Package ‘CRF’

February 12, 2014

Version 0.3-5

Date 2014-02-08

Title CRF - Conditional Random Fields

Author Ling-Yun Wu <wulingyun@gmail.com>

Maintainer Ling-Yun Wu <wulingyun@gmail.com>

Imports Rglpk (>= 0.3-5)

Depends R (>= 2.12.0)

Description Library to decode/infer/sample/train Conditional Random Fields

License GPL (>= 2)

R topics documented:

`CRF-package` *CRF - Conditional Random Fields*

Description

Library to decode/infer/sample/train Conditional Random Fields

Details

CRF is R package for various computational tasks of conditional random fields as well as other probabilistic undirected graphical models of discrete data with pairwise (and unary) potentials. The decoding/inference/sampling tasks are implemented for general discrete undirected graphical models with pairwise potentials. The training task is less general, focusing on conditional random fields with log-linear potentials and a fixed structure. The code is written entirely in R and C++. The initial version is ported from UGM written by Mark Schmidt.

Decoding: Computing the most likely configuration

- `decode.exact` Exact decoding for small graphs with brute-force search

- decode.chain Exact decoding for chain-structured graphs with the Viterbi algorithm
- decode.tree Exact decoding for tree- and forest-structured graphs with max-product belief propagation
- decode.conditional Conditional decoding (takes another decoding method as input)
- decode.cutset Exact decoding for graphs with a small cutset using cutset conditioning
- decode.junction Exact decoding for low-treewidth graphs using junction trees
- decode.sample Approximate decoding using sampling (takes a sampling method as input)
- decode.marginal Approximate decoding using inference (takes an inference method as input)
- decode.lbp Approximate decoding using max-product loopy belief propagation
- decode.trbp Approximate decoding using max-product tree-reweighted belief propagtion
- decode.greedy Approximate decoding with greedy algorithm
- decode.icm Approximate decoding with the iterated conditional modes algorithm
- decode.block Approximate decoding with the block iterated conditional modes algorithm
- decode.ilp Exact decoding with an integer linear programming formulation and approximate using LP relaxation

Inference: Computing the partition function and marginal probabilities

- infer.exact Exact inference for small graphs with brute-force counting
- infer.chain Exact inference for chain-structured graphs with the forward-backward algorithm
- infer.tree Exact inference for tree- and forest-structured graphs with sum-product belief propagation
- infer.conditional Conditional inference (takes another inference method as input)
- infer.cutset Exact inference for graphs with a small cutset using cutset conditioning
- infer.junction Exact decoding for low-treewidth graphs using junction trees
- infer.sample Approximate inference using sampling (takes a sampling method as input)
- infer.lbp Approximate inference using sum-product loopy belief propagation
- infer.trbp Approximate inference using sum-product tree-reweighted belief propagation

Sampling: Generating samples from the distribution

- sample.exact Exact sampling for small graphs with brute-force inverse cumulative distribution
- sample.chain Exact sampling for chain-structured graphs with the forward-filter backward-sample algorithm
- sample.tree Exact sampling for tree- and forest-structured graphs with sum-product belief propagation and backward-sampling
- sample.conditional Conditional sampling (takes another sampling method as input)
- sample.cutset Exact sampling for graphs with a small cutset using cutset conditioning
- sample.junction Exact sampling for low-treewidth graphs using junction trees
- sample.gibbs Approximate sampling using a single-site Gibbs sampler

Training: Given data, computing the most likely estimates of the parameters

- train.crf Train CRF model
- train.mrf Train MRF model

Author(s)

Ling-Yun Wu <wulingyun@gmail.com>

References

J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *the proceedings of International Conference on Machine Learning (ICML)*, pp. 282-289, 2001.

Mark Schmidt. UGM: Matlab code for undirected graphical models. http://www.di.ens.fr/~mschmidt/Software/UGM.html

See Also

make.crf

Examples

```
library(CRF)
data(Small)
decode.exact(Small$crf)
infer.exact(Small$crf)
sample.exact(Small$crf, 100)
```

Chain *Chain CRF example*

Description

This data set gives a chain CRF example

Usage

```
data(Chain)
```

Format

A list containing two elements:

- crf The CRF
- answer A list of 4 elements:
  - decode The most likely configuration
  - node.bel The node belief
  - edge.bel The edge belief
  - logZ The logarithmic value of CRF normalization factor Z

`clamp.crf` *Make clamped CRF*

Description

Generate clamped CRF by fixing the states of some nodes

Usage

```
clamp.crf(crf, clamped)
```

Arguments

`crf`	The CRF generated by `make.crf`
`clamped`	The vector of fixed states of nodes

Details

The function will generate a clamped CRF from a given CRF by fixing the states of some nodes. The vector `clamped` contains the desired state for each node while zero means the state is not fixed. The node and edge potentials are updated to the conditional potentials based on the clamped vector.

Value

The function will return a new CRF with additional components:

`original`	The original CRF.
`clamped`	The vector of fixed states of nodes.
`node.id`	The vector of the original node ids for nodes in the new CRF.
`node.map`	The vector of the new node ids for nodes in the original CRF.
`edge.id`	The vector of the original edge ids for edges in the new CRF.
`edge.map`	The vector of the new edge ids for edges in the original CRF.

See Also

`make.crf`, `sub.crf`, `clamp.reset`

Examples

```
library(CRF)
data(Small)
crf <- clamp.crf(Small$crf, c(0, 0, 1, 1))
```

clamp.reset *Reset clamped CRF*

Description

Reset clamped CRF by changing the states of clamped nodes

Usage

```
clamp.reset(crf, clamped)
```

Arguments

`crf`	The clamped CRF generated by `clamp.crf`
`clamped`	The vector of fixed states of nodes

Details

The function will reset a clamped CRF by changing the states of fixed nodes. The vector `clamped` contains the desired state for each node while zero means the state is not fixed. The node and edge potentials are updated to the conditional potentials based on the clamped vector.

Value

The function will return the same clamped CRF.

See Also

`make.crf`, `clamp.crf`

Examples

```
library(CRF)
data(Small)
crf <- clamp.crf(Small$crf, c(0, 0, 1, 1))
clamp.reset(crf, c(0,0,2,2))
```

Clique *Clique CRF example*

Description

This data set gives a clique CRF example

Usage

```
data(Clique)
```

Format

A list containing two elements:

- `crf` The CRF
- `answer` A list of 4 elements:
  - `decode` The most likely configuration
  - `node.bel` The node belief
  - `edge.bel` The edge belief
  - `logZ` The logarithmic value of CRF normalization factor Z

---

crf.nll *Calculate CRF negative log likelihood*

Description

Calculate the negative log likelihood of CRF model

Usage

```
crf.nll(par, crf, instances, node.fea = NaN, edge.fea = NaN,
  node.ext = NaN, edge.ext = NaN, infer.method = infer.chain, ...)
```

Arguments

`crf`	The CRF
`par`	The parameter vector of CRF
`instances`	The training data matrix of CRF model
`node.fea`	The list of node features
`edge.fea`	The list of edge features
`node.ext`	The list of extended information of node features
`edge.ext`	The list of extended information of edge features
`infer.method`	The inference method used to compute the likelihood
`...`	Other parameters need by the inference method

Details

This function calculates the negative log likelihood of CRF model as well as the gradient. This function is intended to be called by optimization algorithm in training process.

In the training data matrix `instances`, each row is an instance and each column corresponds a node in CRF. The variables `node.fea`, `edge.fea`, `node.ext`, `edge.ext` are lists of length equal to the number of instances, and their elements are defined as in `crf.update` respectively.

Value

This function will return the value of CRF negative log-likelihood.

See Also

`crf.update`, `train.crf`

`crf.update` *Update CRF potentials*

Description

Update node.pot and edge.pot of CRF model

Usage

```
crf.update(crf, node.fea = NaN, edge.fea = NaN, node.ext = NaN,
  edge.ext = NaN)
```

Arguments

`crf`	The CRF
`node.fea`	The node features matrix with dimension `(n.nf, n.nodes)`
`edge.fea`	The edge features matrix with dimension `(n.ef, n.edges)`
`node.ext`	The extended information of node features
`edge.ext`	The extended information of edge features

Details

This function updates `node.pot` and `edge.pot` of CRF model by using the current values of parameters and features.

There are two ways to model the relationship between parameters and features. The first one exploits the special structure of features to reduce the memory usage. However it may not suitable for all circumstances. The other one is more straighforward by explicitly specifying the coefficients of each parameter to calculate the potentials, and may use much more memory. Two approaches can be used together.

The first way uses the objects `node.par` and `edge.par` to define the structure of features and provides the feature information in variables `node.fea` and `edge.fea`. The second way directly provides the feature information in variables `node.ext` and `edge.ext` without any prior assumption on feature structure. `node.ext` is a list and each element has the same structure as `node.pot`. `edge.ext` is a list and each element has the same structure as `edge.pot`.

In detail, the node potential is updated as follows:

$$node.pot[n,i] = \sum_f par[node.par[n,i,f]] * node.fea[f,n] + \sum_k par[k] * node.ext[[k]][n,i]$$

and the edge potential is updated as follows:

$$edge.pot[[e]][i,j] = \sum_f par[edge.par[[e]][i,j,f]] * edge.fea[f,e] + \sum_k par[k] * edge.ext[[k]][[e]][i,j]$$

Value

This function will directly modify the CRF and return the same CRF.

See Also

`crf.nll`, `train.crf`

decode.block *Decoding method using block iterated conditional modes algorithm*

Description

Computing the most likely configuration for CRF

Usage

```
decode.block(crf, blocks, decode.method = decode.tree, restart = 0,
  start = apply(crf$node.pot, 1, which.max), ...)
```

Arguments

`crf`	The CRF
`blocks`	A list of vectors, each vector containing the nodes in a block
`decode.method`	The decoding method to solve the clamped CRF
`restart`	Non-negative integer to control how many restart iterations are repeated
`start`	An initial configuration, a good start will significantly reduce the seraching time
`...`	The parameters for `decode.method`

Details

Approximate decoding with the block iterated conditional modes algorithm

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.block(Small$crf, list(c(1,3), c(2,4)))
```

decode.chain *Decoding method for chain-structured graphs*

Description

Computing the most likely configuration for CRF

Usage

```
decode.chain(crf)
```

Arguments

`crf`	The CRF

Details

Exact decoding for chain-structured graphs with the Viterbi algorithm.

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.chain(Small$crf)
```

`decode.conditional` *Conditional decoding method*

Description

Computing the most likely configuration for CRF

Usage

```
decode.conditional(crf, clamped, decode.method, ...)
```

Arguments

`crf`	The CRF
`clamped`	The vector of fixed values for clamped nodes, 0 for unfixed nodes
`decode.method`	The decoding method to solve clamped CRF
`...`	The parameters for `decode.method`

Details

Conditional decoding (takes another decoding method as input)

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.conditional(Small$crf, c(0,1,0,0), decode.exact)
```

---

decode.cutset *Decoding method for graphs with a small cutset*

Description

Computing the most likely configuration for CRF

Usage

```
decode.cutset(crf, cutset, engine = "default", start = apply(crf$node.pot,
  1, which.max))
```

Arguments

`crf`	The CRF
`cutset`	A vector of nodes in the cutset
`engine`	The underlying engine for cutset decoding, possible values are "default", "none", "exact", "chain", and "tree".
`start`	An initial configuration, a good start will significantly reduce the seraching time

Details

Exact decoding for graphs with a small cutset using cutset conditioning

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.cutset(Small$crf, c(2))
```

---

decode.exact *Decoding method for small graphs*

Description

Computing the most likely configuration for CRF

Usage

```
decode.exact(crf)
```

Arguments

`crf`	The CRF

Details

Exact decoding for small graphs with brute-force search

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.exact(Small$crf)
```

---

decode.greedy — *Decoding method using greedy algorithm*

Description

Computing the most likely configuration for CRF

Usage

```
decode.greedy(crf, restart = 0, start = apply(crf$node.pot, 1, which.max))
```

Arguments

`crf`	The CRF
`restart`	Non-negative integer to control how many restart iterations are repeated
`start`	An initial configuration, a good start will significantly reduce the seraching time

Details

Approximate decoding with greedy algorithm

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.greedy(Small$crf)
```

decode.icm — *Decoding method using iterated conditional modes algorithm*

Description

Computing the most likely configuration for CRF

Usage

```
decode.icm(crf, restart = 0, start = apply(crf$node.pot, 1, which.max))
```

Arguments

`crf`	The CRF
`restart`	Non-negative integer to control how many restart iterations are repeated
`start`	An initial configuration, a good start will significantly reduce the seraching time

Details

Approximate decoding with the iterated conditional modes algorithm

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.icm(Small$crf)
```

decode.ilp — *Decoding method using integer linear programming*

Description

Computing the most likely configuration for CRF

Usage

```
decode.ilp(crf, lp.rounding = FALSE)
```

Arguments

`crf`	The CRF
`lp.rounding`	Boolean variable to indicate whether LP rounding is need.

Details

Exact decoding with an integer linear programming formulation and approximate using LP relaxation

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.ilp(Small$crf)
```

`decode.junction` *Decoding method for low-treewidth graphs*

Description

Computing the most likely configuration for CRF

Usage

```
decode.junction(crf)
```

Arguments

`crf`	The CRF

Details

Exact decoding for low-treewidth graphs using junction trees

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.junction(Small$crf)
```

decode.lbp *Decoding method using loopy belief propagation*

Description

Computing the most likely configuration for CRF

Usage

```
decode.lbp(crf, max.iter = 10000, cutoff = 1e-04, verbose = 0)
```

Arguments

crf	The CRF
max.iter	The maximum allowed iterations of termination criteria
cutoff	The convergence cutoff of termination criteria
verbose	Non-negative integer to control the tracing informtion in algorithm

Details

Approximate decoding using max-product loopy belief propagation

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.lbp(Small$crf)
```

decode.marginal *Decoding method using inference*

Description

Computing the most likely configuration for CRF

Usage

```
decode.marginal(crf, infer.method, ...)
```

Arguments

crf	The CRF
infer.method	The inference method
...	The parameters for `infer.method`

Details

Approximate decoding using inference (takes an inference method as input)

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.marginal(Small$crf, infer.exact)
```

`decode.sample` *Decoding method using sampling*

Description

Computing the most likely configuration for CRF

Usage

```
decode.sample(crf, sample.method, ...)
```

Arguments

`crf`	The CRF
`sample.method`	The sampling method
`...`	The parameters for `sample.method`

Details

Approximate decoding using sampling (takes a sampling method as input)

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.sample(Small$crf, sample.exact, 10000)
```

decode.trbp — *Decoding method using tree-reweighted belief propagation*

Description

Computing the most likely configuration for CRF

Usage

```
decode.trbp(crf, max.iter = 10000, cutoff = 1e-04, verbose = 0)
```

Arguments

`crf`	The CRF
`max.iter`	The maximum allowed iterations of termination criteria
`cutoff`	The convergence cutoff of termination criteria
`verbose`	Non-negative integer to control the tracing informtion in algorithm

Details

Approximate decoding using max-product tree-reweighted belief propagtion

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.trbp(Small$crf)
```

decode.tree — *Decoding method for tree- and forest-structured graphs*

Description

Computing the most likely configuration for CRF

Usage

```
decode.tree(crf)
```

Arguments

`crf`	The CRF

Details

Exact decoding for tree- and forest-structured graphs with max-product belief propagation

Value

This function will return the most likely configuration, which is a vector of length `crf$n.nodes`.

Examples

```
library(CRF)
data(Small)
d <- decode.tree(Small$crf)
```

`duplicate` *Duplicate CRF*

Description

Duplicate an existing CRF

Usage

```
duplicate(crf)
```

Arguments

`crf`	The existing CRF

Details

This function will duplicate an existing CRF. Since CRF is implemented as an environment, normal assignment will only copy the pointer instead of the real data. This function will generate a new CRF and really copy all data.

Value

The function will return a new CRF with copied data

See Also

`make.crf`

`get.logPotential` *Calculate the log-potential of CRF*

Description

Calculate the logarithmic potential of a CRF with given configuration

Usage

```
get.logPotential(crf, configuration)
```

Arguments

`crf`	The CRF
`configuration`	The vector of states of nodes

Details

The function will calculate the logarithmic potential of a CRF with given configuration, i.e., the assigned states of nodes in the CRF.

Value

The function will return the log-potential of CRF with given configuration

See Also

`get.potential`

`get.potential` *Calculate the potential of CRF*

Description

Calculate the potential of a CRF with given configuration

Usage

```
get.potential(crf, configuration)
```

Arguments

`crf`	The CRF
`configuration`	The vector of states of nodes

Details

The function will calculate the potential of a CRF with given configuration, i.e., the assigned states of nodes in the CRF.

Value

The function will return the potential of CRF with given configuration

See Also

get.logPotential

---

infer.chain *Inference method for chain-structured graphs*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.chain(crf)
```

Arguments

crf	The CRF

Details

Exact inference for chain-structured graphs with the forward-backward algorithm

Value

This function will return a list with components:

node.bel	Node belief. It is a matrix with `crf$n.nodes` rows and `crf$max.state` columns.
edge.bel	Edge belief. It is a list of matrices. The size of list is `crf$n.edges` and the matrix i has `crf$n.states[crf$edges[i,1]]` rows and `crf$n.states[crf$edges[i,2]]` columns.
logZ	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.chain(Small$crf)
```

`infer.conditional` *Conditional inference method*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.conditional(crf, clamped, infer.method, ...)
```

Arguments

`crf`	The CRF
`clamped`	The vector of fixed values for clamped nodes, 0 for unfixed nodes
`infer.method`	The inference method to solve the clamped CRF
`...`	The parameters for `infer.method`

Details

Conditional inference (takes another inference method as input)

Value

This function will return a list with components:

`node.bel`	Node belief. It is a matrix with `crf$n.nodes` rows and `crf$max.state` columns.
`edge.bel`	Edge belief. It is a list of matrices. The size of list is `crf$n.edges` and the matrix i has `crf$n.states[crf$edges[i,1]]` rows and `crf$n.states[crf$edges[i,2]]` columns.
`logZ`	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.conditional(Small$crf, c(0,1,0,0), infer.exact)
```

infer.cutset *Inference method for graphs with a small cutset*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.cutset(crf, cutset, engine = "default")
```

Arguments

crf	The CRF
cutset	A vector of nodes in the cutset
engine	The underlying engine for cutset decoding, possible values are "default", "none", "exact", "chain", and "tree".

Details

Exact inference for graphs with a small cutset using cutset conditioning

Value

This function will return a list with components:

node.bel	Node belief. It is a matrix with `crf$n.nodes` rows and `crf$max.state` columns.
edge.bel	Edge belief. It is a list of matrices. The size of list is `crf$n.edges` and the matrix `i` has `crf$n.states[crf$edges[i,1]]` rows and `crf$n.states[crf$edges[i,2]]` columns.
logZ	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.cutset(Small$crf, c(2))
```

infer.exact *Inference method for small graphs*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.exact(crf)
```

Arguments

`crf`	The CRF

Details

Exact inference for small graphs with brute-force counting

Value

This function will return a list with components:

`node.bel`	Node belief. It is a matrix with `crf$n.nodes` rows and `crf$max.state` columns.
`edge.bel`	Edge belief. It is a list of matrices. The size of list is `crf$n.edges` and the matrix i has `crf$n.states[crf$edges[i,1]]` rows and `crf$n.states[crf$edges[i,2]]` columns.
`logZ`	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.exact(Small$crf)
```

`infer.junction` *Inference method for low-treewidth graphs*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.junction(crf)
```

Arguments

`crf`	The CRF

Details

Exact decoding for low-treewidth graphs using junction trees

Value

This function will return a list with components:

`node.bel`	Node belief. It is a matrix with `crf$n.nodes` rows and `crf$max.state` columns.
`edge.bel`	Edge belief. It is a list of matrices. The size of list is `crf$n.edges` and the matrix i has `crf$n.states[crf$edges[i,1]]` rows and `crf$n.states[crf$edges[i,2]]` columns.
`logZ`	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.junction(Small$crf)
```

`infer.lbp` *Inference method using loopy belief propagation*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.lbp(crf, max.iter = 10000, cutoff = 1e-04, verbose = 0)
```

Arguments

`crf`	The CRF
`max.iter`	The maximum allowed iterations of termination criteria
`cutoff`	The convergence cutoff of termination criteria
`verbose`	Non-negative integer to control the tracing informtion in algorithm

Details

Approximate inference using sum-product loopy belief propagation

Value

This function will return a list with components:

`node.bel`	Node belief. It is a matrix with `crf$n.nodes` rows and `crf$max.state` columns.
`edge.bel`	Edge belief. It is a list of matrices. The size of list is `crf$n.edges` and the matrix i has `crf$n.states[crf$edges[i,1]]` rows and `crf$n.states[crf$edges[i,2]]` columns.
`logZ`	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.lbp(Small$crf)
```

infer.sample *Inference method using sampling*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.sample(crf, sample.method, ...)
```

Arguments

crf	The CRF
sample.method	The sampling method
...	The parameters for sample.method

Details

Approximate inference using sampling (takes a sampling method as input)

Value

This function will return a list with components:

node.bel	Node belief. It is a matrix with crf$n.nodes rows and crf$max.state columns.
edge.bel	Edge belief. It is a list of matrices. The size of list is crf$n.edges and the matrix i has crf$n.states[crf$edges[i,1]] rows and crf$n.states[crf$edges[i,2]] columns.
logZ	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.sample(Small$crf, sample.exact, 10000)
```

infer.trbp *Inference method using tree-reweighted belief propagation*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.trbp(crf, max.iter = 10000, cutoff = 1e-04, verbose = 0)
```

Arguments

`crf`	The CRF
`max.iter`	The maximum allowed iterations of termination criteria
`cutoff`	The convergence cutoff of termination criteria
`verbose`	Non-negative integer to control the tracing informtion in algorithm

Details

Approximate inference using sum-product tree-reweighted belief propagation

Value

This function will return a list with components:

`node.bel`	Node belief. It is a matrix with `crf$n.nodes` rows and `crf$max.state` columns.
`edge.bel`	Edge belief. It is a list of matrices. The size of list is `crf$n.edges` and the matrix `i` has `crf$n.states[crf$edges[i,1]]` rows and `crf$n.states[crf$edges[i,2]]` columns.
`logZ`	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.trbp(Small$crf)
```

`infer.tree` *Inference method for tree- and forest-structured graphs*

Description

Computing the partition function and marginal probabilities

Usage

```
infer.tree(crf)
```

Arguments

`crf`	The CRF

Details

Exact inference for tree- and forest-structured graphs with sum-product belief propagation

Value

This function will return a list with components:

node.bel	Node belief. It is a matrix with crf$n.nodes rows and crf$max.state columns.
edge.bel	Edge belief. It is a list of matrices. The size of list is crf$n.edges and the matrix i has crf$n.states[crf$edges[i,1]] rows and crf$n.states[crf$edges[i,2]] columns.
logZ	The logarithmic value of CRF normalization factor Z.

Examples

```
library(CRF)
data(Small)
i <- infer.tree(Small$crf)
```

Loop *Loop CRF example*

Description

This data set gives a loop CRF example

Usage

```
data(Loop)
```

Format

A list containing two elements:

- crf The CRF
- answer A list of 4 elements:
  - decode The most likely configuration
  - node.bel The node belief
  - edge.bel The edge belief
  - logZ The logarithmic value of CRF normalization factor Z

make.crf *Make CRF*

Description

Generate CRF from the adjacent matrix

Usage

```
make.crf(adj.matrix, nstates)
```

Arguments

adj.matrix	The adjacent matrix of CRF network
nstates	The state numbers of nodes

Details

The function will generate an empty CRF from a given adjacent matrix. If the length of `nstates` is less than `n.nodes`, it will be used repeatly. All node and edge potentials are initilized as 1.

Since the CRF data are often very huge, CRF is implemented as an environment. The assignment of environments will only copy the addresses instead of real data, therefore the variables using normal assignment will refer to the exactly same CRF. For complete duplication of the data, please use `duplicate`.

Value

The function will return a new CRF, which is an environment with components:

n.nodes	The number of nodes.
n.edges	The number of edges.
n.states	The number of states for each node. It is a vector of length `n.nodes`.
max.state	The maximum number of states. It is equal to `max(n.states)`.
edges	The node pair of each edge. It is a matrix with 2 columns and `n.edges` rows. Each row denotes one edge. The node with smaller id is put in the first column.
n.adj	The number of adjacent nodes for each node. It is a vector of length `n.nodes`.
adj.nodes	The list of adjacent nodes for each node. It is a list of length `n.nodes` and the i-th element is a vector of length `n.adj[i]`.
adj.edges	The list of adjacent edges for each node. It is similiar to `adj.nodes` while contains the edge ids instead of node ids.
node.pot	The node potentials. It is a matrix with dimmension `(n.nodes,max.state)`. Each row `node.pot[i,]` denotes the node potentials of the i-th node.
edge.pot	The edge potentials. It is a list of `n.edges` matrixes. Each matrix `edge.pot[[i]]`, with dimension `(n.states[edges[i,1]], n.states[edges[i,2]])`, denotes the edge potentials of the i-th edge.

See Also

`duplicate`, `clamp.crf`, `sub.crf`

Examples

```
library(CRF)

nNodes <- 4
nStates <- 2

adj <- matrix(0, nrow=nNodes, ncol=nNodes)
for (i in 1:(nNodes-1))
{
adj[i,i+1] <- 1
adj[i+1,i] <- 1
}

crf <- make.crf(adj, nStates)

crf$node.pot[1,] <- c(1, 3)
crf$node.pot[2,] <- c(9, 1)
crf$node.pot[3,] <- c(1, 3)
crf$node.pot[4,] <- c(9, 1)

for (i in 1:crf$n.edges)
{
   crf$edge.pot[[i]][1,] <- c(2, 1)
   crf$edge.pot[[i]][2,] <- c(1, 2)
}
```

---

make.features *Make CRF features*

Description

Make the data structure of features

Usage

```
make.features(crf, n.nf = 1, n.ef = 1)
```

Arguments

crf	The CRF
n.nf	The number of node features
n.ef	The number of edge features

Details

This function makes the data structure of features need for modeling and training CRF.

The parameters `n.nf` and `n.ef` specify the number of node and edge features, respectively.

The objects `node.par` and `edge.par` define the corresponding parameters used with each feature. `node.par` is a 3-dimensional arrays, and element `node.par[n,i,f]` is the index of parameter associated with the corresponding node potential `node.pot[n,i]` and node feature `f`. `edge.par` is a list of 3-dimensional arrays, and element `edge.par[[e]][i,j,f]` is the index of parameter associated with the corresponding edge potential edge.pot[[e]][i,j] and edge feature f. The value 0 is used to indicate the corresponding node or edge potential does not depend on that feature.

For detail of calculation of node and edge potentials from features and parameters, please see crf.update.

Value

This function will directly modify the CRF and return the same CRF.

See Also

crf.update, make.par, make.crf

---

make.par *Make CRF parameters*

Description

Make the data structure of parameters

Usage

```
make.par(crf, n.par = 1)
```

Arguments

crf	The CRF
n.par	The number of parameters

Details

This function makes the data structure of parameters need for modeling and training CRF. The parameters are stored in par, which is a numeric vector of length n.par.

Value

This function will directly modify the CRF and return the same CRF.

See Also

crf.update, make.features, make.crf

`mrf.nll` *Calculate MRF negative log-likelihood*

Description

Calculate the negative log-likelihood of MRF model

Usage

```
mrf.nll(par, crf, instances, infer.method = infer.chain, ...)
```

Arguments

`crf`	The CRF
`par`	The parameter vector of CRF
`instances`	The training data matrix of MRF model
`infer.method`	The inference method used to compute the likelihood
`...`	Other parameters need by the inference method

Details

This function calculates the negative log-likelihood of MRF model as well as the gradient. This function is intended to be called by optimization algorithm in training process. Before calling this function, the MRF sufficient statistics must be calculated and stored in object `par.stat` of CRF.

In the training data matrix `instances`, each row is an instance and each column corresponds a node in CRF.

Value

This function will return the value of MRF negative log-likilihood.

See Also

`mrf.stat`, `mrf.update`, `train.mrf`

`mrf.stat` *Calculate MRF sufficient statistics*

Description

Calculate the sufficient statistics of MRF model

Usage

```
mrf.stat(crf, instances)
```

Arguments

`crf`	The CRF
`instances`	The training data matrix of MRF model

Details

This function calculates the sufficient statistics of MRF model. This function much be called before the first calling to `mrf.nll`. In the training data matrix `instances`, each row is an instance and each column corresponds a node in CRF.

Value

This function will return the value of MRF sufficient statistics.

See Also

`mrf.nll`, `train.mrf`

`mrf.update` *Update MRF potentials*

Description

Update node.pot and edge.pot of MRF model

Usage

```
mrf.update(crf)
```

Arguments

`crf`	The CRF

Details

The function updates `node.pot` and `edge.pot` of MRF model.

Value

This function will directly modify the CRF and return the same CRF.

See Also

`mrf.nll`, `train.mrf`

`Rain` *Rain data*

Description

This data set gives an example of rain data used to train CRF and MRF models

Usage

```
data(Rain)
```

Format

A list containing two elements:

- `rain` A matrix of 28 columns containing raining data (1: rain, 2: sunny). Each row is an instance of 28 days for one month.
- `months` A vector containing the months of each instance.

References

Mark Schmidt. UGM: Matlab code for undirected graphical models. http://www.di.ens.fr/~mschmidt/Software/UGM.html

`sample.chain` *Sampling method for chain-structured graphs*

Description

Generating samples from the distribution

Usage

```
sample.chain(crf, size)
```

Arguments

`crf`	The CRF
`size`	The sample size

Details

Exact sampling for chain-structured graphs with the forward-filter backward-sample algorithm

Value

This function will return a matrix with `size` rows and `crf$n.nodes` columns, in which each row is a sampled configuration.

Examples

```
library(CRF)
data(Small)
s <- sample.chain(Small$crf, 100)
```

`sample.conditional` *Conditional sampling method*

Description

Generating samples from the distribution

Usage

```
sample.conditional(crf, size, clamped, sample.method, ...)
```

Arguments

`crf`	The CRF
`size`	The sample size
`clamped`	The vector of fixed values for clamped nodes, 0 for unfixed nodes
`sample.method`	The sampling method to solve the clamped CRF
`...`	The parameters for `sample.method`

Details

Conditional sampling (takes another sampling method as input)

Value

This function will return a matrix with `size` rows and `crf$n.nodes` columns, in which each row is a sampled configuration.

Examples

```
library(CRF)
data(Small)
s <- sample.conditional(Small$crf, 100, c(0,1,0,0), sample.exact)
```

`sample.cutset` *Sampling method for graphs with a small cutset*

Description

Generating samples from the distribution

Usage

```
sample.cutset(crf, size, cutset, engine = "default")
```

Arguments

`crf`	The CRF
`size`	The sample size
`cutset`	A vector of nodes in the cutset
`engine`	The underlying engine for cutset sampling, possible values are "default", "none", "exact", "chain", and "tree".

Details

Exact sampling for graphs with a small cutset using cutset conditioning

Value

This function will return a matrix with `size` rows and `crf$n.nodes` columns, in which each row is a sampled configuration.

Examples

```
library(CRF)
data(Small)
s <- sample.cutset(Small$crf, 100, c(2))
```

`sample.exact` *Sampling method for small graphs*

Description

Generating samples from the distribution

Usage

```
sample.exact(crf, size)
```

Arguments

`crf`	The CRF
`size`	The sample size

Details

Exact sampling for small graphs with brute-force inverse cumulative distribution

Value

This function will return a matrix with `size` rows and `crf$n.nodes` columns, in which each row is a sampled configuration.

Examples

```
library(CRF)
data(Small)
s <- sample.exact(Small$crf, 100)
```

`sample.gibbs` *Sampling method using single-site Gibbs sampler*

Description

Generating samples from the distribution

Usage

```
sample.gibbs(crf, size, burn.in = 1000, start = apply(crf$node.pot, 1,
  which.max))
```

Arguments

`crf`	The CRF
`size`	The sample size
`burn.in`	The number of samples at the beginning that will be discarded
`start`	An initial configuration

Details

Approximate sampling using a single-site Gibbs sampler

Value

This function will return a matrix with `size` rows and `crf$n.nodes` columns, in which each row is a sampled configuration.

Examples

```
library(CRF)
data(Small)
s <- sample.gibbs(Small$crf, 100)
```

`sample.junction` *Sampling method for low-treewidth graphs*

Description

Generating samples from the distribution

Usage

```
sample.junction(crf, size)
```

Arguments

`crf`	The CRF
`size`	The sample size

Details

Exact sampling for low-treewidth graphs using junction trees

Value

This function will return a matrix with `size` rows and `crf$n.nodes` columns, in which each row is a sampled configuration.

Examples

```
library(CRF)
data(Small)
s <- sample.junction(Small$crf, 100)
```

`sample.tree` *Sampling method for tree- and forest-structured graphs*

Description

Generating samples from the distribution

Usage

```
sample.tree(crf, size)
```

Arguments

`crf`	The CRF
`size`	The sample size

Details

Exact sampling for tree- and forest-structured graphs with sum-product belief propagation and backward-sampling

Value

This function will return a matrix with `size` rows and `crf$n.nodes` columns, in which each row is a sampled configuration.

Examples

```
library(CRF)
data(Small)
s <- sample.tree(Small$crf, 100)
```

`Small` *Small CRF example*

Description

This data set gives a small CRF example

Usage

```
data(Small)
```

Format

A list containing two elements:

- `crf` The CRF
- `answer` A list of 4 elements:
  - `decode` The most likely configuration
  - `node.bel` The node belief
  - `edge.bel` The edge belief
  - `logZ` The logarithmic value of CRF normalization factor Z

`sub.crf` *Make sub CRF*

Description

Generate sub CRF by selecting some nodes

Usage

```
sub.crf(crf, subset)
```

Arguments

`crf`	The CRF generated by `make.crf`
`subset`	The vector of selected node ids

Details

The function will generate a new CRF from a given CRF by selecting some nodes. The vector `subset` contains the node ids selected to generate the new CRF. Unlike `clamp.crf`, the potentials of remainning nodes and edges are untouched.

Value

The function will return a new CRF with additional components:

`original`	The original CRF data.
`node.id`	The vector of the original node ids for nodes in the new CRF.
`node.map`	The vector of the new node ids for nodes in the original CRF.
`edge.id`	The vector of the original edge ids for edges in the new CRF.
`edge.map`	The vector of the new edge ids for edges in the original CRF.

See Also

`make.crf`, `clamp.crf`

Examples

```
library(CRF)
data(Small)
crf <- sub.crf(Small$crf, c(2, 3))
```

`train.crf` *Train CRF model*

Description

Train the CRF model to estimate the parameters

Usage

```
train.crf(crf, instances, node.fea = NaN, edge.fea = NaN, node.ext = NaN,
  edge.ext = NaN, nll = crf.nll, trace = 0)
```

Arguments

`crf`	The CRF
`instances`	The training data matrix of CRF model
`node.fea`	The list of node features
`edge.fea`	The list of edge features
`node.ext`	The list of extended information of node features
`edge.ext`	The list of extended information of edge features
`nll`	The function to calculate negative log likelihood
`trace`	Non-negative integer to control the tracing informtion of the optimization process

Details

This function train the CRF model.

In the training data matrix `instances`, each row is an instance and each column corresponds a node in CRF. The variables `node.fea`, `edge.fea`, `node.ext`, `edge.ext` are lists of length equal to the number of instances, and their elements are defined as in `crf.update` respectively.

Value

This function will directly modify the CRF and return the same CRF.

See Also

`crf.update`, `crf.nll`, `make.crf`

`train.mrf` *Train MRF model*

Description

Train the MRF model to estimate the parameters

Usage

```
train.mrf(crf, instances, nll = mrf.nll, trace = 0)
```

Arguments

`crf`	The CRF
`instances`	The training data matrix of CRF model
`nll`	The function to calculate negative log likelihood
`trace`	Non-negative integer to control the tracing informtion of the optimization process

Details

This function trains the Markov Random Fields (MRF) model, which is a simple variant of CRF model.

In the training data matrix `instances`, each row is an instance and each column corresponds a node in CRF.

Value

This function will directly modify the CRF and return the same CRF.

See Also

`mrf.update`, `mrf.stat`, `mrf.nll`, `make.crf`

`Tree` *Tree CRF example*

Description

This data set gives a tree CRF example

Usage

```
data(Tree)
```

Format

A list containing two elements:

- `crf` The CRF
- `answer` A list of 4 elements:
  - decode The most likely configuration
  - `node.bel` The node belief
  - `edge.bel` The edge belief
  - `logZ` The logarithmic value of CRF normalization factor Z

Index