arXiv:1404.4893v1 [cs.AI] 18 Apr 2014

# CTBNCToolkit: Continuous Time Bayesian Network Classifier Toolkit

Daniele Codecasa ‡
codecasa@disco.unimib.it
DISCo, Università degli Studi di Milano-Bicocca

Fabio Stella ‡
stella@disco.unimib.it
DISCo, Università degli Studi di Milano-Bicocca

**Abstract**

Continuous time Bayesian network classifiers are designed for temporal classification of multivariate streaming data when time duration of events matters and the class does not change over time. This paper introduces the CTBNCToolkit: an open source Java toolkit which provides a stand-alone application for temporal classification and a library for continuous time Bayesian network classifiers. CTBNCToolkit implements the inference algorithm, the parameter learning algorithm, and the structural learning algorithm for continuous time Bayesian network classifiers. The structural learning algorithm is based on scoring functions: the marginal log-likelihood score and the conditional log-likelihood score are provided. CTBNCToolkit provides also an implementation of the expectation maximization algorithm for clustering purpose. The paper introduces continuous time Bayesian network classifiers. How to use the CTBNToolkit from the command line is described in a specific section. Tutorial examples are included to facilitate users to understand how the toolkit must be used. A section dedicate to the Java library is proposed to help further code extensions.

## ‡ Authors' contributions

The toolkit was developed by Daniele Codecasa, who also wrote the paper. Fabio Stella read the paper and made valuable suggestions. A former MATLAB implementation of the CTBNC inference algorithm was made available by Fabio Stella in order to test the correctness of the inference using the CTBNCToolkit[1].

[1] CTBNCToolkit website and the repository will be updated soon.

# 1 Introduction

The relevance of streaming data is increasing year after year with the emerging of new sources of data. Data streams are important in engineering, with reference to image, audio and video processing (Yilmaz *et al.*, 2006) and in *computer science*, with reference to system error logs, web search query logs, network intrusion detection, and social networks (Simma and Jordan, 2010). In finance, data streams are deeply studied for high frequency trading (Dacorogna, 2001). Biology and medicine offer other examples of analysis of data that change over time. Biologists analyze streaming data to model the evolution of infections (Barber and Cemgil, 2010) and to learn and analyze metabolic networks (Voit, 2012). While in medicine streaming data are used in patient monitoring based on sensor data and in continuous time diagnosis, including the study of computational firing pattern of neurons (Truccolo *et al.*, 2005).

Data streaming problems may be approached with many algorithms and models. Among these, Dynamic Bayesian Networks (DBNs)(Dean and Kanazawa, 1989) and Hidden Markov Models (HMMs) (Rabiner, 1989) have received great attention for modeling temporal dependencies. However, DBNs are concerned with discrete time and thus suffer from several limitations due to the fact that it is not clear how timestamps should be discretized. In the case where a too slow sampling rate is used the data will be poorly represented, while a too fast sampling rate rapidly makes learning and inference prohibitive. Furthermore, it has been pointed out (Gunawardana *et al.*, 2011) that when allowing long term dependencies, it is necessary to condition on multiple steps into the past. Thus the choice of a too fast sampling rate will increase the number of such steps that need to be conditioned on. Continuous Time Bayesian Networks (CTBNs) (Nodelman *et al.*, 2005), Continuous Time Noisy-OR (CT-NOR) (Simma *et al.*, 2008), Poisson cascades (Simma and Jordan, 2010) and Poisson networks (Rajaram *et al.*, 2005) together with the Piecewise-constant Conditional Intensity Model (PCIM) (Gunawardana *et al.*, 2011) are interesting models to represent and analyze continuous time processes. CT-NOR and Poisson cascades are devoted to model event streams while they require the modeler to specify a parametric form for temporal dependencies. This aspect significantly impacts performance, and the problem of model selection in CT-NOR and Poisson cascades has not been addressed yet. This limitation is overcome by PCIMs which perform structure learning to model how events in the past affect future events of interest. CTBNs are continuous time homogeneous Markov processes which allow to represent joint trajectories of discrete finite variables, rather than models of event streams in continuous time.

Classification of data stream is an important area in data stream analysis with applications in many domains, such as medicine where the classification of multivariate trajectories is used for gesture recognition purpose related to the post-stroke rehabilitation (Tormene *et al.*, 2009). In this paper the problem of *temporal classification*, where data stream measurements are available over a period of time in history while the class is expected to occur in the future, is considered. Temporal classification can be addressed by discrete and continuous time models. Discrete time models include dynamic latent classification models (Zhong *et al.*, 2012), a specialization of the latent classification model (LCM) (Langseth and Nielsen, 2005), and DBNs (Dean and Kanazawa, 1989). Continuous time models, as Continuous Time Bayesian Network Classifiers (CTBNCs) (Stella and Amer, 2012; Codecasa and Stella, 2013), overcame the problem of timestamps discretization.

This work describes the CTBNCToolkit: an open source Java toolkit which allows temporal classification and clustering using the CTBNCs introduced by Codecasa and Stella (2013); Codecasa (2014). CTBNCToolkit is a Java library for CTBNCs which can be used as a stand-alone application through the command line interface. CTBNCToolkit can be used for scientific purposes, such as model comparison and temporal classification of interesting scientific problems, but it can be used as well as a prototype to address real world problems.

CTBNCToolkit provides:

- the CTBNC inference algorithm (Stella and Amer, 2012);
- the CTBNC parameter learning algorithm (Stella and Amer, 2012);
- the scoring function based structural learning algorithm for CTBNCs (Codecasa and Stella, 2013);
- two scoring functions: the marginal log-likelihood score and the conditional log-likelihood score (Codecasa and Stella, 2013);
- the expectation maximization algorithm with soft and hard assignment for clustering purposes (Codecasa, 2014);
- three different validation methods: hold out, cross validation and a validation method for the clustering;
- an extended set of performance measures for the supervised classification (see Section 3.3);
- an extended set of external performance measures for clustering evaluation (see Section 3.3);
- an extendable command line interface (see Section 3.2).

Section 2 describes the basic notions about CTBNCs. A guide along the stand-alone usage of CTBNCToolkit is provided in Section 3, which explains how to download the library (Section 3.1), how to use the command line interface (Section 3.2), and how to read the classification and the clustering results (Section 3.3). Section 4 provides tutorial examples of CTBNCtoolkit command line usage. All the examples are replicable. The structure of the Java library and the details of the implementation are addressed in Section 5, while Section 6 contains the conclusions and the possible future steps.

# 2 Basic notions

## 2.1 Continuous time Bayesian networks

Dynamic Bayesian networks (DBNs) model dynamical systems discretizing the time through several time slices. Nodelman *et al.* (2002a) pointed out that "*since DBNs slice time into fixed increments, one must always propagate the joint distribution over the variables at the same rate*" . Therefore, if the system consists of processes which evolve at different time granularities and/or the obtained observations are irregularly spaced in time, the inference process may become computationally intractable.

Continuous time Bayesian networks (CTBNs) overcome the limitations of DBNs by explicitly representing temporal dynamics using exponential distributions. Continuous time Bayesian networks (CTBNs) exploit the conditional independencies in continuous time Markov processes. A continuous time Bayesian network (CTBN) is a probabilistic graphical model whose nodes are associated with random variables and whose state evolves continuously over time.

**Definition 2.1.** (Continuous time Bayesian network). (Nodelman *et al.*, 2002a). Let $\mathbf{X}$ be a set of random variables $X_1, X_2, ..., X_N$. Each $X_n$ has a finite domain of values $Val(X_n) = \{x_1, x_2, ..., x_I\}$. A continuous time Bayesian network $\aleph$ over $\mathbf{X}$ consists of two components: the first is an initial distribution $P^0_{\mathbf{X}}$, specified as a Bayesian network $\mathcal{B}$ over $\mathbf{X}$. The second is a continuous transition model, specified as:

- a directed (possibly cyclic) graph $\mathcal{G}$ whose nodes are $X_1, X_2, ..., X_N$; $Pa(X_n)$ denotes the parents of $X_n$ in $\mathcal{G}$.
- a conditional intensity matrix, $\mathbf{Q}^{Pa(X_n)}_{X_n}$, for each variable $X_n \in \mathbf{X}$.

Given the random variable $X_n$, the *conditional intensity matrix* (CIM) $\mathbf{Q}^{Pa(X_n)}_{X_n}$ consists of a set of intensity matrices, one intensity matrix

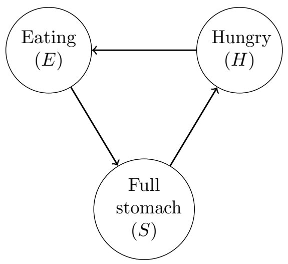
$$\mathbf{Q}^{pa(X_n)}_{X_n} = \begin{bmatrix} -q^{pa(X_n)}_{x_1} & q^{pa(X_n)}_{x_1x_2} & . & q^{pa(X_n)}_{x_1x_I} \\ q^{pa(X_n)}_{x_2x_1} & -q^{pa(X_n)}_{x_2} & . & q^{pa(X_n)}_{x_2x_I} \\ . & . & . & . \\ q^{pa(X_n)}_{x_Ix_1} & q^{pa(X_n)}_{x_Ix_2} & . & -q^{pa(X_n)}_{x_I} \end{bmatrix},$$

for each instantiation $pa(X_n)$ of the parents $Pa(X_n)$ of node $X_n$, where $q^{pa(X_n)}_{x_i} = \sum\limits_{x_j \neq x_i} q^{pa(X_n)}_{x_ix_j}$ is the rate of leaving state $x_i$ for a specific instantiation $pa(X_n)$ of $Pa(X_n)$, while $q^{pa(X_n)}_{x_ix_j}$ is the rate of arriving to state $x_j$ from state $x_i$ for a specific instantiation $pa(X_n)$ of $Pa(X_n)$. Matrix $\mathbf{Q}^{pa(X_n)}_{X_n}$ can equivalently be summarized by using two types of parameters, $q^{pa(X_n)}_{x_i}$ which is associated with each state $x_i$ of the variable $X_n$ when its' parents are set to $pa(X_n)$, and

$\theta_{x_i x_j}^{pa(X_n)} = \frac{q_{x_i x_j}^{pa(X_n)}}{q_{x_i}^{pa(X_n)}}$ which represents the probability of transitioning from state $x_i$ to state $x_j$, when it is known that the transition occurs at a given instant in time.

**Example 2.1.** Figure 1 shows a part of the drug network introduced in Nodelman *et al.* (2002a). It contains a cycle, indicating that whether a person is hungry ($H$) depends on how full his/her stomach ($S$) is, which depends on whether or not he/she is eating ($E$), which in turn depends on whether he/she is hungry.

Assume that $E$ and $H$ are binary variables with states *no* and *yes* while the variable $S$ can be in one of the following states; *full*, *average* or *empty*. Then, the variable $E$ is fully specified by the [2×2] CIM matrices $\mathbf{Q}_E^n$, and $\mathbf{Q}_E^y$, the variable $S$ is fully specified by the [3×3] CIM matrices $\mathbf{Q}_S^n$ and $\mathbf{Q}_S^y$, while the the variable $H$ is fully specified by the [2×2] CIM matrices $\mathbf{Q}_H^f$, $\mathbf{Q}_H^a$ and, $\mathbf{Q}_H^e$.



$$\begin{aligned}\mathbf{Q}_S^y &= \begin{bmatrix} -q_f^y & q_{f,a}^y & q_{f,e}^y \\ q_{a,f}^y & -q_a^y & q_{a,e}^y \\ q_{e,f}^y & q_{e,a}^y & -q_e^y \end{bmatrix} \\ &= \begin{bmatrix} -0.03 & 0.02 & 0.01 \\ 5.99 & -6.00 & 0.01 \\ 1.00 & 5.00 & -6.00 \end{bmatrix} \quad (1)\end{aligned}$$

$$\begin{aligned}\mathbf{Q}_S^y &= \begin{bmatrix} q_f^y & 0 & 0 \\ 0 & q_a^y & 0 \\ 0 & 0 & q_e^y \end{bmatrix} \left( \begin{bmatrix} 0 & \theta_{f,a}^y & \theta_{f,e}^y \\ \theta_{a,f}^y & 0 & \theta_{a,e}^y \\ \theta_{e,f}^y & \theta_{e,a}^y & 0 \end{bmatrix} - \mathbf{I} \right) \\ &= \begin{bmatrix} 0.03 & 0 & 0 \\ 0 & 6.00 & 0 \\ 0 & 0 & 6.00 \end{bmatrix} \left( \begin{bmatrix} 0 & \frac{0.02}{0.03} & \frac{0.01}{0.03} \\ \frac{5.99}{6.00} & 0 & \frac{0.01}{6.00} \\ \frac{1.00}{6.00} & \frac{5.00}{6.00} & 0 \end{bmatrix} - \mathbf{I} \right) \quad (2)\end{aligned}$$

Figure 1: A part of the drug network and the two equivalent parametric representations of $\mathbf{Q}_S^y$ where $\mathbf{I}$ is the identity matrix.

If the hours are the units of time, then a person who has an empty stomach ($S$=*empty*) and is eating ($E$=*yes*) is expected to stop having an empty stomach in 10 minutes ($\frac{1.00}{6.00}$ hour). The stomach will then transition from state *empty* ($S$=*empty*) to state *average* ($S$=*average*) with probability $\frac{5.00}{6.00}$ and to state $full$ ($S$=*full*) with probability $\frac{1.00}{6.00}$. Equation 1 is a compact representation of the CIM while Equation 2 is useful because it explicitly represents the transition probability value from state $x$ to state $x'$, i.e. $\theta_{xx'}^{pa(X)}$.

## 2.2 Continuous time Bayesian network classifiers

Continuous time Bayesian network classifiers (CTBNCs) (Stella and Amer, 2012) are a specialization of CTBNs. CTBNCs allow polynomial time classification of a class variable that does not change over time, while general inference for CTBNs is NP-hard. Classifiers from this class explicitly represent the evolution in continuous time of the set of random variables $X_n$, $n = 1, 2, ..., N$ which are assumed to depend on the static class node $Y$. A continuous time Bayesian network classifier (CTBNC) is defined as follows.

**Definition 2.2.** (Continuous time Bayesian network classifier). (Codecasa and Stella, 2013). A continuous time Bayesian network classifier is a pair $\mathcal{C} = \{\aleph, P(Y)\}$ where $\aleph$ is a CTBN model with attribute nodes $X_1, X_2, ..., X_N$, $Y$ is the class node with marginal probability $P(Y)$ on states $Val(Y) = \{y_1, y_2, ..., y_K\}$, $\mathcal{G}$ is the graph of the CTBNC, such that the following conditions hold:

- $Pa(Y) = \emptyset$, the class variable $Y$ is associated with a root node;
- $Y$ is fully specified by $P(Y)$ and does not depend on time.

**Example 2.2.** Figure 2a depicts the structure of a CTBNC to diagnose eating disorders from the eating process (Figure 1). An example of the eating process is shown in Figure 2b.

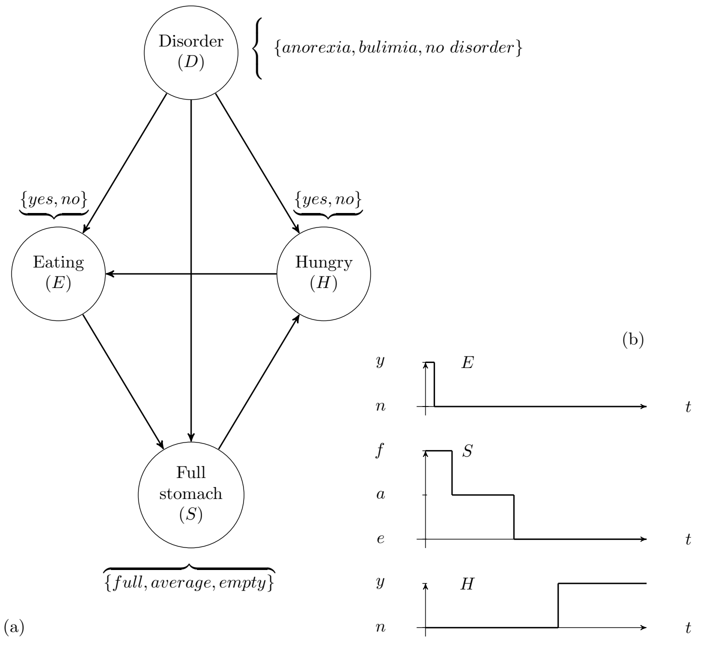


Figure 2: CTBNC to diagnose eating disorders (a) observing the eating process (b).

CTBNCs have been successfully used by Codecasa and Stella (2013); Codecasa (2014) in gesture recognition for post-stroke rehabilitation purposes (Tormene *et al.*, 2009).

### 2.2.1 Models

Stella and Amer (2012) defined the continuous time naive Bayes to overcome the effort of learning the structure of a CTBNC.

**Definition 2.3.** (Continuous time naive Bayes classifier). (Stella and Amer, 2012). A continuous time naive Bayes classifier is a continuous time Bayesian network classifier $\mathcal{C} = \{\aleph, P(Y)\}$ such that $Pa(X_n) = \{Y\}$, $n = 1, 2, ..., N$.

While Codecasa and Stella (2013) addressed for the first time the structural learning problem for CTBNCs defining two classifiers where the maximum number of parents (i.e. $k$) is bounded.

**Definition 2.4.** (Max-$k$ Continuous Time Bayesian Network Classifier). (Codecasa and Stella, 2013). A max-$k$ continuous time Bayesian network classifier is a couple $\mathcal{M} = \{\mathcal{C}, k\}$, where $\mathcal{C}$ is a continuous time Bayesian network classifier $\mathcal{C} = \{\aleph, P(Y)\}$ such that the number of parents $|Pa(X_n)|$ for each attribute node $X_n$ is bounded by a positive integer $k$. Formally, the following condition holds; $|Pa(X_n)| \leq k$, $n = 1, 2, ..., N$, $k \geq 0$.

**Definition 2.5.** (Max-$k$ Augmented Continuous Time Naive Bayes). (Codecasa and Stella, 2013). A max-$k$ augmented continuous time naive Bayes classifier is a max-$k$ continuous time Bayesian network classifier such that the class node $Y$ belongs to the parents set of each attribute node $X_n$, $n = 1, 2, ..., N$. Formally, the following condition holds; $Y \in Pa(X_n)$, $n = 1, 2, ..., N$.

### 2.2.2 Parameter learning

CTBNC parameter learning corresponds to CTBN parameter learning (Nodelman *et al.*, 2002b) with the exception of the class variable which has to be taken in account (Stella and Amer, 2012; Codecasa and Stella, 2013).

Given a data set $\mathcal{D}$ and a fixed structure of a CTBNC, parameter learning is based on *marginal log-likelihood estimation.* Parameter learning accounts for the *imaginary counts* of the hyperparameters $\alpha_x^{pa(X)}$, $\alpha_{xx'}^{pa(X)}$ and, $\tau_x^{pa(X)}$. The parameters $q_x^{pa(X)}$ and $\theta_{xx'}^{pa(X_n)}$ can be estimated as follows:

- $q_x^{pa(X)} = \frac{\alpha_x^{pa(X)} + M[x|pa(X)]}{\tau_x^{pa(X)} + T[x|pa(X)]}$;
- $\theta_{xx'}^{pa(X)} = \frac{\alpha_{xx'}^{pa(X)} + M[x,x'|pa(X)]}{\alpha_x^{pa(X)} + M[x|pa(X)]}$.

where $M[x, x' \mid pa(X)]$, $M[x \mid pa(X)]$ and $T[x \mid pa(X)]$ are the *sufficient statistics* computed over $\mathcal{D}$:

- $M[x, x' \mid pa(X)]$: number of times $X$ transitions from state $x$ to state $x'$ when the state of its parents $Pa(X)$ is set to $pa(X)$;
- $M[x \mid pa(X)] = \sum_{x' \neq x} M[x, x' \mid pa(X)]$: number of transitions from state $x$ of variable $X$ when the state of its parents $Pa(X)$ is set to $pa(X)$;

- $T[x \mid pa(X)]$: amount of time spent in state $x$ by variable $X$ when the state of its parents $Pa(X)$ is set to $pa(X)$.

The only difference between CTBNC parameter learning and CTBN parameter learning is related to the necessity of learning the probability distribution over the class node. Because the class node is a static node, this can be done easily as follows:

$$\theta_y = \frac{\alpha_y + M[y]}{\sum_{y'} \alpha_{y'} + M[y']}$$

where $M[y]$ is the number of trajectories in the training set with class $Y$ sets to $y$ and, $\alpha_y$ are the imaginary counts related to the class variable.

### 2.2.3 Structural learning

Learning a CTBNC from data consists of learning a CTBN where a specific node, i.e. the class node $Y$, does not depend on time. In such a case, the learning algorithm runs, for each attribute node $X_n$, $n = 1, 2, ..., N$, a local search procedure to find its optimal set of parents, i.e. the set of parents which maximizes a given score function. Furthermore, for each attribute node $X_n$, $n = 1, 2, ..., N$, no more than $k$ parents are selected.

Codecasa and Stella (2013) learned CTBNCs using the same local search algorithm proposed by Nodelman *et al.* (2002b) for CTBNs. Codecasa and Stella (2013) proposed two possible scores: a marginal log-likelihood score, corresponding to the score introduced by Nodelman *et al.* (2002b), and a conditional log-likelihood score, especially designed to improve the classification performances. Because no closed form solution exists to compute the optimal value of the model's parameters when the conditional log-likelihood is used, Codecasa and Stella (2013) followed the approach introduced and discussed by Grossman and Domingos (2004) for static Bayesian classifiers. The scoring function is computed by using the conditional log-likelihood, while parameters values are obtained by using the Bayesian approach as described by Nodelman *et al.* (2002b).

The structural learning algorithm for CTBNCs and the conditional log-likelihood scoring function are described in (Codecasa and Stella, 2013).

### 2.2.4 Classification

According to Stella and Amer (2012) a CTBNC $\mathcal{C} = \{\aleph, P(Y)\}$ classifies a stream of continuous time evidence $\mathbf{z} = (x_1, x_2, ..., x_N)$ for the attributes $\mathbf{Z} = (X_1, X_2, ..., X_N)$ over $J$ contiguous time intervals, i.e. a stream of continuous time evidence $\mathbf{Z}^{[t_1,t_2)} = \mathbf{z}^{[t_1,t_2)}$, $\mathbf{Z}^{[t_2,t_3)} = \mathbf{z}^{[t_2,t_3)}$, $\ldots$, $\mathbf{Z}^{[t_{J-1},t_J)} = \mathbf{z}^{[t_{J-1},t_J)}$, by selecting the value $y^*$ for the class $Y$ which maximizes the posterior probability

$$P(Y|\mathbf{z}^{[t_1,t_2)}, \mathbf{z}^{[t_2,t_3)}, ..., \mathbf{z}^{[t_{J-1},t_J)}),$$

which is proportional to

$$P(Y) \prod_{j=1}^{J} q^{pa(X_{m_j})}_{x^j_{m_j} x^{j+1}_{m_j}} \prod_{n=1}^{N} exp\left(-q^{pa(X_n)}_{x^j_n} \delta_j\right), \tag{3}$$

where:

- $q^{pa(X_n)}_{x^j_n}$ is the parameter associated with state $x^j_n$, in which the variable $X_n$ was during the $j^{th}$ time interval, given the state of its parents $pa(X_n)$ during the $j^{th}$ time intervals;
- $q^{pa(X_m)}_{x^j_m x^{j+1}_m}$ is the parameter associated with the transition from state $x^j_m$, in which the variable $X_m$ was during the $j^{th}$ time interval, to state $x^{j+1}_m$, in which the variable $X_m$ will be during the $(j+1)^{th}$ time interval, given the state of its parents $pa(X_m)$ during the $j^{th}$ and the $(j+1)^{th}$ time intervals,

while $\delta_j = t_j - t_{j-1}$ is the length of the $j^{th}$ time interval of the stream $\mathbf{z}^{[t_1,t_2)}, \mathbf{z}^{[t_2,t_3)}, ..., \mathbf{z}^{[t_{J-1},t_J)}$ of continuous time evidence.

The inference algorithm for CTBNCs (Algorithm 1) is described in (Stella and Amer, 2012).

### 2.2.5 Clustering

Clustering for CTBNCs was introduced by Codecasa (2014) using the *Expectation Maximization (EM)* algorithm. The parameter learning algorithm relies on the same formulas showed in Section 2.2.2 for supervised learning, but where the expected sufficient statistics (i.e. $\bar{M}$ and $\bar{T}$) are used.

The expected sufficient statistics can be calculated in the EM expectation step by summing the contributions of the sufficient statistics, occurrences (i.e. $M^i$) and times (i.e. $T^i$) for each trajectory of the training set (i.e. $i \in \{1, \ldots, \mid \mathcal{D} \mid\}$) as follows. Contribution to the class count sufficient statistics for class $Y = y$ is:

$$\bar{M}[y] = \bar{M}[y] + P(y \mid \mathbf{z}^{i,1}, \ldots, \mathbf{z}^{i,J}).$$

Contribution to the occurrence count sufficient statistics of attribute $X_n$ such that $Y \in Pa(X_n)$ and $Y = y$ is:

$$\begin{aligned}\bar{M}[x_n, x'_n \mid pa(X_n)] = \bar{M}[x_n, x'_n \mid pa(X_n)] \\ + M^i[x_n, x'_n \mid pa(X_n)/y] \cdot P(y \mid \mathbf{z}^{i,1}, \ldots, \mathbf{z}^{i,J}),\end{aligned}$$

where $M^i[x_n, x'_n \mid pa(X_n)/y]$ represents the number of times $X_n$ transitions from state $x_n$ to state $x'_n$ in the $i^{th}$ trajectory when $X_n$'s parents without the class node (i.e. $Pa(X_n)/Y$) are set to $pa(X_n)/y$. Contribution to the time count sufficient statistics of attribute $X_n$ such that $Y \in Pa(X_n)$ and $Y = y$ is:

$$\begin{aligned}\bar{T}[x_n \mid pa(X_n)] =& \bar{T}[x_n \mid pa(X_n)] \\ &+ T^i[x_n \mid pa(X_n)/y] \cdot P(y \mid \mathbf{z}^{i,1}, \ldots, \mathbf{z}^{i,J}),\end{aligned}$$

**Algorithm 1** Inference algorithm for CTBNCs.

**Require:** a CTBNC $\mathcal{C} = \{\aleph, P(Y)\}$ consisting of $N$ attribute nodes and a class node $Y$ such that $Val(Y) = \{y_1, y_2, ..., y_K\}$, a *fully observed evidence stream* $\left(\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^J\right)$.

**Ensure:** the maximum aposteriori classification $y^*$ for the *fully observed J-evidence-stream* $\left(\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^J\right)$.

```
for $k = 1$ to $K$ do
  $logp(y_k) \leftarrow \log P(y_k)$
end for
for $k = 1$ to $K$ do
  for $j = 1$ to $J$ do
    for $n = 1$ to $N$ do
      $logp(y_k) := logp(y_k) - q_{x_n^j}^{pa(X_n)}(t_j - t_{j-1})$
      if $x_n^j \neq x_n^{j+1}$ then
        $logp(y_k) := logp(y_k) + log\left(q_{x_n^j x_n^{j+1}}^{pa(X_n)}\right)$
      end if
    end for
  end for
end for
$y^* \leftarrow \arg\max_{y \in Val(Y)} logp(y)$.
return $y^*$
```

where $T^i[x_n \mid pa(X_n)/y]$ is the amount of time along the $i^{th}$ trajectory in which attribute $X_n$ is in state $x_n$ when its parents without the class node (i.e. $Pa(X_n)/Y$) are set to $pa(X_n)/y$.

The calculation of the expected sufficient statistics as showed before correspond to the soft assignment expectation step in the EM algorithm. CTBNC-Toolkit provides also the hard assignment EM algorithm (Koller and Friedman, 2009) where the trajectory contributions to the expected sufficient statistics are calculated taking in account only the case of the most probable class. Once calculated the expected sufficient statistics, the EM maximization step corresponds to the application of the formula in Section 2.2.2 using the expected sufficient statistics.

# 3 CTBNCToolkit how to

## 3.1 Download

CTBNCToolkit can be downloaded as a stand-alone application. It requires *opencsv-2.3* library[2] to read the csv files and *commons-math3-3.0* library[3] for the Gamma function calculation.

To download and use the compiled CTBNCToolkit follow these steps:

- download CTBNCToolkit `.jar` file from `http://dcodecasa.wordpress.com/ctbnc/ctbnctoolkit/` website;
- download opencsv library from `http://sourceforge.net/projects/opencsv/` web site (tests were made with version 2.3);
- download commons math library from `http://commons.apache.org/proper/commons-math/download_math.cgi` web site (tests were made with version 3.0).

The compiled CTBNCToolkit can be used as a stand-alone application, as showed in the next sections. On the same website where the `.jar` file is released, it is possible to find the published papers related to CTBNCs and a collection of free data sets to test CTBNCToolkit (see Section 4).

CTBNCToolkit source code is released under GPL v2.0[4] license. The source code is available on GitHub at the following url: `https://github.com/dcodecasa/CTBNCToolkit`. It can be freely used in accordance with the GPL v2.0 license.

## 3.2 Run experiments from the command line

Once downloaded the CTBNCToolkit jar file, or generated from the source code, it can be run as follows:

```
java -jar CTBNCToolkit.jar <parameters> <data>
```

where `<parameters>` are the CTBNCToolkit parameters (i.e modifiers), addressed in the following, and `<data>` is a directory containing a data set. This data set is used to generate both the training set and the test set (see Section 3.2.4), unless `--training` or `--testset` modifier are used. In this case `<data>` is considered the test set, and it can refer to a single file (see Section 3.2.19 and Section 3.2.20). Hereafter the terms parameters and modifiers will be used interchangeably.

For sake of simplicity, the paths of the required libraries are defined in the manifest file contained in the `.jar` archive. Libraries are supposed to be saved in `lib/` directory. Tests are made using the Java virtual machine version 1.7.0_25 in ubuntu.

[2] `http://opencsv.sourceforge.net/`

[3] `http://commons.apache.org/proper/commons-math/download_math.cgi`

[4] GPL v2.0 license: `http://www.gnu.org/licenses/gpl-2.0.html`

It is worthwhile to note that it is often necessary to add the Java virtual machine parameter -Xmx to increase the heap space and to avoid out of memory exceptions (for example -Xmx2048m increases the heap dimension to 2Gb).

CTBNCToolkit parameters can be without any arguments, if specified as --modifier; or with any number of arguments, if specified as --modifier=arg1,arg2,..,argN. Table 1 summarizes all the parameters while Table 2 shows parameter incompatibilities and dependencies. The next sections address each parameter separately.

| Parameter | Method | Arguments | Section |
|---|---|---|---|
| --help | printHelp | no | 3.2.1 |
| --CTBNC | setCTBNCModels | yes | 3.2.2 |
| --model | setModels | yes | 3.2.3 |
| --validation | setValidationMethod | yes | 3.2.4 |
| --clustering | setClustering | yes | 3.2.5 |
| --1vs1 | setModelToClass | no | 3.2.6 |
| --bThreshold | setBinaryDecider | yes | 3.2.7 |
| --testName | setTestName | yes | 3.2.8 |
| --ext | setFileExt | yes | 3.2.9 |
| --sep | setFileSeparator | yes | 3.2.10 |
| --className | setClassColumnName | yes | 3.2.11 |
| --timeName | setTimeColumnName | yes | 3.2.12 |
| --trjSeparator | setTrjSeparator | yes | 3.2.13 |
| --validColumns | setValidColumns | yes | 3.2.14 |
| --cvPartitions | setCVPartitions | yes | 3.2.15 |
| --cvPrefix | setCVPrefix | yes | 3.2.16 |
| --cutPercentage | setCutPercentage | yes | 3.2.17 |
| --timeFactor | setTimeFactor | yes | 3.2.18 |
| --training | setTrainingSet | yes | 3.2.19 |
| --testset | setTestSet | yes | 3.2.20 |
| --rPath | setResultsPath | yes | 3.2.21 |
| --confidence | setConfidence | yes | 3.2.22 |
| --noprob | disableProbabilities | no | 3.2.23 |
| --v | setVerbose | no | 3.2.24 |

Table 1: It is shown for each modifier: the method of the CommandLine class that manages it, the presence or absence of arguments, and the Section in which the modifier is described.

### 3.2.1 Help

--help prints on the screen the help that shows the allowed parameters. For each parameter a short description is provided. When help is shown all the other parameters are ignored, and the program terminates after printing help.

```
java -jar CTBNCToolkit.jar --help
```

| Idx | Parameter | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | `--help` |  | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| 2 | `--CTBNC` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 | `--model` | X |  |  |  |  | X |  |  |  |  | D |  |  | D |  |  |  |  |  |  |  |  |  |  |
| 4 | `--validation` | X |  |  |  | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 | `--clustering` | X |  |  | X |  | X | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6 | `--1vs1` | X |  | X |  | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 | `--bThreshold` | X |  |  |  | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8 | `--testName` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 | `--ext` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 10 | `--sep` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 11 | `--className` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 12 | `--timeName` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 13 | `--trjSeparator` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 14 | `--validColumns` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 15 | `--cvPartitions` | X |  |  | D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 16 | `--cvPrefix` | X |  |  | D |  |  |  |  |  |  |  |  |  |  | D |  |  |  |  |  |  |  |  |  |
| 17 | `--cutPercentage` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 18 | `--timeFactor` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 19 | `--training` | X |  |  | D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 20 | `--testset` | X |  | D | D |  |  |  |  |  |  |  |  |  |  |  |  |  |  | D |  |  |  |  |  |
| 21 | `--rPath` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 22 | `--confidence` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 23 | `--noprob` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 24 | `--v` | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Table 2: Incompatibilities (X marks) and dependencies (D marks) between the modifiers. Incompatibility relations are symmetric; this does not hold for dependency relations, i.e. a modifier requires a particular value for another modifier, but the opposite it is not necessarily true. Modifiers are indicated by an index for due to the problems of space (first column).

### 3.2.2 Models to learn

`--CTBNC` modifier allows to specify the list of CTBNCs to test.

The models allowed follow:

- `CTNB`: Continuous Time Naive Bayes (Definition 2.3) (Stella and Amer, 2012; Codecasa and Stella, 2013);
- `ACTNBk-f`: Max-*k* Augmented Continuous Time Naive Bayes (Definition 2.5) (Codecasa and Stella, 2013) where `k` is the number of parents ($\geq 2$) and `f` is the scoring function used to learn the structure (`LL` or `CLL`);
- `CTBNCk-f`: Max-*k* Continuous Time Bayesian Network Classifier (Definition 2.4) (Codecasa and Stella, 2013) where `k` is the number of parents ($\geq 1$) and `f` is the scoring function used to learn the structure (`LL` or `CLL`);

where

- `LL` stands for marginal log-likelihood scoring function (Nodelman *et al.*, 2002b; Codecasa and Stella, 2013);

- `CLL` stands for conditional log-likelihood scoring function (Codecasa and Stella, 2013; Codecasa, 2014).

After each model definition, it is possible to specify the modifier parameters, which define the imaginary counts of the hyperparameters (Section 2.2.2):

- `Mk`: `k` are the imaginary counts related to the number of transitions for each variable (default value: `1.0`);
- `Tk`: `k` is the imaginary amount of time spent in a variable state (default value: `0.005`);
- `Pk`: `k` are the imaginary counts related to the class occurrences (default value: `1.0`).

It is a good habit to avoid zero values for the imaginary counts. Indeed, when the data set is not big enough, some model parameters can be 0 with the risk of a division by 0 error. The parameters should be tuned in accord to the specific classification problem, since the parameters can have a big impact in the model performances.

In addition to each model learned with the structural learning, it is possible to add the following parameter:

- `penalty`: adds the dimension penalty during the structural learning process; if omitted the penalty is disabled[5].

Here is an example:

```
java -jar CTBNCToolkit.jar --CTBNC=CTNB,M0.1,T0.001,ACTNB2-CLL,CTBNC4-LL,
penalty <data>
```

The previous line enables the following tree models:

- a CTNB with 0.1 as prior for the variable counts, 0.001 as the time prior and the standard 1.0 as class imaginary counts;
- a Max-2 ACTNB learned maximizing the conditional log-likelihood score and with the default parameter priors;
- a Max-4 CTBNC learned maximizing the marginal log-likelihood, using the default priors and enabling the dimension penalty during the learning process.

### 3.2.3 Model loading

`--model` modifier allows to specify the file paths of CTBNC models to load[6]. The models must be in the `.ctbn` format (see Section 5.3.3). If a training set is defined, the training set will be used only to learn models defined with the `--CTBNC` modifier (see Section 3.2.2).

Here are some examples:

[5] The `penalty` flag is ignored if applied with the CTNB model.

[6] The `--model` modifier will be soon implemented.

- tests the model stored in `model.ctbn` file:
  ```
  java -jar CTBNCToolkit.jar --model=models/model.ctbn
  <other_parameters>
  <data>
  ```

- tests the models stored in `model1.ctbn` and `model2.ctbn` files, and learn and test a CTNB model:
  ```
  java -jar CTBNCToolkit.jar --model=models/model1.ctbn,models/model2.ctbn
  --CTBNC=CTNB <other_parameters> <data>
  ```

It is not possible to apply the `--1vs1` modifier or to load models generated using the `--1vs1` modifier (see Section 3.2.6). The loaded models must be compatible with the input of `--className` modifier (see Section 3.2.11) and `--validColumns` modifier (see Section 3.2.14).

### 3.2.4 Validation method

The `--validation` modifier allows to specify the validation method to use. The CTBNCToolkit provides three validation methods: *hold-out*, *cross-validation* (Witten and Frank, 2005) and a validation method used for the clustering tests. The clustering validation method is automatically used when the clustering is enabled (see Section 3.2.5). While hold-out and cross-validation can be activated as follows:

- `--validation=HO,0.6`: enables the hold out validation method with a random partitioning of the data set in the training set (60%) and test set (40%) (default value: `0.7`);

- `--validation=CV,k`: enables the cross-validation with `k` folds (default value: `10`).

Here are some examples:

- 70%-30% hold-out partitioning:

  ```
  java -jar CTBNCToolkit.jar --validation=HO <other_parameters> <data>
  ```

- 60%-40% hold-out partitioning:

  ```
  java -jar CTBNCToolkit.jar --validation=HO,0.6 <other_parameters> <data>
  ```

- 10-folds cross-validation:

  ```
  java -jar CTBNCToolkit.jar --validation=CV <other_parameters> <data>
  ```

- 8-folds cross-validation:

  ```
  java -jar CTBNCToolkit.jar --validation=CV,8 <other_parameters> <data>
  ```

No validation method can be specified in case of clustering (see Section 3.2.5).

### 3.2.5 Clustering

The `--clustering` modifier disables the supervised learning and enables clustering (Codecasa, 2014) (Section 2.2.5). Tests require labeled data sets because the performances are calculated over the complete data set using external measures (see Section 3.3.2), i.e. the *Rand index (R)* (Rand, 1971), *Jaccard's coefficient (J)* (Halkidi *et al.*, 2001) and, the *Fowlkes-Mallows index (FM)* (Fowlkes and Mallows, 1983).

CTBNC clustering is implemented using the Expectation Maximization (EM) algorithm (Koller and Friedman, 2009). Both *soft-assignment* and *hard-assignment* clustering are implemented (Section 2.2.5). The parameters allow to specify the clustering method, the termination criterion and the number of clusters as follows:

- `hard/soft`: enables the clustering method (default value: `soft`);
- an integer number (i.e. `15`): sets to `15` the maximum number of iteration in the EM algorithm (default value: `10`);
- a `double` number (i.e. `0.1`): percentage of the data set trajectories; if less trajectories change class the EM algorithm is interrupted (default value: `0.01`);
- a `C` followed by an integer number (i.e. `C10`): sets the number of clusters (the default value is the number of classes of the class variable in the data set).

All these parameter values are optional and can be inserted in any order.

Here are some examples:

- soft clustering, `10` iterations, 1% as the trajectory threshold:

  ```
  java -jar CTBNCToolkit.jar --clustering <other_parameters> <data>
  ```

- soft clustering, `15` iterations, 10% as the trajectory threshold:

  ```
  java -jar CTBNCToolkit.jar --clustering=soft,0.1,15 <other_parameters>
  <data>
  ```

- soft clustering, `10` iterations, 3% as the trajectory threshold:

  ```
  java -jar CTBNCToolkit.jar clustering=0.03 <other_parameters> <data>
  ```

- hard clustering, `6` iterations, 5% as the trajectory threshold, `5` clusters:

  ```
  java -jar CTBNCToolkit.jar --clustering=6,0.05,hard,C5 <other_parameters>
  <data>
  ```

When the clustering is enabled a dedicated validation method is used. The used validation method learns and tests the model on the whole data set. For this reason the validation modifier cannot be used (see Section 3.2.4). Also the classification threshold for binary class problem cannot be used (see Section 3.2.7).

### 3.2.6 One model one class

--1vs1 modifier enables the one model one class modality. This modality generates for each model specified with the --CTBNC modifier (see Section 3.2.2) a set of models, one for each class. For example, if a CTNB is required using the --CTBNC modifier over a 10 class data set, this modifier will force the generation of 10 CTNB models. Each one of the ten generated models will discriminate one class against the others. During the classification process each model returns the probability related to the class for which it is specialized. The classification class is the one with the highest probability.

```
java -jar CTBNCToolkit.jar --1vs1 <other_parameters> <data>
```

### 3.2.7 Binary class threshold

The --bThreshold modifier changes the probability threshold used for assigning the class in supervised classification of binary problems.

Here are some examples:

- the first class is assigned to a trajectory only if its posterior probability to belong to that class is greater or equal than 0.4:

  ```
  java -jar CTBNCToolkit.jar --bThreshold=0.4 <other_parameters> <data>
  ```

- the first class is assigned to a trajectory only if its posterior probability to belong to that class is greater or equal than 0.6:

  ```
  java -jar CTBNCToolkit.jar --bThreshold=0.6 <other_parameters> <data>
  ```

Classes are ordered alphabetically. For example, if "A" and "B" are the classes, the first example gives advantage to class "A", while the second example gives advantage to class "B".

This modifier can be used only for binary classification and cannot be used in the case of clustering (see Section 3.2.5).

### 3.2.8 Test name

The --testName modifier specifies the name of the test. This name is used during the printing of the results to identify the particular test. The default value is specified by the current time using the "yyMMddHHmm_Test" format.

```
java -jar CTBNCToolkit.jar --testName=CTNBTest1 <other_parameters> <data>
```

### 3.2.9 File extension

The --ext modifier allows to specify the extension of the files to load in the data set directory (default value: .csv).

With the following command all the files in the data set directory with .txt extension are loaded:

```
java -jar CTBNCToolkit.jar --ext=.txt <other_parameters> <data>
```

### 3.2.10 Column separator

The `--sep` modifier allows to specify the column separator of the files to load (default value: `,`).

```
java -jar CTBNCToolkit.jar --sep=; <other_parameters> <data>
```

### 3.2.11 Class column name

The `--className` modifier specifies the name of the class column in the files to load (default value: `class`).

```
java -jar CTBNCToolkit.jar --className=weather <other_parameters> <data>
```

### 3.2.12 Time column name

The `--timeName` modifier specifies the name of the time column in the files to load (default value: `t`).

```
java -jar CTBNCToolkit.jar --className=time <other_parameters> <data>
```

### 3.2.13 Trajectory separator column name

Many data sets provide multiple trajectories in the same file. The `--trjSeparator` enables trajectory separation using the information provided by a target column. For example, if the trajectories are indexed by an incremental number ("trjIndex" column) that indicates the trajectories in the file, it is possible to split the trajectories automatically, as follows:

```
java -jar CTBNCToolkit.jar --trjSeparator=trjIndex <other_parameters> <data>
```

By default this modifier is not used, and a one file - one trajectory matching is assumed.

### 3.2.14 Data columns

The `--validColumns` modifier allows to specify the columns of the files that correspond to the variables in the models to be generated. By default, i.e. when the modifier is not specified, all the columns except the time column (see Section 3.2.12) and the trajectory separator column (see Section 3.2.13) are considered variables.

It is possible to specify any number of columns as arguments of the modifier. The following command line specifies the columns named `clmn1`, `clmn2`, and `clmn3` as the variables in the models to test:

```
java -jar CTBNCToolkit.jar --validColumns=clmn1,clmn2,clmn3 <other_parameters>
<data>
```

### 3.2.15 Cross-validation partitions

The `--cvPartitions` modifier allows to specify a partitioning when the cross-validation method is enabled (see Section 3.2.4). This modifier indicates a file in which a cross-validation partitioning is specified. The cross-validation method will follow the partitioning instead of generating a new one.

The partitioning file can be a result file, generated by the CTBNCToolkit (see Section 3.3.1):

```
Test1
trj12.txt: True Class: 4, Predicted: 4, Probability: 0.893885
...
trj87.txt: True Class: 2, Predicted: 2, Probability: 0.494983
Test2
trj26.txt: True Class: 2, Predicted: 2, Probability: 0.611254
...
trj96.txt: True Class: 2, Predicted: 3, Probability: 0.637652
Test3
trj1.txt: True Class: 4, Predicted: 4, Probability: 0.5697770
...
trj80.txt: True Class: 1, Predicted: 1, Probability: 0.938935
Test4
trj15.txt: True Class: 4, Predicted: 4, Probability: 0.624698
...
trj8.txt: True Class: 2, Predicted: 2, Probability: 0.7586410
Test5
trj11.txt: True Class: 4, Predicted: 4, Probability: 0.911368
...
trj99.txt: True Class: 4, Predicted: 4, Probability: 0.413442
```

or a text file where the word `Test` identifies the cross-validation folders, and the following lines specify the trajectories to load for each folder[7]:

```
Test 1 of 5:
trj12.txt
...
trj87.txt
Test 2 of 5:
trj26.txt
...
trj96.txt
Test 3 of 5:
trj1.txt
...
trj80.txt
Test 4 of 5:
```

[7] The one trajectory - one file matching is supposed.

```
trj15.txt
...
trj8.txt
Test 5 of 5:
trj11.txt
...
trj99.txt
```

Here is an example where the file `partition.txt` in the `CV` directory is loaded:

```
java -jar CTBNCToolkit.jar --cvPartitions=CV/partition.txt <other_parameters>
<data>
```

This modifier requires to enable cross-validation (see Section 3.2.4).

### 3.2.16 Cross-validation prefix

The `--cvPrefix` allows to specify a prefix to remove from the trajectory names in the partition file.

For example, if the partition file loaded with the `--cvPartitions` modifier has for some reasons the following form:

```
Test 1 of 5:
ex-trj12.txt
...
ex-trj87.txt
Test 2 of 5:
ex-trj26.txt
...
ex-trj96.txt
Test 3 of 5:
ex-trj1.txt
...
ex-trj80.txt
Test 4 of 5:
ex-trj15.txt
...
ex-trj8.txt
Test 5 of 5:
ex-trj11.txt
...
ex-trj99.txt
```

the `ex-` prefix can be removed automatically as follows:

```
java -jar CTBNCToolkit.jar --cvPrefix=ex- <other_parameters> <data>
```

This modifier requires the definition of a cross-validation partition file (see Section 3.2.15).

### 3.2.17 Data sets reduction

The `--cutPercentage` modifier allows to reduce the data set dimension in terms of number of trajectories and trajectory length. With only one parameter (which is a percentage) it is possible to perform tests over reduced data sets in order to evaluate the ability of the models to work with a restricted amount of data.

The following line forces a random selection of 60% of the data set trajectories and reduces each selected trajectory to 60% of its original length:

```
java -jar CTBNCToolkit.jar --cutPercentage=0.6 <other_parameters> <data>
```

The default value is `1.0` that does not change the data amount.

### 3.2.18 Time modifier

The `--timeFactor` modifier specifies a time factor used to scale the trajectory timing.

Here are some examples:

- doubles the trajectory timing:

  ```
  java -jar CTBNCToolkit.jar --timeFactor=2.0 <other_parameters> <data>
  ```

- reduces the trajectory timing by a factor of ten:

  ```
  java -jar CTBNCToolkit.jar --timeFactor=0.1 <other_parameters> <data>
  ```

The default value is `1.0`, which implies no time transformations.

### 3.2.19 Training set

The `--training` modifier specifies the directory that contains the training set. Using this modifier the `<data>` path in the command line is used as the test set. If the `--training` modifier is used, `<data>` can be a file or a directory.

Here are some examples:

- the files in the `trainingDir/` directory compose the training set, while the files in `testDir/` compose the test set:

  ```
  java -jar CTBNCToolkit.jar --training=trainingDir/ <other_parameters>
  testDir/
  ```

- the files in the `training/` directory compose the training set, while the models are tested on the `testDir/trj.txt` file:

  ```
  java -jar CTBNCToolkit.jar --training=training/ <other_parameters>
  testDir/trj.txt
  ```

This modifier requires the use of the hold out validation method.

### 3.2.20 Test set

The `--testset` modifier forces to consider `<data>` as the test set[8]. `<data>` can be a folder or a file. This modifier can be used with the `--model` modifier (see Section 3.2.3) and the hold out validation method to avoid splitting the input data set into training and test set, when the definition of a training set is not necessary (i.e. when `--CTBNC` modifier is not used). This modifier can be omitted, if `--training` modifier (see Section 3.2.19) is used.

Here are some examples:

- the files in the `trainingDir/` directory compose the training set, while the files in `testDir/` compose the test set:

```
java -jar CTBNCToolkit.jar --training=trainingDir/ <other_parameters>
testDir/
```

- the files in the `trainingDir/` directory compose the training set, while the files in `testDir/` compose the test set:

```
java -jar CTBNCToolkit.jar --training=trainingDir/ --testset
<other_parameters> testDir/
```

- no training set is specified, `model.ctbn` is loaded and tested on `testDir/trj.txt` file:

```
\code{java -jar CTBNCToolkit.jar --testset --model=model.ctbn
<other\_parameters> testDir/trj.txt
```

This modifier requires the use of the hold out validation method.

### 3.2.21 Results path

The `--rPath` modifier specifies the directory where to store the results (see Section 3.3). If it is not specified, the results are stored in a directory named as the test name (see Section 3.2.8) under the directory of the data (i.e. `<data>`) specified in the CTBNCToolkit command line.

Here are some examples:

- save the results in `resultDir/` directory:

```
java -jar CTBNCToolkit.jar --rPath=resultDir --testName=Test1
<other_parameters> <data>
```

- save the results in `testDir/Test2/` directory:

```
java -jar CTBNCToolkit.jar --testName=Test2 <other_parameters> testDir/
```

- save the results in `testDir/Test3/` directory:

```
java -jar CTBNCToolkit.jar --testName=Test3 <other_parameters>
testDir/file.txt
```

[8] `--testset` modifier will be soon implemented.

### 3.2.22 Confidence interval

The `--confidence` modifier allows to specify the confidence level to use in the model evaluation. The confidence level is used for model comparison in the case of supervised classification and for generating the error bars in the macro averaging curves (see Section 3.3.1). Models are compared using the test of difference of accuracy means proposed in (Witten and Frank, 2005). Supposes that two models $A$ and $B$ are compared with a test based on 90% of confidence. If the test informs that the models are statistically different with 90% of confidence, knowing that the mean accuracy of $A$ is greater than the mean accuracy of $B$ means that with 95% of confidence model $A$ is better than model $B$ (for more details see Section 3.3.1). This should be taken into account when the confidence level is chosen.

The allowed levels of confidence are: 99.9%, 99.8%, 99%, 98%, 95%, 90%, and 80%. 90% is the default value.

Here are some examples:

- 90% confidence:

  ```
  java -jar CTBNCToolkit.jar <other_parameters> <data>
  ```

- 90% confidence:

  ```
  java -jar CTBNCToolkit.jar --confidence=90% <other_parameters> <data>
  ```

- 99% confidence:

  ```
  java -jar CTBNCToolkit.jar --confidence=99% <other_parameters> <data>
  ```

### 3.2.23 Disable class probability

The `--noprob` modifier disables the calculation of the class probability distribution all across the trajectory. The only probability values are calculated at the end of the trajectory. This allows to speed up the computation.

```
java -jar CTBNCToolkit.jar --noprob <other_parameters> <data>
```

### 3.2.24 Verbose

The `--v` modifier enables the verbose modality. This modality prints more information that helps to follow the execution of the CTBNCToolkit.

```
java -jar CTBNCToolkit.jar --v <other_parameters> <data>
```

## 3.3 Read the results

Once a test is executed, the performances are printed in the results directory (see Section 3.2.21). The directory contains the performances of all the tested models. The performances of each model are identified by its name. Model names start with `Mi`, where `i` is the index that identifies the model in the command line.

For example, the command line:

```
java -jar CTBNCToolkit.jar --CTBNC=CTNB,M0.1,T0.001,ACTNB2-CLL,CTBNC4-LL,
penalty <data>
```

generates the following model names: `M0_CTNB`, `M1_ACTNB2-CLL`, `M2_CTBNC4-LL`.

In the case where external models are loaded (Section 3.2.3), the `i` index first refers to the model defined with the `--CTBNC` modifier (Section 3.2.2), then it refers to the external models.

Here is another example. The command line:

```
java -jar CTBNCToolkit.jar --model=models/model1.ctbn,models/model2.ctbn
--CTBNC=CTNB <other_parameters> <data>
```

generates the following model names: `M0_CTNB`, `M1_model1`, `M2_model2`.

The following sections describe the performances calculated in the case of classification and clustering. In both cases the results directory contains a file that shows the modifiers used in the test.

### 3.3.1 Classification

*Results file*

For each tested model a results file is provided. It can be found in the test directory and contains the results for each classified data set instance. For each instance the name, the true class, the predicted class and the probability of the predicted class is shown.

Here is an example:

```
trj1: True Class: s2, Predicted: s2, Probability: 0.9942336124116266
trj10: True Class: s4, Predicted: s2, Probability: 0.9896614159579054
trj100: True Class: s1, Predicted: s1, Probability: 0.9955018513870293
trj11: True Class: s4, Predicted: s4, Probability: 0.977733953650775
trj12: True Class: s3, Predicted: s3, Probability: 0.9997240904249852
...
```

*Metrics*

In the results directory the metric file is stored. It is the `.csv` file that contains the following performances for each tested model (Witten and Frank, 2005; Fawcett, 2006; Japkowicz and Shah, 2011):

- Accuracy: accuracy value with the confidence interval calculated in accordance with the defined level of confidence (Section 3.2.22);
- Error: the percentage of wrong classified instances;
- Precision: precision value for each class;
- Recall: recall value for each class[9];
- F-Measure: balanced f-measure for each class (i.e. precision and recall weigh equally);

[9] Sensitivity, TP-rate and Recall are the same.

- PR AUC: precision-recall AUC (Area Under the Curve) for each class;
- Sensitivity: sensitivity for each class[9];
- Specificity: specificity for each class;
- TP-Rate: true positive rate for each class[9];
- FP-Rate: false positive rate for each class;
- ROC AUC: ROC Area Under the Curve for each class;
- Brier: brier value;
- Avg learning time: learning time or average learning time in case of multiple tests (i.e. cross-validation);
- Var learning time: learning time variance in case of multiple tests (i.e. cross-validation);
- Avg inference time: average inference time between all the tested instances;
- Var inference time: inference time variance between all the tested instances;

Some information in the metric file depends on the validation method. In the case of cross validation (Section 3.2.4), the file contains both the micro-averaging and the macro-averaging performances.

*Model comparison*

The model comparison file in the results directory contains the comparison matrices between the tested models.

A comparison matrix is a squared matrix that compares each pair of tested models by using their corresponding average accuracy values. The comparison file contains the comparison matrices for the following confidence levels 99%, 95%, 90%, 80%, and 70%.

Here is an example of the comparison matrix with 90% confidence:

| Comparison test | 90% | | | | | | |
|---|---|---|---|---|---|---|---|
| | M0 | M1 | M2 | M3 | M4 | M5 | M6 |
| M0 | | UP | UP | UP | UP | UP | 0 |
| M1 | LF | | UP | 0 | UP | 0 | LF |
| M2 | LF | LF | | LF | 0 | LF | LF |
| M3 | LF | 0 | UP | | UP | 0 | LF |
| M4 | LF | LF | 0 | LF | | LF | LF |
| M5 | LF | 0 | UP | 0 | UP | | LF |
| M6 | 0 | UP | UP | UP | UP | UP | |

UP indicates that the upper model is statistically better than the left model, LF indicates that the left model is statistically better than the upper model, while 0 indicates that the models are indistinguishable.

The tests are based on the difference of the average accuracy of the pair of considered models, as proposed in (Witten and Frank, 2005). This means that when UP or LF are showed we know that the two models are statistically different with a given confidence (i.e. 90% in the previous table). Consider the previous table and the pair of models M0 and M1. We know that they are statistically different with 90% of confidence. Because the average accuracy of M1 is bigger then the average accuracy of M0 the table shows that M1 is better than M0. Knowing that the test is a two tails test we actually know that M1 is statistically better of M0 with 95% of confidence because only one tail must be considered. This must be taken in account when the comparison matrices are read.

*Single run data*

The results directory contains a directory for each model. In the case of cross-validation, the model directory stores a directory named "runs" that contains a metric file with the performances of each single run. In this directory all the models generated by the learning process are stored.

*ROC curve*

The model directory also contains a directory named "ROCs" which stores the ROC curves for each class (Fawcett, 2006). In the case of cross-validation the curves calculated in micro and macro averaging are stored. The macro averaging curves show the confidence interval due to the vertical averaging. The confidence interval can be set using the `--confidence` modifier (see Section 3.2.22).

*Precision-Recall curve*

The model directory contains a directory named "Precision-Recall" which stores the precision-recall curves for each class (Fawcett, 2006). In the case of cross-validation the curves calculated in micro and macro averaging are stored. The macro averaging curves show the confidence interval due to the vertical averaging. The confidence interval can be set using the `--confidence` modifier (see Section 3.2.22).

*Lift chart and cumulative response*

The directories of the models also contain a directory named "CumulativeResp&LiftChar" which stores the cumulative response curves and the lift charts for each class (Fawcett, 2006). In the case of cross-validation the curves calculated in micro and macro averaging are stored. The macro averaging curves show the confidence interval due to the vertical averaging. The confidence interval can be set using the `--confidence` modifier (see Section 3.2.22).

### 3.3.2 Clustering

In the case of clustering, the results directory contains a directory for each tested model. Each directory contains a read me file with some information related to the test and the following files.

*Results file*

As in the case of supervised classification for each tested model a result file is provided. It contains the results for each classified data set instance. For each instance the name, the true class, the predicted cluster and the probability of the predicted cluster is shown. Of course there is no correspondence between the class names and the cluster names.

Here is an example of the results file:

```
trj1.txt: True Class: 3, Predicted: 2, Probability: 0.9998733687935368
trj10.txt: True Class: 3, Predicted: 2, Probability: 0.9999799985527281
trj100.txt: True Class: 3, Predicted: 2, Probability: 0.9998951904099552
trj11.txt: True Class: 3, Predicted: 2, Probability: 0.9999999967591123
trj12.txt: True Class: 4, Predicted: 4, Probability: 0.9999999983049304
...
```

*Performances*

The file performances contains the following external measures (Halkidi *et al.*, 2001; Gan *et al.*, 2007; Xu and Wunsch, 2008):

- the Rand index ($R$) (Rand, 1971);
- Jaccard's coefficient ($J$) (Halkidi *et al.*, 2001);
- the Fowlkes-Mallow index ($FM$) (Fowlkes and Mallows, 1983).

In addition to these measures, association matrix, clustering-partition matrix, precision matrix, recall matrix, and f-measure matrix are shown. Learning time, average inference time and inference time variance are also shown.

When the number of clusters is set manually (see Section 3.2.5) the performance file is not generated.

*Model file*

The model file shows the CTBNC learned in the test. Since the clustering validator learns one model all over the data set, just one model file is generated. `.ctbn` is the format used (see Section 5.3.3).

# 4 Tutorial examples

This section provides replicable examples of CTBNCToolkit usage. To execute the tests follow the next steps:

- download the compiled version of the CTBNCToolkit or download and compile the source code (see Section 3.1);

- create a directory for the tests where the compiled CTBNCToolkit must be copied (let's call the directory "tutorial");
- create a sub-directory named "lib" containing the required libraries (see Section 3.1)[10];
- copy and unzip in the "tutorial" directory, for each of the following tests, the data sets used (see Section 3.1).

## 4.1 Classification

In this section the tutorial examples of classification are given. For these tests "naiveBayes1" synthetic data set is used. The data set has to be downloaded, unzipped and copied under the "tutorial" directory. The trajectories will be available in the "tutorial/naiveBayes1/dataset/" directory.

### 4.1.1 Hold out

Assume we are interested in making classifications using CTNB, ACTNB learned by maximizing conditional log-likelihood and CTBNC learned by maximizing marginal log-likelihood. The last two models learned setting to two the maximum number of parents. To evaluate the performances of the models we want to use the hold out validation method.

The data set is stored in `.txt` files. Trajectories use "Class" as the class column name and "t" as the time column name. To execute the test the following command must be used setting "tutorial" as the current directory[11]:

```
java -jar CTBNCToolkit.jar --CTBNC=CTNB,ACTNB2-CLL,CTBNC2-LL --validation=HO
--ext=.txt --className=Class naiveBayes1/dataset/
```

where:

- `--CTBNC=CTNB,ACTNB2-CLL,CTBNC2-LL` specifies the models to test (Section 3.2.2);
- `--validation=HO` specifies hold out as the validation method (Section 3.2.4);
- `--ext=.txt` specifies that the data set files have `.txt` extension (Section 3.2.9);
- `--className=Class` specifies that the class column has the name "Class" ("class" is the default value, see Section 3.2.11);
- `naiveBayes1/dataset/` specifies the directory which contains the data set (Section 3.2).

[10] Other directories can be used once the manifest file in the CTBNCToolkit`.jar`, is modified.

[11] This test may takes a while.

It is worthwhile to notice that the time column name is not specified because "t" is the default name (Section 3.2.12). Eventually the `--v` modifier can be added to enable the verbose modality in order to monitor the program execution (Section 3.2.24).

The results are stored in a directory with a name similar to "131014_1041_Test". To specify a different name use the `--testName` modifier (Section 3.2.8).

The results follow the guidelines described in Section 3.3.1.

### 4.1.2 Cross validation loading a partitioning

Assume we are interested in making classifications using just a CTNB. To evaluate the performances we use a 10 fold cross-validation, loading a pre-defined partitioning. The partitions are described by the `NB-results.txt` file in the "naiveBayes1/" directory.

Similarly to the previous example, here is the command line to execute the test:

```
java -jar CTBNCToolkit.jar --CTBNC=CTNB --validation=CV
--cvPartitions=naiveBayes1/NB-results.txt --ext=.txt --className=Class
naiveBayes1/dataset/
```

The difference from the hold out classification example relies on the following two points:

- `--validation=CV` enables cross-validation to validate the models (Section 3.2.4);
- `--cvPartitions=naiveBayes1/NB-results.txt` specifies the file which contains the cross-validation partitioning (Section 3.2.15).

`NB-results.txt` is the results file (Section 3.3) of a CTNB model learned on the same data set using the specified cross validation partitioning. The results file can be loaded as a partitioning file (Section 3.2.15). This allows to replicate the executed tests.

Because the test which generated the "naiveBayes1/NB-results.txt" result file was made using the same parameters, the same model and the same cross-validation partitioning, the results just obtained from the previous example should be exactly the same. This can be seen comparing the "naiveBayes1/NB-results.txt" file with the results file just generated.

## 4.2 Clustering

In this section a tutorial example of clustering is described. Also in this case the "naiveBayes1" synthetic data set is used. The data set has to be downloaded, unzipped and copied under the "tutorial" directory. The trajectories will be contained in "tutorial/naiveBayes1/dataset/" directory.

Assume we are interested in making clustering using CTNB and CTBNC learned by maximizing marginal log-likelihood when the maximum number of parents is set to two.

The data set is stored in `.txt` files. Trajectories use "Class" as the class column name and "t" as the time column name. To execute the test the following command must be used setting "tutorial" as the current directory[11]:

```
java -jar CTBNCToolkit.jar --CTBNC=CTNB,CTBNC2-LL --clustering --ext=.txt
--className=Class naiveBayes1/dataset/
```

The command is similar to the previous ones, the main difference is the use of the `--clustering` modifier which substitutes the validation modifier (Section 3.2.5).

The results are stored in a directory with a name similar to "131014_1252_Test". To specify a different name use the `--testName` modifier (Section 3.2.8).

The results follow the guidelines described in Section 3.3.2. Since learning and testing in case of clustering are done using the whole data set, it is easy to replicate the clustering experiments, but because of the random instantiation of the EM algorithm the results can be quite different especially if there is not a clear distinction between clusters.

# 5 CTBNCToolkit library

CTBNCToolkit is released under the GPL v2.0[12] license. It is available on GitHub at the following url: `https://github.com/dcodecasa/CTBNCToolkit`. See section 3.1 for more information regarding the CTBNCToolkit download.

The CTBNCToolkit is a library to manage CTBNCs. It provides:

- a CTBNC model representation which can be easily extended to define other types of models (Section 5.3);
- a supervised learning algorithm (Section 5.4);
- soft and hard assignment Expectation Maximization (EM) algorithms for clustering purposes (Section 5.5);
- two different scoring functions to learn CTBNCs (Section 5.4);
- CTBNCs inference algorithm for static classification (Section 5.6);
- three different validation methods to realize experiments (Section 5.7),
- a rich set of performance measures for supervised classification and clustering as well (Section 5.8);
- a set of utilities for data set and experiment managing (Sections 5.1 and 5.9);

[12] GPL v2.0 license: `http://www.gnu.org/licenses/gpl-2.0.html`

- an extendable command line front-end (Section 5.10).

In the following part of this section the main software components are analyzed from the development point of view.

## 5.1 Input trajectories

Figure 3 depicts the simplified class diagram of interfaces and classes used for representing trajectories.

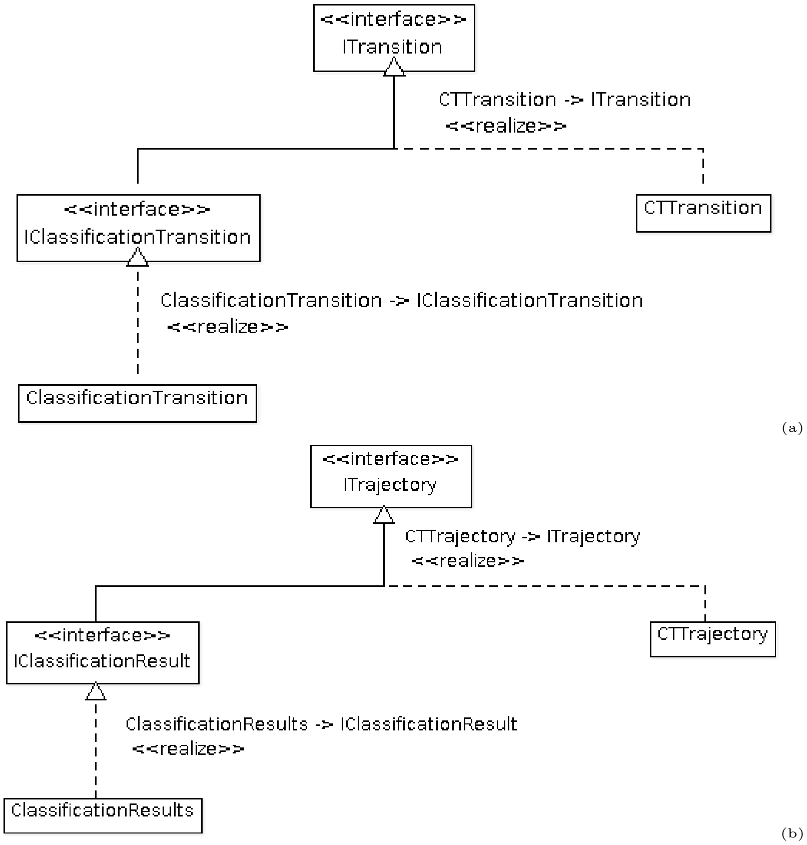


Figure 3: Simplified class diagram of the trajectory managing components.

`ITrajectory` is the interface that defines the trajectories. Each trajectory consists of a sequence of transitions. Each transition (i.e. `ITransition`) occurs at a particular time. Transition time is generalized using `<TimeType extends Number>` generic. Since CTBNCs model continuous time trajectories, in the stand-alone CTBNCToolkit implementation the `TimeType` generic is filled with `Double` class. Nevertheless, each extension of `Number` and `Comparable` classes can be potentially used as time representation; i.e. the `Integer` class can be specified to represent discrete time trajectories.

`CTTransition` and `CTTrajectory` provide a standard implementation of `ITransition` and `ITrajectory`.

`IClassificationTransition` and `IClassificationResult` interfaces define an extension of `ITransition` and `ITrajectory` which allows to mange classified trajectories. They allow to insert the probability distribution for each transition into a trajectory.

`ClassificationTransition` and `ClassificationResults` implement a standard version of the interfaces for the classified transitions and trajectories.

**Example 5.1.** Here is how to generate a trajectory.

```
0.0        Va12        Vb1        Vc123
0.2        Va12        Vb2        Vc123
0.6        Va3         Vb3        Vc123
1.3        Va4         Vb4        Vc4
```

The previous trajectory shows multiple changes of states for each time instant, even if theoretically, in accord with the CTBN definition, this cannot happen. This can be implemented by the following code:

```
// Nodes-column name definition
String[] nodeNames = new String[3];
nodeNames[0] = "A"; nodeNames[1] = "B"; nodeNames[2] = "C";
NodeIndexing nodeIndexing = NodeIndexing.getNodeIndexing(
        "IndexingName", nodeNames, nodeNames[0], null);

// Time jump definition
String[] v;
List<Double> times = new Vector<Double>();
List<String[]> values = new Vector<String[]>();
times.add(0.0);times.add(0.2);times.add(0.6);times.add(1.3);

// State definition
v = new String[3];
v[0] = "Va12"; v[1] = "Vb1"; v[2] = "Vc123";
values.add(v);
v = new String[3];
v[0] = "Va12"; v[1] = "Vb2"; v[2] = "Vc123";
values.add(v);
v = new String[3];
v[0] = "Va3"; v[1] = "Vb3"; v[2] = "Vc123";
values.add(v);
v = new String[3];
v[0] = "Va4"; v[1] = "Vb4"; v[2] = "Vc4";
```

```
values.add(v);

// Trajectory creation
CTTrajectory<Double> tr = new CTTrajectory<Double>(nodeIndexing, times,
        values);
```

Section 5.2 provides a clarification about `NodeIndexing` class.

The `CTBNCTestFactory` class, a class used in test management (see Section 5.9), provides a set of `static` methods which are useful for dealing with data sets:

- `loadDataset`: loads a data set;
- `partitionDataset`: partitions a data set in accordance with a partitioning file (i.e. cross-validation folding, Section 3.2.15);
- `loadResultsDataset`: loads a data set composed of classified trajectories; for each trajectory the probability distribution of the class has to be specified;
- `partitionResultDataset`: partitions a data set of results in accordance with a partitioning file (i.e. cross-validation folding);
- `permuteDataset`: randomly permutes a data set;
- `cutDataset`: reduces the original data set in terms of number and length of trajectories (Section 3.2.17).

## 5.2 Global node indexing

Before introducing the models in section 5.3, it is necessary to deal with the global node indexing system. Models nodes, variables, and columns in a trajectory use the same names. Indeed, objects like model nodes or state values in a trajectory have to be stored by using the same names. To allow a really efficient method to recover these objects the `NodeIndexing` class is developed. A model node can be recovered using its name, which corresponds to a column name in the trajectories, or using its index. Recovering objects by name uses a tree structure. This is efficient, but it is possible to improve it. The index solution is developed to be an $O(1)$ entity recovery system.

To guarantee the same indexing for all the trajectory columns and all the model nodes, the class `NodeIndexing` is used. The idea is that each trajectory and each model has to be synchronized with the same `NodeIndexing` instance. To ensure this, `NodeIndexing` provides a static method (i.e. `getNodeIndexing`) which allows to obtain or create a `NodeIndexing` instance using an unique name. This name usually corresponds to the test name.

**Example 5.2.** Here is an example of how to create a new `NodeIndexing` class.

```
String[] nodeNames = new String[3]; nodeNames[0] = "class";
nodeNames[1] = "N1"; nodeNames[2] = "N2";
NodeIndexing nodeIndexing = NodeIndexing.getNodeIndexing("IndexingName",
        nodeNames, nodeNames[0], null);
```

The first argument is the unique key associated with the index. Using the same key in successive calls of the `getNodeIndexing` method allows to recover the same indexing instance in accordance with the singleton pattern. The second argument (i.e. `nodeNames`) is the array of all the node names, while the third argument is the class node name. The last argument is the set of all the node names in the `nodeNames` array to which an index will be associated. This argument allows to select just a subset of the names specified in the second argument. If it is left to `null`, all the names are loaded.

When two objects are synchronized with the same `NodeIndexing` instance, they use the same loaded indexing. This allows the direct communication index-to-index between all the objects with the same synchronization.

## 5.3 CTBNCToolkit models

Figure 4 depicts the simplified class diagram of the interfaces and classes to be used to work with the models.

### 5.3.1 Nodes

A model consists of nodes which represent the model variables. Node objects are shown in Figure 4b. `INode` interface defines the generic properties of a node, while `IDiscreteNode` interface defines the properties of a discrete state space node.

`Node` class is an abstract class which implements the properties that all the nodes require. Those properties are mainly related to the naming and the dependency relations between nodes. `DiscreteNode` class implements all the requirements of a discrete node. It does not specify any quantitative component as Conditional Intensity Matrix (CIM) or Conditional Probability Table (CPT), but only manages the discrete states of a variable.

`CTDiscreteNode` class implements all the properties which the CTBNC nodes require. It allows the CIMs definition and can implement both a continuous time node and a static node for continuous time models (i.e. the class node).

**Example 5.3.** Here is an example of how to create a continuous time discrete node (i.e. `CTDiscreteNode`).

```
// State definition
Set<String> states2;Set<String> states3;
states2 = new TreeSet<String>();
states2.add("n1_1");states2.add("n1_2");
states3 = new TreeSet<String>();
states3.add("n1_1");states3.add("n1_2");states3.add("n1_3");
```

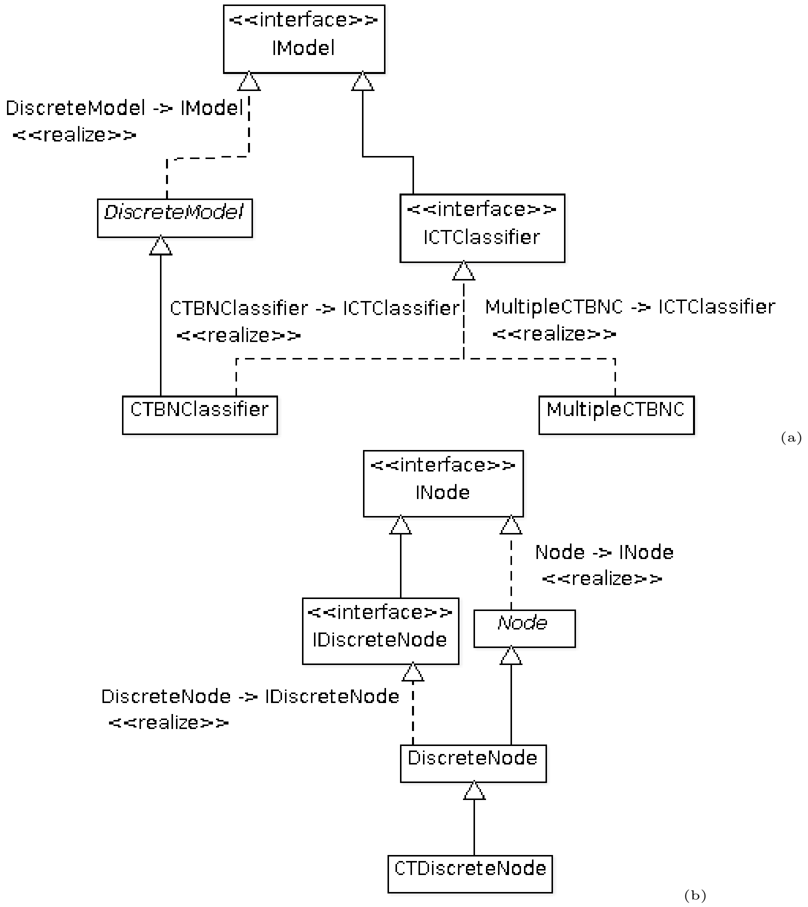


Figure 4: Simplified class diagram of the model managing components.

```
// Node creation
CTDiscreteNode node = new CTDiscreteNode("node",states3,false);
CTDiscreteNode parent = new CTDiscreteNode("parent",states2,true);
parent.addChild(node);

// CIM definition
double[][] cim = new double[3][3];
cim[0][0] = -2; cim[0][1] = 1; cim[0][2] = 1;
cim[1][0] = 2; cim[1][1] = -4; cim[1][2] = 2;
cim[2][0] = 2; cim[2][1] = 1; cim[2][2] = -3;
node.setCIM(0, cim);
node.setCIM(1, cim);
assertTrue( node.checkCIMs() == -1);
```

### 5.3.2 Models

Generic model properties are defined by the IModel interface. Part of these properties are implemented by the DiscreteModel abstract class. Instead, the ICTClassifier interface defines the properties related to the classification process, which a continuous time classifier has to satisfy.

CTBNCs are implemented by CTBNClassifier class. While the MultipleCTBNC class implements a continuous time classifier composed of a set of binary CTBNCs, each one is specialized to recognize a class against the others (see Section 3.2.6).

Models are simple Java classes used like containers of nodes. The real complexity related to the learning and the classification processes relies on the algorithm classes.

**Example 5.4.** Here is an example that shows how to create a CTBNClassifier.

```
// Node names and indexing definition
String[] nodesNames = new String[4];
nodesNames[0] = "Class"; nodesNames[1] = "A";
nodesNames[2] = "B"; nodesNames[3] = "C";
NodeIndexing nodeIndexing = NodeIndexing.getNodeIndexing("IndexingName",
        nodesNames, nodesNames[0], null);

// State generation
CTDiscreteNode classNode, aNode, bNode, cNode;
Set<String> states2 = new TreeSet<String>();
Set<String> states3 = new TreeSet<String>();
Set<CTDiscreteNode> nodes = new TreeSet<CTDiscreteNode>();
states2.add("s1");states2.add("s2");
states3.add("s1");states3.add("s2");states3.add("s3");

// Node generation
nodes.add(classNode = new CTDiscreteNode(nodesNames[0], states2, true));
nodes.add(aNode = new CTDiscreteNode(nodesNames[1], states2, false));
nodes.add(bNode = new CTDiscreteNode(nodesNames[2], states3, false));
nodes.add(cNode = new CTDiscreteNode(nodesNames[3], states3, false));

// Model structure definition
classNode.addChild(aNode);
classNode.addChild(bNode);
classNode.addChild(cNode);

// Node CIM definition
double[][] cim;
cim = new double[1][2]; cim[0][0] = 0.5; cim[0][1] = 0.5;
classNode.setCIM(0, cim);
assertTrue(classNode.checkCIMs() == -1);
```

```
cim = new double[2][2];
cim[0][0] = -0.1; cim[0][1] = 0.1;
cim[1][0] = 0.1; cim[1][1] = -0.1;
aNode.setCIM(0, cim);
cim = new double[2][2];
cim[0][0] = -5; cim[0][1] = 5;
cim[1][0] = 5; cim[1][1] = -5;
aNode.setCIM(1, cim);
assertTrue(aNode.checkCIMs() == -1);

cim = new double[3][3];
cim[0][0] = -0.7; cim[0][1] = 0.5; cim[0][2] = 0.2;
cim[1][0] = 1.0; cim[1][1] = -1.6; cim[1][2] = 0.6;
cim[2][0] = 2; cim[2][1] = 1.3; cim[2][2] = -3.3;
bNode.setCIM(0, cim);cNode.setCIM(0, cim);
cim = new double[3][3];
cim[2][2] = -0.7; cim[2][1] = 0.5; cim[2][0] = 0.2;
cim[1][0] = 1.0; cim[1][1] = -1.6; cim[1][2] = 0.6;
cim[0][2] = 2; cim[0][1] = 1.3; cim[0][0] = -3.3;
bNode.setCIM(1, cim); cNode.setCIM(1, cim);
assertTrue(bNode.checkCIMs() == -1);
assertTrue(cNode.checkCIMs() == -1);

// Model generation
CTBNClassifier model = new CTBNClassifier(nodeIndexing, "classifier", nodes);
```

### 5.3.3 .ctbn format

`.ctbn` is the space-separated text format in which the CTBNCToolkit saves the learned models. The `toString()` method in the `CTBNClassifier` class is the method which provides the model text representation.

`.ctbn` format starts with the following lines:

```
-----------------------
BAYESIAN NETWORK
-----------------------
BBNodes N
-----------------------
```

where `N` is the number of nodes in the CTBNC (class node included).

Then each node is specified with the couple `<node_name states_number>` as follows:

```
Class         4
N01        2
N02        2
```

```
N03        2
...
N15        4
-----------------------
```

where data in the same line are separated by spaces or tabs, and the first node is the class.

The next lines in the `.ctbn` format define the Bayesian Network (BN) structure, which represents the initial probability distribution of the CTBNC. Each line starts with the considered node and then lists all its parents, using the space-separator format. Each line has to stop with a zero. Here is an example:

```
Class        0
N01        Class        0
N02        Class        N01        0
N03        Class        0
...
N15        Class        N07        0
-----------------------
```

The `.ctbn` format allows to define the initial probability distribution by using a Bayesian Network. On the contrary, the CTBNCToolkit does not support an initial probability distribution yet, but assumes an initial uniform distribution between the variables states. The only exception is for the class which has its own probability distribution, as specified later. For this reason, for the moment the initial distribution is represented as a disconnected Bayesian network.

```
Class 0
N01 0
N02 0
N03 0
...
N15 0
-----------------------
```

Note that lines composed of minus signs separate the different file sections.

The next lines inform about the conditional probability distributions of the defined BN. Each CPT is written following the ordering of the parent nodes defined in the previous section. The CPTs are separated by a line composed by minus signs. Here is an example in case of uniform distribution for a disconnected BN:

```
Class
0 0 0 0
-----------------------
N01
0.5 0.5
-----------------------
```

```
...
-----------------------
N15
0.25 0.25 0.25 0.25
-----------------------
```

where the line of the class probability distribution contains zeros, due to the fact that the class prior will be defined in the next section of the file format. Since CTBNCToolkit currently supports only an initial uniform distribution, the CPTs shown are straightforward. Nevertheless, the `.ctbn` format supports complex CPTs, where each line is the probability distribution setting the value of the parent set. Considering the definition of the parent set of node `N02`:

```
N02         Class         N01         0
```

imaging that `Class` node is binary, node `N01` is ternary, and `N02` has 4 states. Node `N02` CPTs can be defined by the probability distribution lines in the following order:

| Class | N01 | probability distribution line |
|---|---|---|
| 0 | 0 | `0.2 0.3 0.1 0.4` |
| 1 | 0 | `0.15 0.2 0.35 0.3` |
| 0 | 1 | `0.42 0.08 0.23 0.27` |
| 1 | 1 | `0.15 0.2 0.35 0.3` |
| 0 | 2 | `0.2 0.3 0.1 0.4` |
| 1 | 2 | `0.23 0.5 0.15 0.12` |

With the previous lines we defined the model variables and then the initial probability distribution. In the next part of the file format the temporal model is defined.

First the graph structure is defined in the same way as previously done for the BN:

```
-----------------------
DIRECTED GRAPH
-----------------------
Class        0
N01         Class         N05         0
N02         Class         N01         0
N03         Class         N01         0
...
N15         Class         N11         0
-----------------------
```

For each node its parents are specified.

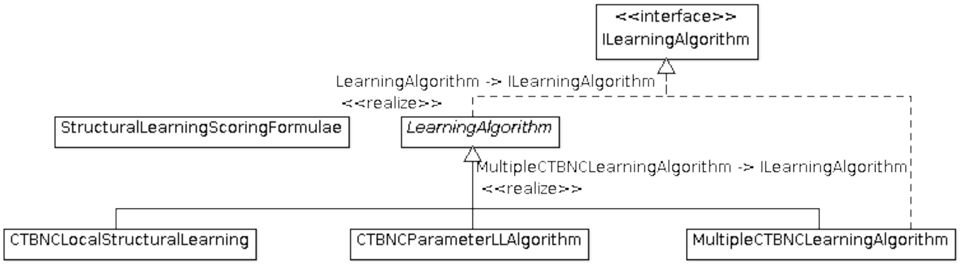
Then the CIMs are defined for each node. Each line represents a complete CIM given the parent set instantiation. The parent sets are defined iterating the parent values starting from the left as previously shown for the Bayesian Network CPTs.

```
-----------------------
CIMS
-----------------------
Class
0.2323008849557522 0.2488938053097345 0.2610619469026549 0.2577433628318584
-----------------------
N01
-1.7437542178131549 1.7437542178131549 1.6677084125959616 -1.6677084125959616
-1.1359184948768346 1.1359184948768346 1.468624833995604 -1.468624833995604
-1.655441390834388 1.655441390834388 1.9083015334745217 -1.9083015334745217
-1.992223211031885 1.992223211031885 1.128540290987483 -1.128540290987483
-1.4597541436405608 1.4597541436405608 1.3875353532121044 -1.3875353532121044
-1.8514888977211081 1.8514888977211081 1.2036550623637277 -1.2036550623637277
-1.5536818371118852 1.5536818371118852 1.06784175008451 -1.06784175008451
-1.2675007732387165 1.2675007732387165 1.8220901564788115 -1.8220901564788115
-----------------------
```

It is worthwhile to note that for the class node the prior probability distribution is specified and not a temporal model since the class is a static node.

## 5.4 Learning algorithms

Figures 5 and 6 depict the simplified class diagram of the components used to provide learning algorithms.



<<interface>>
ILearningAlgorithm
LearningAlgorithm -> ILearningAlgorithm
<<realize>>
StructuralLearningScoringFormulae
LearningAlgorithm
MultipleCTBNCLearningAlgorithm -> ILearningAlgorithm
<<realize>>
CTBNCLocalStructuralLearning
CTBNCParameterLLAlgorithm
MultipleCTBNCLearningAlgorithm

Figure 5: Learning algorithms simplified class diagram.

Learning algorithms are modeled by the `ILearningAlgorithm`interface, which defines the learning methods and the methods to manage the parameters of a learning algorithm.

`LearningAlgorithm` is an abstract class which provides an initial implementation of the parameter managing methods. While `CTBNCParameterLLAlgorithm`, `CTBNCLocalStructuralLearning`, and `MultipleCTBNCLearningAlgorithm` are the learning algorithm implementations.

`CTBNCParameterLLAlgorithm` implements the parameter learning Stella and Amer (2012). This algorithm need in advance the definition of the CTBNC

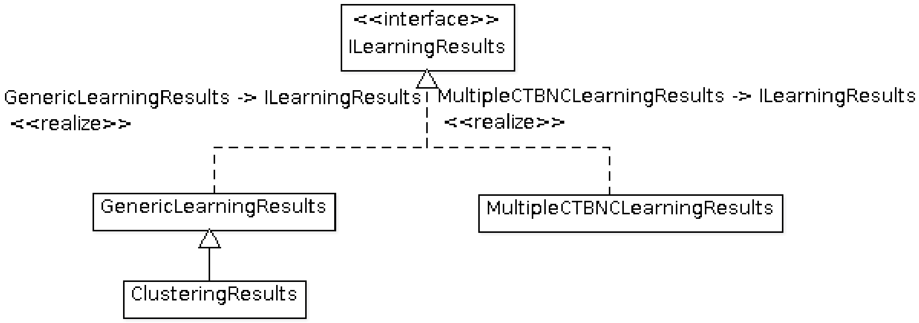

<<interface>>
ILearningResults
GenericLearningResults -> ILearningResults
<<realize>>
MultipleCTBNCLearningResults -> ILearningResults
<<realize>>
GenericLearningResults
MultipleCTBNCLearningResults
ClusteringResults

Figure 6: Simplified class diagram of the learning result components.

structure that specifies the dependencies between the variables. The parameters learning algorithm is the basis for all the structural learning algorithms.

`CTBNCLocalStructuralLearning` implements the structural learning for CTBNCs using the local search (Nodelman *et al.*, 2002b; Codecasa and Stella, 2013). The searching algorithm relies on the optimization classes shown in Figure 7.

`MultipleCTBNCLearningAlgorithm` is the implementation of the local searching algorithm for `MultipleCTBNC` models (see Section 3.2.6). Its implementation is strictly related to the classical structural learning.

All the learning algorithms have to return the results of the learning process implemented by the `ILearningResults` interface.
`ILearningResults` defines the learning results as a container for the sufficient statistics (i.e. `SufficientStatistics` class).

`GenericLearningResults` and `MultipleCTBNCLearningResults` provide a standard implementation of learning results, respectively for CTBNCs and for `MultipleCTBNC` models. `ClusteringResults` extends the `GenericLearningResults` class to implement the learning results for the clustering algorithm (see Section 5.5).

Figure 7 shows the simplified class diagram of the classes used for the local search in structural learning.

CTBNCs structural learning can be seen as an optimization problem. The CTBNCToolkit provides two scoring functions defined as `static` methods in `StructuralLearningScoringFormulae` class. The scoring functions rely on marginal log-likelihood (Nodelman *et al.*, 2002b; Codecasa and Stella, 2013) and conditional log-likelihood calculation (Codecasa and Stella, 2013).

Since the learning procedure solves a maximization problem respect to a selected scoring function, the `IOptimizationElement` interface is defined. The interface provides a simple definition of an element that must be evaluated for optimization purposes. The `ILocalSearchIndividual` interface extends the `IOptimizationElement` interface, defining the properties of a local search element such as the ability to find the best neighbor (i.e. `getBestNeighbor` method).

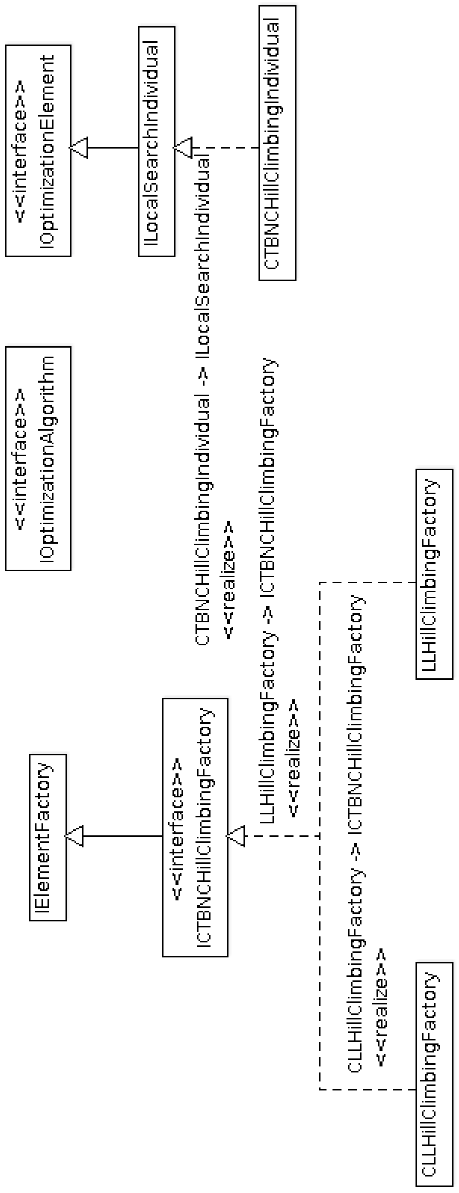

<<interface>>
IOptimizationElement
<<interface>>
IOptimizationAlgorithm
ILocalSearchIndividual
CTBNCHillClimbingIndividual
CTBNCHillClimbingIndividual -> ILocalSearchIndividual
<<realize>>
IElementFactory
<<interface>>
ICTBNCHillClimbingFactory
LLHillClimbingFactory -> ICTBNCHillClimbingFactory
<<realize>>
LLHillClimbingFactory
CLLHillClimbingFactory -> ICTBNCHillClimbingFactory
<<realize>>
CLLHillClimbingFactory

Figure 7: Hill climbing simplified class diagram.

CTBNCHillClimbingIndividual is the actual implementation of an individual in the local search procedure. It is the core of the local search algorithm implementation.

To generalize the local search algorithm, implemented in the

CTBNCLocalStructuralLearning class, the factory pattern is used to generate local search individuals.

IElementFactory is the interface that defines the properties of a factory of IOptimizationElement. The use of Java generics helps the code generalization. The ICTBNCHillClimbingFactoryinterface extends the IElementFactory interface to define the requirements for generating CTBNCHillClimbingIndividual.

LLHillClimbingFactory and CLLHillClimbingFactory are the two factories that generate CTBNCHillClimbingIndividual. The first factory generates the individuals for the marginal log-likelihood score maximization, while the second factory generates the individuals for the conditional log-likelihood score maximization.

**Example 5.5.** Here is an example of parameter learning.

```
// Model definition
CTBNClassifier clModel = new CTBNClassifier(nodeIndexing, "classifier",
        nodes);

// Structure definition
boolean[][] adjMatrix = new boolean[4][4];
for(int i = 0; i < adjMatrix.length; ++i)
        for(int j = 0; j < adjMatrix[i].length; ++j)
                adjMatrix[i][j] = false;
adjMatrix[0][1] = true;adjMatrix[0][2] = true;
adjMatrix[0][3] = true;

// Learning algorithm instantiation
CTBNCParameterLLAlgorithm  alg = new CTBNCParameterLLAlgorithm();

// Learning algorithm parameter (i.e. priors) definition
Map<String,Object> params = new TreeMap<String,Object>();
params.put("Mxx_prior", 1.0);
params.put("Tx_prior", 0.01);
params.put("Px_prior", 1.0);

// Learning algorithm parameters and structure setting
alg.setParameters(params);
alg.setStructure(adjMatrix);

// Parameter learning
Collection<ITrajectory<Double>> dataset = generateDataset();
alg.learn(clModel, dataset);
```

**Example 5.6.** Here is an example of structural learning using the marginal log-likelihood scoring.

```
// Parameter learning algorithm to use in the structural learning
CTBNCParameterLLAlgorithm  paramsAlg = new CTBNCParameterLLAlgorithm();
```

```
Map<String,Object> params = new TreeMap<String,Object>();
params.put("Mxx_prior", 1.0);
params.put("Tx_prior", 0.01);
params.put("Px_prior", 1.0);
paramsAlg.setParameters(params);

// Hill climbing factory definition
LLHillClimbingFactory elemFactory = new LLHillClimbingFactory(paramsAlg, 3,
        false, false);

// Definition of the local search starting structure
int iClass = nodeIndexing.getClassIndex();
int iA = nodeIndexing.getIndex("A");
int iB = nodeIndexing.getIndex("B");
int iC = nodeIndexing.getIndex("C");
boolean[][] adjMatrix = new boolean[4][4];
for(int i = 0; i < adjMatrix.length; ++i)
        for(int j = 0; j < adjMatrix[i].length; ++j)
                adjMatrix[i][j] = false;
adjMatrix[iClass][iA] = true;adjMatrix[iClass][iB] = true;
adjMatrix[iClass][iC] = true;

// Definition of the structural learning algorithm and setting of the initial
// structure
CTBNCLocalStructuralLearning<String,CTBNCHillClimbingIndividual> alg =
        new CTBNCLocalStructuralLearning<String,CTBNCHillClimbingIndividual>(
                elemFactory);
alg.setStructure(adjMatrix);

// Structural learning of a target model
ICTClassifier<Double, CTDiscreteNode> model = generateClassifierModel();
alg.learn(model, trainingSet);
boolean[][] learnedStructure = model.getAdjMatrix();
```

## 5.5 Clustering

Figure 8 depicts the simplified class diagram of the clustering learning.

IClusteringAlgorithm is the interface that defines the clustering learning algorithms. It extends the ILearningAlgorithm interface (Section 5.4).

ClusteringAlgorithm is an abstract class which implements the basis functionality of the clustering learning algorithm, while CTBNClusteringParametersLLAlgorithm implements the soft and hard assignment EM algorithms for clustering purposes (Codecasa, 2014). The structural learning algorithm relies on the optimization algorithms, used also in the case of supervised learning (Section 5.4).

IStopCriterion interface defines the stopping criterion for the EM iterative

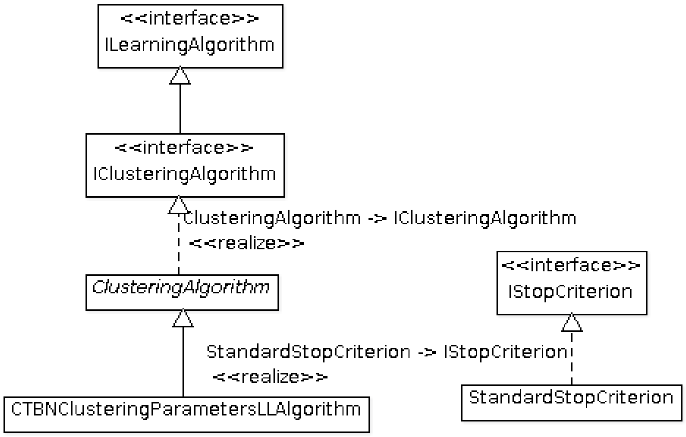


Figure 8: Simplified class diagram of the clustering learning.

algorithm, and the `StandardStopCriterion` class provides a basic stop criterion implementation.

**Example 5.7.** Here is an example of soft-assignment EM clustering.

```
// Parameter definition
Map<String, Object> params = new TreeMap<String,Object>();
params.put("Mxx_prior", 1.0);
params.put("Tx_prior", 0.005);
params.put("Px_prior", 1.0);
params.put("hardClustering", false);
StandardStopCriterion stopCriterion = new StandardStopCriterion(
        iterationNumber, 0.1);

// Classification algorithm definition
Map<String, Object> paramsClassifyAlg = new TreeMap<String,Object>();
paramsClassifyAlg.put("probabilities", true);
CTBNCClassifyAlgorithm classificationAlg = new CTBNCClassifyAlgorithm();
classificationAlg.setParameters(paramsClassifyAlg);

// Clustering algorithm initialization
CTBNClusteringParametersLLAlgorithm cAlg =
        new CTBNClusteringParametersLLAlgorithm();
cAlg.setParameters(params);
cAlg.setClassificationAlgorithm(classificationAlg);
cAlg.setStopCriterion( stopCriterion);

// Clustering learning
ClusteringResults<Double> results = cAlg.learn(model, dataSet);
```

A classification algorithm is required for the expectation step of the EM algorithm in order to calculate the expected sufficient statistics (i.e. `CTBNCClassifyAlgorithm` in the previous code).

## 5.6 Inference algorithms

Figure 9 depicts the simplified class diagram of the inference algorithm classes.

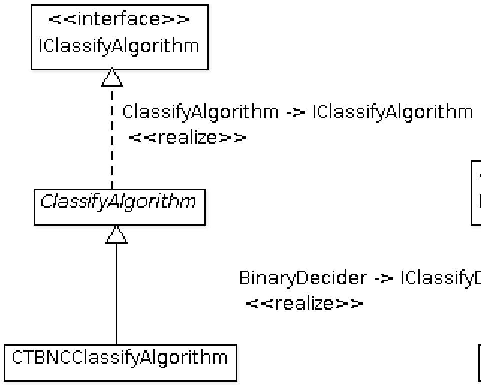


Figure 9: Simplified class diagram of inference components.

`IClassifyAlgorithm` is the interface that defines the properties of an inference algorithm. It mainly defines properties related to parameter managing and classification methods. Classification methods return `IClassificationResult`. As for the learning algorithms (Section 5.4), an abstract class is developed to manage the parameters, i.e. `ClassifyAlgorithm`. Generics are used to generalize the time representation and the nodes in the model to classify.

`CTBNCClassifyAlgorithm` implements the classification algorithm for CTBNCs, as described by Stella and Amer (2012).

**Example 5.8.** Here is an example of classification inference.

```
// Generate the trajectory to classify
ITrajectory<Double> testTrajectory = new CTTrajectory<Double>(nodeIndexing,
        times, values);

// Classification
CTBNCClassifyAlgorithm clAlgorithm = new CTBNCClassifyAlgorithm();
IClassificationResult<Double> result = clAlgorithm.classify(learnedModel,
        testTrajectory, 0.9);
```

Usually the classification relies on the most probable class, but defining a `IClassifyDecider` it is possible to use another classification criteria. `BinaryDecider` implements the `IClassifyDecider` interface to allow an unbalanced classification in the case of two class problems. The decider allows to define a threshold to use in the classification in order to give an advantage to one of the two classes (see Section 3.2.7).

## 5.7 Validation methods

Figure 10 depicts the simplified class diagram of the validation methods used to realize tests and to calculate performances.

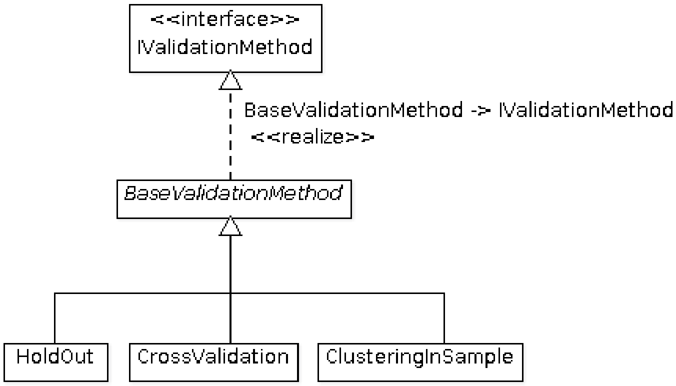


Figure 10: Simplified class diagram of the validation methods.

To execute the tests over a data set, validation methods are implemented. `IValidationMethod` is the interface that defines the validation method properties. `BaseValidationMethod` is the abstract class that implements the base functions of the validation methods.

`HoldOut`, `CrossValidation` and `ClusteringInSample` are the classes that implement different validation methods (Section 3.2.4). Each validation approach, one executed using `validate` method, returns the performances of the tested algorithm (Section 5.8).

`HoldOut` class implements the hold out validation, where training set and test set are used to calculate the performances. `CrossValidation` class implements the cross validation method (Witten and Frank, 2005). `ClusteringInSample` class implements a validation method, where learning and testing are both realized over the same complete data set; this can be used to test clustering approaches (Section 3.2.5).

**Example 5.9.** Here is an example of classification inference.

```
// Instantiation of the cross validation method
CrossValidation<Double,CTDiscreteNode,CTBNClassifier,
        MicroMacroClassificationPerformances<Double,
                ClassificationStandardPerformances<Double>>,
        ClassificationStandardPerformances<Double>> validationMethod =
        new CrossValidation<...>(performanceFactory, 10, true);
```

```
validationMethod.setVerbose(true);

// Test execution using a validation method
performances = validationMethod.validate( model, learningAlgorithm,
        inferenceAlgorithm, dataset);
```

performanceFactory is a factory to generate new test performances. For more details about performances see Section 5.8.

## 5.8 Performances

The CTBNCToolkit provides a rich set of performances to evaluate the experiments. Different performances are provided in the case of classification (Section 3.3.1) and clustering (Section 3.3.2). The simplified class diagrams of the classification and the clustering performances are depicted in Figures 11 and 12. In both cases the class hierarchy is the same. Different classes are provided to calculate single run and aggregate performances (i.e. for cross-validation multiple runs). To allow the best possible generalization, factory classes are provided to generate the performances. The simplified class diagram of the performance factories is depicted in Figure 13. A factory argument is required in all the validation methods in order to generate the performances during the tests (see Section 5.7).

IPerformances is the interface which defines a generic performance, while ISingleRunPerformances and IAggregatePerformances respectively define the single run and the aggregate performances.

In the case of classification the IClassificationPerformances interface is provided. In the case of clustering the performances definitions relies on IExternalClusteringPerformances; only external clustering measures are provided (see Section 3.3.2).

IClassificationSingleRunPerformances is the interface which defines the single run classification performances, while the IClassificationAggregatePerformancesinterface defines the aggregate classification performances. Similarly, the clustering performances are defined by IExternalClusteringSingleRunPerformances and IExternalClusteringAggregatePerformances interfaces.

The ClassificationStandardPerformances class, together with the ClusteringExternalPerformances class, implements the performances for single runs in the case of classification and clustering. Aggregate performances are provided in micro and macro averaging. Micro averaging performances are calculated extending the single run performances. MicroAvgAggregatePerformances is the class that implements the micro averaging in the case of classification, while MicroAvgExternalClusteringAggregatePerformances implements the micro averaging performances in the case of clustering.

Macro averaging classification performances are provided by the MacroAvgAggregatePerformances class. Macro averaging performances

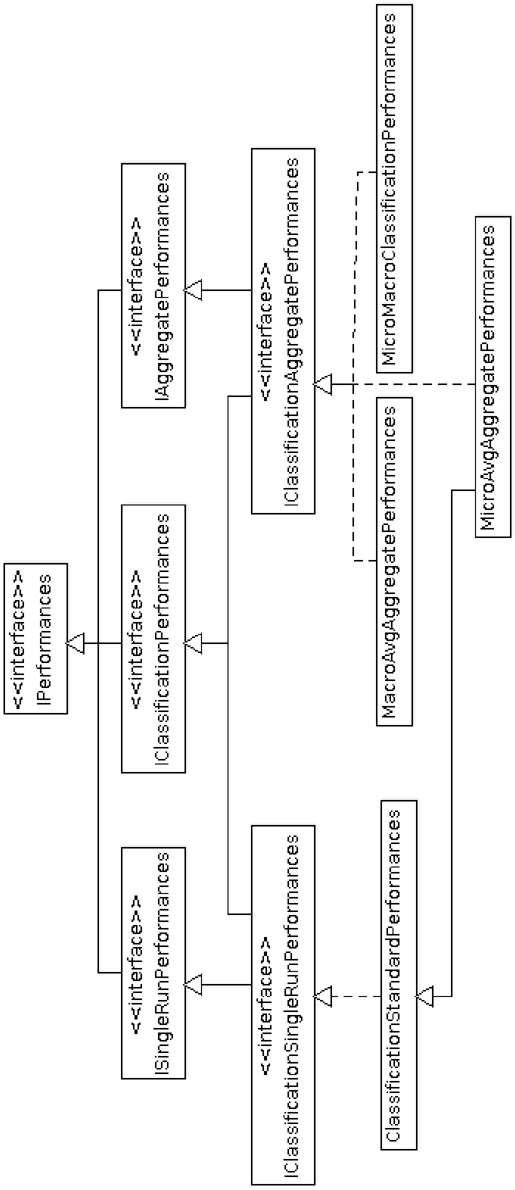

<<interface>>
IPerformances
<<interface>>
ISingleRunPerformances
<<interface>>
IClassificationPerformances
<<interface>>
IAggregatePerformances
<<interface>>
IClassificationSingleRunPerformances
<<interface>>
IClassificationAggregatePerformances
ClassificationStandardPerformances
MacroAvgAggregatePerformances
MicroMacroClassificationPerformances
MicroAvgAggregatePerformances

Figure 11: Simplified class diagram of the supervised classification performances.

for unsupervised learning (i.e. clustering) are provided by the `MarcoAvgExternalClusteringAggregatePerformances` class.

In order to have the possibility to calculate both micro and macro averaging performances, two classes that hide both the averaging approaches are provided

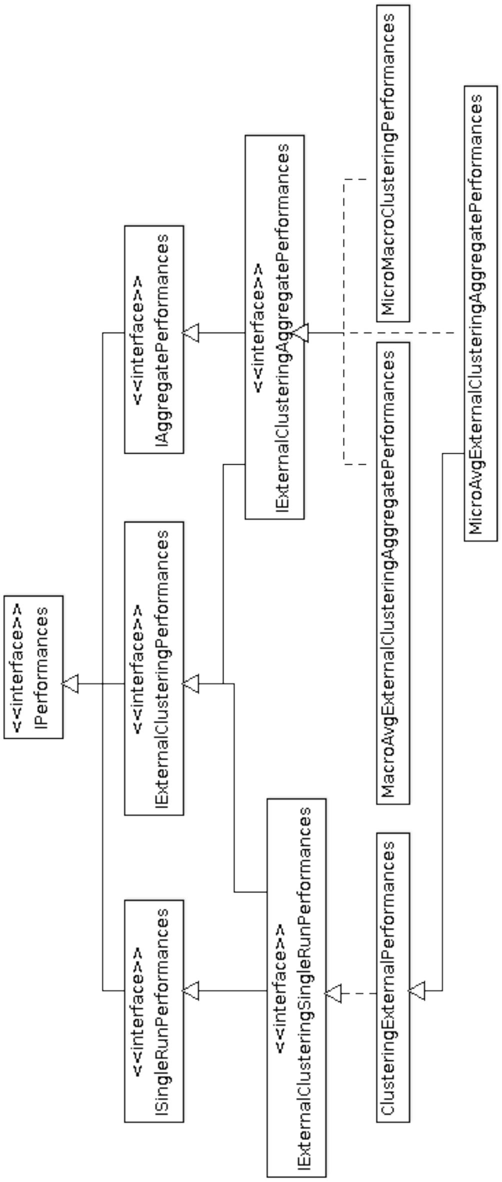

<<interface>>
IPerformances
<<interface>>
IAggregatePerformances
<<interface>>
IExternalClusteringPerformances
<<interface>>
ISingleRunPerformances
<<interface>>
IExternalClusteringAggregatePerformances
<<interface>>
IExternalClusteringSingleRunPerformances
MicroMacroClusteringPerformances
MacroAvgExternalClusteringAggregatePerformances
ClusteringExternalPerformances
MicroAvgExternalClusteringAggregatePerformances

Figure 12: Simplified class diagram of the clustering performances.

in the case of classification (i.e. `MicroMacroClassificationPerformances`) and in the case of clustering (i.e. `MicroMacroClusteringPerformances`).

Figure 13 depicts the simplified class diagram of the performance factory classes.

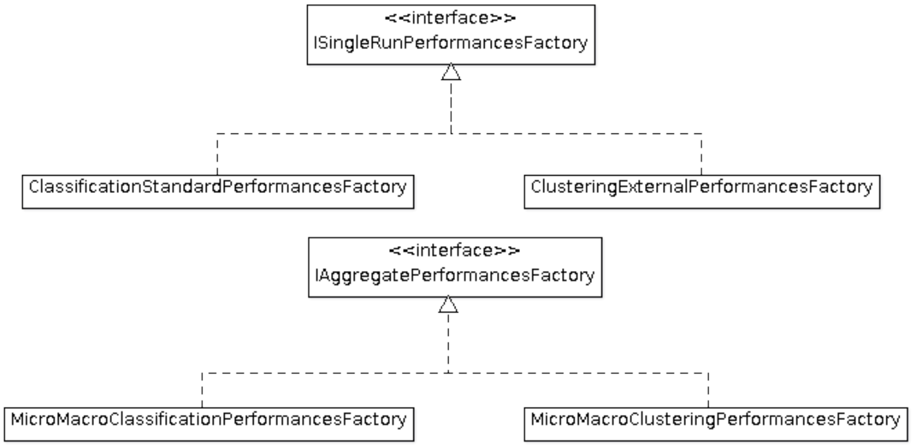

<<interface>>
ISingleRunPerformancesFactory
ClassificationStandardPerformancesFactory
ClusteringExternalPerformancesFactory
<<interface>>
IAggregatePerformancesFactory
MicroMacroClassificationPerformancesFactory
MicroMacroClusteringPerformancesFactory

Figure 13: Simplified class diagram of the performance factories.

The `ISingleRunPerformancesFactory` interface defines the factory for single run performances. The factory for aggregate performances is defined by the `IAggregatePerformancesFactory` interface. In the case of classification and in the case of clustering two factories are provided: one for the single run performances and one for the combination of micro and macro averaging aggregate performances. In the case of classification `ClassificationStandardPerformancesFactory` is provided for single runs, and `MicroMacroClassificationPerformancesFactory`is provided for aggregate performances. In the case of clustering the provided classes are: `ClusteringExternalPerformancesFactory` and `MicroMacroClusteringPerformancesFactory`.

## 5.9 Tests

To perform the test experiments a set of utilities have been developed. Figure 14 depicts the simplified class diagram of these utilities.

`IModelFactory` interface defines a factory to generate new models. `CTBNClassifierFactory` implements a simple way to generate synthetic CTBNCs. This class has been used to generate synthetic models for evaluation and test purposes.

**Example 5.10.** Here is an example of factory for the CTNB models.

```
// Number of variables
int N = 16;
// Number of states for each variable
int[] nStates = new int[N];
nStates[0] = 10;                // class variable
nStates[1] = 2;  ...;  nStates[15] = 4;
```

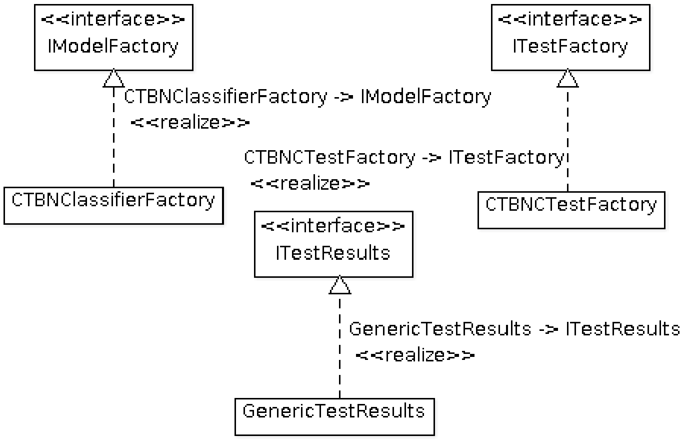


Figure 14: Simplified class diagram of the classes used to test CTBNCs.

```
// Range in which sample the lambda values of the exponential distribution
double[][] lambdaRanges = new double[2][N];
lambdaRanges[0][1] = 10; lambdaRanges[1][1] = 20;
...
lambdaRanges[0][15] = 40; lambdaRanges[1][15] = 80;

// Initialize the factory
CTBNClassifierFactory modelFactory = CTBNClassifierFactory( "naiveBayes",
        nStates, lambdaRanges);

// Generate a model
CTBNClassifier model = modelFactory.newInstance();
```

`ITestFactory` is an interface which defines a factory for tests. The idea is to provide a factory instance with the parameters of the tests, for example the learning and inference algorithms, and then to run a test calling a method. `CTBNCTestFactory` implements a test factory for CTBNCs. It provides the possibility to use a model factory to generate and test different data sets created from different model instances, and it provides the possibility to execute the tests over a data set in input.

In both cases `GenericTestResults` class is returned. This class implements a test result defined by the `ITestResults` interface and contains the performances of the tests.

**Example 5.11.** Here is an example of `CTBNCTestFactory` use. Generics are widely used to have a complete generalization.

```
// Factory instantiation
CTBNCTestFactory<Double,CTDiscreteNode,CTBNClassifier,
        MicroMacroClassificationPerformances<Double,
                ClassificationStandardPerformances<Double>>> testFactory;
testFactory = new CTBNCTestFactory<Double,CTDiscreteNode,CTBNClassifier,
        MicroMacroClassificationPerformances<Double,
                ClassificationStandardPerformances<Double>>>(
                        modelFactory, validationMethod, nbParamsLearningAlg,
                        classificationAlg);

// Test execution
GenericTestResults<Double,CTDiscreteNode,CTBNClassifier,
        MicroMacroClassificationPerformances<Double,
                ClassificationStandardPerformances<Double>>> resultNB =
                        testFactory.newTest("NB", datasetDim);

// Performances recovering
String perfSum = resultNB.performancesSummary(testName, confidenceLevel,
        datasetDim, kFolds);
```

In addition to the basic methods, GenericTestResults and CTBNCTestFactory classes provide a set of general utilities to manage the tests. GenericTestResults provides a set of static methods to print the classification and clustering performances. CTBNCTestFactory provides a set of static methods to load and to manage the input data sets (see Section 5.1).

## 5.10 Command line front-end

CTBNCToolkit.frontend is the package that contains the front-ends. Currently, only a command line front-end is developed. The package consist of two classes: Main and CommandLine. Main class is due to start the command line front-end implemented by the CommandLine class.

### 5.10.1 Command line parameters

The CommandLine class was developed to easily add new command line parameters (i.e. modifiers).

The command line parameters use an array of Strings that gathers the information necessary to manage the modifiers (i.e. modifiersList).

```
private String[][] modifiersList = {
   {"help", "printHelp", "print the CTBNCToolkit help", "enabled"},
   ... ,
   {"validation", "setValidationMethod", "specify the validation method:\n" +
      "\t--validation=CV,k  \tk-folds cross validation is used\n" +
      "\t--validation=HO,0.6\thold out is used", "enabled"},
   ... ,
   {"v", "setVerbose", "enable the verbose comments", "enabled"}
};
```

Each line consists of four columns:

i. name of the modifier, i.e. `"help"` is the modifier name that can be called from the command line writing `--help`;

ii. name of the function called to manage the modifier, i.e. when the `--help` modifier is inserted, the `printHelp()` function is called;

iii. modifier description, automatically printed when the `--help` modifier is inserted;

iv. flag to enable or not the modifier: if the fourth column contains `"enabled"` the modifier is enabled, otherwise it is completely ignored, even in the printing of the help.

When the command line is started, the input parameters are analyzed in order to call the appropriate methods. It is possible to define a method without parameters, i.e. `printHelp()`:

```
public void printHelp() {
   System.out.println("CTBNCToolkit for classification");
   ....
}
```

or a method with parameters, i.e. `setValidationMethod( vMethod)`:

```
public void setValidationMethod( LinkedList<String> vMethod) {
   ....
}
```

The inputs are passed to the method through a linked list automatically generated from the command line parameters. The command line parameters with arguments have to be in the following form:

```
--modifier=arg1,arg2,..,argN
```

this generates a linked list of the following `String`s: `"arg1"`, `"arg2"`, ..., `"argN"`. Using the example of the validation method (3.2.4): the command line input `--validation=HO,0.6` generates a linked list where `"HO"` is the first argument and `"0.6"` is the second.

Using the `modifiersList` and the automatic managing of the modifier parameters it is possible to add a new modifier just adding in the code a line in the `modifiersList` and the corresponding method, which can have in input any number of arguments.

### 5.10.2 Program execution

The command line parameters are used to set the parameters required during the program execution, while the real CTBNCToolkit starter relies on the `CommandLine()` constructor.

The constructor first initializes the modifier methods using Java reflexivity. This is done in the `initModifiers()` method. It then analyzes the command line parameters, calling the right methods and checking the compatibilities between the parameters.

Once the modifiers are managed, the CTBNCToolkit starts with the following steps:

- data set loading and generation of a CTNB, provided by the `loadDatasets(..)` method (see Section 5.1);
- loading of the learning algorithms to test, provided by the `loadLearningAlgorithms(..)` method (see Section 5.4);
- loading of the classification algorithm, provided by the `loadInferenceAlgorithm()` method (see Section 5.6);
- test generation, provided by the `generateTestFactories(..)` method (see Section 5.9);
- test execution, provided by the `executeTests(..)` method (see Section 5.9).

# 6 Conclusion and future works

In this paper the CTBNCToolkit has been presented. CTBNCToolkit is an open source toolkit for the temporal classification of a static variable using Continuous Time Bayesian Network Classifiers (CTBNCs). Once introduced the main concepts about the CTBNCs (Stella and Amer, 2012; Codecasa and Stella, 2013; Codecasa, 2014), the stand-alone usage of the toolkit was provided. The performances description and the tutorial examples allowed to replicate the examples and to test the toolkit, while the description of the Java library introduced the code giving the possibility to use the CTBNCToolkit as an external library for related application.

This paper describes the first version of CTBNCToolkit, for this reason many future developments are be planned. First the `--model` (Section 3.2.3) and the `--testset` (Section 3.2.20) modifiers will be implemented. The current version of CTBNCToolkit does not allow to the class node to have parents. One of the possible future implementation consists in the possibility to add static nodes as parents of another static node.

Further extension will be provided in parallel with the planned research on CTBNCs. Currently the research directions are focused on the possibility of extending the model in the case of partially observable trajectories.

# References

Barber D, Cemgil A (2010). "Graphical models for time-series." *Signal Processing Magazine, IEEE*, **27**(6), 18–28.

Codecasa D (2014). *Continuous Time Bayesian Network Classifiers*. Ph.D. thesis, Universitá degli Studi di Milano-Bicocca. To appear in BOA (Bicocca Open Archive).

Codecasa D, Stella F (2013). "Conditional Log-Likelihood for Continuous Time Bayesian Network Classifiers." In *International Workshop NFMCP held at ECML-PKDD 2013*.

Dacorogna M (2001). *An introduction to high-frequency finance*. AP.

Dean T, Kanazawa K (1989). "A model for reasoning about persistence and causation." *Computational intelligence*, **5**(2), 142–150.

Fawcett T (2006). "An introduction to ROC analysis." *Pattern recognition letters*, **27**(8), 861–874.

Fowlkes EB, Mallows CL (1983). "A method for comparing two hierarchical clusterings." *Journal of the American statistical association*, **78**(383), 553–569.

Gan G, Ma C, Wu J (2007). *Data clustering: theory, algorithms, and applications*, volume 20. Siam.

Grossman D, Domingos P (2004). "Learning Bayesian network classifiers by maximizing conditional likelihood." In *Proc. of the 21st Int. Conf. on Machine Learning*, pp. 361–368. ACM Press.

Gunawardana A, Meek C, Xu P (2011). "A Model for Temporal Dependencies in Event Streams." In J Shawe-Taylor, R Zemel, P Bartlett, F Pereira, K Weinberger (eds.), *Advances in Neural Information Processing Systems 24*, pp. 1962–1970.

Halkidi M, Batistakis Y, Vazirgiannis M (2001). "On clustering validation techniques." *Journal of Intelligent Information Systems*, **17**(2-3), 107–145.

Japkowicz N, Shah M (2011). *Evaluating learning algorithms: a classification perspective*. Cambridge University Press.

Koller D, Friedman N (2009). *Probabilistic graphical models: principles and techniques*. The MIT Press.

Langseth H, Nielsen TD (2005). "Latent classification models." *Machine Learning*, **59**(3), 237–265.

Nodelman U, Koller D, Shelton C (2005). "Expectation Propagation for Continuous Time Bayesian Networks." In *Proc. of the 21st Conf. on Uncertainty in Artificial Intelligence*, pp. 431–440. Edinburgh, Scotland, UK.

Nodelman U, Shelton C, Koller D (2002a). "Continuous time Bayesian networks." In *Proc. of the 18th Conf. on Uncertainty in Artificial Intelligence*, pp. 378–387. Morgan Kaufmann Publishers Inc.

Nodelman U, Shelton C, Koller D (2002b). "Learning continuous time Bayesian networks." In *Proc. of the 19th Conf. on Uncertainty in Artificial Intelligence*, pp. 451–458. Morgan Kaufmann Publishers Inc.

Rabiner LR (1989). "A tutorial on hidden Markov models and selected applications in speech recognition." *Proceedings of the IEEE*, **77**(2), 257–286.

Rajaram S, Graepel T, Herbrich R (2005). "Poisson-networks: A model for structured point processes." In *Proc. of the 10th Int. Workshop on Artificial Intelligence and Statistics.*

Rand WM (1971). "Objective criteria for the evaluation of clustering methods." *Journal of the American Statistical association*, **66**(336), 846–850.

Simma A, Goldszmidt M, MacCormick J, Barham P, Black R, Isaacs R, Mortier R (2008). "CT-NOR: Representing and Reasoning About Events in Continuous Time." In DA McAllester, P Myllym (eds.), *Proc. of the 24th Conf. on Uncertainty in Artificial Intelligence*, pp. 484–493. AUAI Press.

Simma A, Jordan M (2010). "Modeling Events with Cascades of Poisson Processes." In *Proc. of the 26th Conf. on Uncertainty in Artificial Intelligence*, pp. 546–555. AUAI Press.

Stella F, Amer Y (2012). "Continuous time Bayesian network classifiers." *Journal of Biomedical Informatics*, **45**(6), 1108 – 1119.

Tormene P, Giorgino T, Quaglini S, Stefanelli M (2009). "Matching incomplete time series with dynamic time warping: an algorithm and an application to post-stroke rehabilitation." *Artificial Intelligence in Medicine*, **45**(1), 11–34.

Truccolo W, Eden U, Fellows M, Donoghue J, Brown E (2005). "A point process framework for relating neural spiking activity to spiking history, neural ensemble, and extrinsic covariate effects." *Journal of neurophysiology*, **93**(2), 1074–1089.

Voit E (2012). *A First Course in Systems Biology.* Garland Science: NY.

Witten IH, Frank E (2005). *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann.

Xu R, Wunsch D (2008). *Clustering*, volume 10. Wiley. com.

Yilmaz A, Javed O, Shah M (2006). "Object tracking: A survey." *Acm Computing Surveys (CSUR)*, **38**(4).

Zhong S, Langseth H, Nielsen TD (2012). "Bayesian networks for dynamic classification." *Technical report*, http://idi.ntnu.no/˜shket/dLCM.pdf.