Tutorial: Conditional Markov random fields (CRFs) in Matlab

Stephen Gould
sgould@stanford.edu

February 1, 2009

1 Overview

The **STAIR Vision Library** provides a Matlab interface for learning and running inference on medium-sized[1] Markov random fields (MRFs) and conditional Markov random fields (CRFs) over discrete variables. The models are assumed to be log-linear, i.e.,models of the form

$$\mathrm{P}\left(\mathcal{Y} \mid \mathcal{X}; \Theta\right) = \frac{1}{Z(\mathcal{X})} \prod_{m=1}^{M} \Psi_m(\mathbf{Y}_m, \mathbf{X}_m; \theta) = \frac{1}{Z(\mathcal{X})} \exp\left\{ \sum_{m=1}^{M} \theta(\mathbf{y}_m)^T \mathbf{x}_m(\mathbf{y}_m) \right\} \tag{1}$$

where the $\mathbf{Y}_m \subseteq \mathcal{Y}$ are cliques over variables and $\mathbf{X}_m$ is the corresponding subsets of features for the m-th clique, and where $\theta(\mathbf{y}_m)$ and $\mathbf{x}(\mathbf{y}_m)$ indicate the subset of parameters and features contributing to the factor when $\mathbf{Y}_m = \mathbf{y}_m$.

Note that in many cases the structure of the Ψ_m will repeat, e.g.,pairwise CRFs. In addition, each instance of a problem may have a different structure (number of variables, etc), but share the same types of factors and model parameters. Therefore we will make use of factor templates for describing how features and parameters map to the various factor entries. Thus we have $\Psi_m = \Psi_{t(m)}$ where $t(m) \in \{0, \ldots, T-1\}$ indexes a pre-specified template.

We separate a CRF into a *model* and one or more *instances*. The model defines the CRF weights and templates for each clique potential. For example, a dense stereo algorithm can be defined by two potential templates; the first template corresponds to the data term (and is defined over single variables) and the second template corresponds to the smoothness term (and is defined over pairs of adjacent variables). A particular instantiation of the CRF model (e.g., a given pair of left and right stereo images) is then defined by the instance.

Formally, we specify the model by

$$\mathcal{M} = \{\Theta, \Psi_t : t = 0, \ldots, T1\} \tag{2}$$

and an instance by

$$\mathcal{I} = \{y_n, k_n, \mathbf{Y}_m, \mathbf{x}_m, t_m : n = 0, \ldots, N-1; m = 0, \ldots, M-1\} \tag{3}$$

where we have an assignment y_n and cardinality k_n for each variable and a set of variables $\mathbf{Y}_m$, feature vector $\mathbf{x}_m$, and template index t_m for each clique.

These concepts will become more concrete in the examples below.

[1] Here "medium-sized" relates to the size of the largest clique (potential). Typically models with a few hundred variables of cardinality between 2 and 50, and with cliques of no more than 3 or 4 variables can be handled efficiently.

1.1 Compilation

In order to use the Matlab interface you need to compile the **SVL** library as well as Matlab mex applications. You should also have setup the Matlab compiler by calling `mex -setup` from the Matlab command prompt.

```
cd $CODEBASE
make
make mex
```

This will compile `mexCRFLearn` and `mexCRFInfer` and place them in the bin directory. Alternatively, you can add `BUILD_MEX_APPS = 1` to your `make.local` file (see the installation instructions for more details). Running make from `$CODEBASE` will then automatically build the mex functions.

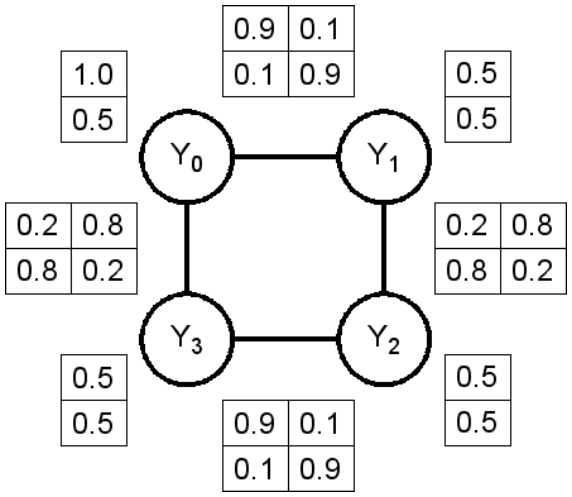
1.2 Starting Matlab

You can automatically add the path to the **SVL** mex functions (as well as all the project directories) to the Matlab search path by calling `addSVLPaths.m` from the Matlab command prompt:

```
run('$CODEBASE/svl/scripts/addSVLPaths');
```

2 Example 1: MRF Inference

Consider the example of running inference on a very simple MRF comprising of four binary variables arranged in a loop. We have a singleton potential on each variable as well as pairwise potentials on each pair of adjacent variables. Suppose that our singleton and pairwise potentials are as shown below (in log-space):



Now recall that we need to split the MRF into a model and an instance. While this may seem overkill for such a simple example, it becomes extremely useful when models get large. Notice that, although we have defined eight potentials (four singleton and four pairwise), there are only four different templates. We specify these templates as follows:

```
% initialize model weights
model.weights = [];

% construct first template
model.weights = [model.weights, log([1.0, 0.5])];
model.templates(1).cards = [2];                    % potential over a single binary var
model.templates(1).entries(1).wi = [0];            % model.weights(1)
```

```
model.templates(1).entries(2).wi = [1];                 % model.weights(2)
[model.templates(1).entries(1:2).xi] = deal(-1);        % no features

% construct second template
model.weights = [model.weights, log([0.5, 0.5])];
model.templates(2).cards = [2];                         % potential over a single binary var
model.templates(2).entries(1).wi = [2];                 % model.weights(3)
model.templates(2).entries(2).wi = [3];                 % model.weights(4)
[model.templates(2).entries(1:2).xi] = deal(-1);        % no features

% construct third template
model.weights = [model.weights, log([0.9, 0.1])];
model.templates(3).cards = [2, 2];                      % potential over a two binary vars
model.templates(3).entries(1).wi = [4];                 % model.weights(5)
model.templates(3).entries(2).wi = [5];                 % model.weights(6)
model.templates(3).entries(3).wi = [5];                 % model.weights(6)
model.templates(3).entries(4).wi = [4];                 % model.weights(5)
[model.templates(3).entries(1:4).xi] = deal(-1);        % no features

% construct third template
model.weights = [model.weights, log([0.2, 0.8])];
model.templates(4).cards = [2, 2];                      % potential over a two binary vars
model.templates(4).entries(1).wi = [6];                 % model.weights(6)
model.templates(4).entries(2).wi = [7];                 % model.weights(7)
model.templates(4).entries(3).wi = [7];                 % model.weights(7)
model.templates(4).entries(4).wi = [6];                 % model.weights(6)
[model.templates(4).entries(1:4).xi] = deal(-1);        % no features
```

Constructing the instance is now a simple matter of specifying the variables and cliques:

```
instance.values = [];                                   % variables are unknown
instance.cards = [2, 2, 2, 2];                          % instance has four binary variables

instance.cliques(1).Cm = [0];                           % singleton clique over Y[0]
instance.cliques(1).Tm = 0;                             % use first template for potential

for i = 2:4,
    instance.cliques(i).Cm = [i - 1];                   % singleton clique over Y[i - 1]
    instance.cliques(i).Tm = 1;                         % use second template for potential
end;

instance.cliques(5).Cm = [0, 1];                        % pairwise clique over Y[0] and Y[1]
instance.cliques(5).Tm = 2;                             % use third template for potential

instance.cliques(6).Cm = [1, 2];                        % pairwise clique over Y[1] and Y[2]
instance.cliques(6).Tm = 3;                             % use forth template for potential

instance.cliques(7).Cm = [2, 3];                        % pairwise clique over Y[2] and Y[3]
instance.cliques(7).Tm = 2;                             % use third template for potential

instance.cliques(8).Cm = [3, 0];                        % pairwise clique over Y[3] and Y[0]
instance.cliques(8).Tm = 3;                             % use forth template for potential

[instance.cliques(:).Xm] = deal([]);                    % no features for any clique
```

Running inference is now a simply:

```
options = struct('verbose', 1, 'maxIterations', 100);
output = mexCRFInfer(model, instance, options);
disp([output.values]);
```

The code returns assignment $Y = [0, 0, 1, 1]$.

You can turn verbose output off by setting `verbose` to 0. The library uses sum-product loopy belief propagation. You can change the maximum number of iterations by setting the `maxIterations` option.

3 Example 1: CRF Inference

In this example we'll consider very simple (and efficient) chain CRFs. Note, however, that the **SVL** supports arbitrary graph topologies.

TODO

4 Example 3: CRF Learning

TODO

5 References

TODO