set.

At each step, the feature selection scheme ranks the features by how much they will increase the likelihood of the training data when added to the feature set. This ranking is updated at each step and the algorithm stops when even the top ranked features don't add much to the likelihood.

Such schemes have been used in text classification, where the ranking metric is not the increase in likelihood but the increase in some classification score (e.g. F1). When we attempt at using this scheme in the domain of linear CRFs, the following issues arise:

1. The number of features can be in millions or even more. So we can't afford to select just one feature per iteration. Also, the rank scores have to be computed very efficiently.

2. CRFs learn a weight vector for the feature set. On adding a new feature, a new vector has to be learnt from scratch.

For ranking a feature, we need to compute the increase in likelihood on adding it. Strictly speaking, the two likelihoods (before and after) use entirely different weight vectors, as produced by the training algorithm. But for the sake of efficiency, the weights of the old features are assumed to be fixed and we only optimize over the new feature's weight. The 'gain' score of a new feature g (with associated weight μ) is thus given by:

$$G_{\mathbf{\Lambda}}(g) = \max_{\mu} LL_{\mathbf{\Lambda}+(g,\mu)} - LL_{\mathbf{\Lambda}} - \mu^2/2\sigma^2 \tag{61}$$

In order to make the gain computations tractable, the likelihood is approximated by the pseudo likelihood ([McC03]):

$$\begin{aligned} P_{\mathbf{\Lambda}+(\mathbf{g},\mu)}(\mathbf{y}|\mathbf{x}) &\approx \prod_j P_{\mathbf{\Lambda}+(\mathbf{g},\mu)}(y_j|\mathbf{x}) \\ &= \prod_j \frac{P_{\mathbf{\Lambda}}(y_j|\mathbf{x})\exp(\mu g(y_j,\mathbf{x},j))}{\sum_{\mathbf{y_j'}} P_{\mathbf{\Lambda}}(y_j'|\mathbf{x})\exp(\mu g(y_j',\mathbf{x},j))} \end{aligned}$$

The optimal value of μ can now be computed using any iterative method. The algorithm in [McC03] uses other optimizations to cut down on the computation costs. It calculates the gain only over those tokens which were misclassified by the current weight vector. It also selects multiple features per iteration. Further, after adding new feature(s), while retraining the CRF, it discards the last few iterations of the LBFGS (or whatever method is used) to avoid overfitting.

4.3 Constrained Inferencing

So far we have performed unconstrained inferencing on CRFs. However, in many scenarios, we are interested in computing the best segmentation or labeling that satisfies a given set of constraints. Examples of such constraints are:

- The number of occurences of a particular label y should be atleast (or atmost) k.
- If a particular label y is present then some other label y' should be present (or absent).
- A label y_a should be followed be the label y_b.

4.3.1 Constrained Viterbi

Only some of the constraints can be incorporated into the Viterbi algorithm. Specifically, constraints that deal with adjacent labels can be dealt with by modifying the matrices M_i (ref. Section 1.2). For example, the third constraint can be enforced by using the modified matrix M_i':

$$M_i'(y',y) = \begin{cases} M_i(y',y) & [\![\ y' = y_a\ \&\ y = y_b]\!] \\ 0 & \text{otherwise} \end{cases}$$

In general, this approach can only work with 'local' constraints that affect the labels of adjacent positions. We next look at integer linear programs (ILP) that deal with both local as well as global constraints in a natural way.

4.3.2 Integer linear programming based inferencing

We introduce ILP variables of the kind $z_{i,yy'}$ $(1 \le i \le |\mathbf{x}|,\ y,y' \in \mathcal{Y})$ which is set to 1 if the word at position i is labeled y', with the previous label being y and 0 otherwise. These variables can be seen as the nodes of a graph (called the trellis graph), where there is an edge between every pair of nodes of the form $(z_{i,yy'}, z_{i+1,y'y''})$, with weight $\mathbf{\Lambda}^T\mathbf{f}(y'',y',\mathbf{x},i+1)$.

Computing the best constraint labeling for the string $\mathbf{x}$ is equivalent to finding the maxmimum weight path from the start node to the end node in this trellis.

Any constraints, global or local, can be represented as linear constraints on the $z_{i,yy'}$'s. The unconstrained labeling problem can easily be shown to be equivalent to the following ILP ([RY05]):

$$\begin{aligned} \max_z \sum_{i,y,y'} \mathbf{\Lambda}^T\mathbf{f}(y',y,\mathbf{x},i) z_{i,yy'} \\ \text{s.t.} \sum_y z_{i-1,yy'} = \sum_y^{''} z_{i,y'y''} \quad \forall i,y' \\ \sum_y z_{0,0y} = 1, \quad \sum_y z_{|\mathbf{x}|+1,y0} = 1 \\ z_{i,yy'} \in \{0,1\} \end{aligned}$$

Note that the LP relaxation of this ILP will still have an integral optima because of the total unimodularity of the constraint matrix.

User-defined constraints can be appended to this ILP to yield the best constrained solution. However, the total unimodularity of the matrix can no longer be guaranteed. In that case, either the ILP must be solved optimally or the solution to the LP relaxation be rounded intelligently.

As an example, the linear inequalities corresponding to the constraints discussed before are given by:

1. $\sum_i \sum_{y'} z_{i,y'y} \leq k$.
2. $\sum_{y''} z_{i,y''y} \leq \sum_{j,y''} z_{j,y''y'} \quad \forall i$.
3. $\sum_y z_{i,yy_a} \leq z_{i+1,y_a y_b} \quad \forall i$.

Examples of more complex constraints are given in [RY05]. Non ILP-based techniques like the A* algorithm can also be used to perform constrained inferencing. However, they too suffer from having no bounds on the running time.

5 Exact inference in arbitrary undirected models

Given an undirected model $G = (V, E)$, we are often interested in computing the following quantities:

1. Marginal probability of a subset of nodes, $P(Y_S)$ where $S \subseteq V$.
2. Maximum a-posteriori configuration of the model (MAP configuration), i.e. $\arg\max_{\mathbf{y}} P(Y_V = \mathbf{y})$.
3. Conditional probability $P(Y_S|Y_T)$.

The task of computing conditional probabilities is reduced to that of computing marginals so let us focus only on the first two problems.

Inference algorithms for general graphs fall into three major categories - (a) exact methods, (b) sampling based methods, and (c) variational methods. Here, we look at two of the exact methods viz. Sum Product algorithm and the Junction Tree algorithm.

5.1 Sum Product Algorithm

Let us first look at the case where the given graph is a tree (or a forest). In the later sections, we will see how to adapt the inference techniques to general graphs.

Now, since the graph is a tree, we will work only with potentials over nodes and edges. The Sum-Product algorithm (also called the message passing algorithm) begins by defining an ordering of the vertices by rooting the tree at a node r. Then 'belief' messages are passed from child nodes to their parent nodes. A node v sends messages to its (unique) parent only when v has in turn received messages from all its children. The marginal probability of the root node is proportional to the messages it receives from its children. The message from node u to its parent v is denoted by $m_{uv}(y_v)$. One can think that u sends a message to v for each value of y_v or sends a single message which is a function of y_v. The message is given by:

$$m_{uv}(y_v) = \sum_{y_u} \psi_u(y_u)\psi_{uv}(y_u, y_v) \prod_{w \in Ch(u)} m_{wu}(y_u) \tag{62}$$

where $Ch(u)$ denotes the children nodes of u. On 'unrolling' the messages, it can be seen that the message passed by a node u to its parent v is simply the unnormalized contribution of the subtree rooted at u to the marginal $P(Y_v = y_v)$. Therefore, once the root receives all the messages, its marginal is proportional to their product.

$$\begin{aligned} p(y_r) &\propto \psi_r(y_r) \prod_{u \in Ch(r)} m_{ur}(y_r) \\ &= \frac{\psi_r(y_r) \prod_{u \in Ch(r)} m_{ur}(y_r)}{Z}, \; Z = \sum_{y_r} \psi_r(y_r) \prod_{u \in Ch(r)} m_{ur}(y_r) \end{aligned} \tag{63}$$

Let $|\mathcal{Y}|$ denote the number of possible values of a single label y_u. Then it is obvious that $\theta(|E||\mathcal{Y}|^2)$ work is performed while sending all the messages. Also, the number of iterations taken by the algorithm depends on the diameter of the tree.

The key thing to note here is that the intermediate messages once computed can be stored at their destination nodes and be reused for computing any other marginal. Infact after running the message passing algorithm over the entire tree, we can compute the marginal of any arbitrary node without doing any extra work!

5.1.1 Viterbi Algorithm

The Sum Product algorithm can also be used to compute the MAP configuration (where it is called the Viterbi algorithm). To achieve this, simply replace the sum by the max operator. Whenever node u communicates with node v, instead of passing a message that quantifies its 'consolidated belief' in y_v, the node will pass a message that says whats the best that it can do to be consistent with y_v. The modified message is:

$$m_{uv}(y_v) = \max_{y_u} \psi_u(y_u)\psi_{uv}(y_u, y_v) \prod_{w \in Ch(u)} m_{wu}(y_u) \tag{64}$$

When these messages reach the root r, it knows what it is best possible configuration of other nodes given y_r. Thus, by iterating over y_r, we can find the probability of the most probable global configuration (the choice of root node doesn't matter). Along with the max, we can also store the argmax, which can be used to find the best assignment.

Note that in the case where the graph is a linear chain, the Viterbi algorithm reduces to the algorithm given in Section 1.2.2. Further, note how the forward and backward vectors correspond to the belief messages.

5.2 Junction Tree Algorithm

Let us see why the Sum Product algorithm fails on general graphs. Consider the graph G which is a cycle in four nodes $Y_1, \ldots, Y_4$. 'Rooting' at node 1, the message that the root receives is:

$$p(y_1) \propto \psi_1(y_1)[\sum_{y_2} \psi_{12}(y_1, y_2)\psi_2(y_2) \sum_{y_3} \psi_3(y_3)\psi_{23}(y_2, y_3)][\sum_{y_4} \psi_{14}(y_1, y_4)\psi_4(y_4) \sum_{y_3} \psi_3(y_3)\psi_{43}(y_4, y_3)] \tag{65}$$

while if we use the definition of marginal probability to compute $p(y_1)$, we get:

$$\begin{aligned} p(y_1) &\propto \sum_{y_{2,3,4}} \psi_{12}(y_1, y_2)\psi_{23}(y_2, y_3)\psi_{34}(y_3, y_4)\psi_{41}(y_1, y_4) \prod_i \psi_i(y_i) \\ &= \sum_{y_{2,4}} \psi_1(y_1)\psi_2(y_2)\psi_4(y_4)\psi_{12}(y_1, y_2)\psi_{14}(y_1, y_4)m_3(y_2, y_4) \\ &= \cdots \end{aligned}$$

Clearly the two are not equal. The key problem is caused by the presence of multiple paths. The potential of some nodes may reach their ancestors more than once (e.g. $\psi_3(y_3)$ in the example). Further, due to the lack of acyclicity, there is no stopping criterion for the message passing algorithm. Although, one can go ahead and execute the algorithm and empirically demonstrate acceptable convergence, examples can be given where the algorithm doesn't converge at all.

To tackle this problem, the natural approach is to first make the graph acyclic and then run the Sum-Product algorithm on the transformed graph. The most intuitive step in this direction is to construct a hypertree, whose nodes correspond to vertices of maximal cliques in the original graph. Two nodes in the tree are adjacent if the corresponding cliques share vertices. Clique-wise division is performed because the potentials are defined over cliques (or portions thereof), which helps us in defining the potential of a node of a hypertree. The potential of a clique is the product of potentials defined over its subgraphs. It should be ensured that a given potential is not assigned to two different clique nodes.

Note that two distant nodes V_1 and V_2 in the hypertree can have some common vertices, say T, in the original graph. After we run the message passing algorithm on this tree, it is imperative that the marginals of T be equal, viz. $P(T) = \sum_{V_1 \setminus T} P(V_1) = \sum_{V_2 \setminus T} P(V_2)$. However, the algorithm ensures only local consistency, viz. between adjacent nodes. Therefore, to obtain global consistency from local consistency, we constrain the hypertree to have the *running intersection property*. This property states that if two non-adjacent hypertree nodes V_1 and V_2 share a set of vertices T, then T should be present in all the nodes in the (unique) path from V_1 to V_2. Any hypertree that satisfies this property is called a Junction-Tree.

It can be shown that the family of undirected graphs that possess a Junction-Tree is the family of triangulated graphs (graphs where each cycle of length more than four has a chord). For example, consider a 4-node cycle, with edge potentials specified. It is easily seen that any hypertree of this graph will not satisfy the running intersection property. However, after the introduction of a chord, we have two cliques of size three each. This can be trivially represented as a hypertree. Note that triangulation can increase the size of the maximal clique of the graph. From now on, we can assume that the graph is triangulated apriori.

The message passing protocol on a Junction-Tree works as follows:

$$m_{cc'}(y_{c'}) = \sum_{c} \psi_c(y_c) \prod_{c_1 \in Ch(c)} m_{c_1 c}(y_c) \tag{66}$$

where ψ_c includes all the product of all the potentials associated with the clique c. The Viterbi algorithm for this case is analogous:

$$m_{cc'}(y_{c'}) = \max_{y_c \sim y'_c} \psi_c(y_c) \prod_{c_1 \in Ch(c)} m_{c_1 c}(y_c) \tag{67}$$

where $y_c \sim y'_c$ denotes that y_c and y'_c assign the same values to the common vertices.

Junction-Tree algorithm's running time depends on the size of the biggest clique in the triangulated graph (or the treewidth, which is defined as one less than the size of the biggest clique). The runtime is exponential in the treewidth, and consequently the onus of doing efficient inferencing lies on the triangulation performed. Unfortunately, it is NP-hard to find a triangulation that achieves minimum treewidth. Hence heuristics are used to come up with good enough triangulations.

5.3 Linear Programming based inferencing

Computation of the MAP labeling can also be done using Integer Linear Programming. Let $\mathcal{C}$ denote the set of maximal cliques and $c \in \mathcal{C}$ be any maximal clique. Then, the binary variable $\mu_c(\mathbf{y}_c)$ denotes whether the clique vertices of c have the labeling $\mathbf{y}_c$ or not. For a valid global assignment $\mathbf{y}$, we need the μ variables to be consistent. Moreover, for a fixed clique, only one of its μ variables can be 1. Thus, the ILP for MAP labeling can be written as:

$$\begin{aligned}
& \max \sum_{c, \mathbf{y'}_c} \mu_c(\mathbf{y}_c) \log \psi_c(\mathbf{y}_c) \\
\forall c, \mathbf{y}_c : \quad & \mu_c(\mathbf{y}_c) \in \{0,1\}, \ \sum_{\mathbf{y}_c} \mu_c(\mathbf{y}_c) = 1 \\
\forall c, \mathbf{y}_c, c' \supset c : \quad & \sum_{\mathbf{y'}_{c'} : \mathbf{y'}_{c'} \sim \mathbf{y}_c} \mu_{c'}(\mathbf{y'}_{c'}) = \mu_c(\mathbf{y}_c)
\end{aligned} \tag{68}$$

The LP relaxation of this program is obtained by letting the μ's to vary between 0 and 1. Looking at the type of constraints, it is clear that the μ variables behave like marginal probabilities over the clique vertices. Assume that the graph was triangulated and the original ILP had a unique solution. In that case, relaxing the ILP doesn't change the optimal solution, viz. the marginals have all their 'probabilistic' mass located at either 1 or 0. One possible way to prove this would be to demonstrate that the constraint matrix is totally unimodular (determinant of every square submatrix is 0 or ± 1). In case of multiple MAP labelings, any convex combination of the optimal labelings would also be optimal for the LP relaxation.

Since c ranges over elements of $\mathcal{C}$, one can immediately see the parallels between this formulation and the Junction-Tree algorithm. For example, the consistency constraint is equivalent to the running intersection property. However, while the ILP is valid for any arbitrary graph, the LP relaxation admits invalid marginals as feasible solutions in the case of untriangulated graphs. This is analogous to the absence of any Junction-Tree for untriangulated graphs. Hence, to deal with such cases, we have to triangulate the graph, which translates to adding more μ variables (over potentially bigger cliques) and more consistency constraints. Again, the number of extra variables (and constraints) created by this process depends on the tree-width and solving the LP becomes more and more intractable with increasing tree-width.

6 Associative Markov Networks

Recently, [TCK04] looked a restricted class of graphical models, called Associative Markov Networks (AMN). In such models, the potential function over a clique favours common labeling of all the vertices in that clique. Approximate inference in such models can be done very efficiently ([TCK04]), and polynomial time exact inferencing is possible for the case of two labels. Note that such potentials can arise in real-life tasks like classifying hyperlinked documents. In this case, linked documents would tend to have same topic labels and the corresponding potential would be high. AMNs can be trained using the standard max-margin formulation, so we focus on inferencing here. Consider potential functions of the form:

$$\psi_c(\mathbf{y}_c) = \begin{cases} \eta_k^c & (\mathbf{y}_c = [k\ k\ \ldots\ k\ k]) \\ 1 & \text{otherwise} \end{cases} \tag{69}$$

where $\eta_k^c \geq 1$. For AMNs, we can replace the generic inference variable $\mu_c(\mathbf{y}_c)$ by $\mu_c(k)$, which is true only if all the vertices in the clique c are labeled k. Thus, the global potential of a labeling $\mathbf{y}$ is given by:

$$\psi(\mathbf{y}) = \prod_{v,i} \psi_v(i)^{\mu_v(i)} \prod_{c,k} \psi_c(k)^{\mu_c(k)} \tag{70}$$

Also, note that if $u, v \in c$, then $\mu_c(k) \leq \mu_u(k) \wedge \mu_v(k)$. Thus, for an AMN, the MAP labeling can be computed using the following ILP:

$$\begin{aligned} & \max \sum_{v \in V,i} \mu_v(i) \log \psi_v(i) + \sum_{c \in \mathcal{C} \setminus V,k} \mu_c(k) \log \psi_v(k) \\ \forall v, c, k: \quad & \mu_v(k), \mu_c(k) \in \{0,1\}, \quad \forall v : \sum_i \mu_v(i) = 1 \\ \forall c, v \in c, k: \quad & \mu_c(k) \leq \mu_v(k) \end{aligned} \tag{71}$$

It can be shown that if the number of labels ($= m$) is 2, then the ILP can be relaxed and the optima would still lie at an integral point. Infact, the LP relaxation for $m = 2$ can be reduced to a graph min-cut which can be solved very efficiently using combinatorial methods. If $m > 2$, then the relaxation has an optimum whose value is atleast $\frac{1}{|c_{max}|}$ times the ILP optimum, $|c_{max}|$ being the size of the biggest clique in $\mathcal{C}$.

For $m > 2$, we can perform an iterative min-cut. The algorithm begins with any arbitrary labeling of the vertices. At step i, an 'i-expansion' step is performed, where we do a min-cut on the vertex set. The first partition of the min-cut preserves its current label while the vertices in the second partition switch to label i. The min-cut is performed so that we achieve a maximum improvement in the objective. Repeated min-cuts are performed by varying i, until we can no longer improve the objective. Experimental evidence in [TCK04] shows that only a few iterations are sufficient for convergence in practice.

7 Collective Classification

Collective classification is the task of jointly assigning labels to a set of correlated entities. One example is assigning topic labels to a set of hyperlinked webpages ([TAK02]) . The labels of linked webpages

are obviously correlated. Another example is information extraction while exploting the fact that the individual extractions are correlated ([BM04]).

To achieve this, [TAK02] defines the notion of a *clique template*. A clique template chooses a set of entities according to a user defined criteria. A clique between these entites is introduced in the Markov field. For example, in the webpage classification task, we may introduce a clique between every pair of documents that have a hyperlink between them, based on the belief that labels of linked pages are similar. Similarly, we can introduce a clique between every pair of documents that are pointed to by a common page. For the information extraction scenario, we may be interested in introducing a clique between every pair of words (or phrases) which have a high textual similarity ([BM04]), mirroring our belief that similar words should get similar labels..

Training these models through likelihood maximization leads to the same gradient as in Equation 25, where the global feature vector $\mathbf{F}(.)$ is given by:

$$\mathbf{F}(\mathbf{x}, \mathbf{y}) = \sum_{C} \mathbf{f}(\mathbf{x}_C, \mathbf{y}_C, C) \tag{72}$$

Again, the tricky part is to compute the expectation of $\mathbf{F}()$ based on the current weight vector. In the case of sequential models, we computed it using the forward-backward vectors. For acyclic models, we can use the Sum-Product algorithm to compute marginal probabilities exactly. However, in the presence of cycles (which is usually the case in such applications), the exact probabilities can be computed using the Junction-Tree algorithm (ref. Section 5.2). The Junction-Tree algorithm provides us with marginals of each clique which can be used to obtain the expectaton.

However, if the cliques are large, then the Junction-Tree algorithm is very expensive to use. In this scenario, we can directly run a slightly modified version of the Sum-Product algorithm on the original model. This variant is called loopy belief propagation ([Pea88]). Loopy belief propagation runs the Sum-Product algorithm by initializing all messages to 1 and repeatedly passing the messages through the cycles in the graph. Although there is no theoretical guarantee on the convergence of this algorithm, it has been found to provide good approximations to the true marginals in experiments.

Both [TAK02] and [BM04] report a significant accuracy improvement by using undirected models as against labeling the data using techniques like SVMs and logistic regression that don't exploit label correlation.

8 Other work

In this section, we briefly discuss the work that could not be covered in this report. From a training point of view, there are atleast two other approaches. [QSM05] presents a method for Bayesian estimation of the model parameters. The posteriors are approximated using a variant of the Expectation-Propagation ([Min01]) algorithm. Another method ([SM05]) divides the training task into multiple pieces of the graph. Each piece is trained separately, using its own partition function. At testing time, the learned weights are combined into a global weight vector.

Inference is a key problem in the usage of graphical models. Some of the training methods, like the cutting plane algorithm for the max-margin formulation, rely on efficient inference algorithms for their feasibility. However, for arbitrarily large tree-width graphs, inference is NP-hard. So, fast approximate inferencing algorithms are of paramount importance. Broadly, the set of inexact algorithms can be divided into three major categories.

The first family comprises of sampling based algorithms, like Monte Carlo Markov Chains, importance sampling, or Gibbs Sampling. Gibbs sampling picks a random vertex at each iteration, and sets its conditional probabilities, given its neighbours.

The second approach, called the variational approach, transforms the inference problem into an optimization problem. The optimization is done using an objective like KL-distance, and it is done over a set of simplified probability distributions. This gives a bound on the desired probability. Another transform

must be used to obtain the matching bound. The quality of bounds depends on the transformation and the simplification enforced in the optimization.

The last set of approaches comes from the theory community. One approach ([NB04]) deals with computing a subgraph of the given model, that is optimal in terms of KL-divergence and belongs to a fixed family of graphs, e.g. subgraphs with d fewer edges than the original graph. In [KT99], a MAP inference algorithm is presented for the case when the potentials are pairwise and metrics. The algorithm has an approximation ratio of $O(\log k \log\log k)$, where k is the number of labels. The ratio further drops to 2 if all the potentials are uniform.

References

[BCTM04] P. L. Bartlett, M. Collins, B. Taskar, and D. McAllester. Exponentiated gradient algorithms for large-margin structured classification. In *NIPs*, 2004. http://www.stat.berkeley.edu/~bartlett/papers/bcmt-lmmsc-04.ps.gz.

[Ber] A. Berger. A brief maxent tutorial. http://www.cs.cmu.edu/afs/cs/user/aberger/www/html/tutorial/tutorial.

[BM04] R. Bunescu and R. Mooney. Collective information extraction with relational markov networks. In *ACL*, 2004. http://www.cs.stanford.edu/~btaskar/papers/bcmt-lmmsc-04.ps.gz.

[Cli90] P. Clifford. Markov random fields in statistics. In *G.R. Grimmett and D.J.A. Welsh (Eds) Disorder in Physical Systems, J.M. Hammersley Festschrift*, pages 19–32. Oxford University Press, 1990. http://www.statslab.cam.ac.uk/~grg/books/hammfest/3-pdc.ps.

[Col02] M. Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *EMNLP*, 2002. http://citeseer.ist.psu.edu/collins02discriminative.html.

[CSO05] T. Cohn, A. Smith, and M. Osborne. Scaling conditional random fields using error correcting codes. In *ACL*, 2005. http://www.cs.mu.oz.au/~tacohn/acl05_scaling.pdf.

[DAB04] T. Dietterich, A. Ashenfelter, and Y. Bulatov. Training conditional random fields via gradient tree boosting. In *ICML*, 2004. http://citeseer.ist.psu.edu/dietterich04training.html.

[Fri01] J. H. Friedman. Greedy function approximation: A gradient boosting machine. In *Annals of Statistics*, volume 29, 2001. http://www-stat.stanford.edu/~jhf/ftp/trebst.ps.

[KT99] J. Kleinberg and E. Tardos. Approximation algorithms for classification problems with pairwise relationships: Metric labeling and markov random fields. In *FOCS*, 1999. http://www.cs.cornell.edu/home/kleinber/focs99-mrf.ps.

[KTR02] S. Kakade, Y. Teh, and S. Roweis. An alternate objective function for markovian fields. In *ICML*, pages 275–282, 2002. http://www.cs.berkeley.edu/~ywteh/research/newcost/icml2002.pdf.

[LMP01] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, pages 282–289, 2001. http://citeseer.ist.psu.edu/lafferty01conditional.html.

[McC03] A. McCallum. Efficiently inducing features of conditional random fields. In *UAI*, 2003. http://citeseer.ist.psu.edu/mccallum03efficiently.html.

[Min01] T. Minka. Expectation propagation for approximate bayesian inference. In *UAI*, 2001. http://research.microsoft.com/~minka/papers/ep/minka-ep-uai.pdf.

[NB04] M. Narasimhan and J. Bilmes. Optimal sub-graphical models. In *NIPS*, 2004. http://ssli.ee.washington.edu/~mukundn/pubs/nips2004.pdf.

[Pea88] J. Pearl. Probabilistic reasoning in intelligent systems. Morgan Kaufmann, San Francisco, 1988.

[QSM05] Y. Qi, M. Szummer, and T. P. Minka. Bayesian conditional random fields. In *AISTATS*, 2005. http://people.csail.mit.edu/alanqi/papers/Qi-Bayesian-CRF-AIstat05.pdf.

[RY05] D. Roth and W. Yih. Integer linear programming inference for conditional random fields. In *ICML*, pages 737–744, 2005. http://l2r.cs.uiuc.edu/~danr/Papers/RothYi05.pdf.

[SCO05] A. Smith, T. Cohn, and M. Osborne. Logarithmic opinion pools for conditional random fields. In *ACL*, 2005. http://www.iccs.informatics.ed.ac.uk/~osborne/papers/acl05a.pdf.

[SM05] C. Sutton and A. McCallum. Piecewise training for undirected models. In *UAI*, 2005. http://www.cs.umass.edu/~mccallum/papers/piecewise-uai05.pdf.

[SP03] F. Sha and F. Pereira. Shallow parsing with conditional random fields. In *NAACL*, 2003. http://www.cis.upenn.edu/~feisha/pubs/shallow03.pdf.

[TAK02] B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *UAI*, 2002. http://www.cs.berkeley.edu/~taskar/pubs/rmn.ps.

[Tas04] B. Taskar. *Learning Structured Prediction Models: A Large Margin Approach*. PhD thesis, Stanford University, 2004. http://www.cs.berkeley.edu/~taskar/pubs/thesis.pdf.

[TCK04] B. Taskar, V. Chatalbashev, and D. Koller. Learning associative markov networks. In *ICML*, 2004. http://www.cs.berkeley.edu/~taskar/pubs/mmamn.ps.

[Wal02] H. Wallach. Efficient training of conditional random fields. Master's thesis, University of Edinburgh, 2002. http://citeseer.ist.psu.edu/wallach02efficient.html.