Journal of Statistical Software

November 2014, Volume 62, Issue 3. *http://www.jstatsoft.org/*

Learning Continuous Time Bayesian Network Classifiers Using MapReduce

Simone Villa
University of Milano-Bicocca

Marco Rossetti
University of Milano-Bicocca

Abstract

Parameter and structural learning on continuous time Bayesian network classifiers are challenging tasks when you are dealing with big data. This paper describes an efficient scalable parallel algorithm for parameter and structural learning in the case of complete data using the MapReduce framework. Two popular instances of classifiers are analyzed, namely the continuous time naive Bayes and the continuous time tree augmented naive Bayes. Details of the proposed algorithm are presented using **Hadoop**, an open-source implementation of a distributed file system and the MapReduce framework for distributed data processing. Performance evaluation of the designed algorithm shows a robust parallel scaling.

Keywords: continuous time Bayesian network classifiers, parameter learning, structural learning, big data analysis, MapReduce, **Hadoop**.

1. Introduction

Bayesian networks are probabilistic graphical models that allow us to represent and reason about an uncertain domain (Pearl 1985). Bayesian networks allow us to learn causal relationships, trying to gain understanding about a problem domain and making predictions in the presence of interventions. They also avoid the overfitting of data (Heckerman 1999). Using these models, it is possible to describe and manage efficiently joint probability distributions in a variety of applications, from health diagnosis to finance modeling and traffic control (see Jensen and Nielsen 2007, for further details). An extension of Bayesian networks to model discrete time stochastic processes is offered by dynamic Bayesian networks (Dean and Kanazawa 1989), while recently continuous time Bayesian networks have been proposed to cope with continuous time stochastic processes (Nodelman, Shelton, and Koller 2002). This latter model explicitly represents temporal dynamics and allows us to query the network for distributions

over the times when particular events occur. Bayesian network models have been exploited to address the classification task, a basic task in data analysis that requires the construction of a classifier, which is a function that assigns a class label to instances described by a set of attributes (see Friedman, Geiger, and Goldszmidt 1997; Pavlovic, Frey, and Huang 1999; Stella and Amer 2012, for further details).

Inference and learning algorithms for continuous time Bayesian networks and their classifiers have been presented in the literature (Nodelman 2007; Stella and Amer 2012), and software implementations have been developed (Shelton, Fan, Lam, Lee, and Xu 2010). Learning algorithms have a main limitation, when the data size grows the learning time becomes unacceptable. To overcome this limitation, several parallelization alternatives are available in the specialized literature. One approach is to use a system with multiple central processing units (CPUs) and a shared-memory or a distributed-memory cluster made up of smaller shared-memory systems. This method requires vast resources and specialized parallel programming expertise. A recent approach consists of using graphics hardware because their performance is increasing more rapidly than that of CPUs. Graphics processing units (GPUs) are designed with a high parallel architecture due to the intrinsic parallel nature of graphics computations. For this reason the GPUs are transformed in general parallel computing devices for a wide range of applications (see Owens, Luebke, Govindaraju, Harris, Krüger, Lefohn, and Purcell 2007, for further details). A different approach is to use the MapReduce framework introduced by Dean and Ghemawat (2004). This framework offers the possibility to implement a parallel application without focusing on the details of data distribution, load balancing and fault tolerance (Dean and Ghemawat 2010). This model is inspired by the map and reduce functions, which are user defined and are used for problem decomposition. In the map phase the input data is processed by parallel mappers and passed to reducers as key-value pairs. The reducers take these pairs in input and aggregate the results. The most famous implementation of MapReduce is Apache **Hadoop** (The Apache Software Foundation 2014), which allows the distributed processing of large data sets across clusters of computers using simple programming models (White 2009). Scalability is one of the main feature of this framework, because it is designed to scale up from single server to thousands of machines, each offering local computation and storage.

Chu, Kim, Lin, Yu, Bradski, Ng, and Olukotun (2006) demonstrated that when an algorithm does sums over the data, the calculations can be easily distributed over multiple processing units. The key point is to divide the data into many pieces, give each core its part of the data, make calculations and aggregate the results at the end. This is called summation form and can be applied to different machine learning algorithms, such as in the field of Bayesian networks. Basak, Brinster, Ma, and Mengshoel (2012) applied the distributed computing of MapReduce to Bayesian parameter learning, both for traditional parameter learning (complete data) and the classical expectation maximization algorithm (incomplete data). However, to the best of our knowledge, no formulation of the MapReduce algorithms for parameter and structural learning of continuous time Bayesian network classifiers is available in the literature.

The paper is laid out as follows. Section 2 introduces the theory underpinning the continuous time Bayesian network classifiers and the learning framework. Section 3 describes the learning algorithm in the MapReduce framework. Section 4 gives the instructions on how to set up **Hadoop** and use our software. Section 5 presents the speedup in comparison to the sequential case, and compares various **Hadoop** configurations when increasing the dataset size, the **Hadoop** nodes, and the attributes. Finally, Section 6 concludes the paper.

2. Background

We introduce the necessary background on continuous time Bayesian network classifiers designed to solve the problem of supervised classification on multivariate trajectories evolving in continuous time. After the introduction of the basic notions on continuous time Bayesian networks (Nodelman *et al.* 2002), we describe two standard classifiers (Stella and Amer 2012), and we present the learning framework for these classifiers in the case of complete data.

2.1. Continuous time Bayesian networks

A continuous time Bayesian network is a graphical model whose nodes are finite state variables in which the state evolves continuously over time, and where the evolution of each variable depends on the state of its parents in the graph. This framework is based on homogeneous Markov processes, but utilizes ideas from Bayesian networks to provide a graphical representation language for these systems (Nodelman *et al.* 2002). Continuous time Bayesian networks have been used to model the presence of people at their computers (Nodelman and Horvitz 2003), for dynamical systems reliability modeling and analysis (Boudali and Dugan 2006), for network intrusion detection (Xu and Shelton 2008), to model social networks (Fan and Shelton 2009), and to model cardiogenic heart failure (Gatti, Luciani, and Stella 2012).

Definition 1 *Continuous time Bayesian network (CTBN; Nodelman* et al. *2002). Let* $\mathbf{X}$ *be a set of random variables* $X_1, X_2, \ldots, X_N$*. Each* X_n *has a finite domain of values* $Val(X_n) = \{x_1, x_2, \ldots, x_{I_n}\}$*. A continuous time Bayesian network* $\aleph$ *over* $\mathbf{X}$ *consists of two components: the first is an initial distribution* $\mathsf{P}^0_{\mathbf{X}}$*, specified as a Bayesian network* $\mathcal{B}$ *over* $\mathbf{X}$*, the second is a continuous time transition model specified as:*

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

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

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

for each instantiation $pa(X_n)$ of the parents $Pa(X_n)$ of node X_n, where I_n is the cardinality of X_n, $q_{x_i}^{pa(X_n)} = \sum_{x_j \neq x_i} q_{x_ix_j}^{pa(X_n)}$ can be interpreted as the instantaneous probability to leave x_i for a specific instantiation $pa(X_n)$ of $Pa(X_n)$, while $q_{x_ix_j}^{pa(X_n)}$ can be interpreted as the instantaneous probability to transition from x_i to x_j for an instantiation $pa(X_n)$ of $Pa(X_n)$. The IM can be represented using two sets of parameters: the set of intensities parameterizing the exponential distributions over when the next transition occurs, i.e., $\boldsymbol{q}_{X_n}^{pa(X_n)} = \{q_{x_i}^{pa(X_n)} : x_i \in Val(X_n)\}$, and the set of probabilities parameterizing the distribution over where the state transitions occur, i.e., $\boldsymbol{\theta}_{X_n}^{pa(X_n)} = \{\theta_{x_i,x_j}^{pa(X_n)} = q_{x_i,x_j}^{pa(X_n)} / q_{x_i}^{pa(X_n)} : x_i, x_j \in Val(X_n), x_i \neq x_j\}$.

As stated in Nodelman *et al.* (2002), a CTBN $\aleph$ is a factored representation of a homogeneous Markov process described by the joint intensity matrix:

$$\mathbf{Q}_{\aleph} = \prod_{X_n \in \mathbf{X}} \mathbf{Q}_{X_n}^{Pa(X_n)}, \tag{1}$$

that can be used to answer any query that involves time. For example, given two time points t_1 and t_2 with $t_2 \geq t_1$, and an initial distribution $\mathsf{P}_{\aleph}(t_1)$, we can compute the joint distribution over the two time points as:

$$\mathsf{P}_{\aleph}(t_1, t_2) = \mathsf{P}_{\aleph}(t_1) \exp(\mathbf{Q}_{\aleph}(t_2 - t_1)). \tag{2}$$

Continuous time Bayesian networks allow point evidence and continuous evidence. Point evidence is an observation of the value x_i of a variable X_n at a particular instant in time, i.e., $X_n(t) = x_i$, while continuous evidence provides the value of a variable throughout an entire interval, which we take to be a half-closed interval $[t_1, t_2)$. CTBNs are instantiated with a J-evidence-stream defined as follows.

Definition 2 *J-time-stream (Stella and Amer 2012). A J-time-stream, over the left-closed time interval $[0, T)$, is a partitioning into J left-closed intervals $[0, t_1); [t_1, t_2); \ldots; [t_{J-1}, T)$.*

Definition 3 *J-evidence-stream (Stella and Amer 2012). Given a CTBN $\aleph$, consisting of N nodes, and a J-time-stream $[0, t_1); [t_1, t_2); \ldots; [t_{J-1}, T)$, a J-evidence-stream is the set of joint instantiations $\mathbf{X} = \mathbf{x}$ for any subset of random variables X_n, $n = 1, 2, \ldots, N$ associated with each of the J time segments. A J-evidence-stream will be referred to as $(\mathbf{X}^1 = \mathbf{x}^1, \mathbf{X}^2 = \mathbf{x}^2, \ldots, \mathbf{X}^J = \mathbf{x}^J)$ or for short as $(\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^J)$.*

A J-evidence-stream is said to be fully observed in the case where the state of all the variables X_n is known along the whole time interval $[0, T)$, a J-evidence-stream which is not fully observed is said to be partially observed. With the CTBN model, it is possible to perform inference, structural learning and parameter learning.

Exact inference in a CTBN is intractable as the state space of the dynamic system grows exponentially with the number of variables, and thus several approximate algorithms have been proposed: Nodelman, Koller, and Shelton (2005a) introduced the expectation propagation algorithm, which allows both point and continuous evidence; Saria, Nodelman, and Koller (2007) provided an expectation propagation algorithm, which utilizes a general cluster graph architecture where clusters contain distributions that can overlap in both space and time; Fan, Xu, and Shelton (2010) proposed an approximate inference algorithm based on importance sampling; and El-Hay, Friedman, and Kupferman (2008) developed a Gibbs sampling procedure for CTBNs, which iteratively samples a trajectory for one of the components given the remaining ones.

Structural learning can be performed with complete and incomplete data. In the first case, Nodelman, Shelton, and Koller (2003) proposed a score based approach defining a Bayesian score for evaluating different candidate structures, and then using a search algorithm to find a structure that has a high score, while in the second case Nodelman, Shelton, and Koller (2005b) applied a structural expectation maximization to CTBNs. Maximum likelihood parameter learning can be done both for complete data (Nodelman *et al.* 2003), and for incomplete data via the expectation maximization algorithm (Nodelman *et al.* 2005b).

2.2. Continuous time Bayesian network classifiers

The continuous time Bayesian network model has been exploited to perform classification, a basic task in data analysis that assigns a class label to instances described by a set of values, which explicitly represent the evolution in continuous time of a set of random variables X_n, $n = 1, 2, \ldots, N$. These random variables are also called attributes in the context of classification.

Definition 4 *Continuous time Bayesian network classifier (CTBNC; Stella and Amer 2012). A continuous time Bayesian network classifier is a pair* $\mathcal{C} = \{\aleph, \mathsf{P}(Y)\}$ *where* $\aleph$ *is a CTBN model with attribute nodes* $X_1, X_2, \ldots, X_N$*, class node* Y *with marginal probability* $\mathsf{P}(Y)$ *on states* $Val(Y) = \{y_1, y_2, \ldots, y_K\}$*, and* $\mathcal{G}$ *is the graph, such that:*

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

In this paper we focus on the continuous time version of two popular classifiers: the naive Bayes, and the tree augmented naive Bayes, described in Friedman *et al.* (1997). The first is the simplest classifier in which all the attributes X_n are conditionally independent given the value of the class Y. This assumption is represented by its simple structure depicted in Figure 1(a) where each attribute (leaf in the graph) is only connected with the class variable (root in the graph). Since the conditional independence assumption is often unrealistic, a more general classifier has been introduced in order to capture the dependencies among attributes. These dependencies are approximated by using a tree structure imposed on the naive Bayes structure as shown in Figure 1(b). Formally, the two classifiers are defined as follows.

Definition 5 *Continuous time naive Bayes (CTBNC-NB; Stella and Amer 2012). A continuous time naive Bayes is a continuous time Bayesian network classifier* $\mathcal{C} = \{\aleph, \mathsf{P}(Y)\}$ *such that* $Pa(X_n) = Y, n = 1, 2, \ldots, N$.

Definition 6 *Continuous time tree augmented naive Bayes (CTBNC-TANB; Stella and Amer 2012). A continuous time tree augmented naive Bayes is a continuous time Bayesian network classifier* $\mathcal{C} = \{\aleph, \mathsf{P}(Y)\}$ *such that the following conditions hold:*

- $Y \in Pa(X_n), n = 1, 2, \ldots, N$*;*
- *the attribute nodes* X_n *form a tree, i.e.,* $\exists\ j \in \{1, 2, \ldots, N\}\ :\ |Pa(X_j)| = 1$*, while for* $i \neq j$*,* $i = 1, 2, \ldots, N\ :\ |Pa(X_i)| = 2$.

With the CTBNC model, it is possible to perform inference, structural learning and parameter learning. Given a dataset of fully observed trajectories, the parameter learning task consists of estimating the prior probability associated with the class node $\mathsf{P}(Y)$ and estimating the conditional intensity matrix for each attribute node X_n, while for the structural learning task an additional step of model search is required. Once the learning problem has been solved, the trained CTBNC can be used to classify fully observed J-evidence-streams using the exact inference algorithm presented by Stella and Amer (2012).

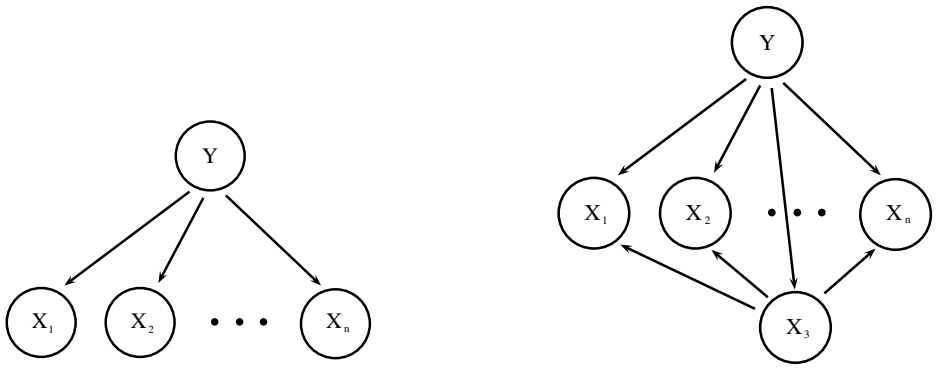


Figure 1: An instance of a continuous time naive Bayes classifier (a) in which all the attributes X_n are conditionally independent given the value of the class Y; and an instance of a continuous time tree augmented naive Bayes classifier (b) in which the dependencies among attributes are approximated by using a tree structure imposed on the naive Bayes structure.

2.3. Learning framework

A continuous time Bayesian network classifier $\mathcal{C}$ can be learned on a dataset $\mathcal{D}$ of fully observed J-evidence-streams using the standard Bayesian learning framework (see Koller and Friedman 2009, for further details). This paradigm requires to define a prior probability distribution $\mathsf{P}(\mathcal{C})$ over the space of possible classifiers and to update it using the Bayesian conditioning to obtain the posterior probability $\mathsf{P}(\mathcal{C}|\mathcal{D})$ over this space.

For continuous time Bayesian network classifiers, a model $\mathcal{C}$ consists of three components:

- The marginal probability associated with the class node $\mathsf{P}(Y)$: This marginal probability is independent from the classifier's structure $\mathcal{G}$ and it is not time-dependent. Given a prior probability on the class node, such as a uniform distribution over classes, it can be updated exploiting the available dataset.
- The structure $\mathcal{G}$: In our settings, two structures are possible: naive or tree augmented. Two examples are shown in Figures 1(a)–1(b).
- The values of the parameters $\boldsymbol{q}$ and $\boldsymbol{\theta}$ associated with the structure $\mathcal{G}$: These parameters define exponential $\boldsymbol{q}_{X_n}^{Pa(X_n)}$ and multinomial distributions $\boldsymbol{\theta}_{X_n}^{Pa(X_n)}$ for each variable X_n and each assignment of values $pa(X_n)$ of $Pa(X_n)$ in the graph $\mathcal{G}$.

In order to define the prior $\mathsf{P}(\mathcal{C})$ over the space of possible classifiers, we need to specify a probability distribution over graph structures $\mathsf{P}(\mathcal{G})$ and, for each possible graph $\mathcal{G}$, a density measure over possible values of the parameters $\mathsf{P}(\boldsymbol{q}, \boldsymbol{\theta}|\mathcal{G})$.

The prior over structures $\mathsf{P}(\mathcal{G})$ does not grow with the size of the data; a simple prior such as a uniform is often chosen. A key property of this prior is that it must satisfy structure modularity (Friedman and Koller 2000), so that the prior decomposes into a product with a term for each parent in $\mathcal{G}$:

$$\mathsf{P}(\mathcal{G}) = \prod_{X_n} \mathsf{P}(Pa(X_n) = Pa_{\mathcal{G}}(X_n)). \tag{3}$$

The prior over parameters $\mathsf{P}(\boldsymbol{q}, \boldsymbol{\theta}|\mathcal{G})$ is selected in order to satisfy the following three assumptions.

Parameter modularity: If a node X_n has the same parents $Pa(X_n)$ in two distinct graphs, then the probability density functions of the parameters associated with this node must be identical, i.e., for each $\mathcal{G}$ and $\mathcal{G}'$ such that $Pa_{\mathcal{G}}(X_n) = Pa_{\mathcal{G}'}(X_n)$:

$$\mathsf{P}(\boldsymbol{q}_{X_n}^{Pa_{\mathcal{G}}(X_n)}, \boldsymbol{\theta}_{X_n}^{Pa_{\mathcal{G}}(X_n)}|\mathcal{G}) = \mathsf{P}(\boldsymbol{q}_{X_n}^{Pa_{\mathcal{G}'}(X_n)}, \boldsymbol{\theta}_{X_n}^{Pa_{\mathcal{G}'}(X_n)}|\mathcal{G}'). \tag{4}$$

Global parameter independence: The parameters associated with each variable in a network structure are independent. So the prior over parameters can be decomposed by variable as follows:

$$\mathsf{P}(\boldsymbol{q}, \boldsymbol{\theta}|\mathcal{G}) = \prod_{X_n} \mathsf{P}(\boldsymbol{q}_{X_n}^{Pa(X_n)}, \boldsymbol{\theta}_{X_n}^{Pa(X_n)}|\mathcal{G}). \tag{5}$$

Local parameter independence: The parameters associated with each state of the parents of a variable are independent. So the parameters of each variable X_n are decomposable by parent configuration $pa(X_n)$ in $Pa(X_n)$ as follows:

$$\mathsf{P}(\boldsymbol{q}_{X_n}^{Pa(X_n)}, \boldsymbol{\theta}_{X_n}^{Pa(X_n)}|\mathcal{G}) = \prod_{pa(X_n)} \prod_{x_i} \mathsf{P}(q_{x_i}^{pa(X_n)}|\mathcal{G}) \times \prod_{pa(X_n)} \prod_{x_i} \mathsf{P}(\boldsymbol{\theta}_{x_i}^{pa(X_n)}|\mathcal{G}). \tag{6}$$

Once the prior is defined, it is possible to compute the form of the posterior probability using the Bayes rule $\mathsf{P}(\mathcal{G}|\mathcal{D}) \propto \mathsf{P}(\mathcal{D}|\mathcal{G})\mathsf{P}(\mathcal{G})$. The marginal likelihood $\mathsf{P}(\mathcal{D}|\mathcal{G})$ is defined as the integral of the likelihood function over all the possible parameter values for $\mathcal{G}$:

$$\mathsf{P}(\mathcal{D}|\mathcal{G}) = \int_{\boldsymbol{q},\boldsymbol{\theta}} \mathsf{P}(\mathcal{D}|\mathcal{G}, \boldsymbol{q}, \boldsymbol{\theta})\mathsf{P}(\boldsymbol{q}, \boldsymbol{\theta}|\mathcal{G}) d\boldsymbol{q} d\boldsymbol{\theta}. \tag{7}$$

In the case of no missing values, the probability of the data given a classifier $\mathsf{P}(\mathcal{D}|\mathcal{G}, \boldsymbol{q}, \boldsymbol{\theta})$ can be decomposed as a product of likelihoods of the parameters $\boldsymbol{q}$ and $\boldsymbol{\theta}$:

$$\mathsf{P}(\mathcal{D}|\mathcal{G}, \boldsymbol{q}, \boldsymbol{\theta}) = \prod_{X_n} L(\boldsymbol{q}_{X_n}^{Pa(X_n)}|\mathcal{D}) \times \prod_{X_n} L(\boldsymbol{\theta}_{X_n}^{Pa(X_n)}|\mathcal{D}). \tag{8}$$

Using the global parameter independence assumption (5) and the decomposition (8), the marginal likelihood $\mathsf{P}(\mathcal{D}|\mathcal{G})$ can be written as the product of marginal likelihoods of $\boldsymbol{q}$ and $\boldsymbol{\theta}$:

$$\begin{aligned} \mathsf{P}(\mathcal{D}|\mathcal{G}) &= \prod_{X_n} \int_{\boldsymbol{q}} L(\boldsymbol{q}_{X_n}^{Pa(X_n)}|\mathcal{D})\mathsf{P}(\boldsymbol{q}_{X_n}^{Pa(X_n)}) d\boldsymbol{q} \times \prod_{X_n} \int_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_{X_n}^{Pa(X_n)}|\mathcal{D})\mathsf{P}(\boldsymbol{\theta}_{X_n}^{Pa(X_n)}) d\boldsymbol{\theta} \\ &= \prod_{X_n} ML(\boldsymbol{q}_{X_n}^{Pa(X_n)}|\mathcal{D}) \times \prod_{X_n} ML(\boldsymbol{\theta}_{X_n}^{Pa(X_n)}|\mathcal{D}). \end{aligned} \tag{9}$$

It is possible to extend the Bayesian-Dirichlet equivalent metric introduced by Heckerman, Geiger, and Chickering (1995) for the marginal likelihood $\mathsf{P}(\mathcal{D}|\mathcal{G})$ and to compute it in a closed form solution from the prior and the sufficient statistics over the data. These statistics are the same as for the continuous time Bayesian networks.

Definition 7 *Sufficient statistics (Nodelman et al. 2003). The sufficient statistics for the transition dynamics of a CTBN over* $\mathbf{X}$ *decompose as a set for each variable* $X_n \in \mathbf{X}$ *as:*

- $T_{x_i}^{pa(X_n)}$, *amount of time that* $X_n = x_i$, *while* $Pa(X_n) = pa(X_n)$, *and*
- $MM_{x_i,x_j}^{pa(X_n)}$, *number of transitions from* $X_n = x_i$ *to* $X_n = x_j$, *while* $Pa(X_n) = pa(X_n)$.

From the last statistic, we can define $M_{x_i}^{pa(X_n)} = \sum_{i \neq j} MM_{x_i,x_j}^{pa(X_n)}$, *the number of transitions leaving the state* $X_n = x_i$, *while* $Pa(X_n) = pa(X_n)$.

Given the sufficient statistics, it is possible to define the parameters of each entry of the intensity matrix as:

$$\hat{q}_{x_i}^{pa(X_n)} = \frac{\alpha_{x_i}^{pa(X_n)} + M_{x_i}^{pa(X_n)}}{\tau_{x_i}^{pa(X_n)} + T_{x_i}^{pa(X_n)}}, \tag{10}$$

$$\hat{\theta}_{x_i,x_j}^{pa(X_n)} = \frac{\alpha_{x_i,x_j}^{pa(X_n)} + MM_{x_i,x_j}^{pa(X_n)}}{\alpha_{x_i}^{pa(X_n)} + M_{x_i}^{pa(X_n)}}, \tag{11}$$

where M, MM, and T are the sufficient statistics previously defined, while α and τ denote the pseudocounts (prior counts) of the number of transitions from one state to another state and the amount of time spent in a state respectively (see Nodelman 2007, for further details). The closed form solution of the marginal likelihood of $\boldsymbol{q}$ reported in Equation 9 is:

$$ML(\boldsymbol{q}_{X_n}^{Pa(X_n)}|\mathcal{D}) = \prod_{pa(X_n)} \prod_{x_i} \frac{\Gamma\left(\alpha_{x_i}^{pa(X_n)} + M_{x_i}^{pa(X_n)} + 1\right) \left(\tau_{x_i}^{pa(X_n)}\right)^{(\alpha_{x_i}^{pa(X_n)}+1)}}{\Gamma\left(\alpha_{x_i}^{pa(X_n)} + 1\right) \left(\tau_{x_i}^{pa(X_n)} + T_{x_i}^{pa(X_n)}\right)^{(\alpha_{x_i}^{pa(X_n)}+M_{x_i}^{pa(X_n)}+1)}}, \tag{12}$$

and the marginal likelihood of $\boldsymbol{\theta}$ is:

$$ML(\boldsymbol{\theta}_{X_n}^{Pa(X_n)}|\mathcal{D}) = \prod_{pa(X_n)} \prod_{x_i = x_j} \frac{\Gamma\left(\alpha_{x_i}^{pa(X_n)}\right)}{\Gamma\left(\alpha_{x_i}^{pa(X_n)} + M_{x_i}^{pa(X_n)}\right)} \times \prod_{x_i \neq x_j} \frac{\Gamma\left(\alpha_{x_i,x_j}^{pa(X_n)} + MM_{x_i,x_j}^{pa(X_n)}\right)}{\Gamma\left(\alpha_{x_i,x_j}^{pa(X_n)}\right)}. \tag{13}$$

The task of inferring the set of conditional dependencies is expressed only for the tree augmented naive Bayes since the structure of the naive Bayes is fixed by definition. This task can be accomplished evaluating the Bayesian score of different tree structures and searching the structure that has the higher score. The Bayesian score BS is obtained taking the logarithm of the marginal likelihood $\mathsf{P}(\mathcal{D}|\mathcal{G})$ and the logarithm of the prior $\mathsf{P}(\mathcal{G})$:

$$\begin{aligned} BS(\mathcal{G}|\mathcal{D}) &= \ln \mathsf{P}(\mathcal{D}|\mathcal{G}) + \ln \mathsf{P}(\mathcal{G}) \\ &= \sum_{X_n} \ln ML(\boldsymbol{q}_{X_n}^{pa(X_n)}|\mathcal{D}) + \ln ML(\boldsymbol{\theta}_{X_n}^{pa(X_n)}|\mathcal{D}) + \ln \mathsf{P}(\mathcal{G}). \end{aligned} \tag{14}$$

The search space over the tree structures can be done in polynomial time (given a fixed number of parents of a node) and it is possible to optimize the parent set for each variable independently. The search can be easily performed enumerating each possible tree structure compliant with the definition of continuous time tree augmented naive Bayes classifier.

3. Learning in the MapReduce framework

We present the learning algorithms for continuous time Bayesian network classifiers in the MapReduce framework. Since structural and parameter learning rely on the computation of the sufficient statistics, we present one map function and one reduce function for both tasks. The primary motivation of using this programming model is that it simplifies large scale data processing tasks allowing programmers to express concurrent computations while hiding low level details of scheduling, fault tolerance, and data distribution (Dean and Ghemawat 2004).

MapReduce programs are expressed as sequences of map and reduce operations performed by the mapper and the reducer respectively. A mapper takes as input parts of the dataset, applies a function (e.g., a partition of the data), and produces as output key-value pairs, while a reducer takes as input a list indexed by a key of all corresponding values and applies a reduction function (e.g., aggregation or sum operations) on the values. Once a reducer has terminated its work, the next set of mappers can be scheduled. Since a reducer must wait for all mapper outputs, the synchronization is implicit in the reducer operation, while fault tolerance is achieved by rescheduling mappers that time out.

3.1. Key concepts

The design of the learning algorithms is based on some basic patterns used in MapReduce (Lin and Dyer 2010). The main idea is to exploit the peculiarities of the continuous time Bayesian network classifiers presented in Section 2 to parallelize the operations of structural and parameter learning. Through appropriate structuring of keys and values it is possible to use the MapReduce execution framework to bring together all the pieces of data required to perform the learning computation. In our case, the key-value pairs are constructed in order to encode all the information relevant for the description of the classifier, i.e., the marginal probability of the class, the structure, and the parameters associated with the structure. Two types of key-value pairs are used: a key with the identifier of the class node and a value containing the structure for the computation of the marginal probability, and a key with the identifier of the node given its parents and a value containing the structure for the calculation of the sufficient statistics.

We use the strips approach introduced by Lin (2008) to generate the output keys of the mapper, instead of emitting intermediate key-value pairs for each interval, this information is first stored in a map denoted as *paMap*. The mapper emits key-value pairs with text as keys and corresponding maps as values. The MapReduce execution framework guarantees that all associative arrays with the same key will be brought together in the reduce step. This last phase aggregates the results by computing the sufficient statistics and the estimation of the parameters of the conditional intensity matrix and of the Bayesian score. It is possible to further increase the performance by means of the use of combiners. This approach assumes that the map *paMap* fits into memory; such a condition is reasonable since the number of transitions of each variable, given the instantiation of its parents, is generally bounded.

As in Basak *et al.* (2012), we have tested the correctness of the algorithms by comparing the results generated by MapReduce against sequential versions. The algorithms were executed on Linux EC2 computers to compute the learning tasks given the same data. We found that the outputs of the two versions of the algorithms were exactly the same and thus we concluded that the MapReduce algorithms were the correct implementation of the sequential versions.

Algorithm 1 Map

Require: fully observed J-evidence-stream $(id, y, \mathbf{x}^1, \ldots, \mathbf{x}^J)$ and structure $\mathcal{G}$ (optional).
Ensure: key-value pairs in the following forms: if the key is denoted by *CLASS* then the value is $\langle id, y \rangle$, while if the key is $(X_n|Pa(X_n))$ then the value is the map *paMap*.

1: **emit**$\langle CLASS, \langle id, y \rangle \rangle$
2: **for** $n \leftarrow 1$ to N **do**
3: **for** $p \leftarrow 1$ to N **do**
4: $Pa(X_n) \leftarrow (Y, X_p)$
5: **if** $n = p$ **then**
6: $Pa(X_n) \leftarrow (Y, \emptyset)$
7: **end if**
8: **if** *AnalyzeParents* $(Pa(X_n))$ **then**
9: $paMap \leftarrow Map()$
10: **for** $j \leftarrow 2$ to J **do**
11: $paMap \leftarrow IncrementT\ (paMap, pa(X_n)^{j-1}, (x_n^{j-1}, x_n^j), t_j - t_{j-1})$
12: $paMap \leftarrow IncrementM\ (paMap, pa(X_n)^{j-1}, (x_n^{j-1}, x_n^j), 1)$
13: **end for**
14: **emit**$\langle (X_n|Pa(X_n)), paMap \rangle$
15: **end if**
16: **end for**
17: **end for**

3.2. Map function

The main task of the map function is to count the transitions and the relative amount of time in the fully observed J-evidence-stream of each variable X_n given the instantiation $pa(X_n)$ of its parents $Pa(X_n)$. In the case of structural learning, every possible combination of parents for each node must be computed subject to the constraint of the tree structure, while in the case of parameter learning the structure $\mathcal{G}$ is defined as input. The key-value pairs for the class probability are constructed as textual keys denoted by *CLASS* and values containing the identifier of the J-evidence-stream and the relative class. The key-value pairs for the parameters are constructed as a textual keys encoding the variable name and the names of its parents, i.e., $(X_n|Pa(X_n))$, and values containing a two level association of an ID corresponding to the instantiation of parents, another ID corresponding to a transition, and the count and the elapsed time of that transition, i.e., $\langle pa(X_n), \langle (x_n^{j-1}, x_n^j), (count, time) \rangle \rangle$.

In Algorithm 1 the pseudo code of the map function is given. It takes as input a fully observed J-evidence-stream $(id, y, \mathbf{x}^1, \ldots, \mathbf{x}^J)$ with the corresponding identifier id and class y, and the structure $\mathcal{G}$ (only for the parameter learning), while it produces as output key-value pairs previously described. In line 1, a key-value pair is emitted for the computation of the class probability. The for statement in line 2 ranges over the N attributes to analyze every node, while the for statement in line 3 ranges over the the N attributes to compute every possible parent combination. In line 8, the parent configuration is analyzed: in the structural learning this function gives always true because every combination of parents must be analyzed, while in the parameter learning only the combinations compatible with the input structure are analyzed. In line 9, the map for the count and time statistics is initialized. The for statement in line 10 ranges over the stream and the functions in line 11 and 12 update the map.

Algorithm 2 Reduce

Require: a key and a list of maps $\{paMap_1, \ldots, paMap_S\}$, α and τ parameters (optional).
Ensure: class probability, conditional intensity matrix, and Bayesian score (optional).

1: **for** $s \leftarrow 1$ to S **do**
2: $paMap \leftarrow Merge\ (paMap,\ paMap_s)$
3: **end for**
4: **if** $key = CLASS$ **then**
5: $marg \leftarrow \emptyset$
6: **for** $y \leftarrow Values\ (paMap)$ **do**
7: $marg(y) \leftarrow marg(y) + 1$
8: **end for**
9: **emit**$\langle CLASS,\ marg \rangle$
10: **else**
11: $bs \leftarrow 0$
12: **for** $pa(X_n) \leftarrow Keys\ (paMap)$ **do**
13: $trMap \leftarrow paMap\ [\ pa(X_n)\ ]$
14: $T \leftarrow \emptyset, M \leftarrow \emptyset, MM \leftarrow \emptyset$
15: **for** $(x_i,\ x_j) \leftarrow Keys\ (trMap)$ **do**
16: $T(x_i) \leftarrow T(x_i) + GetT\ (trMap\ [\ (x_i,\ x_j)\ ])$
17: $MM(x_i,\ x_j) \leftarrow MM(x_i,\ x_j) + GetMM\ (trMap\ [\ (x_i,\ x_j)\ ])$
18: **if** $x_i \neq x_j$ **then**
19: $M(x_i) \leftarrow M(x_i) + getM\ (trMap\ [\ (x_i,\ x_j)\ ])$
20: **end if**
21: **end for**
22: $im \leftarrow ComputeIM\ (T,\ M,\ MM,\ \alpha,\ \tau)$
23: $bs \leftarrow bs + ComputeBS\ (T,\ M,\ MM,\ \alpha,\ \tau)$
24: **emit**$\langle CIM, \langle (key, pa(X_n)),\ im \rangle \rangle$
25: **end for**
26: **emit**$\langle BS, \langle key, bs \rangle \rangle$
27: **end if**

3.3. Reduce function

The task of the reduce function is to provide the basic elements for the description of the classifier, namely the class probability and the conditional intensity matrices. In Algorithm 2 the pseudo code of the reduce function is given. It takes as input key-value pairs where the keys can be *CLASS* or $(X_n|Pa(X_n))$ and the values are collections of data computed by mappers with the same key, while it produces as output key-value pairs for the model description. From line 1 to line 3 the values are merged in a single map named *paMap*. If the key is *CLASS* (from line 4 to 9), then the marginal probability of the class node is computed, otherwise (from line 10 to line 26) the BS and the CIM are calculated according to Equation 14. The for statement in line 12 ranges over all the possible instantiations $pa(X_n)$ and the for statement in line 15 ranges over all the possible transitions (x_i, x_j) to compute the sufficient statistics T, M, and MM. The functions *getT* and *getM* get time and counts from the map *trMap*. The function in line 22 computes the IM related to the current parent instantiation, and the function in line 23 calculates the BS only in the case of structural learning.

Algorithm 3 ComputeIM

Require: maps containing the counting values T, M and MM of the node X_n when $Pa(X_n) = pa(X_n)$, α and τ parameters (optional).

Ensure: the intensity matrix $\boldsymbol{Q}_{X_n}^{pa(X_n)}$ for the node X_n when $Pa(X_n) = pa(X_n)$.

1: **for** $(x_i, x_j) \in Transitions\ (X_n)$ **do**
2: **if** $x_i \neq x_j$ **then**
3: $q(x_i) \leftarrow \frac{M(x_i) + \alpha(x_i)}{T(x_i) + \tau(x_i)}$
4: **else**
5: $q(x_i,\ x_j) \leftarrow \frac{MM(x_i,\, x_j) + \alpha(x_i,\, x_j)}{T(x_i) + \tau(x_i)}$
6: **end if**
7: **end for**

Algorithm 4 ComputeBS

Require: maps containing the counting values T, M and MM of the node X_n when $Pa(X_n) = pa(X_n)$, α and τ parameters (optional).

Ensure: Bayesian score of the node X_n when $Pa(X_n) = pa(X_n)$.

1: $bs \leftarrow 0$
2: **for** $(x_i, x_j) \in Transitions\ (X_n)$ **do**
3: **if** $x_i \neq x_j$ **then**
4: $bs \leftarrow bs + \ln\Gamma(\alpha(x_i,\, x_j) + MM(x_i,\, x_j)) - \ln\Gamma(\alpha(x_i,\, x_j))$
5: **else**
6: $bs \leftarrow bs + \ln\Gamma(\alpha(x_i)) - \ln\Gamma(\alpha(x_i) + M(x_i))$
7: $bs \leftarrow bs + \ln\Gamma(\alpha(x_i) + M(x_i) + 1) + (\alpha(x_i) + 1) \times \ln(\tau(x_i))$
8: $bs \leftarrow bs - \ln\Gamma(\alpha(x_i) + 1) - (\alpha(x_i) + M(x_i) + 1) \times \ln(\tau(x_i) + T(x_i))$
9: **end if**
10: **end for**

3.4. Auxiliary functions

The mapper and the reducer rely on auxiliary functions in order to compute their outputs, the main functions are *ComputeIM* and *ComputeBS* used by the reducer. The first function computes the intensity matrix of the variable X_n given an instantiation of its parents by means of counting maps according to Equations 10–11. Its pseudo code is reported in Algorithm 3. The second function computes the Bayesian score of the variable X_n given an instantiation of its parents according to Equation 14. Its pseudo code is reported in Algorithm 4.

In our implementation of the learning framework, we made some modifications in order to improve the performance. We use the MapReduce environment to split the dataset into slices $\mathcal{D}_1, \ldots, \mathcal{D}_D$ and then send these parts to the mappers instead of each single stream. The map function has been slightly modified accordingly: if we have a fully observed J-evidence-stream, we emit the key-value pair for the calculation of the marginal probability, otherwise we elaborate the stream in the usual way. We use combiners that receive as input all data emitted by the mappers on a given computational node and their outputs are sent to the reducers. The combiner code is the same as that reported in Algorithm 2 from line 1 to 3. Moreover, when the Bayesian score is computed for every parent configuration, the driver chooses the structure that maximizes the Bayesian score subject to the model constraints. The final structure and the corresponding conditional intensity matrices are given as output.

4. Program installation and usage

This section gives the instructions to set up a single-node **Hadoop** installation, to run our software on users computers, and to set up Amazon Web Service (AWS; Amazon.com, Inc. 2014) to run our implementation on multiple nodes. Moreover, an example of using the software is given. The instructions for the single computer installation work on a Linux OS, but can also work on a Mac OS or a Windows PC following some modifications which can be found on the web (The Apache Software Foundation 2014). The **Hadoop** cluster can run in one of the three supported modes.

- Local mode: the non-distributed mode to run **Hadoop** on a single Java process.
- Pseudo-distributed mode: the local pseudo-distributed mode to run **Hadoop** where each **Hadoop** daemon runs in a separate process. It is an emulation of the real distribution.
- Fully-distributed mode: the real fully-distributed mode where **Hadoop** processes run on different clusters.

In the pseudo and fully distributed mode, the system works with a single master (jobtracker), which coordinates many slaves (tasktrackers): the jobtracker receives the job from the user, distributes map and reduce tasks to the tasktrackers and checks the work flow handling failures and exceptions. Data is managed by the **Hadoop** distributed file system (HDFS), which is responsible for the distribution of it to all the tasktrackers.

4.1. Pseudo-distributed installation

To use this software the following requirements must be satisfied:

1. A supported version of GNU/Linux with Java VM 1.6.x, preferably from Sun.
2. Secure shell (SSH) must be installed and its daemon (SSHD) must be up and running to use the **Hadoop** scripts that manage remote **Hadoop** daemons.
3. The Apache **Hadoop** platform (release 1.0.3 or higher).

To run the proposed software, you need to install, configure and run **Hadoop**, then load the data into the HDFS, and finally run our implementation.

To run **Hadoop**, download the binary version and decompress it into a folder. From here on, let us assume that we extract **Hadoop** into the folder `/opt/hadoop/`. After that, edit the file `conf/hadoop-env.sh` to set the variable `JAVA_HOME` to be the root of the Java installation and create a system environment variable called `HADOOP_PREFIX` which points to the root of the **Hadoop** installation folder `/opt/hadoop/`. In order to invoke **Hadoop** from the command line, edit the `PATH` variable adding the `$HADOOP_PREFIX/bin`.

These operations can be done by editing the `.bashrc` file in the user home directory and adding the following lines:

```
export HADOOP_PREFIX=/opt/hadoop
export PATH=$PATH:$HADOOP_PREFIX/bin
```

The next step is to edit three configuration files, `core-site.xml`, `hdfs-site.xml`, and `mapred-site.xml`, in the directory `conf` as specified by the following configuration files:

```
<!-- content of core-site.xml -->
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:54310</value>
  </property>
</configuration>

<!-- content of hdfs-site.xml -->
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
<!-- content of mapred-site.xml -->
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:54311</value>
  </property>
</configuration>
```

To communicate with each other, the nodes need to use the command `ssh` without a passphrase. If the system is already configured, the command `ssh localhost` will work. If the command `ssh` on `localhost` does not work, use the following two commands:

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

When the SSH is configured, the namenode must be formatted with the following command:

```
$ hadoop namenode -format
```

Finally, **Hadoop** can be started with the following command:

```
$ start-all.sh
```

While **Hadoop** is up and running, the data can be loaded into the HDFS file system with:

```
$ hadoop fs -put /opt/Trajectories/dataset.csv /Trajectories/dataset.csv
```

With this command, **Hadoop** copies the file `dataset.csv` from the local directory named `/opt/Trajectories/` to the HDFS directory `/Trajectories/`, while the full HDFS path is:

```
hdfs://localhost:54310/Trajectories/dataset.csv
```

4.2. Distributed installation

In order to use the software with a real distributed environment, we used AWS to run our experiments into the cloud. AWS is a set of services and infrastructures which gives the possibility to run software in the cloud on the Amazon infrastructure. Specifically, we used:

- Amazon elastic MapReduce (EMR): This service is a simple tool which can be used to run **Hadoop** clusters on Amazon instances already configured with **Hadoop**. In the following section, we explain how to use this service in order to run the software.
- Amazon simple storage service (S3): This service is an on-line repository which can be used to host the user's files on the Amazon infrastructure. In the following section, we explain how to use this service to load the software and the dataset.

In order to run the proposed software on the Amazon infrastructure, the user has to load the software and the dataset in an S3 bucket. This operation can be done using the S3 section of the AWS console available in a web browser. Once in the S3 page, click on the `Create Bucket` button and give a name to the bucket. When the bucket is created, use the `Upload` button to load the dataset and the software. Let us assume that our bucket name is `ctbnc` and we loaded the software and the dataset in the root directory, their paths will be `s3n://ctbnc/ctbnc_mr.jar` (software) and `s3n://ctbnc/dataset.csv` (dataset).

When the software and the dataset are loaded, use the AWS console to reach the Amazon EMR section. This page gives the list of the last clusters run or that are still running and gives the choice to start a cluster that stays alive after the execution of a task or an infrastructure that terminates as soon as the task is completed. Hereafter the second case is described.

Press the button `Create Cluster` to enter in a wizard which gives the possibility to configure and start the cluster. The parameters to be configured belong to the following groups:

- Cluster configuration: The user has to define the name of the cluster (Cluster name), deactivate the Termination protection, activate the Logging in a given S3 folder (with write permissions) and deactivate the Debugging option.
- Software configuration: The user has to define which kind of **Hadoop** distribution must be used and the software to be installed. In this case the Amazon distribution with Amazon machine images (AMI) version 2.4.2 is needed and no additional software is required. If any software is already selected, it can be removed.
- Hardware configuration: The user has to define the Master instance group type (Large), the Core instance group count ($\langle N \rangle$), and Core instance group type (Large).
- Steps: The user has to specify that the cluster must execute a custom JAR. The S3 path of the JAR must be given with all the parameters needed. The user can choose the option to terminate on failure or to auto-terminate the cluster after the last step is completed.

The other parameters are not relevant for our case, so the user can click the `Create Cluster` button. In the main EMR dashboard the user can see his cluster as a row in the table. If there are no problems, the execution phase passes through: `STARTING`, `RUNNING`, `SHUTTING_DOWN` and `COMPLETED`. If something goes wrong, the user can check the console messages and logs.

4.3. Usage

To run the software on **Hadoop** under Linux, use the following command:

```
$ hadoop jar ctbnc_mr.jar <inputFile> <tempFolder> \
>    <outputFile> <-P|-S> [structureFile]
```

where `inputFile`, `tempFolder`, `outputFile` and `structureFile` are full HDFS paths. The launch command should be executed in the same directory where `ctbnc_mr.jar` is located. If Amazon EMR is used, then the S3 path of `ctbnc_mr.jar` must be specified in the `JAR S3 location` field, while the software arguments can be passed in the `Arguments` field. Note that in this specific case, the aforementioned paths must be valid S3 paths.

The parameter `inputFile` must point to a flat csv file containing fully observed J-evidence-streams in which each row corresponds to the temporal evolution of each stream, while each column consists of the trajectory identifier id, the elapsed time from the beginning of the trajectory, the relative class y, and the instantiations of the variables $X_1, \ldots, X_n$. An excerpt of a dataset with one binary class node and six binary attribute nodes is reported hereafter.

```
trajectory,time,class,a1,a2,a3,a4,a5,a6
1,0.00,2,1,2,1,2,1,2
1,0.08,2,1,1,1,2,1,2
1,0.12,2,1,1,2,2,1,2
1,0.16,2,1,1,2,2,1,1
1,0.20,2,1,1,2,2,2,1
2,0.00,1,1,1,1,1,1,1
2,0.05,1,1,1,1,1,1,2
2,0.10,1,1,2,1,1,1,2
2,0.20,1,1,2,1,2,1,2
```

The `tempFolder` is the directory where the software saves the intermediate results (the output of the reducers), while `outputFile` is the final result of the software, which is a tab delimited file containing the marginal class probability and the conditional intensity matrix of each variable X_n decomposed by $q_{x_i x_j}^{pa(X_n)}$ as shown hereafter.

```
MARG class 1 _ 0.25
MARG class 2 _ 0.75
CIM a1|class,a2 1,1 1,1 -134.5999
CIM a1|class,a2 1,1 1,2  134.5999
```

The parameter `-P` or `-S` indicates if the parameter learning or the structural learning must be executed. If the parameter learning is selected, then the user can specify an optional structure file with the tree augmented naive Bayes structure, otherwise the naive Bayes will be used. The structure file represents the edge list in the form $parent, child$ as in the following example:

```
a1,a2
a2,a3
a3,a4
a4,a1
a5,a6
```

The classifier's model can be also represented using the data structures provided by R (R Core Team 2014). The function `readCTBNCmodel` contained in the file `readCTBNC.R` reads the output of the **Hadoop** software and produces the R version of the classifier's model in the form of nested lists.

```
R> source(sourceFile)
R> results <- readCTBNCmodel(outputFile)
```

The marginal probability of the class node is contained in `results$MARG`.

```
R> print(results$MARG)

$class
  Prob
1 0.25
2 0.75
```

The conditionally intensity matrices of the model are stored in `results$CIM`. Each conditional intensity matrix can be reached by giving the variable name, while each intensity matrix can be indexed by the variable name and the instantiations of its parents.

```
R> print(results$CIM$a1)

$'class=1,a2=1'
          1         2
1 -134.5999  134.5999
2  918.6448 -918.6448

$'class=1,a2=2'
          1         2
1 -978.6699  978.6699
2  110.5335 -110.5335

$'class=2,a2=1'
          1          2
1 -222.0433   222.0433
2 1337.4396 -1337.4396

$'class=2,a2=2'
          1          2
1 -1167.4237 1167.4237
2    70.4580  -70.4580

R> print(results$CIM$a1$'class=1,a2=1')

          1         2
1 -134.5999  134.5999
2  918.6448 -918.6448
```

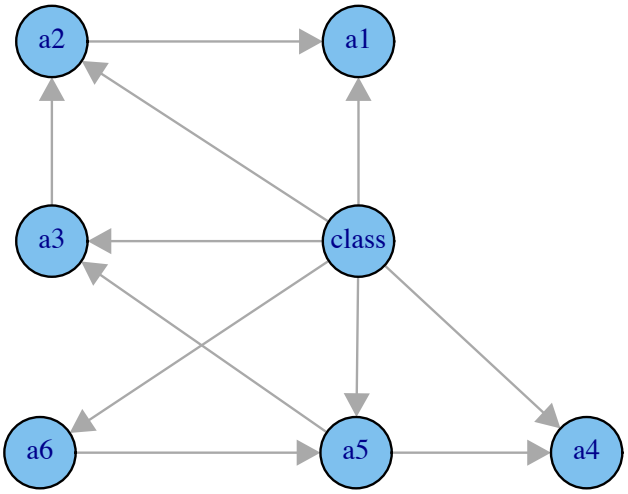


Figure 2: Visualization of the classifier's structure using the package **igraph**. It is an instance of a continuous time tree augmented naive Bayes classifier with six attribute nodes. The dependencies among these attributes are approximated by using a tree in which the root is the node named a6.

The graph structure in the form of an adjacency list is stored in `results$GRAPH`.

```
R> print(results$GRAPH)
```

```
$ADJLIST
    from to
1  class a1
2     a2 a1
3  class a2
4     a3 a2
5  class a3
6     a5 a3
7  class a4
8     a5 a4
9  class a5
10    a6 a5
11 class a6
```

It is possible to visualize the graph using the package **igraph** (Csardi and Nepusz 2006) and its `plot` functions. Figure 2 shows the graph obtained by the execution of the following code.

```
R> library("igraph")
R> g <- graph.data.frame(results$GRAPH$ADJLIST, directed = TRUE)
R> plot(g, layout = layout.fruchterman.reingold(g))
R> tkplot(g)
```

5. Experiments

We tested the proposed software in the parameter learning of a continuous time naive Bayes classifier and in the structural learning of a continuous time tree augmented naive Bayes. We made three types of experiments changing the dataset size, the number of **Hadoop** nodes, and the number of attributes. We compared the speedup of the proposed software versus the sequential version of the algorithm described in Stella and Amer (2012). The dataset is composed of a text file containing fully observed J-evidence-streams. These streams concern high frequency transaction data of the Foreign Exchange market (see Villa and Stella 2014, for further details). Our tests are performed using M1 Large instances of Amazon EMR, while the training and output data are stored in Amazon S3.

5.1. Increasing the dataset size

In the first experiment, we measure the performance of the MapReduce algorithm in the case of parameter learning of a continuous time naive Bayes classifier. We use 1 Master instance and 5 Core instances against the sequential algorithm using only one instance. The dataset consists of 1 binary class attribute and 6 binary attributes. We increase the dataset size using 25K to 200K trajectories with a step size of 25K training samples to learn each classifier.

Figure 3(a) illustrates the learning time compared to the dataset size. The figure shows the time taken by the algorithms and the regression lines which interpolate the data points. Intuitively, the increase of the size of the training samples leads to increased training time because the MapReduce implementation has a computational overhead, which with little data only led to bad performance. Figure 3(a) shows that the MapReduce algorithm performs better than the sequential algorithm also with the smallest dataset, but when the number of trajectories increases, the gap between the two algorithms starts growing.

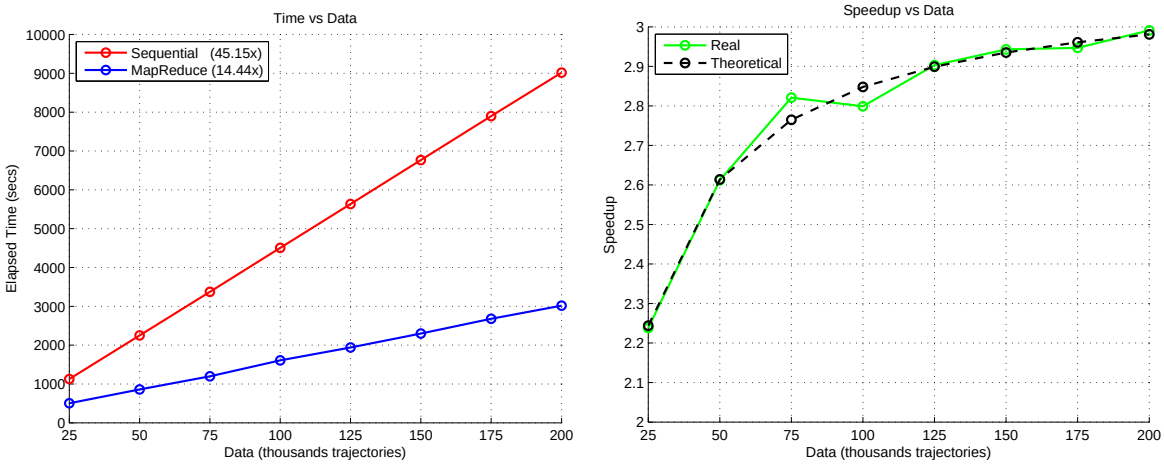
Figure 3(b) illustrates the speedup between the sequential and MapReduce algorithms. The points were calculated according to this equation: $S_p = T_s/T_{mr}$, where T_s is the sequential time and T_{mr} is the MapReduce time. As the data size increases, the speedup grows quickly at the beginning, while it become more stable when the data size is already big enough. For example, with 200K trajectories we have a speedup of about 3. In order to better understand this trend, the figure illustrates the speedup between the two regression lines (theoretical). This line shows very well how the speedup behaves in this case with a logarithmic trend.

5.2. Increasing the Hadoop nodes

In the second experiment, we varied the number of **Hadoop** nodes to assess the parallel performance of the MapReduce algorithm in the case of parameter learning. We used the same training set for the experiments using 200K trajectories.

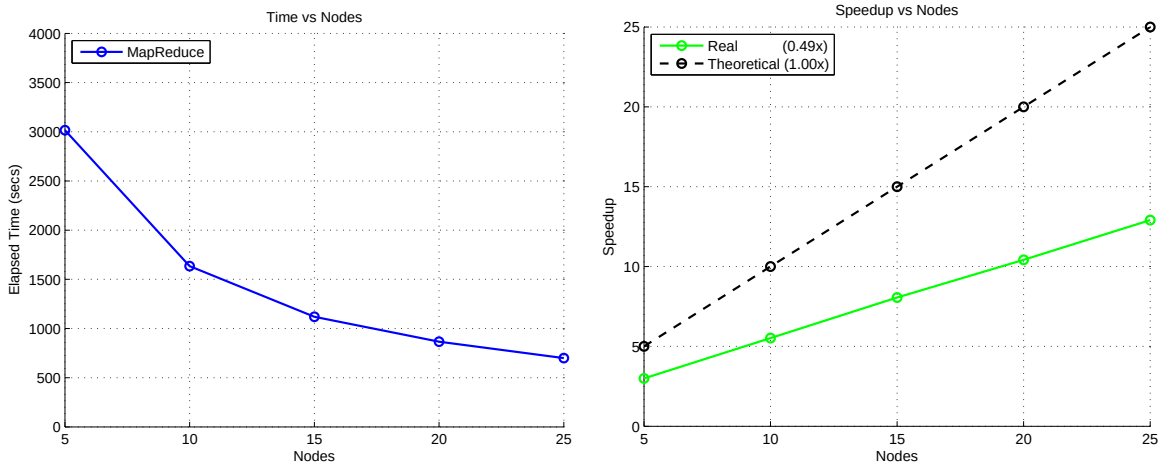
Figure 4(a) shows the changes in the training time using different numbers of **Hadoop** nodes from 5 to 25 with a step size of 5 nodes. As expected, increasing the number of **Hadoop** nodes significantly reduces the learning time, but this reduction is not equal to the ratio between the number of nodes.

Figure 4(b) reports the real speedup versus the theoretical one against the sequential algorithm. As illustrated, the trend is similar, but the speedup is almost half the value of the theoretical. This effect is due to the **Hadoop** overhead that is not present in the sequential version of the algorithm.



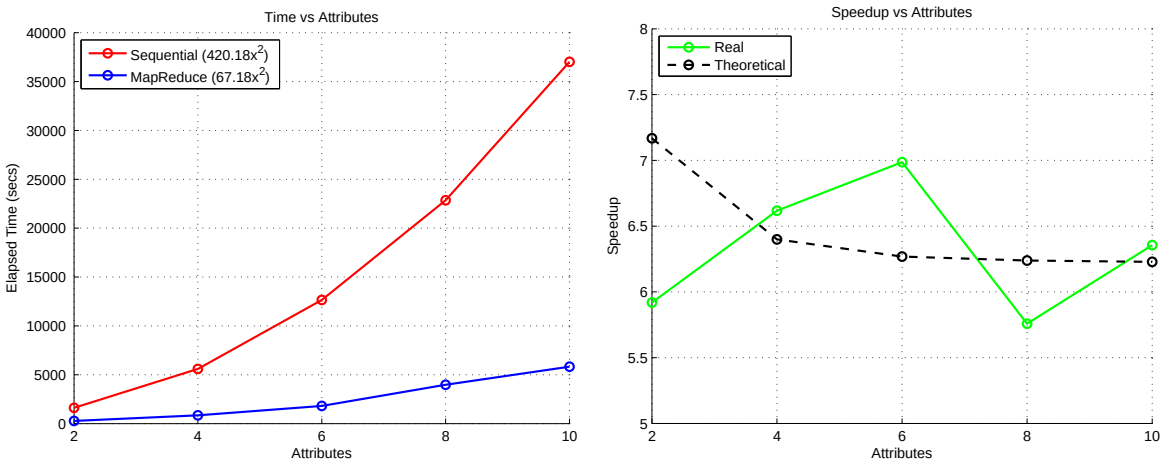
(a) (b)

Figure 3: Chart (a) shows the elapsed time for the sequential and MapReduce algorithms with 5 nodes with respect to the data size in the case of parameter learning of a continuous time naive Bayes classifier. It also show the regression lines which represents the trend of the elapsed time used by the two algorithms. Chart (b) illustrates the real speedup between the sequential and MapReduce algorithms versus its theoretical value. In this case the speedup behaves with a logarithmic trend.



(a) (b)

Figure 4: Chart (a) shows the elapsed time for the MapReduce algorithm on 200K trajectories with respect to the number of nodes in the case of parameter learning of a continuous time naive Bayes classifier. Chart (b) illustrates the real versus the theoretical speedup between the MapReduce and the sequential algorithm. In this case the speedup is almost half the theoretical value.



(a) (b)

Figure 5: Chart (a) shows the elapsed time for the MapReduce algorithm with 25 nodes and the sequential algorithm on 100K trajectories with respect to the number of attributes in the case of structural learning of a continuous time tree augmented naive Bayes classifier. Chart (b) illustrates the real versus the theoretical speedup. In this case the real speedup against the second order polynomial regression is not perfect when the number of attributes is less then 10.

5.3. Increasing the number of attributes

In the third experiment, we measured the performance of the MapReduce algorithm in the case of structural learning of a continuous time tree augmented naive Bayes classifier varying the number of attributes.

In this last experiment, we used 1 Master instance and 25 Core instances against the sequential algorithm using only one instance. We used the dataset with 100K trajectories varying the number of attributes from 2 to 10 with a step size of 2.

Figure 5(a) illustrates the learning time compared to the number of attributes. This figure shows the elapsed time for the computation of the MapReduce algorithm versus the sequential one. In both cases, the trend is clearly quadratic because we are testing the structural learning part of the algorithms. In this configuration, every possible parents combination given a variable is analyzed, for this reason we have two inner loops ranging over the number of variables (as described in Algorithm 1 for the map function). The quadratic coefficient of the polynomial regression for the MapReduce algorithm is 67.18 against 420.18 of the sequential version.

Figure 5(b) shows the speedup between the sequential algorithm and MapReduce algorithm. We can see that the real speedup against the second order polynomial regression is not perfect when the number of attributes is less then 10. Indeed, with only 2 attributes the MapReduce algorithm is slightly penalized because the real speedup is only 5.92 against 7.17 of its theoretical value, while with 10 attributes the speedup is stabilized around its theoretical value of 6.25.

6. Conclusion

In recent years, we have seen the explosion in the quantity of available data as result of the recent advancements in data recording and storage technology; this phenomenon is commonly defined as big data (Diebold 2003). In this context, the amount of data that needs to be elaborated, which also needs to be processed efficiently, is massive. The MapReduce framework has proved to be a very good way of parallelizing machine learning applications due to its characteristics of simplicity and fault tolerance (Lin and Dyer 2010).

In this paper, we have designed and implemented a MapReduce algorithm for the learning task of continuous time Bayesian network classifiers in the case of complete data. Different experiments have shown that our design scales well in the distributed processing environment. For example, as shown in Figure 3(b), using a dataset of 200K trajectories with six binary attributes and 5 **Hadoop** nodes, it is possible to reach a speedup of about 3 compared to the sequential version for the parameter learning task. As shown in Figure 5(b), using a dataset of 100K trajectories with ten binary attributes and 25 **Hadoop** nodes, it is possible to reach a speedup of about 6.25 compared to the sequential version for the structural learning task. These performances can be improved if the algorithms are executed on big data using many nodes. The advantage of MapReduce depends not only on the size of the input data, but also on the structure of the graph and on the number of states of the variables. In fact, it is not necessary to maintain the counting map in memory, which can be critical in large networks with many states.

Future research will focus on extending the learning task of continuous time Bayesian network classifiers in the case of incomplete data using the MapReduce framework.

Acknowledgments

The authors acknowledge the many helpful suggestions of anonymous referees which helped to improve the paper clarity and quality.

References

Amazoncom, Inc (2014). "Amazon Web Services (AWS)." URL http://aws.amazon.com/.

Basak A, Brinster I, Ma X, Mengshoel OJ (2012). "Accelerating Bayesian Network Parameter Learning Using **Hadoop** and MapReduce." In *1st International Workshop on Big Data, Streams and Heterogeneous Source Mining (BigMine'12)*, pp. 101–108.

Boudali H, Dugan JB (2006). "A Continuous-Time Bayesian Network Reliability Modeling, and Analysis Framework." *IEEE Transactions on Reliability*, **55**(1), 86–97.

Chu CT, Kim SK, Lin YA, Yu Y, Bradski GR, Ng AY, Olukotun K (2006). "Map-Reduce for Machine Learning on Multicore." In *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pp. 281–288.

Csardi G, Nepusz T (2006). "The **igraph** Software Package for Complex Network Research." *InterJournal*, **Complex Systems**, 1695. URL http://igraph.org/.

Dean J, Ghemawat S (2004). "MapReduce: Simplified Data Processing on Large Clusters." In *6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI04)*.

Dean J, Ghemawat S (2010). "MapReduce: A Flexible Data Processing Tool." *Communications of the ACM*, **53**(1), 72–77.

Dean T, Kanazawa K (1989). "A Model for Reasoning about Persistence and Causation." *Computational Intelligence*, **5**(3), 142–150.

Diebold FX (2003). "Big Data Dynamic Factor Models for Macroeconomic Measurement and Forecasting." In M Dewatripont, LP Hansen, S Turnovsky (eds.), *Advances in Economics and Econometrics: Theory and Applications. Eighth World Congress of the Econometric Society*, pp. 115–122. Cambridge University Press.

El-Hay T, Friedman N, Kupferman R (2008). "Gibbs Sampling in Factorized Continuous-Time Markov Processes." In *24nd Conference on Uncertainty in AI (UAI)*, pp. 169–178.

Fan Y, Shelton CR (2009). "Learning Continuous-Time Social Network Dynamics." In *25th Conference on Uncertainty in AI (UAI)*, pp. 161–168.

Fan Y, Xu J, Shelton CR (2010). "Importance Sampling for Continuous Time Bayesian Networks." *Journal of Machine Learning Research*, **11**(August), 2115–2140.

Friedman N, Geiger D, Goldszmidt M (1997). "Bayesian Network Classifiers." *Machine Learning*, **29**(2–3), 131–163.

Friedman N, Koller D (2000). "Being Bayesian about Bayesian Network Structure: A Bayesian Approach to Structure Discovery in Bayesian Networks." *Machine Learning*, **50**(1–2), 95–125.

Gatti E, Luciani D, Stella F (2012). "A Continuous Time Bayesian Network Model for Cardiogenic Heart Failure." *Flexible Services and Manufacturing Journal*, **24**(4), 496–515.

Heckerman D (1999). "A Tutorial on Learning with Bayesian Networks." In *Learning in Graphical Models*, pp. 301–354. MIT Press.

Heckerman D, Geiger D, Chickering DM (1995). "Learning Bayesian Networks: The Combination of Knowledge and Statistical Data." *Machine Learning*, **20**(3), 197–243.

Jensen FV, Nielsen TD (2007). *Bayesian Networks and Decision Graphs*. Springer-Verlag.

Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.

Lin J (2008). "Scalable Language Processing Algorithms for the Masses: A Case Study in Computing Word Co-Occurrence Matrices with MapReduce." In *2008 Conference on Empirical Methods in Natural Language Processing (EMNLP 2008)*, pp. 419–428.

Lin J, Dyer C (2010). *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers.

Nodelman U (2007). *Continuous Time Bayesian Networks*. Ph.D. thesis, Stanford University.

Nodelman U, Horvitz E (2003). "Continuous Time Bayesian Networks for Inferring Users' Presence and Activities with Extensions for Modeling and Evaluation." *Technical Report MSR-TR-2003-97*, Microsoft Research.

Nodelman U, Koller D, Shelton CR (2005a). "Expectation Propagation for Continuous Time Bayesian Networks." In *21st Conference on Uncertainty in AI (UAI)*, pp. 431–440.

Nodelman U, Shelton CR, Koller D (2002). "Continuous Time Bayesian Networks." In *18th Conference on Uncertainty in AI (UAI)*, pp. 378–387.

Nodelman U, Shelton CR, Koller D (2003). "Learning Continuous Time Bayesian Networks." In *19th Conference on Uncertainty in AI (UAI)*, pp. 451–458.

Nodelman U, Shelton CR, Koller D (2005b). "Expectation Maximization and Complex Duration Distributions for Continuous Time Bayesian Networks." In *21st Conference on Uncertainty in AI (UAI)*, pp. 421–430.

Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell T (2007). "A Survey of General-Purpose Computation on Graphics Hardware." *Computer Graphics Forum*, **26**(1), 80–113.

Pavlovic V, Frey BJ, Huang TS (1999). "Time-Series Classification Using Mixed-State Dynamic Bayesian Networks." In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'99)*, volume 2, pp. 609–615.

Pearl J (1985). "Bayesian Networks: A Model of Self-Activated Memory for Evidential Reasoning." In *Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, CA*, pp. 329–334.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Saria S, Nodelman U, Koller D (2007). "Reasoning at the Right Time Granularity." In *23rd Conference on Uncertainty in AI (UAI)*.

Shelton CR, Fan Y, Lam W, Lee J, Xu J (2010). "Continuous Time Bayesian Network Reasoning and Learning Engine." *Journal of Machine Learning Research*, **11**(March), 1137–1140.

Stella F, Amer Y (2012). "Continuous Time Bayesian Network Classifiers." *Journal of Biomedical Informatics*, **45**(6), 1108–1119.

The Apache Software Foundation (2014). "Apache **Hadoop**." URL http://hadoop.apache.org/.

Villa S, Stella F (2014). "Continuous Time Bayesian Network Classifiers for Intraday FX Prediction." *Quantitative Finance*, **14**(12), 2079–2092.

White T (2009). ***Hadoop**: The Definitive Guide*. O'Reilly, Sebastopol.

Xu J, Shelton CR (2008). "Continuous Time Bayesian Networks for Host Level Network Intrusion Detection." In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2008, Antwerp, Belgium, September 15–19, 2008, Proceedings, Part II*, pp. 613–627.

Affiliation:

Simone Villa, Marco Rossetti
Department of Informatics, Systems and Communication
Faculty of Mathematics, Physics and Natural Sciences
University of Milano-Bicocca
20126 Milano, Italy
E-mail: simone.villa@disco.unimib.it, marco.rossetti@disco.unimib.it
URL: http://www.mad.disco.unimib.it/doku.php/people/people

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 62, Issue 3 *Submitted:* 2013-03-01
November 2014 *Accepted:* 2014-06-26