Journal of Statistical Software

December 2025, Volume 115, Issue 6. *doi: 10.18637/jss.v115.i06*

dbnR: Gaussian Dynamic Bayesian Network Learning and Inference in R

David Quesada
Universidad Politécnica de Madrid

Pedro Larrañaga
Universidad Politécnica de Madrid

Concha Bielza
Universidad Politécnica de Madrid

Abstract

Dynamic Bayesian networks are a type of multivariate time series forecasting model capable of a level of interpretability thanks to their graphical representation. They have been reported extensively in the literature in a variety of areas, but their application has usually involved an ad hoc implementation or adaptation of existing Bayesian network software to a dynamic case. In this paper, we present **dbnR**, an R package that encapsulates the whole process of learning the model and parameters from data and performing inference. The package provides three different structure learning algorithms, exact and approximate inference and a visualization tool that allows inspection of the graphical structure of the networks. The aim of **dbnR** is to provide a tool that enables fast deployment of dynamic Bayesian network models and to make them readily available as general purpose forecasting models.

Keywords: dynamic Bayesian networks, multivariate time series, structure learning, forecasting, R.

1. Introduction

In recent years, the use of dynamic Bayesian networks (DBNs) has gained popularity in several fields. Their applications range from more scientific environments such as bioinformatics (Wang, Berceli, Garbey, and Wu 2019) and neuroscience (Bielza and Larrañaga 2014) to more industrial settings such as traffic management (Chaudhary, Indu, and Chaudhury 2017), fatigue assessment of construction (Zhu, Zhang, and Li 2019) or estimation of the remaining useful life of structures (Cai *et al.* 2019). These applications benefit from a white-box, interpretable model capable of performing multivariate forecasting and inference.

However, none of these works present the option to apply their DBN code to other problems or show the use of some public DBN library. Most of the time, authors resort to implementing the

models themselves. Other times, researchers opt for adapting existing static Bayesian network packages, such as **bnlearn** (Scutari 2010) in R or **pgmpy** (Ankan and Panda 2015) in Python. Both of these options can be very time-consuming and result in ad hoc implementations that, for the most part, are not extendable to new applications.

It is uncommon to find software packages specifically designed for DBNs. In R, we can find **dbnlearn** (Fernandes 2020), a package that allows creating univariate DBNs and making predictions of the next instant with them. These univariate DBNs only have a single variable repeated in several instances of time and always have the same fixed structure, where nodes are connected to the node in the next instant and to the objective variable. Underneath, the package uses the parameter learning and inference offered by **bnlearn**. In Python, there is the **pyAgrum** package (Ducamp, Gonzales, and Wuillemin 2020), which is a wrapper of the C++ library **aGruM**. It offers learning, inference and visualization of Bayesian networks (BNs) and supports DBNs, but it only allows the use of discrete variables and discretizes continuous ones. If we opted for adapting existing BN software for a dynamic scenario, there are several possibilities. The most versatile one is the use of **bnlearn** with certain restrictions and pre-processing steps for learning DBN structures with BN methods. Afterwards, one can use either **gRain** (Højsgaard 2012) for inference with discrete variables or **rbmn** (Denis and Scutari 2021) for inference with continuous ones. In Python, one can use either **pgmpy** or **bnlearn** (Taskesen 2020), which has the same name but a different author from the original R package by Scutari, to potentially fit a DBN model. However, both packages only support discrete variables, and only **pgmpy** has the skeleton of the classes for a potential extension to DBNs. In the case of discrete DBNs, these options could offer a solution after the user adapts them, but neither offers an option for continuous variables. More recently, the **BiDAG** (Suter, Kuipers, Moffa, and Beerenwinkel 2023) package in R and the **PyBNesian** (Atienza, Bielza, and Larrañaga 2022) package in Python have been released. These two packages offer new alternatives in the form of DBN structure and parameter learning. In the case of **BiDAG**, it presents a framework for learning DBN structures with Markov chain Monte Carlo methods for both discrete and continuous variables. This offers a unique alternative for learning DBN structures from data, and these methods scale well to bigger networks of over 100 nodes. However, **BiDAG** has no inference motor of its own apart from sampling methods. On the other hand, **PyBNesian** offers a very complete package that allows dealing with discrete and continuous variables, and even allows creating hybrid DBNs. It also has the unique feature of allowing the use of kernel density estimation for the nodes inside the DBNs, which constitutes an alternative to the usual discrete and Gaussian distributions. However, it does not have an inference motor either and to make predictions the user would need to use a sampling procedure. To summarize the listed available software, we have compiled all the cited packages in Table 1, where we point out all their capabilities and features.

The process of training a DBN model from data and forecasting has several intermediate steps (Koller and Friedman 2009): adapting the dataset for time series (TS) learning, applying a structure learning algorithm, visualising the network, using an exact or approximate inference method and running a forecasting motor. With **dbnR**, our objective is to create a simple pipeline where upon providing some data, we can obtain a model that we can visualize and use to perform inference. All the intermediate steps are encapsulated and parametrized inside the package to allow both a simple deployment and the possibility of tuning the learning and inference process to the user's needs. Compared with all the listed software packages, **dbnR** offers its users a package specifically designed to work with TS of continuous variables that

Package	BNs	DBNs	HO	Disc.	Cont.	Hybrid	Exact	Approx.	Visual
R:									
dbnR	–	✓	✓	–	✓	–	✓	✓	✓
bnlearn	✓	–	–	✓	✓	✓	–	✓	✓
dbnlearn	–	✓	–	–	✓	–	–	✓	✓
BiDAG	✓	✓	–	✓	✓	–	–	✓*	✓
Python:									
PyBNesian	✓	✓	✓	✓	✓	✓	–	✓*	–
pgmpy	✓	✓	–	✓	–	–	✓	✓	✓
bnlearn	✓	–	–	✓	–	–	–	✓	✓
pyAgrum	✓	✓	–	✓	–	–	✓	✓	✓

*: A sampling method for BNs is provided, but no proper approximate inference motor defined.

Table 1: Overview of various BN and DBN packages in R and Python. In order, each column means package name, support for BNs, DBNs, high Markovian order, discrete variables, continuous variables, hybrid networks, exact inference, approximate inference and visualization tools. Names where shortened to fit the table inside the page length.

allows high order DBN learning and exact inference readily applicable in real world datasets. Its forecasting functions allow performing predictions of future values and plotting the results in a comprehensive manner, and the DBN visualization tool provides an interactive graph html widget that helps with interpreting network structures.

The stable **dbnR** (Quesada 2026) source code and binaries can be found in the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=dbnR`, while active development is underway in GitHub at `https://github.com/dkesada/dbnR`.

The rest of the paper is organized as follows. Section 2 covers some background of the DBN model. Section 3 discusses the structure learning algorithms available in **dbnR** and the visualization tool. Section 4 presents the inference and forecasting methods. Section 5 describes the main functions and classes of **dbnR** and its compatibility with the **bnlearn** package. Section 6 showcases a complete example of applying DBNs for forecasting some data. Finally, Section 7 offers some final remarks and conclusions.

2. Bayesian network background

In this section, we briefly introduce the concept of BNs, the different types of networks depending on the data, and the extension to the dynamic scenario where time is taken into consideration. For a more in-depth discussion of these models, we refer the readers to Koller and Friedman (2009) for a general use text that covers most of the theory on BNs and to Murphy (2002) for specific information regarding DBN models.

2.1. Joint probability distribution

Bayesian networks (Pearl 1988; Koller and Friedman 2009) are probabilistic graphical models that represent conditional independence relationships between variables using a directed acyclic graph, where each node in the graph represents a random variable of our domain. We say that an arc between two nodes X_i and X_j inside a graph is directed when it has either

the direction $X_i \to X_j$ or $X_i \leftarrow X_j$, as opposed to no direction at all $X_i - X_j$. When we consider a path that traverses from one node X_i to some other node X_j, possibly passing through other nodes in between, such path can only traverse arcs in the direction that they point towards. In a directed graph, all arcs are directed and no undirected arc is present. A cycle appears in a graph when we can find a path $X_i \to \ldots \to X_j$ where $X_i = X_j$, that is, a path where we end up in the same node we started from. Inside an acyclic graph, no such cycles can be found for any variable in the graph. In the case of BNs, their constraints require graphs that are both directed and acyclic. In a BN, the model represents a joint probability distribution $p(\mathbf{X})$ of all the variables factorized as:

$$p(\mathbf{X}) = \prod_{i=1}^{n} p(X_i|\mathbf{Pa}_i), \tag{1}$$

where $\mathbf{X} = \{X_1, \ldots, X_n\}$ is the set of all the nodes in the network and $\mathbf{Pa}_i = \{X_{1(i)}, \ldots, X_{k(i)}\}$ is the set of all the parent nodes of X_i in the graph. We say that a node X_i is the parent of another node X_j when there is a directed arc $X_i \to X_j$ that connects them. This type of relationship means that the values of X_i will directly affect the values that X_j will take, creating a probabilistic dependence relationship between them. In cases where triplets of variables are present, as in a structure like $X_i \to X_j \to X_k$, we say that X_i and X_k are conditionally independent given X_j because knowing the value of X_j would make X_i and X_k independent from one another. These conditional independences can also be present among other ancestors and their descendants, making the graph structure a reflection of the conditional dependence relationships between all variables in our BNs. This in turn is also what makes BNs interpretable models, because we know for a fact the effect that each node has on its descendants, and we can know exactly how likely some variable will take a specific value given its parents. Depending on the type of variables we are handling, each node in the network will have a different kind of conditional probability distribution (CPD).

There are three popular types of BN models depending on the type of data we have: discrete BNs for categorical data, Gaussian BNs for continuous data and conditional linear Gaussian BNs for mixed data. In the literature, the most well-known package that offers support for all the aforementioned types of BNs is the R package **bnlearn**. It allows training and inference with all of them, but it does not include DBNs. In our package, we opted to focus on Gaussian DBNs for two reasons. First, real-world data recovered from sensors are usually real valued. Second, exact inference with Gaussian BNs is much faster to compute than with their discrete counterparts due to the CPD of their nodes. In Gaussian BNs, we assume that all the variables in our system follow a normal distribution represented as a linear Gaussian model:

$$p(x_i|\mathbf{Pa}_i) = \mathcal{N}(\beta_{0i} + \beta_{1i}x_{1(i)} + \cdots + \beta_{ki}x_{k(i)}; \sigma_i^2) = \mathcal{N}(\mu_i^{Pa_i}; \sigma_i^2), \tag{2}$$

where $\beta_{0i}, \ldots, \beta_{ki}$ are regression parameters associated with each parent node of X_i in $\mathbf{Pa}_i$ and σ_i^2 is the (unconditional) variance of X_i. Given that all the nodes in the network have this kind of CPD, we can write the joint density in Equation 1 using Equation 2 as:

$$p(\mathbf{X}) = \prod_{i=1}^{n} p(x_i|\mathbf{Pa}_i) = \prod_{i=1}^{n} \mathcal{N}(\mu_i^{Pa_i}; \sigma_i^2). \tag{3}$$

In Equation 3, we can see that the joint density has the same form as a multivariate Gaussian

distribution:

$$\mathcal{N}(\boldsymbol{\mu}; \boldsymbol{\Sigma}) = \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \vdots \\ \mu_n \end{bmatrix}, \quad \begin{bmatrix} \sigma_1^2 & \cdots & \sigma_{1n} \\ \vdots & \ddots & \vdots \\ \sigma_{n1} & \cdots & \sigma_n^2 \end{bmatrix} \right), \tag{4}$$

where σ_{ij} is the covariance between variables X_i and X_j. This transformation into the multivariate Gaussian form is why performing exact inference in these models is faster than in the discrete case.

Given the popularity of **bnlearn**, we opted to extend some of the functionality of this package to the case of Gaussian DBNs. As a result, all **dbnR** networks extend the S3 classes '`bn`' for the graph structure and '`bn.fit`' for the fitted networks offered by **bnlearn**. The new resulting '`dbn`' and '`dbn.fit`' classes enable all the graph operations and the score functions coded in **bnlearn** to work with the DBN models in **dbnR**. The network structure and parameter learning in **bnlearn** requires a dataset as a '`data.frame`' to be provided to these functions in order to learn a BN from data. This '`data.frame`' stores the information about the relationships between variables in different instances of recorded values for each variable. To allow compatibility with the use of '`data.frame`' and improve efficiency in terms of making queries, operating with the data and passing it down between functions, we switched the use of '`data.frame`' to '`data.table`' (Dowle and Srinivasan 2021). Additionally, to allow fast exact inference, we switched to C++ using **rcpp** (Eddelbuettel and François 2011) to calculate the mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ seen in Equation 4, and we store them as attributes of our S3 object so that they can be used in place of the graph structure when forecasting with the network. We use C++ several times in our package, especially when we need to perform heavy computations or matrix operations inside structure learning algorithms.

2.2. Dynamic Bayesian networks

When we are dealing with time series (TS) data, we need to extend the BN model to take time into consideration. To do this, we discretize time into time slices to define DBNs. Each time slice can have a local BN structure with intra-slice arcs between its nodes, and it can be connected to previous and posterior time slices with inter-slice arcs, which must always go from older time slices to more recent ones. The inter-slice arcs represent the effect that the past state of the variables has on the present. An example of the graph structure of a DBN can be seen in Figure 1. In this context, the dynamic part of the model refers only to the ability of DBNs to model the time component, and the structure of the inter-slice arcs remains unchanged through time. Models like time-varying dynamic Bayesian networks (Song, Kolar, and Xing 2009), which allow this kind of behaviour, are outside the scope of **dbnR**.

This new DBN structure can be represented in the '`data.frame`' table that **bnlearn** uses by simply adding the new nodes and arcs in and between each time slice. However, restrictions have to be implemented to new learning algorithms to avoid introducing invalid arcs backwards in time.

The new joint probability distribution of the network has to consider all the $T+1$ time slices and the effect they have on the more recent ones:

$$p(\mathbf{X}^0, \ldots, \mathbf{X}^T) = p(\mathbf{X}^{0:T}) = p(\mathbf{X}^0) \prod_{t=0}^{T-1} p(\mathbf{X}^{t+1} | \mathbf{X}^{0:t}), \tag{5}$$

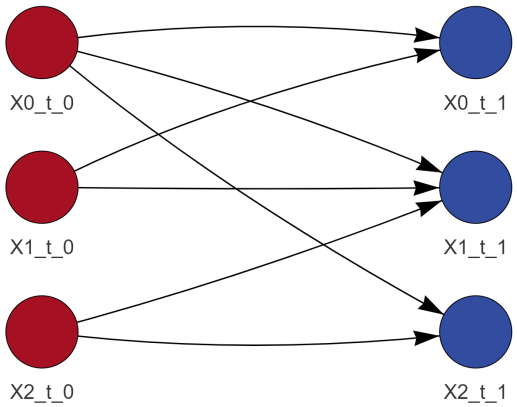

Figure 1: Example of a DBN network with two time slices in red and blue, and three nodes per time slice. There are several inter-slice arcs and no intra-slice arcs. The network was plotted using the visualising tool included in the **dbnR** package.

where $\mathbf{X}^t = \{X_1^t, \ldots, X_n^t\}$ is the set of nodes in time slice t and T is a time horizon. This representation becomes increasingly unfeasible the more time slices we take into consideration. To fix this issue, it is very common to introduce the Markovian order assumption, where we assume the present to be independent of the past after a certain number of time slices. The most common order in the literature is one, and it transforms the joint probability distribution in Equation 5 into:

$$p(\mathbf{X}^{0:T}) = p(\mathbf{X}^0) \prod_{t=0}^{T-1} p(\mathbf{X}^{t+1}|\mathbf{X}^t), \tag{6}$$

where only the last instant is used to calculate the probability of the variables in the next instant. The Markovian order of the network should be chosen based on the autoregressive order of our TS data, so setting this order to one is not always the best option. In **dbnR**, the order of the network can be freely chosen by the user with a parameter when learning the network structure from the data.

After extending the '`bn`' class from **bnlearn** to the DBN scenario, we opted for taking a different route in terms of the structure learning algorithms and the inference motor inside **dbnR**.

3. Structure learning

To learn the effect that past values of the variables have on the present, the first step we need to take is to adapt our datasets to the time discretization of DBNs. In TS data, instances are arranged in chronological order, where the oldest instance is usually the first row. Depending on the desired Markovian order, we need to shift the rows in our dataset to ensure that the values of several rows are grouped into a single row. To illustrate this process, we show an example in Figure 2. Unlike in Equation 5, we set $t = 0$ as the most recent time slice instead of the oldest. This is merely a convention change motivated by an easier implementation of the package architecture. Given that we allow an arbitrary Markovian order, it is more convenient to have the most recent time slice always named t_0 during inference regardless of

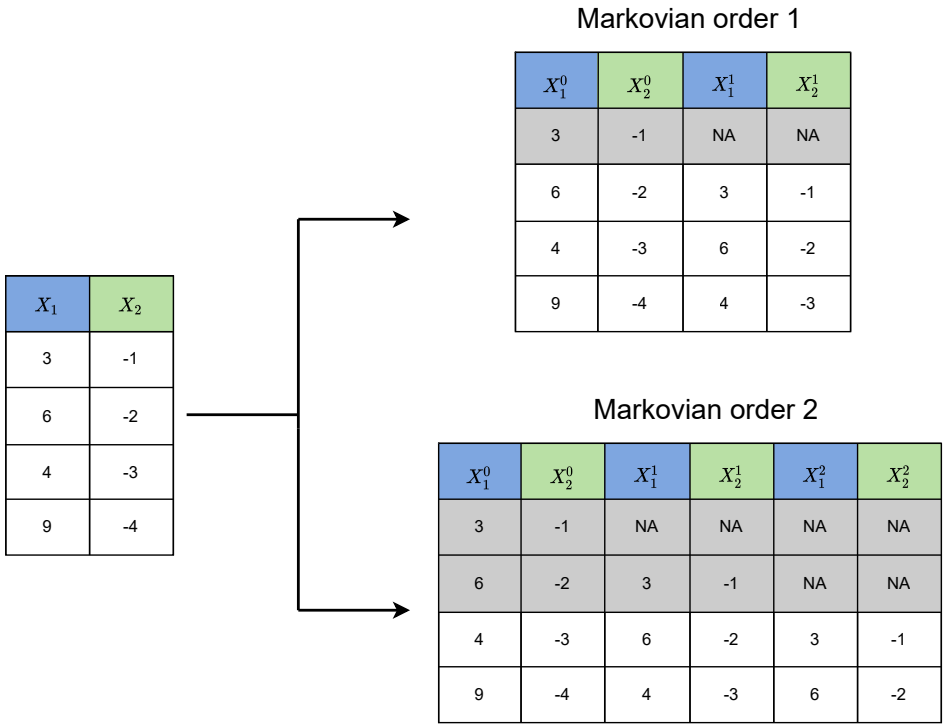

Figure 2: Example of transforming a dataset with two variables X_1 and X_2 into several variables depending on the desired Markovian order. The rows in the original data are ordered from the oldest recorded values, $X_1 = 3$ and $X_2 = -1$, to the newest. The grey rows contain missing values and should be deleted.

the order. On a side note, please keep in mind that **dbnR** does not perform training, test and validation splits of any kind, and so this falls under the responsibility of the user. If a dataset is provided to a training function, the whole dataset will be used for training, and should have been partitioned beforehand by the user. Similarly, forecasting functions will predict the values in the datasets provided to them without doing any splitting. This behaviour is reflected in the examples shown in this work.

In **dbnR**, the function `fold_dt` performs this dataset modification automatically. To begin the learning process, we call this function to adapt our data to the desired format. The example in Figure 2 can be replicated with the following code:

```
R> df <- data.frame(X1 = c(3, 6, 4, 9), X2 = c(-1, -2, -3, -4))
R> df

  X1 X2
1  3 -1
2  6 -2
3  4 -3
4  9 -4

R> fold_dt(df, size = 2)
```

```
   X1_t_0 X2_t_0 X1_t_1 X2_t_1
1:      6     -2      3     -1
2:      4     -3      6     -2
3:      9     -4      4     -3
```

```
R> fold_dt(df, size = 3)
```

```
   X1_t_0 X2_t_0 X1_t_1 X2_t_1 X1_t_2 X2_t_2
1:      4     -3      6     -2      3     -1
2:      9     -4      4     -3      6     -2
```

Note that due to restrictions in variable names, the time slice that each variable corresponds to is written as `t_x`. The `size` argument determines the total number of time slices in the network, that is, the Markovian order plus one. Additionally, if one has a dataset with several independent time series repetitions, there is a specific `filtered_fold_dt()` function that allows us to create a folded dataset for learning a DBN without mixing instances from different time series in the same row. We can see an example of this in the following code:

```
R> df <- data.frame(X1 = c(3, 6, 4, 9, 8, 2), X2 = c(-1, -2, -3, -4, -5, -6),
+    idx = c(1, 1, 1, 2, 2, 2))
R> df
```

```
  X1 X2 idx
1  3 -1   1
2  6 -2   1
3  4 -3   1
4  9 -4   2
5  8 -5   2
6  2 -6   2
```

```
R> fold_dt(subset(df, select=-idx), size = 2)
```

```
   X1_t_0 X2_t_0 X1_t_1 X2_t_1
1:      6     -2      3     -1
2:      4     -3      6     -2
3:      9     -4      4     -3
4:      8     -5      9     -4
5:      2     -6      8     -5
```

```
R> filtered_fold_dt(df, size = 2, id_var = 'idx')[]
```

```
   X1_t_0 X2_t_0 X1_t_1 X2_t_1
1:      6     -2      3     -1
2:      4     -3      6     -2
3:      8     -5      9     -4
4:      2     -6      8     -5
```

This example uses a similar dataset to the previous folding example, with the `X1` and `X2` variables, and an additional column `idx` that identifies two time series marked with 1 and 2 respectively. If we use the regular `fold_dt()` function, we can see that the third row contains values from the first and second time series, which is wrong: values from independent time series should not be mixed together. This is done properly in the following call to `filtered_fold_dt()` where we specify that the index variable `idx` should be used to fold the dataset taking care not to mix different time series together. The temporal window approach of the **dbnR** package allows us to use different instances of time series generated from the same process, which can also be of different lengths, to train our DBN models.

In the most recent **dbnR** version (version 0.8.0), three structure learning algorithms are available: a version of Trabelsi, Leray, Ayed, and Alimi (2013) dynamic max-min hill-climbing (DMMHC), Santos and Maciel (2014) binary particle swarm optimization (PSO) algorithm and Quesada, Bielza, and Larrañaga (2021) natural number order invariant encoding PSO algorithm. There is only a single function that handles the calls to all the different algorithms:

```
learn_dbn_struc(dt, size, method, f_dt, ...)
```

The `learn_dbn_struc` function requires the training dataset and a desired size, and depending on the `method` argument ("dmmhc", "psoho" or "natPsoho", respectively, for the three aforementioned methods), it will call the appropriate non-exported function of the specific structure learning algorithm inside the package. The `f_dt` argument allows the user to pass down a dataset shifted manually or with the `fold_dt` function in case it is needed, and the ellipsis can be used to pass down further algorithm-specific arguments. This function will learn both the static structure and the transition network. The static structure is the BN structure of the first time slice, with only intra-slice arcs, and it represents the effect of the variables in the same time instant. The transition network is the structure that represents only the inter-slice arcs in the DBN and the effects that the past has on the present. Afterwards, it will return an S3 '`dbn`' object that extends the '`bn`' object from **bnlearn**. It is worth noting that while the DMMHC algorithm returns networks with both intra and inter-slice arcs, the particle swarm algorithms return networks with only inter-slice arcs. This is due to the nature of the codification of a DBN structure inside the particles used during the search. As such, these algorithms also enjoy faster execution times in part due to them moving in smaller search spaces by not allowing intra-slice arcs to appear in the resulting networks. This particular characteristic is not necessarily a reason for getting poorer forecasting accuracies with DBNs structures obtained from the PSO algorithms, but interpretation of these structures can only rely on the temporal relationships that variables have from past instants to the next ones, given that no interactions between variables can be found within the same instant of time.

3.1. Dynamic max-min hill-climbing

This was the first algorithm implemented in **dbnR** mainly as an extension of the offered methods in **bnlearn** for the case of DBNs. The basic max-min hill-climbing algorithm introduced by Tsamardinos, Brown, and Aliferis (2006) is a hybrid algorithm that combines a constrained local search of parent nodes to create an undirected skeleton graph and a score to orient the edges and obtain a directed acyclic graph. It begins by defining this initial skeleton graph through the use of conditional independence tests to find the possible parents of each node

in the network. Inside this initial skeleton graph, all edges remain undirected. Once a set of undirected edges has been found for every node in the graph, a hill-climbing search is run in order to select which edges present in the skeleton graph will be added to the network structure. This search procedure starts from an initial network structure, usually an empty graph, and it iterates by converting edges from the skeleton graph into arcs to add them to the network, deleting arcs or reversing arcs in the structure until no operation can be found that generates a graph improving the scoring function used for evaluating the network structure. This algorithm was extended to the dynamic case in Trabelsi *et al.* (2013) by building the DBN structure in two steps: learning the static structure and learning the transition network. The basic max-min hill-climbing algorithm is used to learn both of these structures separately.

In our case, we used the max-min hill-climbing implementation available in **bnlearn** to learn both of these networks, and then combined them into a single structure. To force the additional constraints on arcs imposed by DBNs, we make use of the `blacklist` argument that allows the user to introduce a matrix of forbidden arcs that will not appear in the final network. This `blacklist` argument will be used to ban inter-slice arcs backwards in time while learning the transition network. The construction of the `blacklist` matrix is performed automatically with regular expression operations based on the names of the variables. The user will not have to worry about this process, and additional arcs might be added to the `blacklist` if needed.

As an additional feature, we opted for using the `rsmax2()` function from **bnlearn** instead of directly calling the `mmmhc()` function for the graph learning. This decision was motivated due to the fact that the `rsmax2()` function encapsulates several other structure learning algorithms and allows the user the flexibility of selecting from all of them to learn the initial skeleton graph. It also allows the user to select a maximization procedure between a tabu search and the usual hill-climbing. As a result, a user is able to change the usual structure learning of the DMMHC algorithm for some other procedure if the need arises. This characteristic is defined with the use of the `restriction` and `maximize` arguments of the `rsmax2()` function, which in turn can be given to the `learn_dbn_struc()` function. If, for example, we wanted to use the stable version of the well-known PC algorithm in **bnlearn**, we would use the following call:

```
learn_dbn_struc(dt, size, method = "dmmhc", f_dt,
  restrict = "pc.stable", maximize = "hc")
```

For a list of all structure learning algorithms that can be used in the `restrict` parameter, one can check the strings that identify each one of them in the corresponding help page of the **bnlearn** package with `?bnlearn::‘structure-learning‘`.

Once both networks are built, we combine all their arcs to generate the final network structure that can be used in the `fit_dbn_params` function to fit its parameters. The original algorithm in Trabelsi *et al.* (2013) is defined for Markovian order 1 DBNs, and extending it to higher orders only implies learning a larger transition network with more than two time slices. It performs well for Markovian order 1 or 2 networks, but due to the super-exponential nature of the number of possible BN structures depending on the number of nodes (Robinson 1977), it scales poorly to higher orders.

3.2. Particle swarm optimization algorithms

The two PSO algorithms offered in **dbnR** are a non-deterministic alternative to the DMMHC method. They are better suited at learning the structure of high Markovian order networks, having far lower execution times as we increase the `size` parameter. Given that they are non-deterministic in nature, they will very likely return different network structures each time they are executed unless a seed is provided. They will both score DBN structures encoded as particles until they reach the maximum number of allowed iterations set by the user with the `n_it` parameter, by default set to 50 iterations. The best structure found is then returned as the solution of the structure learning. In contrast with the DMMHC algorithm, both of these algorithms only allow inter-slice arcs to appear in the final network structure in order to facilitate the encoding of individuals and to reduce the search space. As a result, the networks obtained from them will only show temporal relationships and will have no intra-slice arcs. This will not be an issue when performing forecasting and inference with the networks, but they will return less interpretable networks, as only the temporal effects can be seen.

Binary encoding particle swarm optimization

The first alternative to the DMMCH is the PSO algorithm presented by Santos and Maciel (2014). In this case, the problem of finding an optimal DBN structure is transformed into an optimization one, where the quality of each network is evaluated with a score. The graph structure is encoded into a list of lists, where the parents of each node are defined in a binary representation. An arc from one node to another is defined by a `1` in this list, and its absence is defined by a `0`. To drastically reduce the space of possible DBN structures, only inter-slice arcs from older time slices to the most recent one are allowed.

This algorithm has been implemented from scratch in **dbnR** using **R6** classes (Chang 2021) and follows an object-oriented programming paradigm. The '`Particle`' class in the framework contains a '`Position`', which encodes the binary list of lists, and a '`Velocity`', which contains arc additions or deletions with the same binary representation. The custom operations defined for these positions and velocities can be found in Santos and Maciel (2014) and are encapsulated inside the **R6** classes. These operations also switch to C++ when needed. The particles are evaluated by calculating the Bayesian information criterion (BIC) score (Schwarz 1978) or the Bayesian Gaussian equivalent (BGe) score (Geiger and Heckerman 1994) of the graph encoded in each particle. We defined a '`PsoCtrl`' class that initializes and contains all the particles and controls the execution of the search by evaluating the positions of the particles, obtaining the local and global optima, calculating new velocities and moving the particles. The best position found at the end of the process is transformed into its equivalent DBN form and is returned as the solution of the search.

Natural number invariant encoding particle swarm optimization

The implementation of this algorithm is in many ways similar to the binary PSO algorithm, as they share the same pipeline. The main differences between them are the encoding of the DBN structures and the operations between positions and velocities. In this case, the networks are encoded in vectors of natural numbers of constant length regardless of the Markovian order desired. Each particle consists only of a '`numeric`' vector, where each number corresponds to a single node of the network in t_0. This number encodes the information about which arcs from previous nodes point to that specific node. The binary bitwise representation of the nat-

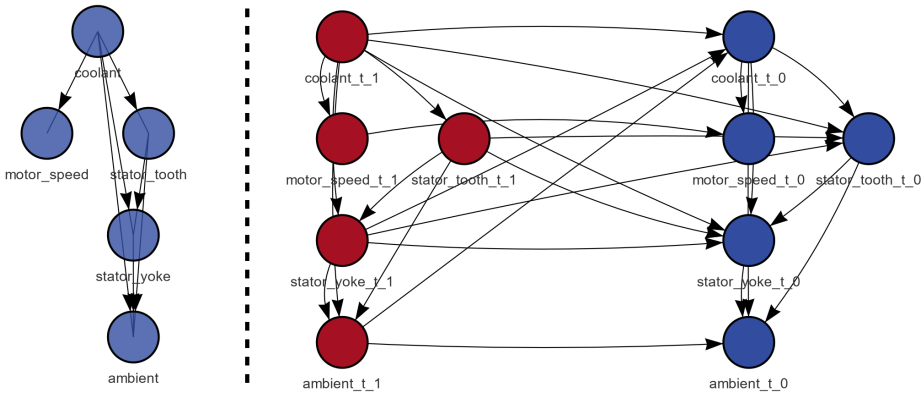

Figure 3: Example of the visualization of the structure of both a BN (left) and a DBN (right). The nodes on both networks can be clicked on to be highlighted and dragged to reposition them.

ural number indicates which arcs are present in the DBN and which arcs are not. With this encoding, a higher Markovian order only generates larger natural numbers, but it does not increase the size of the '`numeric`' vectors. The operations of additions and deletions of arcs are now performed bitwise with custom operators on the natural numbers representing the existing arcs, making this encoding scalable to high orders. Similar **R6** classes '`natParticle`', '`natPosition`' and '`natVelocity`' were generated for this algorithm to follow similar developing procedures. Thanks to this encoding and the custom operators, it has the fastest execution times for high Markovian order and should be considered when learning DBN structures in these scenarios. The details of the encoding and operators of this algorithm, as well as a comparison of execution times and percentage of recovered arcs of learning DBNs with the algorithms in **dbnR** can be found in Quesada *et al.* (2021).

3.3. Structure visualization tool

To offer the possibility of visualizing the graph structure of both the BNs learned with **bnlearn** and the DBNs learned with **dbnR**, we implemented a tool using the **visNetwork** package (Almende, Thieurmel, and Robert 2019). This tool is included in **dbnR**, but the **visNetwork** package is listed as suggested and will only be downloaded in case the user needs to plot some network. By using **visNetwork**, we plot the graph structures as HTML widgets that the user can interact with by highlighting arcs and nodes and by clicking on and dragging the nodes. The tool is intended only for visualization purposes, and any changes to the graph structure have to be made programmatically.

Plotting a BN or a DBN structure can be done with a single function call:

```
plot(structure, ...)
```

The `structure` argument can be a '`bn`', a '`bn.fit`', a '`dbn`' or a '`dbn.fit`' object obtained after learning a network structure with either **bnlearn** or **dbnR**. In the case of '`bn`' and '`bn.fit`' objects, if the user has loaded the **bnlearn** package then the generic `plot` function will call the visualization tool of **bnlearn**. In this case, if the user wants to use the visualization tool from

dbnR with **bnlearn** objects, then the `plot_static_network(structure)` function must be called instead. The ellipsis argument is used to provide three additional parameters for the DBN case: the `offset` argument, which modifies the size of the blank space between time slices in the plot, the `subset_nodes` argument, which allows the user to plot only a certain subgraph from the whole network by providing a vector with the names of the desired nodes, and the Boolean `reverse` argument, which allows the user to reverse the naming convention in **dbnR** plots to show the time slice t_0 as the oldest time slice instead of the most recent, as in Figure 1. An example of two network structures plotted with this tool can be seen in Figure 3.

4. Inference and forecasting

After learning the structure of the DBN, we need to fit its parameters to our data with the following function:

```
fit_dbn_params(net, f_dt, ...)
```

The `fit_db_params` function takes a '`dbn`' object and a dataset shifted with the `fold_dt` function to learn the parameters of the DBN and return a '`dbn.fit`' object. The function allows us to use the maximum likelihood estimation implemented in **bnlearn** to learn the parameters of the provided network from the shifted dataset. The $\boldsymbol{\mu}$ vector and the $\boldsymbol{\Sigma}$ matrix are estimated from the fitted DBN and added along with the `size` of the network as attributes of the '`dbn.fit`' object automatically in the `fit_dbn_params` function for future inference and forecasting. It is important to note that, while the concept of the $\boldsymbol{\mu}$ vector and the $\boldsymbol{\Sigma}$ matrix in DBNs is the same as that defined in Equation 4 for static Gaussian networks, in the case of DBNs the dimensions of this data structures are significantly larger. When we fold our datasets, we will find each variable replicated through time once per time slice. That means that our $\boldsymbol{\mu}$ vector and $\boldsymbol{\Sigma}$ matrix will depend on which `size` parameter we fix when folding our dataset and learning our structure, and so the length of $\boldsymbol{\mu}$ and the number of rows and columns of $\boldsymbol{\Sigma}$ will be `size` times the number of variables in our original dataset.

Once we have the structure and the parameters of a DBN, we can use the model to perform inference over some data. We call inference to the process of, given some evidence of the values of some nodes, calculating the most likely values for the rest of the variables. In **dbnR**, we offer an approximate inference method and an exact one. The approximate inference is performed via the likelihood weighting (Korb and Nicholson 2010) based on Monte Carlo sampling offered in **bnlearn**. This method performs several forward sampling runs, starting from the root nodes with no parents and ending with the leaf nodes with no children in a topological order, where the values of each node are either fixed as evidence or sampled randomly weighted by the probability of each value occurring using the sampled values of their parent nodes as evidence. When all nodes are sampled we finish a run, and when we finish several runs we can average the expected values for each node from all the results. The exact inference algorithm has been implemented in **dbnR** specifically for Gaussian BNs by using the equivalent multivariate joint distribution shown in Equation 4. In the case of Gaussian networks, exact inference is preferable to approximate inference because the execution time is greatly reduced by using the equivalent joint Gaussian distribution and it is usually faster than running several simulations to estimate the averaged results. As such, we will only focus on exact inference in this work.

4.1. Exact inference

One way to perform exact inference in a Gaussian BN is to use the equivalent multivariate joint distribution of the network. To do this, we can use the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$ that we calculated when learning the network structure. When we perform inference, the values of some of the variables are known beforehand and used to predict the most likely values for the rest of the variables in the system. Note that in the degenerated case where no evidence whatsoever is provided, the values predicted for the variables are the marginal means in $\boldsymbol{\mu}$. In the dynamic scenario, usually the variables in the past are observed and will be used to perform inference over the variables in the present.

In the inference process, we start by dividing the parameters into those corresponding to the set of target variables $\mathbf{X}_1$ and to the set of observed variables $\mathbf{X}_2$:

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \ \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}, \tag{7}$$

where $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are the mean vectors of the variables in $\mathbf{X}_1$ and $\mathbf{X}_2$, respectively; $\boldsymbol{\Sigma}_{11}$ is the covariance submatrix of $\mathbf{X}_1$; $\boldsymbol{\Sigma}_{12}$ and $\boldsymbol{\Sigma}_{21}$ are the covariance matrices between $\mathbf{X}_1$ and $\mathbf{X}_2$; and $\boldsymbol{\Sigma}_{22}$ is the covariance submatrix of $\mathbf{X}_2$. After this process, we can perform inference to obtain the expected mean $\boldsymbol{\mu}_{1|2}$ and covariance matrix $\boldsymbol{\Sigma}_{1|2}$ of the variables in $\mathbf{X}_1$ given $\mathbf{X}_2$ (Murphy 2012):

$$\boldsymbol{\mu}_{1|2} = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{x}_2 - \boldsymbol{\mu}_2), \tag{8}$$

$$\boldsymbol{\Sigma}_{1|2} = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}. \tag{9}$$

Multivariate Gaussian inference can be performed in **dbnR** with the function

```
mvn_inference(mu, sigma, evidence)
```

The `mu` and `sigma` arguments, which correspond to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, respectively, in Equation 7 are stored in the '`dbn.fit`' as attributes, and the `evidence` argument is a named vector with the values of the observed variables in $\mathbf{X}_2$. This returns a list with both the calculated $\boldsymbol{\mu}_{1|2}$ from Equation 8 and $\boldsymbol{\Sigma}_{1|2}$ from Equation 9. Typically, the `mvn_inference` function is not used outside of **dbnR** because it is already encapsulated in other exported methods for prediction and forecasting with DBN models, but we exported it too in case the user needs to perform inference over only a specific subset of nodes or wants to perform inference over a multivariate Gaussian distribution outside of the scope of the package.

4.2. Forecasting time series

When using DBNs to deal with TS data, one of the most common operations that we can perform is forecasting up to some horizon T. In **dbnR**, we perform forecasting with DBN models using a sliding window. First, given the initial state vector $\mathtt{s}_0 = ((x_1^0, \ldots, x_n^0), \ldots, (x_1^t, \ldots, x_n^t))$ with values of the variables in our system observed at instant t and at many previous instances as defined by the Markovian order of the network, we perform inference to obtain the values, $\hat{\mathbf{x}}^{t+1} = (\hat{x}_1^{t+1}, \ldots, \hat{x}_n^{t+1})$, that the variables are predicted to take at the next instant. After this, we move all the previous evidence forward in time. We forget the oldest evidence $\mathbf{x}^0$ from the system and introduce $\hat{\mathbf{x}}^{t+1}$ as the new evidence of the last instant to create the new

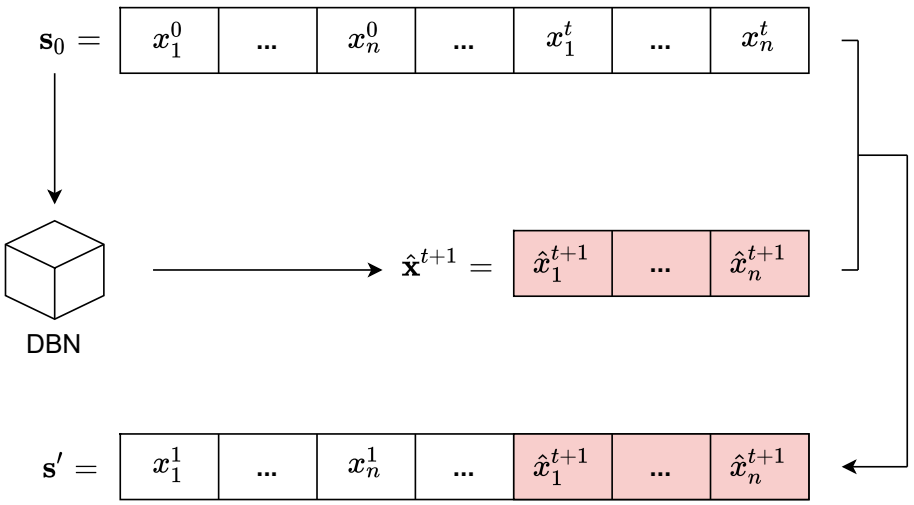

Figure 4: Schematic representation of the sliding window mechanism in an inference step. The new $\hat{\mathbf{x}}^{t+1}$ is predicted with the DBN model and is introduced into the state vector, removing the oldest $\mathbf{x}^0$.

state vector $\mathbf{s}' = ((x_1^1, \ldots, x_n^1), \ldots, (\hat{x}_1^{t+1}, \ldots, \hat{x}_n^{t+1}))$. This completes the current inference step, while $\mathbf{s}'$ is used as the initial state vector of the next inference step, predicting the next state of the system. This process is illustrated in Figure 4. We perform as many inference steps as needed to reach the desired horizon T. Finally, when the forecasting is completed, the values of the target variables at each instant are returned, and the mean absolute error (MAE) is calculated with the original TS if some test data are provided. If the user is making predictions without knowing the future values of the TS, as in a real-world application, no metrics will be calculated.

The whole process of forecasting up to some horizon T is encapsulated in the function

```
forecast_ts(dt, fit, obj_vars, ini, len, rep, num_p,
  print_res, plot_res, mode, prov_ev)
```

By providing a '`data.frame`' and a '`dbn.fit`' with the `dt` and the `fit` arguments, the **dbnR** package handles the moving window procedure underneath. The forecasting can be tuned by defining the target variables with `obj_vars`, setting the initial instance of the forecasting in the '`data.frame`' with `ini`, defining the horizon T regarding how long the forecasting should be with `len` and if either exact or approximate inference should be used with the `mode` parameter. If approximate inference is selected, the number of times that the inference is repeated before returning a mean value is defined by the `num_p` parameter. If test data are provided in `dt` and the `print_res` argument is set to `True`, the MAE of the forecasting compared to the original TS will be printed. The `plot_res` argument shows a plot of both the original and the predicted TS. An example of forecasting a TS and plotting the results can be seen in Figure 5. It is worth noting that, as shown in Equation 9, the conditional variance of the predictions is not affected by how many instants into the future we predict and will remain constant throughout the forecasting. This means that if we were to plot the pointwise prediction intervals of our forecasting, it would show a constant range of values around the

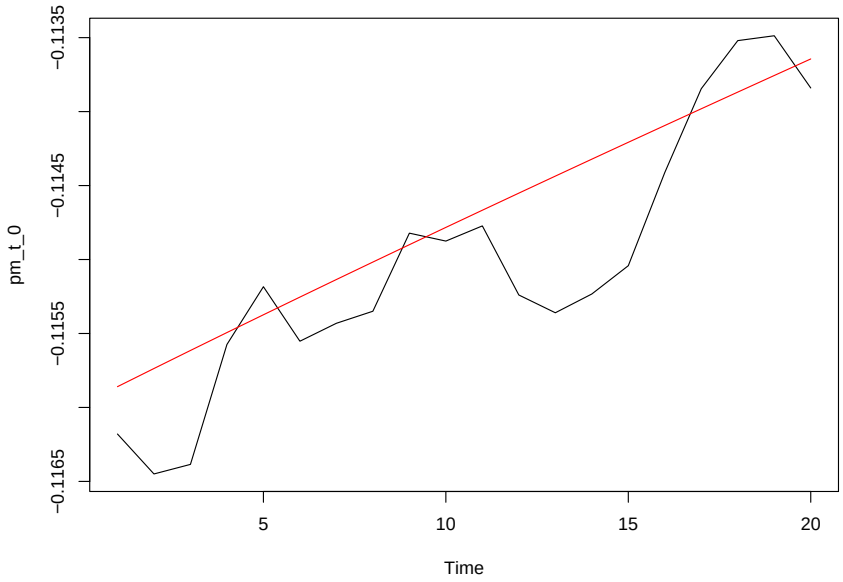

Figure 5: Plot returned by the `forecast_ts` function after forecasting 20 instances of a TS. The *pm* variable represents the magnet temperature inside an electric motor. The black line represents the original values of the TS, and the red line represents the forecasting. Only the values of the variables at the initial instant 0 are known as evidence to the DBN.

predicted mean. One should be careful with this characteristic, because the uncertainty of the predictions is expected to increase as we predict further into the future, we should not expect it to remain constant.

An additional argument `prov_ev` can be provided to the `forecast_ts` function, which allows the user to give specific future evidence to the DBN in each forecasting step. This can be useful when employing a DBN as a simulator to make interventions in the forecasting and see the effects that they have in the prediction profiles.

4.3. Smoothing

Additionally, DBN models can perform smoothing operations over TS (Koller and Friedman 2009). In this context, smoothing refers to, given some initial evidence from instants 1 to t, performing inference over $p(\mathbf{X}^0|\mathbf{X}^{1:t})$. Essentially, we predict the past given the current value of the variables in our system. Afterwards, we move all evidence backwards in the same manner as the sliding window from the forecasting case, but in the opposite direction in time. If we repeat this process up to horizon T in the past, we will obtain a TS that reflects the predicted state of the system along several instants in the past. Typically, this operation is performed when we do not know the past state of the system, for example, due to missing data or when we want to check how much our past data differ from the smoothed values to check for faulty sensor recordings.

Smoothing is called in a similar way to forecasting by using the function

```
smooth_ts(dt, fit, obj_vars, ini, len, mode, print_res, plot_res)
```

5. Package functions and compatibility

The **dbnR** package is focused on making DBN models readily available for application. As such, the main pipeline from some dataset to a fitted DBN model is kept as simple as possible. A diagram of this pipeline can be seen in Figure 6, where with four function calls, we can obtain a fully functional DBN model and perform inference with it.

Apart from the essential functionality, **dbnR** offers other utilities. In Table 2, we show all the exported functions of the package. For the learning task, all three structure learning algorithms are encapsulated and parametrized inside the `learn_dbn_struc()` function. Then, the user will always call the `fit_dbn_params()` function to fit the parameters of the network. As such, these two functions are pivotal in order to learn both the structure and the parameters of a DBN. On the other hand, the `calc_mu()` and the `calc_sigma()` functions are more situational. The user would only need to call these functions if they need to calculate the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$. The most likely scenario for this would be either to perform some specific inference with the `mvn_inference()` function or to perform some operation outside the scope of **dbnR** with the multivariate Gaussian equivalent of the network.

For visualization purposes, all networks can be plotted simply with a call to `plot()`, but the specific plot functions used underneath that call are the `plot_dynamic_network()` for DBNs and the `plot_static_network()` for BNs, which can also be used if the user prefers to do so.

On the inference task, the two functions that a user will most likely use are `forecast_ts()` to forecast a time series up to some horizon, and `predict_dt()` to forecast each row in a dataset a single step into the future. The `smooth_ts()` function allows the uncommon smoothing operation described in Section 4.3, and the `mvn_inference()` function allows the user a less restricted option for inference with the multivariate Gaussian equivalent of the network at the expense of more work on their end to create a forecasting pipeline. The `predict_bn()` function is similar to `predict_dt()`, but it is coded in a way that can be applied to several rows inside a '`data.table`'.

Additionally, the **dbnR** package offers some utility functions that can be of use. The ones needed for dataset transformation are the `fold_dt()` and `filtered_fold_dt()` functions. The `time_rename()` function is a subcomponent of the folding process that renames the columns inside a dataset to the `t_x` format used by **dbnR**, and the `filter_same_cycle()` function is a subcomponent of the filtered folding of a dataset that removes rows where different time series are mixed together. Unless the user is doing some specific operations

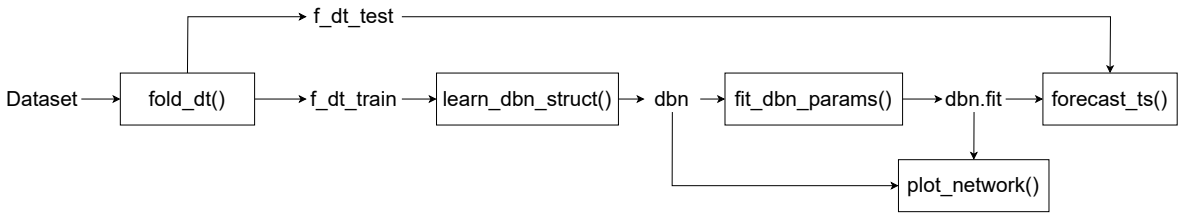

Figure 6: Diagram with the main workflow of the **dbnR** package. We start by preparing a dataset; then, we learn the DBN structure, learn its parameters and typically perform forecasting. Optionally, the network structure can also be plotted.

Task	Function	Output
Learning		
Learn DBN structure	`learn_dbn_struc`	'`dbn`'
Fit DBN parameters	`fit_dbn_params`	'`dbn.fit`'
Calculate $\boldsymbol{\mu}$ from '`dbn.fit`'	`calc_mu`	'`numeric`'
Calculate $\boldsymbol{\Sigma}$ from '`dbn.fit`'	`calc_sigma`	'`matrix`'
Visualization		
Plot DBN	`plot_dynamic_network`	HTML
Plot BN	`plot_static_network`	HTML
Inference		
Forecast a TS	`forecast_ts`	'`list`'
Smooth a TS	`smooth_ts`	'`list`'
Multivariate Gaussian inference	`mvn_inference`	'`list`'
Inference over '`data.table`'	`predict_dt`	'`data.table`'
Inference over '`data.table`'	`predict_bn`	'`data.table`'
Utilities		
Rename variables into `t_x`	`time_rename`	'`data.table`'
Fold dataset	`fold_dt`	'`data.table`'
Fold dataset based on index	`filtered_fold_dt`	'`data.table`'
Dataset folding utility	`filter_same_cycle`	'`data.table`'
Create a random DBN	`generate_random_network_exp`	'`list`'
Reduce frequency of TS	`reduce_freq`	'`data.table`'

Table 2: Overview of all the exported functions in the **dbnR** package ordered by the type of task they perform.

during the folding of the data, these two functions will rarely be used. For testing purposes, the `generate_random_network_exp()` function creates a randomly generated dataset and a '`dbn`' object that reflects the relationships present in the dataset. Finally, the `reduce_freq()` function allows the user to reduce the frequency in their time series by averaging batches of subsequent values to some desired new frequency.

Regarding the compatibility with **bnlearn**, all the functions for the addition or deletion of arcs and nodes offered by **bnlearn** also work for '`dbn`' objects. To show this compatibility, we use the code below to obtain a random DBN structure and a simulated dataset with the `generate_random_network_exp` function, and then apply some graph modification functions.

```
R> dbn_ex <- generate_random_network_exp(n_vars = 3, size = 2,
+    min_mu = -5, max_mu = 5, min_sd = 0.5, max_sd = 2, min_coef = -1,
+    max_coef = 1, seed = 42)
R> names(dbn_ex$net$nodes)

[1] "X0_t_0" "X1_t_0" "X2_t_0" "X0_t_1" "X1_t_1" "X2_t_1"

R> dbn_ex$net$arcs

     from     to
[1,] "X0_t_1" "X0_t_0"
```

```
[2,] "X1_t_1" "X0_t_0"
[3,] "X0_t_1" "X1_t_0"
[4,] "X1_t_1" "X1_t_0"
[5,] "X2_t_1" "X1_t_0"
[6,] "X0_t_1" "X2_t_0"
[7,] "X2_t_1" "X2_t_0"
```

We generated a random DBN with two time slices and three variables per time slice. In total, the network structure has six nodes and seven arcs. If we want to delete the first arc, we can do so with the `drop.arc` function from **bnlearn**, and if we want to delete a node entirely, we can use the `remove.node` function.

```
R> dbn_ex$net <- drop.arc(dbn_ex$net, "X0_t_1", "X0_t_0")
R> dbn_ex$net$arcs

     from     to
[1,] "X1_t_1" "X0_t_0"
[2,] "X0_t_1" "X1_t_0"
[3,] "X1_t_1" "X1_t_0"
[4,] "X2_t_1" "X1_t_0"
[5,] "X0_t_1" "X2_t_0"
[6,] "X2_t_1" "X2_t_0"

R> dbn_ex$net <- remove.node(dbn_ex$net, "X0_t_0")
R> names(dbn_ex$net$nodes)

[1] "X1_t_0" "X2_t_0" "X0_t_1" "X1_t_1" "X2_t_1"
```

Several other auxiliary functions from **bnlearn**, such as obtaining the node ordering of a network with the `node.ordering` function or getting the Markov blanket of a node by calling `mb` can be used in a similar manner.

6. Usage example: sample motor dataset

To show a practical full example of using the **dbnR** package, we use a dataset to learn the structure of three different DBNs, fit their parameters and perform forecasting with them. We use the sample dataset included in the package. The data come from the *electric motor temperature* dataset in Kaggle (Kirchgässner, Wallscheid, and Böcker 2021), from which we selected a sample of the first 3000 instances intended only for testing purposes of the package utilities. For the complete dataset, we refer the readers to the original source (https://www.kaggle.com/wkirgsn/electric-motor-temperature).

```
R> dt <- dbnR::motor
R> summary(dt)

    ambient            coolant              u_d
 Min.   :-0.79598   Min.   :-0.07434   Min.   :-1.6415
```

```
 1st Qu.: 0.01516   1st Qu.: 0.05816   1st Qu.: 0.3122
 Median : 0.05754   Median : 0.09546   Median : 0.3137
 Mean   : 0.05927   Mean   : 0.89587   Mean   : 0.3006
 3rd Qu.: 0.10300   3rd Qu.: 2.15739   3rd Qu.: 0.3157
 Max.   : 0.20956   Max.   : 2.27659   Max.   : 2.2359
      u_q              motor_speed            i_d
 Min.   :-1.332770   Min.   :-1.22243   Min.   :-2.4526
 1st Qu.:-1.328685   1st Qu.:-1.22243   1st Qu.: 0.2333
 Median :-1.326736   Median :-1.22243   Median : 1.0291
 Mean   :-0.601467   Mean   :-0.58853   Mean   : 0.4785
 3rd Qu.: 0.008286   3rd Qu.: 0.02407   3rd Qu.: 1.0291
 Max.   : 1.729171   Max.   : 1.87129   Max.   : 1.0292
      i_q                 pm             stator_yoke
 Min.   :-2.9470   Min.   :-0.14291   Min.   :-0.0564
 1st Qu.:-0.2524   1st Qu.:-0.11738   1st Qu.: 0.1337
 Median :-0.2457   Median : 0.04524   Median : 0.2650
 Mean   :-0.2941   Mean   : 0.01934   Mean   : 0.5307
 3rd Qu.:-0.2457   3rd Qu.: 0.12920   3rd Qu.: 0.9512
 Max.   : 2.2931   Max.   : 0.25813   Max.   : 1.5442
  stator_tooth       stator_winding
 Min.   :-0.31658   Min.   :-0.53658
 1st Qu.: 0.01911   1st Qu.:-0.22645
 Median : 0.31257   Median : 0.13088
 Mean   : 0.27072   Mean   : 0.08419
 3rd Qu.: 0.46184   3rd Qu.: 0.39245
 Max.   : 0.92338   Max.   : 0.71988
```

The sample dataset consists of 11 continuous variables that correspond to different temperatures, voltages and currents inside an electrical motor. Our aim is to use the data to fit the DBN models and showcase the whole process of training a DBN with some dataset and perform forecasting.

Initially, we split our data into training and test sets, and then use the `fold_dt` function to generate the necessary temporal variables in each row.

```
R> dt_train <- dt[1:2800]
R> dt_test <- dt[2801:3000]
R> size <- 2
R> f_dt_train <- fold_dt(dt_train, size)
R> f_dt_test <- fold_dt(dt_test, size)
R> print(names(f_dt_train))
```

```
 [1] "ambient_t_0"        "coolant_t_0"        "u_d_t_0"
 [4] "u_q_t_0"            "motor_speed_t_0"    "i_d_t_0"
 [7] "i_q_t_0"            "pm_t_0"             "stator_yoke_t_0"
[10] "stator_tooth_t_0"   "stator_winding_t_0" "ambient_t_1"
[13] "coolant_t_1"        "u_d_t_1"            "u_q_t_1"
[16] "motor_speed_t_1"    "i_d_t_1"            "i_q_t_1"
```

```
[19] "pm_t_1"             "stator_yoke_t_1"    "stator_tooth_t_1"
[22] "stator_winding_t_1"
```

For the sake of simplicity, we fix the size of the folding to 2 so that we learn Markovian order 1 DBNs. In a real world scenario, this size parameter is very important due to its relationship with the autoregressive order of time series, and as such it defines how many previous instants of time are needed to forecast the next instant. Given that there is no automatic way of determining the appropriate Markovian order of a network for a given dataset, it is better to start by fixing `size = 2` and obtaining the fastest baseline results for Markovian order 1 that can be empirically compared with slower higher orders later on. In practice, accuracies can improve considerably up to `size = 5` depending on the problem at hand, but not all structure learning algorithms are able to learn such high order networks. On datasets with 20 variables and 10.000 instances, the DMMHC algorithm can take more than 24 hours to learn networks of `size > 3`, and the particle swarm algorithms with 50 iterations and 300 particles can take between 1 and 2 hours for learning networks of `size > 7`. For a more in-depth comparison of execution time and performance of the three algorithms, we refer the readers to Quesada *et al.* (2021).

After splitting and folding, we create the necessary variables for the desired size and obtain a dataset that can be used for learning the structure and the parameters of the DBN models.

```
R> t <- Sys.time()
R> net_dmmhc <- learn_dbn_struc(dt_train, size, method = "dmmhc",
+    f_dt = f_dt_train)
R> Sys.time() - t

Time difference of 0.3007278 secs

R> set.seed(42)
R> t <- Sys.time()
R> net_psoho <- learn_dbn_struc(dt_train, size, method = "psoho",
+    f_dt = f_dt_train, n_it = 10)

  |======================================================================
   ======================================================================
   ============================================================| 100%

R> Sys.time() - t

Time difference of 3.022008 secs

R> t <- Sys.time()
R> net_nat <- learn_dbn_struc(dt_train, size, method = "natPsoho",
+    f_dt = f_dt_train, n_it = 10)

  |======================================================================
   ======================================================================
   ============================================================| 100%
```

```
R> Sys.time() - t

Time difference of 2.784051 secs
```

We used the three available structure learning algorithms and printed the execution time spent by each one. For small DBNs, the execution time of the DMMHC algorithm is unrivalled by the particle swarm algorithms, but it scales poorly to a greater number of nodes and higher Markovian orders. Note that the particle swarm algorithms are not deterministic, and we had to set a seed number to obtain reproducible results. They also print a progress bar that shows how much of the process is done in terms of the number of iterations finished from the total number of iterations allowed with the `n_it` parameter.

In terms of parameters, the DMMHC algorithm offers the option to add `blacklist` and `whitelist` parameters, which avoid or force arcs in the resulting structure respectively. This is a useful feature in the case that the user wants to obtain networks that adhere to some previous knowledge of the variables relationships. An additional parameter `blacklist_tr` is also provided to avoid specific inter-slice arcs. Underneath the DMMHC implementation, the algorithm calls the `rsmax2` function from the **bnlearn** package in order to apply the max-min hill-climbing algorithm. As a result, the user can provide further parameters to this function if they want to adjust the behaviour of this function.

In terms of the parameters of the particle swarm algorithms, the number of particles can be defined with the `n_inds` parameter and the number of iterations with the `n_it` parameter. By default they are set to 50, and increasing any of them will result in slower execution time but more exhaustive searches. Increasing the number of particles up to 100 or the number of iterations up to 200 can be a good, simple approach in case the user wants to see if better DBN structures can be found in terms of accuracy. Additionally, if the user is proficient with particle swarm optimization then they can modify the values of the inertia factor, the global best factor and the local best factor with the `in_cte`, `gb_cte` and `lb_cte` parameters respectively. This parameters determine how the particles move through the solution space. An extra parameter `cte` determines whether or not as the search progresses, the inertia and local best factor decrease while the global best factor increases to favour exploration at first and exploitation at the end. This behaviour is not enabled by default, but it can improve the results obtained in some cases.

With regards to the network structures obtained, the one from the DMMHC algorithm is more interpretable than the ones obtained from the particle swarm algorithms due to the presence of intra-slice arcs, as shown in Figure 7. While the particle swarm algorithms only show temporal relationships, in the Markov blanket of the DMMHC network we can see the effects of previous variables in red, current variables in blue and in grey we see variables that are not directly related to `pm_t_0` but affect its value. However, the networks obtained from the particle swarms algorithms tend to be simpler and more accurate for pure forecasting problems. In this example case, we will guide the final network selection by accuracy results, as the DBN structure interpretation is more relevant when working with experts in the problem that can extract information from the relationships shown in the network. As a result, the user should prioritize the PSOHO and natPSOHO algorithms over the DMMHC in cases where learning a high order network is required and where the objective is more focused on forecasting accuracy rather than interpretability.

After obtaining the network structures, we can now learn their parameters from the folded

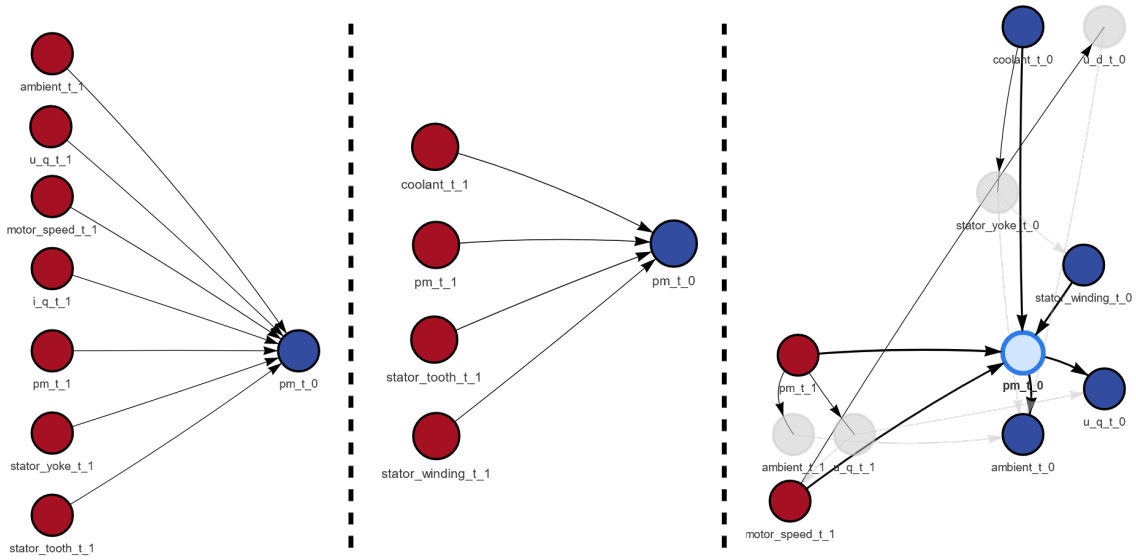

Figure 7: The plotted Markov blanket of the `pm_t_0` variable from the DBNs learned with the natPSOHO, PSOHO and DMMHC algorithms respectively. While the particle swarm algorithms offer much simpler networks where we only see the temporal effect of previous variables, the DMMHC network shows more complex relationships.

dataset with the `fit_dbn_params` function. This returns a '`dbn.fit`' object that can be used to perform inference.

```
R> fit_dmmhc <- fit_dbn_params(net_dmmhc, f_dt_train)
R> fit_psoho <- fit_dbn_params(net_psoho, f_dt_train)
R> fit_nat <- fit_dbn_params(net_nat, f_dt_train)
R> fit_dmmhc$pm_t_0

  Parameters of node pm_t_0 (Gaussian distribution)

Conditional density: pm_t_0 | coolant_t_0 + stator_winding_t_0 +
                              motor_speed_t_1 + pm_t_1
Coefficients:
       (Intercept)         coolant_t_0  stator_winding_t_0
      0.0017474119       -0.0014122266        0.0011111532
   motor_speed_t_1              pm_t_1
      0.0005594395        0.9849412445
Standard deviation of the residuals: 0.004961973
```

We can inspect a fitted model by checking specific nodes. In the previous code chunk, we printed the parameters of the `pm_t_0` node. This variable represents the temperature of the permanent magnet in the rotor of the motor and can be used to predict overheating. By checking its parameters, we can see all its parent nodes, as well as the effects that each one has on it. Note that all variables are normalized, so the scales of the parameters are similar and can be compared. We can see that the values of the last instant `pm_t_1` have a parameter

of approximately 0.985, which hints at the fact that the previous value in the TS is a very good prediction of the next one and has high correlation. If the variables were not normalized, the parameters would need to account for the difference in magnitude of the variables, and they would be much harder to interpret.

In our DBN models, the DMMHC algorithm allows intra-slice arcs, as shown by the `pm_t_0` variable having as parents other variables in t_0. The PSO algorithms, however, do not allow this kind of arc and only learn inter-slice arcs directed to t_0.

```
R> fit_nat$pm_t_1

  Parameters of node pm_t_1 (Gaussian distribution)

Conditional density: pm_t_1
Coefficients:
(Intercept)
 0.02872801
Standard deviation of the residuals: 0.123039

R> summary(f_dt_train[, pm_t_1])

    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.14291 -0.11971  0.06236  0.02873  0.13503  0.25813
```

In the case of variables with no parents, it can be seen that their intercept is equal to the mean of the TS in the original training dataset. This can be seen in the case of the PSO algorithms in each variable outside t_0, given that only variables in t_0 are allowed to have parent nodes in those algorithms.

With the fitted models, we can now perform inference over the test dataset. First, we use the `mvn_inference` function directly to perform a single inference step. We use the values from previous time slices as evidence and perform inference over the variables at t_0.

```
R> ev_vars <- names(f_dt_test)[grepl("t_1$", names(f_dt_test))]
R> ev <- f_dt_test[1, .SD, .SDcols = ev_vars]
R> ev

   ambient_t_1 coolant_t_1   u_d_t_1   u_q_t_1 motor_speed_t_1
1:   0.1377959    2.177947 0.3127252 -1.329747       -1.222431
    i_d_t_1    i_q_t_1     pm_t_1 stator_yoke_t_1 stator_tooth_t_1
1: 1.029135 -0.2457216 -0.1207832        1.471376        0.8375071
   stator_winding_t_1
1:          0.3224269
```

First, we extract the values of all the variables whose name finishes with `"t_1"`, that is, all the variables that are at the t_1 time slice from the first row of the folded test dataset. We can now feed the data to any of the '`dbn.fit`' objects that we trained earlier.

```
R> res <- mvn_inference(attr(fit_dmmhc, "mu"),
+    attr(fit_dmmhc, "sigma"), evidence = ev)
R> res

$mu_p
                       [,1]
coolant_t_0         2.1771681
stator_tooth_t_0    0.8386566
stator_yoke_t_0     1.4720731
stator_winding_t_0  0.3229435
u_d_t_0             0.3156188
pm_t_0             -0.1206166
ambient_t_0         0.1371033
u_q_t_0            -1.3303784
i_d_t_0             1.0286580
motor_speed_t_0    -1.2240255
i_q_t_0            -0.2442530

$sigma_p
                     coolant_t_0 stator_tooth_t_0 stator_yoke_t_0
coolant_t_0         3.730297e-03    -9.341215e-05   -6.949001e-05
stator_tooth_t_0   -9.341215e-05     2.389378e-03    1.943209e-05
stator_yoke_t_0    -6.949001e-05     1.943209e-05   -4.629102e-03
stator_winding_t_0  2.617418e-07    -1.718153e-05   -2.775256e-05
u_d_t_0             6.505213e-18    -1.870926e-17   -6.396793e-18
pm_t_0             -5.267734e-06     1.128278e-07    6.729829e-08
ambient_t_0        -7.032338e-07     1.722317e-07   -4.054521e-05
u_q_t_0            -1.899465e-07     4.068401e-09    2.426676e-09
i_d_t_0             5.745827e-07    -1.480856e-05   -4.137123e-07
motor_speed_t_0    -8.185877e-09     1.694358e-07    4.753232e-09
i_q_t_0            -3.539884e-08     8.669667e-09   -2.040933e-06
                   stator_winding_t_0       u_d_t_0        pm_t_0
coolant_t_0              2.617418e-07 -8.673617e-18 -5.267734e-06
stator_tooth_t_0        -1.718153e-05  4.370690e-18  1.128278e-07
stator_yoke_t_0         -2.775256e-05  1.864828e-17  6.729829e-08
stator_winding_t_0      -2.205694e-03 -1.019150e-17 -2.451233e-06
u_d_t_0                 -4.835542e-17  2.375250e-03 -3.361027e-18
pm_t_0                  -2.451233e-06  6.505213e-19 -4.675682e-04
ambient_t_0             -2.870911e-07 -1.169272e-06 -8.393435e-06
u_q_t_0                 -8.838777e-08  1.821460e-17 -1.685981e-05
i_d_t_0                 -2.332173e-05  1.573537e-05 -2.672546e-08
motor_speed_t_0          2.660363e-07 -1.800034e-07 -1.428645e-07
i_q_t_0                 -1.445137e-08 -2.165141e-05 -4.225023e-07
                     ambient_t_0       u_q_t_0       i_d_t_0
coolant_t_0        -7.032338e-07 -1.899465e-07  5.745827e-07
stator_tooth_t_0    1.722317e-07  4.068400e-09 -1.480856e-05
stator_yoke_t_0    -4.054521e-05  2.426675e-09 -4.137123e-07
```

```
stator_winding_t_0 -2.870911e-07 -8.838777e-08 -2.332173e-05
u_d_t_0            -1.169272e-06 -2.645453e-17  1.573537e-05
pm_t_0             -8.393435e-06 -1.685981e-05 -2.672546e-08
ambient_t_0         2.150180e-04 -3.026546e-07 -1.184961e-08
u_q_t_0            -3.026546e-07 -4.193088e-04 -9.636789e-10
i_d_t_0            -1.184961e-08 -9.636806e-10  1.887835e-02
motor_speed_t_0    -2.434532e-09 -3.560678e-06 -2.159572e-04
i_q_t_0             1.083404e-05 -1.523479e-08 -1.436411e-07
                   motor_speed_t_0       i_q_t_0
coolant_t_0          -8.185875e-09 -3.539884e-08
stator_tooth_t_0      1.694358e-07  8.669667e-09
stator_yoke_t_0       4.753232e-09 -2.040933e-06
stator_winding_t_0    2.660363e-07 -1.445137e-08
u_d_t_0              -1.800034e-07 -2.165141e-05
pm_t_0               -1.428645e-07 -4.225023e-07
ambient_t_0          -2.434532e-09  1.083404e-05
u_q_t_0              -3.560678e-06 -1.523479e-08
i_d_t_0              -2.159572e-04 -1.436411e-07
motor_speed_t_0      -5.066376e-02  1.513799e-09
i_q_t_0               1.513799e-09 -9.822611e-03
```

This returns both the $\boldsymbol{\mu}_{1|2}$ vector and $\boldsymbol{\Sigma}_{1|2}$ matrix calculated in Equation 8 and Equation 9, respectively. The $\boldsymbol{\mu}_{1|2}$ vector is used as the resulting value from the exact inference, that is, the most likely value for our predicted variables given the provided evidence. The $\boldsymbol{\Sigma}_{1|2}$ matrix is also returned, but it is less interesting in the case of Gaussian DBNs given that it remains constant no matter the evidence we provide, as shown by Equation 9, where only the constant values of the covariance matrix are used in its calculation.

The mean vector obtained only corresponds to a single instant prediction. We can automate this process for all the rows in a dataset with the `predict_dt` function in case our objective is to predict only the next time instant.

```
R> res <- predict_dt(fit_dmmhc, f_dt_test, obj_nodes = "pm_t_0")
```

```
MAE:
0.0004112805
SD:
0.0005886638
```

Along with the predictions, the `predict_dt` function prints the average MAE and the standard deviation of the residuals and plots the predictions, as shown in Figure 8. Although the inference to horizon 1 obtains a seemingly low MAE and the plot seems to be a good result, single-step predictions can be misleading. As shown earlier by the parameters, a good prediction of the next instant of a TS is the previous one. Sometimes, a TS model can just be passing forward the values of the variables, incurring good predictions for $T = 1$ but obtains worse results when forecasting.

For forecasting, we use the `forecast_ts` function, which allows us to forecast up to an arbitrary time horizon. We use the `pm_t_0` variable as our target variable, but more than one variable can be selected simultaneously as target variables.

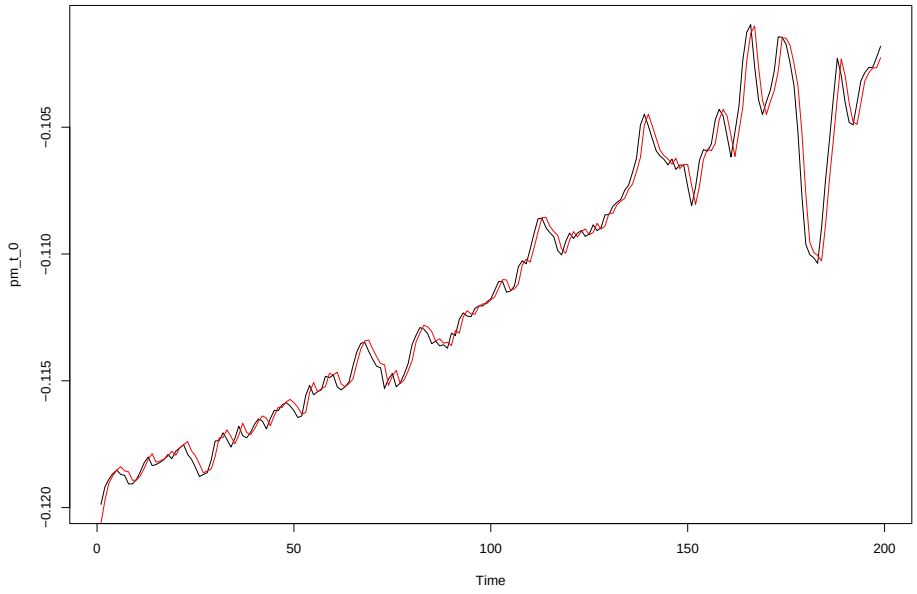

Figure 8: Plot of the predictions (red) returned by the `predict_dt` function and the real values of the time series (black). All the predictions are performed to horizon 1 using the last instant as evidence. They are deceivingly accurate because the last instant is always used as evidence for the next prediction. If looking closely, it can be seen that large changes in the profile of the curve are not properly predicted by the DBN until one instant later when evidence of these changes is provided to the model.

```
R> res <- forecast_ts(f_dt_test, fit_dmmhc, obj_vars = "pm_t_0",
+    ini = 40, len = 30, mode = "exact")

Time difference of -0.045842 secs
The average MAE per execution is:
pm_t_0: 4e-04
```

The obtained forecast shown in Figure 9 follows the tendency of the TS, but it is much smoother than the real values. This is because the exact inference in DBN models returns the most likely value in each instant, which is the mean value of the conditional multivariate Gaussian distribution. As such, the variance shown in $\mathbf{\Sigma}_{1|2}$ is expected to take place, but it remains constant throughout the forecast. We can show this issue in more detail by plotting the prediction intervals of the forecasting.

```
R> plot_pred_int_95 <- function(orig, pred, sigma, col) {
+    u_bound = pred + sigma_p*1.96
+    l_bound = pred - sigma_p*1.96
+    max_val = max(c(u_bound, l_bound))
+    min_val = min(c(u_bound, l_bound))
+    x = seq(length(pred))
+    plot(ts(orig), ylim = c(min_val, max_val), ylab = col)
+    polygon(c(x, rev(x)), c(u_bound, rev(l_bound)),
```

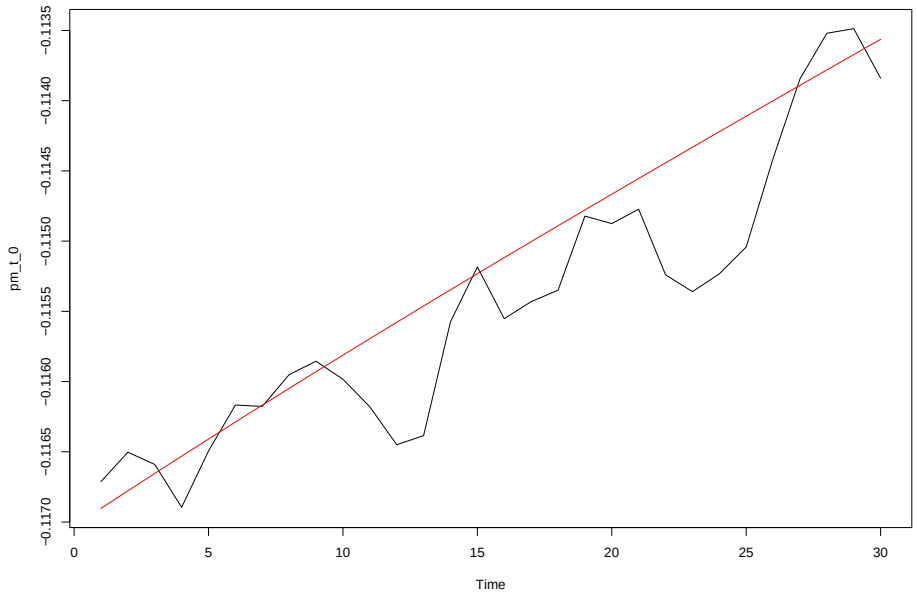

Figure 9: Plot obtained from forecasting 30 instances with a DBN model using only the evidence from the initial point at $t = 0$. The black line represents the original values of the TS, and the red line represents the forecasting.

```
+       col = adjustcolor("purple", alpha.f = 0.1), lty = 0)
+     lines(pred, col = "red")
+     lines(u_bound, col = "purple")
+     lines(l_bound, col = "purple")
+   }
```

The previous function will plot the real values as a black line, the forecasting as a red line and the prediction interval will be shown as two pruple lines and the violet area in between them. This pointwise prediction interval is calculated like that of a Gaussian distribution $[\mu_X - z\sigma_X, \mu_X + z\sigma_X]$.

```
R> cols <- names(f_dt_test)[-8]
R> sigma_p <- mvn_inference(attr(fit, 'mu'), attr(fit, 'sigma'),
+     f_dt_test[1, .SD, .SDcols = cols])$sigma_p[1]
R> plot_pred_int_95(res$orig$pm_t_0, res$pred$pm_t_0, sigma_p, "pm_t_0")
```

The resulting plot can be seen in Figure 10. In this case, the pointwise prediction interval helps us see the implications of the expected variance of the predictions, but we can also see how this interval remains the same in the first instant of the predictions, when we have real evidence values, up to the last instant. It is also important to note that the model only sees the evidence at the first instant of time, which means that any future intervention in the TS will not be seen by the DBN.

If we are applying a DBN model in real time, we do not have the values of the full TS when performing forecasting. In this case, we can provide the model with the evidence of the initial time point, and it will return the forecast without performing MAE calculations.

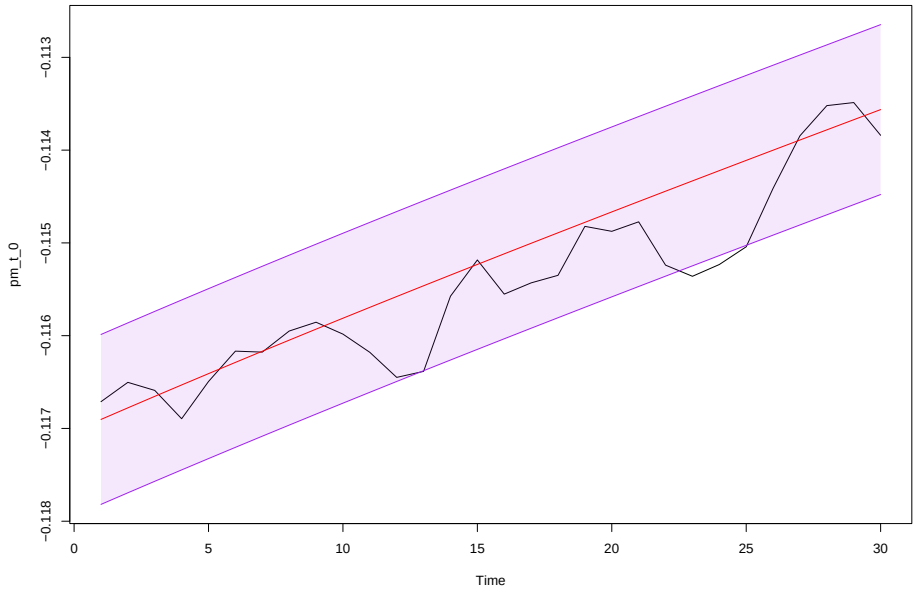

Figure 10: Plot and prediction interval obtained from forecasting 30 instances with a DBN model. The black line represents the original values of the TS, the red line represents the forecasting and the violet area represents the 95% pointwise prediction interval.

```
R> res <- forecast_ts(f_dt_test[40], fit_dmmhc, obj_vars = "pm_t_0",
+    ini = 1, len = 5, mode = "exact", plot_res = FALSE, print_res = FALSE)
R> res$pred$pm_t_0

[1] -0.1169029 -0.1167780 -0.1166541 -0.1165312 -0.1164092
```

We can also use the DBN model as a simulator by providing specific evidence over time during forecasting. Thus, we can test the effects of specific values on some key variables or see how certain interventions affect the system. In our example data, we may want to test how the increasing or decreasing revolutions per minute of the motor represented by the `motor_speed` variable affects our objective temperature. Our first scenario is to fix this value at a very low rate, lower than the real values of the TS. In the second scenario that we propose, we progressively increase the revolutions per minute over time from a very low starting point to see the effects that accelerating a car would have on the permanent magnet temperature.

```
f_dt_i <- f_dt_test[40:70]
summary(f_dt_i$motor_speed_t_0)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.222  -1.222  -1.222  -1.222  -1.222  -1.222

f_dt_i[1:5, motor_speed_t_0]

[1] -1.222428 -1.222428 -1.222430 -1.222432 -1.222431
```

We first extract the 30 instances of our previous forecasting to test both interventions because we already know how the model behaves for that period of time. The `motor_speed` variable is fairly constant at -1.22 on this interval. Note that the variables are normalized, and it does not mean negative revolutions per minute. We will modify this subset of data by fixing the values of `motor_speed_t_0` to a lower value of -1.4, and then to a sequence of values increasing from -1.4 to -1.1 to simulate motor acceleration.

```
R> f_dt_i[, motor_speed_t_0 := -1.4]
R> res <- forecast_ts(f_dt_i, fit_dmmhc, obj_vars = "pm_t_0",
+    ini = 1, len = 30, mode = "exact", prov_ev = "motor_speed_t_0")

Time difference of -0.104531 secs
The average MAE per execution is:
pm_t_0: 9e-04

R> f_dt_i[, motor_speed_t_0 := seq(from = -1.4, to = -1.1, by = 0.3 / 30)]
R> res <- forecast_ts(f_dt_i, fit_dmmhc, obj_vars = "pm_t_0",
+    ini = 1, len = 30, mode = "exact", prov_ev = "motor_speed_t_0")

Time difference of -0.100183 secs
The average MAE per execution is:
pm_t_0: 4e-04
```

The plots resulting from both scenarios can be seen in Figure 11. If we set a low `motor_speed`, the temperature of the magnet increases much more slowly than in the real case. However, we can see the effect that accelerating the motor would have on the temperature, which ramps up when we start increasing the revolutions per minute. This behaviour as simulators of the real world is a powerful tool that can turn DBNs into generative models that offer insight into industrial processes before making an intervention in a system.

As a final example and to evaluate the performance of the different DBN models, we have prepared a pipeline that will perform inference over a span of 20 instants for every row in our test dataset and then calculate the final MAE of the models.

```
R> mae <- function(orig, pred) {
+    res <- 0
+    for(i in 1:(dim(f_dt_test)[1]-len))
+      res_fore <- forecast_ts(f_dt_test, fit, obj_vars = obj_var,
+        ini = i, len = len, plot_res = FALSE,print_res = FALSE)
+      res <- res + mae(res_fore$orig[, get(obj_var)],
+        res_fore$pred[, get(obj_var)])
+    res <- res / (dim(f_dt_test)[1] - len)
+    cat(paste0("The final MAE of the model is: ", res, "\n"))
+    return(res)
+  }
```

Given that the `forecast_ts` function returns both the original values and the predicted ones, we can use any metric to evaluate the results. In our case, we define the MAE in an auxiliary function and then define a function that will perform this testing over the test dataset with any fit provided to it.

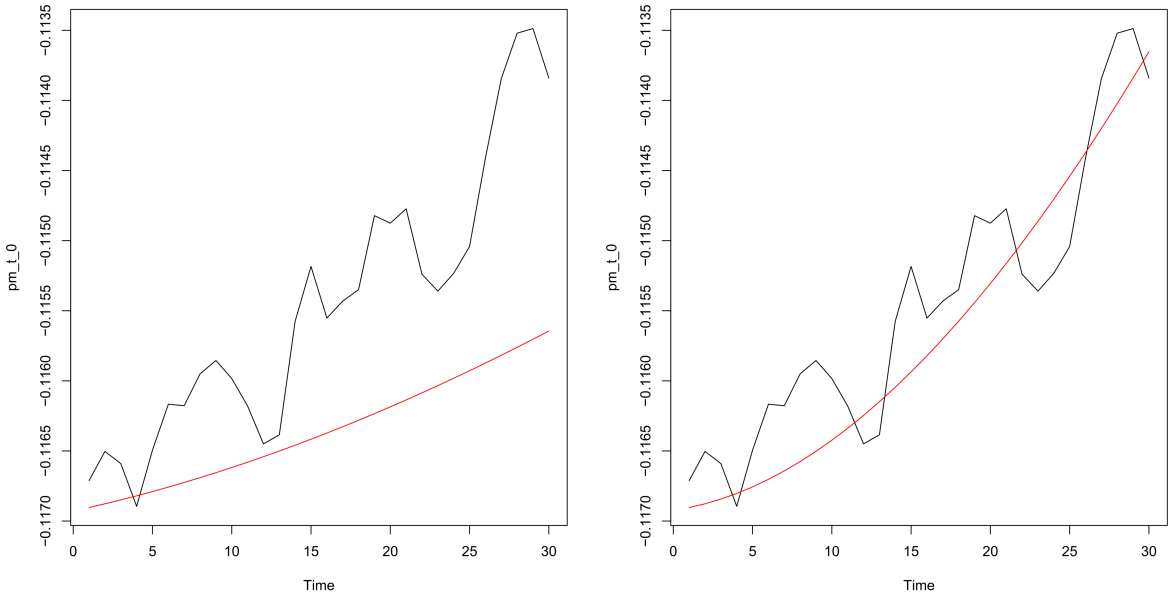

Figure 11: A comparison between the plotted forecasts of fixing the `motor_speed` variable to -1.4 (left) and progressively increasing it from -1.4 to -1.1 (right). The real values of the TS are represented by black lines, and the red lines represent the forecasts. The effects of both actions can be clearly seen in the profile of the predictions.

```
R> res_dmmhc <- eval_fit(f_dt_test, fit_dmmhc)

The final MAE of the model is: 1.29892071902173e-05

R> res_psoho <- eval_fit(f_dt_test, fit_psoho)

The final MAE of the model is: 1.40902160021131e-05

R> res_nat <- eval_fit(f_dt_test, fit_nat)

The final MAE of the model is: 1.29356816394937e-05
```

From this comparison, we can see that the best results by a small margin in terms of MAE are obtained with the DBN learned with the natPSOHO algorithm for this particular example.

7. Conclusions

In this paper, we have presented the **dbnR** package for Gaussian DBN learning, inference and visualization. The package covers the whole process from learning both the structure and parameters of a DBN model to performing inference and forecasting with it. It also extends the functionality of the most popular BN package in R, **bnlearn**, to the case of DBNs. The intermediate steps of the learning and inference process, including the structure learning algorithms, are presented and discussed regarding both their definitions and their implementations.

For future work, we would like to add an option to show an automatically generated user interface with **shiny** (Chang *et al.* 2021). This would give the simulator of DBN models more capacity to interact with the user, as well as generating tools readily available for a data scientist to present prototypes to an expert on a specific problem. This can be especially useful in the case of BNs and DBNs due to their capacity to incorporate expert knowledge into the model itself and to show the results of inference clearly and directly to the model's end users. On the subject of discrete or hybrid networks, **dbnR** is not envisioned to be extended to either of those cases on the foreseeable future. We would like to refer the readers to either the R package **bnlearn** that can be adapted to the case of dicrete or hybrid DBNs with some work on the users end or the Python package **PyBNesian** for their applications that require either discrete nodes, hybrid networks or nodes learned via kernel density estimation.

Computational details

The results in this paper were obtained using R 4.4.1 with the **dbnR** 0.8.0 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/. The **dbnR** package is licensed under GPL-3 or later license. All the examples were run on a 64 bits Windows 10 machine with an Intel i5-6200U CPU at 2.30GHz and 8GB of RAM.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation through the PID2022-139977NB-I00 and TED2021-131310B-I00 projects, and by the grant to the ELLIS Unit Madrid by the Autonomous Region of Madrid.

References

Almende BV, Thieurmel B, Robert T (2019). ***visNetwork**: Network Visualization Using **vis.js** Library*. doi:10.32614/CRAN.package.visNetwork. R package version 2.0.9.

Ankan A, Panda A (2015). "**pgmpy**: Probabilistic Graphical Models Using Python." In *Proceedings of the 14th Python in Science Conference*, volume 10. Citeseer.

Atienza D, Bielza C, Larrañaga P (2022). "**PyBNesian**: An Extensible Python Package for Bayesian Networks." *Neurocomputing*, **504**, 204–209. ISSN 0925-2312. doi:10.1016/j.neucom.2022.06.112.

Bielza C, Larrañaga P (2014). "Bayesian Networks in Neuroscience: A Survey." *Frontiers in Computational Neuroscience*, **8**, 131. doi:10.3389/fncom.2014.00131.

Cai B, Shao X, Liu Y, Kong X, Wang H, Xu H, Ge W (2019). "Remaining Useful Life Estimation of Structure Systems under the Influence of Multiple Causes: Subsea Pipelines as a Case Study." *IEEE Transactions on Industrial Electronics*, **67**(7), 5737–5747. doi:10.1109/tie.2019.2931491.

Chang W (2021). ***R6**: Encapsulated Classes with Reference Semantics*. doi:10.32614/CRAN.package.R6. R package version 2.5.1.

Chang W, Cheng J, Allaire J, Sievert C, Schloerke B, Xie Y, Allen J, McPherson J, Dipert A, Borges B (2021). ***shiny**: Web Application Framework for R*. doi:10.32614/CRAN.package.shiny. R package version 1.7.1.

Chaudhary S, Indu S, Chaudhury S (2017). "Video-Based Road Traffic Monitoring and Prediction Using Dynamic Bayesian Networks." *IET Intelligent Transport Systems*, **12**(3), 169–176. doi:10.1049/iet-its.2016.0336.

Denis JB, Scutari M (2021). ***rbmn**: Handling Linear Gaussian Bayesian Networks*. doi:10.32614/CRAN.package.rbmn. R package version 0.9-4.

Dowle M, Srinivasan A (2021). ***data.table**: Extension of 'data.frame'*. doi:10.32614/CRAN.package.data.table. R package version 1.14.2.

Ducamp G, Gonzales C, Wuillemin PH (2020). "**aGrUM/pyAgrum**: A Toolbox to Build Models and Algorithms for Probabilistic Graphical Models in Python." In *10th International Conference on Probabilistic Graphical Models*, pp. 609–612.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Fernandes R (2020). ***dbnlearn**: Dynamic Bayesian Network Structure Learning, Parameter Learning and Forecasting*. doi:10.32614/CRAN.package.dbnlearn. R package version 0.1.0.

Geiger D, Heckerman D (1994). "Learning Gaussian Networks." In *Uncertainty in Artificial Intelligence Proceedings 1994*, pp. 235–243. Elsevier.

Højsgaard S (2012). "Graphical Independence Networks with the **gRain** Package for R." *Journal of Statistical Software*, **46**(10), 1–26. doi:10.18637/jss.v046.i10.

Kirchgässner W, Wallscheid O, Böcker J (2021). "Estimating Electric Motor Temperatures with Deep Residual Machine Learning." *IEEE Transactions on Power Electronics*, **36**(7), 7480–7488. doi:10.1109/tpel.2020.3045596.

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

Korb KB, Nicholson AE (2010). *Bayesian Artificial Intelligence*. CRC press. doi:10.1201/b10391.

Murphy KP (2002). *Dynamic Bayesian Networks: Representation, Inference and Learning*. University of California, Berkeley.

Murphy KP (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Pearl J (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Quesada D (2026). ***dbnR**: Dynamic Bayesian Network Learning and Inference*. doi:10.32614/CRAN.package.dbnR. R package version 0.8.0.

Quesada D, Bielza C, Larrañaga P (2021). "Structure Learning of High-Order Dynamic Bayesian Networks via Particle Swarm Optimization with Order Invariant Encoding." In *International Conference on Hybrid Artificial Intelligence Systems*, pp. 158–171. Springer-Verlag. doi:10.1007/978-3-030-86271-8_14.

Robinson RW (1977). "Counting Unlabeled Acyclic Digraphs." In *Combinatorial Mathematics V*, pp. 28–43. Springer-Verlag.

Santos FP, Maciel CD (2014). "A PSO Approach for Learning Transition Structures of Higher-Order Dynamic Bayesian Networks." In *5th ISSNIP-IEEE Biosignals and Biorobotics Conference (2014): Biosignals and Robotics for Better and Safer Living*, pp. 1–6. IEEE. doi:10.1109/brc.2014.6880957.

Schwarz G (1978). "Estimating the Dimension of a Model." *The Annals of Statistics*, **6**(2), 461–464. doi:10.1214/aos/1176344136.

Scutari M (2010). "Learning Bayesian Networks with the **bnlearn** R Package." *Journal of Statistical Software*, **35**(3), 1–22. doi:10.18637/jss.v035.i03.

Song L, Kolar M, Xing E (2009). "Time-Varying Dynamic Bayesian Networks." *Advances in Neural Information Processing Systems*, **22**. doi:10.1093/bioinformatics/btp192.

Suter P, Kuipers J, Moffa G, Beerenwinkel N (2023). "Bayesian Structure Learning and Sampling of Bayesian Networks with the R Package **BiDAG**." *Journal of Statistical Software*, **105**(9), 1–31. doi:10.18637/jss.v105.i09.

Taskesen E (2020). ***bnlearn**: Library for Bayesian Network Learning and Inference*. URL https://erdogant.github.io/bnlearn.

Trabelsi G, Leray P, Ayed MB, Alimi AM (2013). "Dynamic MMHC: A Local Search Algorithm for Dynamic Bayesian Network Structure Learning." In *International Symposium on Intelligent Data Analysis*, pp. 392–403. Springer-Verlag. doi:10.1007/978-3-642-41398-8_34.

Tsamardinos I, Brown LE, Aliferis CF (2006). "The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm." *Machine Learning*, **65**(1), 31–78. doi:10.1007/s10994-006-6889-7.

Wang Y, Berceli SA, Garbey M, Wu R (2019). "Inference of Gene Regulatory Network through Adaptive Dynamic Bayesian Network Modeling." In *Contemporary Biostatistics with Biopharmaceutical Applications*, pp. 91–113. Springer-Verlag. doi:10.1007/978-3-030-15310-6_5.

Zhu J, Zhang W, Li X (2019). "Fatigue Damage Assessment of Orthotropic Steel Deck Using Dynamic Bayesian Networks." *International Journal of Fatigue*, **118**, 44–53. doi:10.1016/j.ijfatigue.2018.08.037.

Affiliation:

David Quesada, Pedro Larrañaga, Concha Bielza
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid
Campus de Montegacedo, SN
Madrid, Spain
E-mail: dquesada@fi.upm.es, pedro.larranaga@fi.upm.es, mcbielza@fi.upm.es
URL: https://www.github.com/dkesada,
https://cig.fi.upm.es/CIGmembers/pedro-larranaga,
https://cig.fi.upm.es/CIGmembers/concha_bielza

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
December 2025, Volume 115, Issue 6 *Submitted:* 2023-08-29
doi:10.18637/jss.v115.i06 *Accepted:* 2024-09-26