MaryCrf a Conditional Random Field in Python

Martin Weber

Johannes Kepler University Linz
napster2202@gmail.com

Abstract. MaryCrf is an easy to use library for linear-chain Conditional Random Fields with freely definable feature functions. The math and algorithms will be derived and explained in the first part. The second part shows how to use and adapt MaryCrf for your task.

1 Introduction

Conditional Random Fields (CRF) are undirected graphical models. Like the name suggests they model a probability distribution over a hidden state space under the condition of an observation. A special subclass are linear-chain conditional random fields, which are used for sequence classification. For a deeper introduction to CRFs in general have a look at [1] or [2].

1.1 From HMM to CRF

In contrast to Hidden Markov Models (HMM) a CRF has access to the whole observation sequence $\boldsymbol{x}$ for reasoning about any timestep in the hiden state sequence $\boldsymbol{y}$ as illustrated in figure 1. Because of that CRFs can not perform online reasoning tasks.

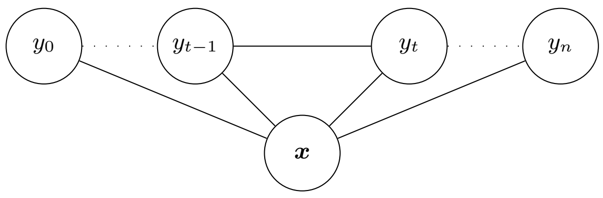

Fig. 1. Graphical structure of a linear-chain CRF

A second difference, where the CRF gives more freedom, besides the visibility of observations, is the style of feature functions. A HMM is restricted to two kinds of functions, the state transition function and the state observation function. Given that the model is able to link an observation to a state belief, but not to a state transition. A CRF does not distinguish feature function types. A feature function can generate a scalar output based on their input parameters. How these parameters affect the output is up to the function itself and can vary across functions even in the same CRF.

1.2 Definition

Conditional Random Fields are discriminative models and therefor model the conditional distribution $\mathcal{P}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x})$. Hidden Markov Models in contrast are generative models and therefor model the joint distribution $\mathcal{P}_{\boldsymbol{\lambda}}(\boldsymbol{y}, \boldsymbol{x})$. A Conditional Random Field is described as

$$\mathcal{P}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \prod_{t=1}^{T} \Psi_{\boldsymbol{\lambda}}(y_{t-1}, y_t, \boldsymbol{x}, t) \tag{1}$$

together with the normalization term

$$Z_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \sum_{\boldsymbol{s} \in S^{T+1}} \prod_{t=1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t-1}, s_t, \boldsymbol{x}, t). \tag{2}$$

$\Psi_{\boldsymbol{\lambda}}$ is a short hand for the exponential sum of feature functions evaluated at a possible transition from s_{from} to s_{to} at t with observation $\boldsymbol{x}$ with a weight vector $\boldsymbol{\lambda}$. Which can be expressed by

$$\Psi_{\boldsymbol{\lambda}}(s_{from}, s_{to}, \boldsymbol{x}, t) = exp(\sum_{k \in K} \lambda_k f_k(s_{from}, s_{to}, \boldsymbol{x}, t)). \tag{3}$$

The sorted set of features functions ($\boldsymbol{f}$) together with the corresponding weights ($\boldsymbol{\lambda}$) and the initial state distribution ($\boldsymbol{\pi}$) define the model. λ_k is the weight which scales the output of f_k. A feature function takes 4 arguments: from-state (s_{from}), to-state (s_{to}), the whole observation sequence ($\boldsymbol{x}$) and the time-index (t).

1.3 Comparison to HMMs

CRFs are less restricted about what they learn than HMMs because of the more flexible feature functions. To illustrate this we use an accumulating adder as an example for a hidden function we want to learn from training data. The input for the adder is a sequence of values that are added to the value the adder is currently holding. The values the adder holds over time represent the state sequence and the inputs represent the observation sequence. For simplicity the adder is bounded between 0 and 3 and the observation sequence is composed of $x_t \in \{-1, 0, 1\}$. Observing a 1 when being in state 3 will let the adder stay in it's previous state. The same holds for the input -1 when being in state 0. A HMM will model that there are only transitions from a state to itself and to the next lower or higher number. The state transition function therefore represents a meaningful property of the adder. In contrast the state observation will learn which state is the most likely that the adder is in when observing $-1, 0$ or 1. When considering the case of the adder this is not meaningful.

The freely definable feature functions of a CRF allow us to link state changes with observations. A CRF with the appropriate feature functions can model the pattern of a state change on a -1 or 1 in the observation sequence and no state change on a 0. Feature functions can be designed to indicate a high probability for a specific state change or just a state change in general.

A more complex HMM would be possible but with the cost of a larger state space.

The additional flexibility of a CRF also comes with some downsides. Because the CRF feature functions can work with the whole observation sequence it is not possible to apply the counting of transitions and state-observation-pairs training algorithm known from HMMs. The optimal model is found by minimizing a cost function. Completely unsupervised training methods like the Baum-Welch algorithm for HMMs do not exist for CRFs. This is because a CRFs model the conditional, not like HMMs the joint distribution.

2 Algorithms

The mathematical derivation of the algorithms is mainly inspired by [3], nevertheless the following derivation uses a different notation. Also it contains more intermediate steps when exploiting properties of the linear chain to get the naive formula's exponential complexity down to linear complexity.

2.1 Notation

- $\boldsymbol{x}$ is an observation sequence and $\boldsymbol{y}$ the corresponding state sequence. The observation sequence has length T and is labeled $\boldsymbol{x} = [x_1, \cdots, x_T]$, whereas the state sequence has length $T+1$ and is labeled $\boldsymbol{y} = [y_0, \cdots, y_T]$.
- $\boldsymbol{\alpha}$ is the Forward message, $\boldsymbol{\beta}$ is the Backward message, $\boldsymbol{\gamma}$ is the Forward-Backward message, $\boldsymbol{\delta}$ is the Viterbi message. All of those sequences have length $T+1$ and are indexed from 0 on.
- $\boldsymbol{bp}$ is the Viterbi path (also called back-pointer) of length T and like the state sequence indexed from 1 on.
- $\boldsymbol{s}, \boldsymbol{y}$ are both sequences of states where $s_t, y_t \in S$. Although we use $\boldsymbol{s}$ when sampling and $\boldsymbol{y}$ when referring to a state sequence of a training sample.

2.2 Inference

Like for HMMs there exist three inference algorithms: Forward, Backward, Viterbi. All three algorithms exploit the linear chain of the model. As seen in from equation 2 the normalization is a sum over all possible state sequences of certain length in the state space. This would result in exponential complexity for the likelihood computation if we would not consider the linear chain property. The Forward algorithm calculates $\mathcal{P}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x})$ starting at the front of the observation, the Backward algorithm starts from the back. Both algorithms update their belief state while iterating over the sequence. The sequence of most likely states in either direction can be inferred, when picking the state with maximum probability at each time frame in the belief state of the Forward respectively Backward algorithm. These sequences are not to be confused with the most likely state sequence, in fact the sequences can even contain transitions prohibited (assigned zero probability) by the model. The most likely state sequence can be computed with the Back pointer path of the viterbi algorithm.

Despite some changes in notation and the omission of the convenient matrix multiplications known from the algorithms applied to an HMMs, the algorithms follow the same principles.

The Forward algorithm

As seen in section 1.2 the formula for the normalization is a sum over all possible state sequences of certain length in the state space. This would result in exponential complexity for the likelihood computation if we would not consider the linear chain property. The Forward algorithm does that by applying the Forward recursion.

We start from the equation 2 and decompose the linear chain in order we get

$$Z_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \sum_{s_T \in S} \underbrace{\sum_{\boldsymbol{s} \in S^{T-1}} \left(\prod_{t=1}^{T-1} \Psi_{\boldsymbol{\lambda}}(s_{t-1}, s_t, \boldsymbol{x}, t) \right)}_{\alpha_{T-1}(s_{T-1}|\boldsymbol{x})} \Psi_{\boldsymbol{\lambda}}(s_{T-1}, s_T, \boldsymbol{x}, T). \tag{4}$$

From there on we already see the recursive structure which can be expressed as

$$\alpha_t(s_{to} \mid \boldsymbol{x}) = \sum_{s_{from} \in S} \Psi_{\boldsymbol{\lambda}}(s_{from}, s_{to}, \boldsymbol{x}, t) \alpha_{t-1}(s_{from} \mid \boldsymbol{x}). \tag{5}$$

Now we can plug recursion into the formula of the normalization term and get

$$Z_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \sum_{s \in S} \alpha_T(s \mid \boldsymbol{x}). \tag{6}$$

The Backward algorithm

The Backward algorithm exploits the linear chain property starting from the end of the sequence leaving the recursion

$$\beta_t(s_{from} \mid \boldsymbol{x}) = \sum_{s_{to} \in S} \Psi_{\boldsymbol{\lambda}}(s_{from}, s_{to}, \boldsymbol{x}, t+1) \beta_{t+1}(s_{to} \mid \boldsymbol{x}), \tag{7}$$

and the normalization

$$Z_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \sum_{s \in S} \beta_0(s \mid \boldsymbol{x}). \tag{8}$$

Combining Forward and Backward algorithm

The Forward and Backward algorithm can be combined via

$$\gamma_t(s \mid \boldsymbol{x}) = \alpha_t(s \mid \boldsymbol{x}) * \beta_t(s \mid \boldsymbol{x}), \tag{9}$$

the hadamard product of their message timeseries'.

The Viterbi algorithm

The Viterbi recursion

$$\delta_t(s_{to} \mid \boldsymbol{x}) = max_{s_{from} \in S}(\Psi_{\boldsymbol{\lambda}}(s_{from}, s_{to}, \boldsymbol{x}, t) * \delta_{t-1}(s_{from} \mid \boldsymbol{x})) \tag{10}$$

passes on the probability of the highest probable state to the next time-slice and

$$bp_t(s_{to} \mid \boldsymbol{x}) = argmax_{s_{from} \in S}(\Psi_{\boldsymbol{\lambda}}(s_{from}, s_{to}, \boldsymbol{x}, t) * \delta_{t-1}(s_{from} \mid \boldsymbol{x})) \tag{11}$$

keeps track of which states provided the highest probability. After iterating over the sequence from the front to the rear end, the most likely state sequence can now be collected by iterating the Back pointer array from the rear to the front, always picking the state of the previous time frame from the value of the Back pointer of the current time frame.

2.3 Training

The training is done by minimizing a cost function composed of a term for the conditional likelihood and a regularization term. Because this is a convex problem a gradient decent algorithm can be used.

Log-Likelihood

For increasing numerical stability, and easing computations by turning products into sums we will use the log-likelihood as in equation 12 and 13 for training.

$$log(\mathcal{P}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x})) = \sum_{t=1}^{T} log(\Psi_{\boldsymbol{\lambda}}(y_{t-1}, y_t, \boldsymbol{x}, t)) - log(Z_{\boldsymbol{\lambda}}(\boldsymbol{x})) \tag{12}$$

$$log(\mathcal{P}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x})) = \sum_{t=1}^{T} \sum_{k \in K} \lambda_k f_k(y_{t-1}, y_t, \boldsymbol{x}, t) - log\left(\sum_{\boldsymbol{s} \in S^{T+1}} \prod_{t=1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t-1}, s_t, \boldsymbol{x}, t)\right) \tag{13}$$

Cost function for Training

Only the ratios between the individual elements (not the absolute values) of the weight vector $\boldsymbol{\lambda}$ influence the likelihood. This implies that scaled versions of $\boldsymbol{\lambda}$ will achieve the same likelihood and that there are infinitely many solutions with maximal likelihood. To deal with this the target function for the optimization

$$\mathcal{C}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x}) = log(\mathcal{P}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x})) - Reg(\boldsymbol{\lambda}) \tag{14}$$

contains a regularization term as well. For regularization we use

$$Reg(\boldsymbol{\lambda}) = \sum_{k \in K} \frac{\lambda_k^2}{2\sigma^2} \tag{15}$$

which forces the sum of squared weights to be small and therefore also fights overfitting. Starting with equation 14 and applying the log-likelihood formula 13 leafs us with the final cost function in equation 16.

$$\mathcal{C}_{\boldsymbol{\lambda}}(\boldsymbol{y} \mid \boldsymbol{x}) = \underbrace{\sum_{t=1}^{T} \sum_{k \in K} \lambda_k f_k(y_{t-1}, y_t, \boldsymbol{x}, t) -}_{A}$$
$$- \underbrace{log\left(\sum_{\boldsymbol{s} \in S^{T+1}} \prod_{t=1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t-1}, s_t, \boldsymbol{x}, t)\right)}_{B} - \underbrace{\sum_{k \in K} \frac{\lambda_k^2}{2\sigma^2}}_{C} \quad (16)$$

Gradient of the Cost function

The derivation is carried out separately for the parts A,B and C. The derivation for A according to λ_k is

$$\frac{\partial}{\partial \lambda_k} \sum_{t=1}^{T} \sum_{k \in K} \lambda_k f_k(y_{t-1}, y_t, \boldsymbol{x}, t) = \sum_{t=1}^{T} f_k(y_{t-1}, y_t, \boldsymbol{x}, t). \quad (17)$$

We start the derivation for B according to λ_k by replacing the logarithm as in equation 18.

$$\frac{\partial}{\partial \lambda_k} log(Z_{\boldsymbol{\lambda}}(\boldsymbol{x})) = \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \frac{\partial Z_{\boldsymbol{\lambda}}(\boldsymbol{x})}{\partial \lambda_k} \quad (18)$$

In equation 19 we plug in the normalization with equation 2. And in the next step depicted in formula 20 we replace the exponential sum with the product operator. In the next step we swap the two sums in the front and derive formula 21.

$$= \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{\boldsymbol{s} \in S^{T+1}} \sum_{t=1}^{T} f_k(s_{t-1}, s_t, \boldsymbol{x}, t) exp\left(\sum_{t'=1}^{T} \sum_{k \in K} \lambda_k f_k(s_{t'-1}, s_{t'}, \boldsymbol{x}, t')\right) \quad (19)$$

$$= \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{\boldsymbol{s} \in S^{T+1}} \sum_{t=1}^{T} f_k(s_{t-1}, s_t, \boldsymbol{x}, t) \prod_{t'=1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t'-1}, s_{t'}, \boldsymbol{x}, t') \quad (20)$$

$$= \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{t=1}^{T} \sum_{\boldsymbol{s} \in S^{T+1}} f_k(s_{t-1}, s_t, \boldsymbol{x}, t) \prod_{t'=1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t'-1}, s_{t'}, \boldsymbol{x}, t') \quad (21)$$

Now we start exploiting the linear chain structure and isolate the edge between s_{t-1} and s_t. Therefor we split up the summation over the state space into the sequence before, the edge itself and the sequence after, as shown in equation 22.

$$= \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{t=1}^{T} \sum_{\boldsymbol{s} \in S^{T+1}_{[0 \ldots t-2]}} \sum_{s_{t-1} \in S} \sum_{s_t \in S} \sum_{\boldsymbol{s} \in S^{T+1}_{[t+1 \ldots T]}} f_k(s_{t-1}, s_t, \boldsymbol{x}, t) \prod_{t'=1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t'-1}, s_{t'}, \boldsymbol{x}, t') \quad (22)$$

We rearrange the summation signs and pull out the computation of f_k so that we derive equation 23.

$$= \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{t=1}^{T} \sum_{s_{t-1}\in S} \sum_{s_t \in S} f_k(s_{t-1}, s_t, \boldsymbol{x}, t) \sum_{\boldsymbol{s}\in S^{T+1}_{[0\ldots t-2]}} \sum_{\boldsymbol{s}\in S^{T+1}_{[t+1\ldots T]}} \prod_{t'=1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t'-1}, s_{t'}, \boldsymbol{x}, t') \tag{23}$$

We split the product into the sequence before, the edge itself and the sequence after. In the same step we pull out the computation of $\Psi_{\boldsymbol{\lambda}}$. We get equation 24 which nicely separates the three parts.

$$\begin{aligned} = \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{t=1}^{T} \sum_{s_{t-1}\in S} \sum_{s_t \in S} & f_k(s_{t-1}, s_t, \boldsymbol{x}, t) \Psi_{\boldsymbol{\lambda}}(s_{t-1}, s_t, \boldsymbol{x}, t) \\ & \sum_{\boldsymbol{s}\in S^{T+1}_{[0\ldots t-2]}} \sum_{\boldsymbol{s}\in S^{T+1}_{[t+1\ldots T]}} \left(\prod_{t'=1}^{t-1} \Psi_{\boldsymbol{\lambda}}(s_{t'-1}, s_{t'}, \boldsymbol{x}, t') \right) \\ & \left(\prod_{t''=t+1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t''-1}, s_{t''}, \boldsymbol{x}, t'') \right) \end{aligned} \tag{24}$$

In formula 25 we pull out the product considering the sequence before the edge $(s_{t-1},\ s_t)$.

$$\begin{aligned} = \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{t=1}^{T} \sum_{s_{t-1}\in S} \sum_{s_t \in S} & f_k(s_{t-1}, s_t, \boldsymbol{x}, t) \Psi_{\boldsymbol{\lambda}}(s_{t-1}, s_t, \boldsymbol{x}, t) \\ & \sum_{\boldsymbol{s}\in S^{T+1}_{[0\ldots t-2]}} \left(\prod_{t'=1}^{t-1} \Psi_{\boldsymbol{\lambda}}(s_{t'-1}, s_{t'}, \boldsymbol{x}, t') \right) \\ & \sum_{\boldsymbol{s}\in S^{T+1}_{[t+1\ldots T]}} \left(\prod_{t''=t+1}^{T} \Psi_{\boldsymbol{\lambda}}(s_{t''-1}, s_{t''}, \boldsymbol{x}, t'') \right) \end{aligned} \tag{25}$$

The sums over the state space of sequences before and after (s_{t-1}, s_t) can be replaced by the Forward and Backward equations as shown in formula 26. The Forward and Backward equations can be calculated efficiently by the respective algorithms.

$$= \frac{1}{Z_{\boldsymbol{\lambda}}(\boldsymbol{x})} \sum_{t=1}^{T} \sum_{s_{t-1}\in S} \sum_{s_t \in S} f_k(s_{t-1}, s_t, \boldsymbol{x}, t) \alpha_{t-1}(s_{t-1} \mid \boldsymbol{x}) \Psi_{\boldsymbol{\lambda}}(s_{t-1}, s_t, \boldsymbol{x}, t) \beta_t(s_t \mid \boldsymbol{x}) \tag{26}$$

Equation 27 shows the derivation of the regularization term C according to λ_k

$$\frac{\partial}{\partial \lambda_k} \sum_{k\in K} \frac{\lambda_k^2}{2\sigma^2} = \frac{2\lambda_k^2}{2\sigma^2} = \frac{\lambda_k^2}{\sigma^2} \tag{27}$$

3 Implementation

The CRF is implemented in Python/Cython which makes it easy to use in a Python environment. A number of commonly used feature functions are implemented and can be parameterized for the intended use case. Furthermore the interface of `FeatureFunction` allows the user to implement arbitrary functions that are considered useful. The value delivered by the feature function can be an arbitrary floating point number.

The calculations for training are carried out in the logarithmic domain. This avoids problems with numeric precision and boosts performance, because products turn into sums. The reasoning algorithms however do not use the log domain to provide true probabilities. Because the Forward and Backward algorithms are also needed for training both variants exist. The methods using the log-domain are distinguishable by the "˙Log" postfix in the method name.

3.1 Defining the CRF

First you have to define your set of feature functions either by using already implemented ones or by creating your own ones. Of course a mixture of both works as well. The feature functions are passed to the CRF as a list. When writing your own functions you can inherit functionality of any of the predefined ones or start from scratch by using the `FeatureFunction` as your basis. The predefined feature functions are:

- FeatureFunctionStateTo: Activated by a transition to a specified state.
- FeatureFunctionStateToIndexedCurrentObservation: Activated by a transition to a specified state and reduces the observation to a single feature. A bunch of those can represent the State-Observation matrices of a HMM.
- FeatureFunctionTransition: Activated by a transition from a specified state to another specified state. With $|S|^2$ functions all the entries of the HMMs state-transition matrix can be represented.
- FeatureFunctionObservation: Activated when a specified observation is observed.
- FeatureFunctionObservationTransition: Activated by a transition from a specified state to if a specified state and a specified observation is observed.

Modeling Gaussian Observations

The following feature functions were designed to give the opportunity to model Gaussian observations. For each state and observation variable you need one identity and one quadratic function of X. The constant function is only needed once per state.

- FeatureFunctionCurrentStateX: Activated by a transition to a specified state and returns the observed value a of single feature.
- FeatureFunctionCurrentStateXSq: Activated by a transition to a specified state and returns the squared observed value a of single feature.

– FeatureFunctionCurrentStateConst: Activated by a transition to a specified state and returns the constant 1.0.

Because Gaussian features in the crf are modelled with three functions and therefore have three independent weights, the functions can express more than Gaussians. To allow only Gaussians one of the weights must be restricted to a value computed from the other two weights. This would require that feature functions have dependencies to each other which conflicts with the actual model. The here implemented model does not apply the mentioned restriction to pure Gaussians. The implementation allows Gaussian like functions with some intercept on the y-axis.

3.2 Training

As already defined in the section 2.1 the state sequence must contain one element more than the corresponding observation sequence. The length of the individual pairs of training sequences can vary.

You can choose between two minimization techniques for training. Either BFGS or the Nelder-Method. The BFGS does an analytic gradient calculation whereas the Nelder-Method uses a stochastic gradient computation.

3.3 Reasoning

Forward and Forward-Backward algorithm deliver the Forward message sequence respectively the hadamard product of Forward message sequence and Backward message sequence. The message sequences exceed the length of the passed observation sequences by one.

The Viterbi algorithm returns a tuple containing the most likely path through the state space as first item. The second item of the tuple is the Viterbi message sequence in the same format as the Forward message sequence.

4 Adder-Example

In this example we will illustrate the advantages of the more flexible feature functions by comparing a CRF restricted to HMM-like feature functions against a CRF with unrestricted feature functions. We choose the example of the adder which was used to illustrate the theoretical advantages in a previous section. Now we train both CRFs on multiple training sequences and evaluate the results on a test sequence.

4.1 Coding

Download MaryCrf from GitHub:

```
https://github.com/napster2202/MaryCrf
```

Compile it using the `setup.py` file via the terminal.

```
python setup.py build_ext --inplace
```

Import the package:

```
from marycrf import *
```

Define feature functions of the HMM like CRF:

```
hmmobservationFunctions =
   [FeatureFunctionObservation(x[1],0,x[0]-1) for x in np.ndindex((3,4))]
hmmtransitionFunctions =
   [FeatureFunctionTransition(x[1], x[0]) for x in np.ndindex((4,4))]
hmmfeaturefunctions =
   np.array(hmmobservationFunctions + hmmtransitionFunctions)

hmmcrf = CyCRF(4,hmmfeaturefunctions)
```

Define feature functions of the CRF:

```
crffeaturefunctions =
   [FeatureFunctionObservationTransition
      (x[2], x[1],0,x[0]-1) for x in np.ndindex((3,4,4))]

crf = CyCRF(4,crffeaturefunctions)
```

Train both CRFs:

```
hmmcrf.train_BFGS(train_y,train_x)
crf.train_BFGS(train_y,train_x)
```

We use the following test sequence:

```
test_x =
    array([[-1, -1, 1, 1, 1, -1, 1, 0, 0, -1, 0, -1, 0, 0, -1, -1, -1,
        -1, 1, 0, 0, 1, 1, 1, 0, -1, -1, 1, 1, 0, -1, 1, -1, -1,
        1, -1, 0, 1, 1, 1, -1, 1, 0, -1, 1, 1, 0, -1, 1, -1]])

test_y =
    array([3, 3, 3, 2, 1, 0, 1, 0, 0, 0, 1, 1, 2, 2, 2, 3, 3, 3, 3, 2, 2,
        2, 1, 0, 0, 0, 1, 2, 1, 0, 0, 1, 0, 1, 2, 1, 2, 2, 1, 0, 0, 1,
        0, 0, 1, 0, 0, 0, 1, 0, 1])
```

Apply the Viterbi algorithm:

```
viterbi_hmmcrf = hmmcrf.viterbi(test_x)
viterbi_crf = crf.viterbi(test_x)
```

4.2 Results

Figure 2 shows the real state sequence of the test example. In figure 3 we see that the HMM like CRF was not able to model the behavior of our hidden function. The CRF with observation indicated transitions could model the function from some point on.

Why did it take until $t = 2$ to get the path right? At $t = 5$ the adder hits the state $y_5 = 3$ after having observed three ones($x_{3,4,5} = 1$). Each of the ones triggered a state change which only is possible if $y_2 = 0$. Before that there were two observations of minus ones($x_{1,2} = -1$). This would fit to multiple sequences ($y_0 = 2, y_1 = 1$),($y_0 = 1, y_1 = 0$) or ($y_0 = 0, y_1 = 0$) with the same likelihood when thinking of the adder and the random construction of the sequences. Of course, the CRF itself could have a preference to one of those combinations, depending on what transitions occurred more often in the training set. In figure 4 of the Viterbi path for the CRF we can clearly see that from $t = 2$ on the path is completely correct. The gray scale background represents a normalized view of the Viterbi message, the red arrows show the path through the state space.

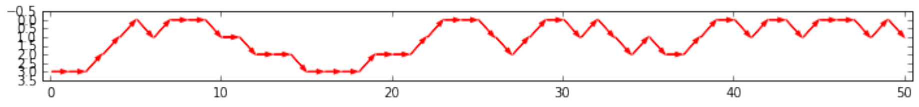

Fig. 2. The real test state sequence

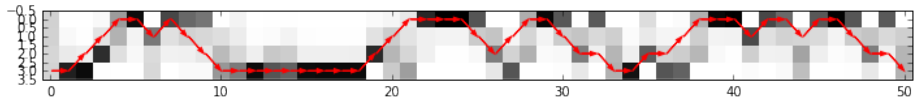

Fig. 3. Trace found by an HMM simulated by an CRF

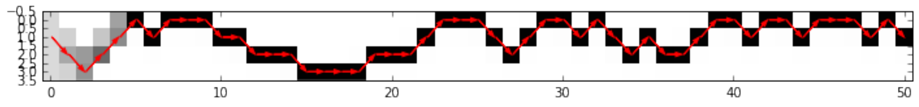

Fig. 4. Trace found by an CRF with observation indicated transitions

References

1. Klinger, R., Tomanek, K.: Classical probabilistic models and conditional random fields. TU, Algorithm Engineering (2007)
2. Sutton, C., McCallum, A.: An introduction to conditional random fields. Machine Learning 4(4), 267–373 (2011)
3. Vail, D.L., Veloso, M.M., Lafferty, J.D.: Conditional random fields for activity recognition. In: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems. p. 235. ACM (2007)