Package ‘CRF’

March 4, 2019

Version 0.4-2
Title Conditional Random Fields
Description Implements modeling and computational tools for conditional random fields (CRF) model as well as other probabilistic undirected graphical models of discrete data with pairwise and unary potentials.

ByteCompile TRUE
Depends R (>= 3.0.0)
Imports Matrix
Suggests knitr, rmarkdown, Rglpk
VignetteBuilder knitr
License GPL (>= 2)

BugReports https://github.com/wulingyun/crf/issues
URL https://github.com/wulingyun/crf
RoxygenNote 6.1.1
Encoding UTF-8
Author Ling-Yun Wu [aut, cre]
Maintainer Ling-Yun Wu <wulingyun@gmail.com>
Repository CRAN
Repository/R-Forge/Project crf
Repository/R-Forge/Revision 50
Repository/R-Forge/DateTimeStamp 2019-03-04 05:00:34
Date/Publication 2019-03-04 10:20:02 UTC
NeedsCompilation yes
### R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRF-package</td>
<td>3</td>
</tr>
<tr>
<td>Chain</td>
<td>5</td>
</tr>
<tr>
<td>clamp.crf</td>
<td>6</td>
</tr>
<tr>
<td>clamp.reset</td>
<td>7</td>
</tr>
<tr>
<td>Clique</td>
<td>8</td>
</tr>
<tr>
<td>crf.nll</td>
<td>8</td>
</tr>
<tr>
<td>crf.update</td>
<td>9</td>
</tr>
<tr>
<td>decode.block</td>
<td>10</td>
</tr>
<tr>
<td>decode.chain</td>
<td>11</td>
</tr>
<tr>
<td>decode.conditional</td>
<td>12</td>
</tr>
<tr>
<td>decode.cutset</td>
<td>12</td>
</tr>
<tr>
<td>decode.exact</td>
<td>13</td>
</tr>
<tr>
<td>decode.greedy</td>
<td>14</td>
</tr>
<tr>
<td>decode.icm</td>
<td>15</td>
</tr>
<tr>
<td>decode.ilp</td>
<td>15</td>
</tr>
<tr>
<td>decode.junction</td>
<td>16</td>
</tr>
<tr>
<td>decode.lbp</td>
<td>17</td>
</tr>
<tr>
<td>decode.marginal</td>
<td>18</td>
</tr>
<tr>
<td>decode.rbp</td>
<td>18</td>
</tr>
<tr>
<td>decode.sample</td>
<td>19</td>
</tr>
<tr>
<td>decode.trbp</td>
<td>20</td>
</tr>
<tr>
<td>decode.tree</td>
<td>21</td>
</tr>
<tr>
<td>duplicate.crf</td>
<td>21</td>
</tr>
<tr>
<td>get.logPotential</td>
<td>22</td>
</tr>
<tr>
<td>get.potential</td>
<td>23</td>
</tr>
<tr>
<td>infer.chain</td>
<td>23</td>
</tr>
<tr>
<td>infer.conditional</td>
<td>24</td>
</tr>
<tr>
<td>infer.cutset</td>
<td>25</td>
</tr>
<tr>
<td>infer.exact</td>
<td>26</td>
</tr>
<tr>
<td>infer.junction</td>
<td>27</td>
</tr>
<tr>
<td>infer.lbp</td>
<td>28</td>
</tr>
<tr>
<td>infer.rbp</td>
<td>29</td>
</tr>
<tr>
<td>infer.sample</td>
<td>30</td>
</tr>
<tr>
<td>infer.trbp</td>
<td>31</td>
</tr>
<tr>
<td>infer.tree</td>
<td>32</td>
</tr>
<tr>
<td>Loop</td>
<td>33</td>
</tr>
<tr>
<td>make.crf</td>
<td>33</td>
</tr>
<tr>
<td>make.features</td>
<td>35</td>
</tr>
<tr>
<td>make.par</td>
<td>36</td>
</tr>
<tr>
<td>mrf.nll</td>
<td>36</td>
</tr>
<tr>
<td>mrf.stat</td>
<td>37</td>
</tr>
<tr>
<td>mrf.update</td>
<td>38</td>
</tr>
<tr>
<td>Rain</td>
<td>39</td>
</tr>
<tr>
<td>sample.chain</td>
<td>39</td>
</tr>
<tr>
<td>sample.conditional</td>
<td>40</td>
</tr>
<tr>
<td>sample.cutset</td>
<td>41</td>
</tr>
</tbody>
</table>
### Description

Library of Conditional Random Fields model

### 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 propagation
- `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 decoding 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

Tools: Tools for building and manipulating CRF data

• **make.crf** Generate CRF from the adjacent matrix
• **make.features** Make the data structure of CRF features
• **make.par** Make the data structure of CRF parameters
• **duplicate.crf** Duplicate an existing CRF
• **clamp.crf** Generate clamped CRF by fixing the states of some nodes
• **clamp.reset** Reset clamped CRF by changing the states of clamped nodes
• **sub.crf** Generate sub CRF by selecting some nodes
• **mrf.update** Update node and edge potentials of MRF model
• **crf.update** Update node and edge potentials of CRF model
**Author(s)**

Ling-Yun Wu <wulingyun@gmail.com>

**References**


**Examples**

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

---

**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 = NULL, edge.fea = NULL,
          node.ext = NULL, edge.ext = NULL, infer.method = infer.chain, ...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>par</td>
<td>The parameter vector of CRF</td>
</tr>
<tr>
<td>crf</td>
<td>The CRF</td>
</tr>
<tr>
<td>instances</td>
<td>The training data matrix of CRF model</td>
</tr>
<tr>
<td>node.fea</td>
<td>The list of node features</td>
</tr>
<tr>
<td>edge.fea</td>
<td>The list of edge features</td>
</tr>
<tr>
<td>node.ext</td>
<td>The list of extended information of node features</td>
</tr>
<tr>
<td>edge.ext</td>
<td>The list of extended information of edge features</td>
</tr>
<tr>
<td>infer.method</td>
<td>The inference method used to compute the likelihood</td>
</tr>
<tr>
<td>...</td>
<td>Extra parameters need by the inference method</td>
</tr>
</tbody>
</table>
**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`

---

**Description**

Update node and edge potentials of CRF model

**Usage**

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

**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 straightforward 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:

\[
node.pot[n, i] = \exp(\sum_f \text{par}[node.par[n, i, f]] * node.fea[f, n] + \sum_k \text{par}[k] * node.ext[[k]][n, i])
\]

and the edge potential is updated as follows:

\[
edge.pot[[e]][i, j] = \exp(\sum_f \text{par}[edge.par[[e]][i, j, f]] * edge.fea[f, e] + \sum_k \text{par}[k] * edge.ext[[k]][[e]][i, j])
\]

**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 searching 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

```r
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

```r
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

```r
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**

```r
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 searching 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

```r
library(crf)
data(small)
d <- decode.cutset(small$crf, c(R))
```

---

**decode.exact**  
*Decoding method for small graphs*

Description

Computing the most likely configuration for CRF

Usage

```r
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

```r
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**

```r
deck.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 searching 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**

```r
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 searching 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

```r
## Not run:
library(CRF)
data(Small)
d <- decode.ilp(Small$crf)
## End(Not run)
```

---

`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`. 
decode.lbp

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

de 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 information 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.rbp

*Decoding method using residual belief propagation*

**Description**

Computing the most likely configuration for CRF

**Usage**

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

**Arguments**

- `crf`: The CRF
- `max.iter`: Maximum number of iterations
- `cutoff`: Cutoff for convergence
- `verbose`: verbosity

**Details**

- Computes the most likely configuration using residual belief propagation

**Value**

- Returns the most likely configuration, a vector of length `crf$n.nodes`
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 information in algorithm

Details

Approximate decoding using max-product residual belief propagation

Value

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

Examples

```r
library(crf)
data(Small)
d <- decode.rbp(Small$crf)
```

---

**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

```r
library(crf)
data(Small)
d <- decode.sample(Small$crfL sampleNexactL 10000)
```

---

decode.trbp Decoding method using tree-rewighted belief propagation

Description

Computing the most likely configuration for CRF

Usage

```r
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 information in algorithm

Details

Approximate decoding using max-product tree-rewighted belief propagation

Value

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

Examples

```r
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**

```r
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**

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

---

**duplicate.crf**

*Duplicate CRF*

**Description**

Duplicate an existing CRF

**Usage**

```r
duplicate.crf(crf)
```

**Arguments**

- **crf**  
The existing CRF
get.logPotential

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

generateLogPotential

Description
Calculate the logarithmic potential of a CRF with given configuration

Usage
generateLogPotential(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
generatePotential
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

```r
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

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

**infer.cutset**

*Inference method for graphs with a small cutset*

**Description**

Computing the partition function and marginal probabilities

**Usage**

```r
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**

```r
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

\hspace{1cm} \textit{Inference method for low-treewidth graphs}

\textbf{Description}
Computing the partition function and marginal probabilities

\textbf{Usage}
\begin{verbatim}
infer.junction(crf)
\end{verbatim}

\textbf{Arguments}
crf The CRF

\textbf{Details}
Exact decoding for low-treewidth graphs using junction trees

\textbf{Value}
This function will return a list with components:

node.bel Node belief. It is a matrix with \texttt{crf$n$nodes} rows and \texttt{crf$\max$state} columns.

edge.bel Edge belief. It is a list of matrices. The size of list is \texttt{crf$n$edges} and the matrix \texttt{i} has \texttt{crf$n$states[crf$edges[i,1]$]} rows and \texttt{crf$n$states[crf$edges[i,2]$]} columns.

logZ The logarithmic value of CRF normalization factor Z.

\textbf{Examples}
\begin{verbatim}
library(CRF)
data(Small)
i <- infer.junction(Small$crf)
\end{verbatim}
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, maximize = FALSE)
```

**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 information in algorithm
- `maximize`: Logical variable to indicate using max-product instead of sum-product

**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**

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

Inference method using residual belief propagation

Description
Computing the partition function and marginal probabilities

Usage
infer.rbp(crf, max.iter = 10000, cutoff = 1e-04, verbose = 0, maximize = FALSE)

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 information in algorithm
- **maximize**: Logical variable to indicate using max-product instead of sum-product

Details
Approximate inference using sum-product residual 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

```r
library(CRF)
data(Small)
i <- infer.rbp(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**
```r
infer.trbp(crf, max.iter = 10000, cutoff = 1e-04, verbose = 0,
maximize = FALSE)
```

**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 information in algorithm
- `maximize`: Logical variable to indicate using max-product instead of sum-product

**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**
```r
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 \(\text{crf$n$nodes}\) rows and \(\text{crf$max.state}\) columns.
- **edge.bel** Edge belief. It is a list of matrices. The size of list is \(\text{crf$n$edges}\) and the matrix \(i\) has \(\text{crf$n.states[crf$edges[i,1]]}\) rows and \(\text{crf$n.states[crf$edges[i,2]]}\) 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**

```r
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**

```r
make.crf(adj.matrix = NULL, n.states = 2, n.nodes = 2)
```

**Arguments**

- **adj.matrix**: The adjacent matrix of CRF network.
- **n.states**: The state numbers of nodes.
- **n.nodes**: The number of nodes, which is only used to generate linear chain CRF when `adj.matrix` is NULL.
Details

The function will generate an empty CRF from a given adjacent matrix. If the length of \texttt{nstates} is less than \texttt{n.nodes}, it will be used repeatedly. All node and edge potentials are initialized 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 \texttt{duplicate.crf}.

Value

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

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

See Also

\texttt{duplicate.crf}, \texttt{clamp.crf}, \texttt{sub.crf}

Examples

```r
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
}
```
make.features <- 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 CRF features

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crf</td>
<td>The CRF</td>
</tr>
<tr>
<td>n.nf</td>
<td>The number of node features</td>
</tr>
<tr>
<td>n.ef</td>
<td>The number of edge features</td>
</tr>
</tbody>
</table>

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 CRF parameters

**Usage**

```r
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**

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

- `par` : The parameter vector of CRF
- `crf` : The CRF
- `instances` : The training data matrix of MRF model
- `infer.method` : The inference method used to compute the likelihood
- `...` : Extra 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-likelihood.

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.
mrf.update

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 and edge potentials 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

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


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
sample.conditional

Value

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

Examples

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

---

sample.conditional  Conditional sampling method

Description

Generating samples from the distribution

Usage

```r
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

```r
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$nodes` columns, in which each row is a sampled configuration.

### Examples
```r
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**

```r
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))
sample.junction

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crf</td>
<td>The CRF</td>
</tr>
<tr>
<td>size</td>
<td>The sample size</td>
</tr>
<tr>
<td>burn.in</td>
<td>The number of samples at the beginning that will be discarded</td>
</tr>
<tr>
<td>start</td>
<td>An initial configuration</td>
</tr>
</tbody>
</table>

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

```r
library(crf)
data(small)
s <- sample.junction(small$crf, 100)
```

Description

Generating samples from the distribution

Usage

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crf</td>
<td>The CRF</td>
</tr>
<tr>
<td>size</td>
<td>The sample size</td>
</tr>
</tbody>
</table>

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

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

**Description**

Generating samples from the distribution

**Usage**

```r
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**

```r
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 remaining nodes and edges are untouched.
### train.crf

**Train CRF model**

Train the CRF model to estimate the parameters.

#### Usage

```r
train.crf(crf, instances, node.fea = NULL, edge.fea = NULL,
          node.ext = NULL, edge.ext = NULL, nll = crf.nll,
          infer.method = infer.chain, ..., 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
- `infer.method` The inference method used to compute the likelihood

---

**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

```r
library(CRF)
data(Small)

# Create a CRF

crf <- sub.crf(Small$crf, c(2, 3))
```
... Extra parameters need by the inference method
trace Non-negative integer to control the tracing information 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, infer.method = infer.chain,
... \trace = 0)

Arguments
crf The CRF
instances The training data matrix of CRF model
nll The function to calculate negative log likelihood
infer.method The inference method used to compute the likelihood
... Extra parameters need by the inference method
trace Non-negative integer to control the tracing information 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

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

*Topic **datasets**
  Chain, 5
  Clique, 8
  Loop, 33
  Rain, 39
  Small, 45
  Tree, 48

*Topic **package**
  CRF-package, 3

Chain, 5
clamp.crf, 4, 6, 7, 34, 45, 46
clamp.reset, 4, 6, 7
Clique, 8
CRF (CRF-package), 3
CRF-package, 3
crf.nll, 8, 10, 47
crf.update, 4, 9, 9, 35, 36, 47
decode.block, 3, 10
decode.chain, 3, 11
decode.conditional, 3, 12
decode.cutset, 3, 12
decode.exact, 3, 13
decode.greedy, 3, 14
decode.icm, 3, 15
decode.ilp, 4, 15
decode.junction, 3, 16
decode.lbp, 3, 17
decode.marginal, 3, 18
decode.rbp, 18
decode.sample, 3, 19
decode.trbp, 3, 20
decode.tree, 3, 21
duplicate.crf, 4, 21, 34
get.logPotential, 22, 23
get.potential, 22, 23
infer.chain, 4, 23
infer.conditional, 4, 24
infer.cutset, 4, 25
infer.exact, 4, 26
infer.junction, 4, 27
infer.lbp, 4, 28
infer.rbp, 29
infer.sample, 4, 30
infer.trbp, 4, 31
infer.tree, 4, 32

Loop, 33
make.crf, 4, 6, 7, 22, 33, 36, 45–48
make.features, 4, 35, 36
make.par, 4, 36
mrf.nll, 36, 37, 38, 48
mrf.stat, 37, 37, 48
mrf.update, 4, 37, 38, 48

Rain, 39
sample.chain, 4, 39
sample.conditional, 4, 40
sample.cutset, 4, 41
sample.exact, 4, 42
sample.gibbs, 4, 42
sample.junction, 4, 43
sample.tree, 4, 44
Small, 45
sub.crf, 4, 6, 34, 45

train.crf, 4, 9, 10, 46
train.mrf, 4, 37, 38, 47
Tree, 48