`bnstruct`: an R package for Bayesian Network Structure Learning with missing data

Francesco Sambo, Alberto Franzin

December 1, 2016

1 Introduction

Bayesian Networks (Pearl [8]) are a powerful tool for probabilistic inference among a set of variables, modeled using a directed acyclic graph. However, one often does not have the network, but only a set of observations, and wants to reconstruct the network that generated the data. The `bnstruct` package provides objects and methods for learning the structure and parameters of the network in various situations, such as in presence of missing data, for which it is possible to perform *imputation* (guessing the missing values, by looking at the data). The package also contains methods for learning using the Bootstrap technique. Finally, `bnstruct`, has a set of additional tools to use Bayesian Networks, such as methods to perform belief propagation.

In particular, the absence of some observations in the dataset is a very common situation in real-life applications such as biology or medicine, but very few software around is devoted to address these problems. `bnstruct` is developed mainly with the purpose of filling this void.

This document is intended to show some examples of how `bnstruct` can be used to learn and use Bayesian Networks. First we describe how to manage data sets, how to use them to discover a Bayesian Network, and finally how to perform some operations on a network. Complete reference for classes and methods can be found in the package documentation.

1.1 Overview

We provide here some general informations about the context for understanding and using properly this document. A more thorough introduction to the topic can be found for example in Koller and Friedman [6].

1.1.1 The data

A *dataset* is a collection of rows, each of which is composed by the same number of values. Each value corresponds to an observation of a *variable*, which is a feature, an event or an entity considered significant and therefore measured.

In a Bayesian Network, each variable is associated to a node. The number of variables is the *size* of the network. Each variable has a certain range of values it can take. If the variable can take any possible value in its range, it is called a *continuous* variable; otherwise, if a variable can only take some values in its range, it is a *discrete* variable. The number of values a variable can take is called its *cardinality*. A continuous variable has infinite cardinality; however, in order to deal with it, we have to restrict its domain to a smaller set of values, in order to be able to treat it as a discrete variable; this process is called *quantization*, the number of values it can take is called the number of *levels* of the quantization step, and we will therefore call the cardinality of a continuous variable the number of its quantization levels, with a little abuse of terminology.

In many real-life applications and contexts, it often happens that some observations in the dataset we are studying are absent, for many reasons. In this case, one may want to "guess" a reasonable (according to the other observations) value that would have been present in the dataset, if the observations was successful. This inference task is called *imputation*. In this case, the dataset with the "holes filled" is called the *imputed dataset*, while the original dataset with missing values is referred to as the *raw dataset*. In section 3.2 we show how to perform imputation in `bnstruct`.

Another common operation on data is the employment of resampling techniques in order to estimate properties of a distribution from an approximate one. This usually allows to have more confidence in the results. We implement the *bootstrap* technique (Efron and Tibshirani [2]), and provide it to generate samples of a dataset, with the chance of using it on both raw and imputed data.

1.1.2 Bayesian Networks

After introducing the data, we are now ready to talk about *Bayesian Networks*. A Bayesian Network (hereafter sometimes simply *network*, *net* or *BN* for brevity) is a probabilistic graphical model that encodes the conditional dependency relationships of a set of variables using a Directed Acyclic Graph (DAG). Each node of the graph represents one variable of the dataset; we will therefore interchange the terms *node* and *variable* when no confusion arises. The set of directed edges connecting the nodes forms the *structure* of the network, while the set of conditional probabilities associated with each variable forms the set of *parameters* of the net.

The DAG is represented as an *adjacency matrix*, a $n \times n$ matrix, where n is the number of nodes, whose cells of indices (i, j) take value 1 if there is an edge going from node i to node j, and 0 otherwise.

The problems of learning the structure and the parameters of a network from data define the *structure learning* and *parameter learning* tasks, respectively.

Given a dataset of observations, the structure learning problem is the problem of finding the DAG of a network that may have generated the data. Several algorithms have been proposed fot this problem, but a complete search is doable only for networks with no more than 20-30 nodes. For larger networks, several

heuristic strategies exist.

The subsequent problem of parameter learning, instead, aims to discover the conditional probabilities that relate the variables, given the dataset of observations and the structure of the network.

In addition to structure learning, sometimes it is of interest to estimate a level of the confidence on the presence of an edge in the network. This is what happens when we apply bootstrap to the problem of structure learning. The result is not a DAG, but a different entity that we call *weighted partially DAG*, which is an adjacency matrix whose cells of indices (i, j) take the number of times that an edge going from node i to node j appear in the network obtained from each bootstrap sample.

As the graph obtained when performing structure learning with bootstrap represents a measure of the confidence on the presence of each edge in the original network, and not a binary response on the presence of the edge, the graph is likely to contain undirected edges or cycles.

As the structure learnt is not a DAG but a measure of confidence, it cannot be used to learn conditional probabilities. Therefore, parameter learning is not defined in case of network learning with bootstrap.

One of the most relevant operations that can be performed with a Bayesian Network is to perform *inference* with it. Inference is the operations that, given a set of observed variables, computes the probabilities of the remaining variables updated according to the new knowledge. Inference answers questions like "How does the probability for variable Y change, given that variable X is taking value x?".

2 Installation

The latest stable version of the package is available on CRAN `https://cran.r-project.org`, and can therefore be installed from an R session using

```
> install.packages("bnstruct")
```

The latest development version of `bnstruct` can be found at `http://github.com/sambofra/bnstruct`.

In order to install the package, it suffices to open a shell and run

```
git clone https://github.com/sambofra/bnstruct.git
cd bnstruct
make install
```

`bnstruct` requires R $\geq$ 2.10, and depends on `bitops`, `igraph`, `Matrix` and `methods`. Package `Rgraphviz` is requested in order to plot graphs, but is not mandatory.

3 Data sets

The class that `bnstruct` provides to manage datasets is `BNDataset`. It contains all of the data and the informations related to it: raw and imputed data, raw and imputed bootstrap samples, and variable names and cardinality.

3.1 Creating a BNDataset

There are two ways to build a BNDataset: using two files containing respectively header informations and data, and manually providing the data table and the related header informations (variable names, cardinality and discreteness).

```
> dataset.from.data <- BNDataset(data = data,
+                                discreteness = rep('d',4),
+                                variables = c("a", "b", "c", "d"),
+                                node.sizes = c(4,8,12,16))
>
> dataset.from.file <- BNDataset("path/to/data.file",
+                                "path/to/header.file")
```

The key informations needed are:

1. the data;
2. the state of variables (discrete or continuous);
3. the names of the variables;
4. the cardinalities of the variables (if discrete), or the number of levels they have to be quantized into (if continuous).

Names and cardinalities/leves can be guessed by looking at the data, but it is strongly advised to provide *all* of the informations, in order to avoid problems later on during the execution.

Data can be provided in form of data.frame or matrix. It can contain NAs. By default, NAs are indicated with '?'; to specify a different character for NAs, it is possible to provide also the `na.string.symbol` parameter. The values contained in the data have to be numeric (real for continuous variables, integer for discrete ones). The default range of values for a discrete variable `X` is `[1,|X|]`, with `|X|` being the cardinality of `X`. The same applies for the levels of quantization for continuous variables. If the value ranges for the data are different from the expected ones, it is possible to specify a different starting value (for the whole dataset) with the `starts.from` parameter. E.g. by `starts.from=0` we assume that the values of the variables in the dataset have range `[0,|X|-1]`. Please keep in mind that the internal representation of Rpackagebnstruct starts from 1, and the original starting values are then lost.

It is possible to use two files, one for the data and one for the metadata, instead of providing manually all of the info. bnstruct requires the data files

to be in a format subsequently described. The actual data has to be in (a text file containing data in) tabular format, one tuple per row, with the values for each variable separated by a space or a tab. Values for each variable have to be numbers, starting from `1` in case of discrete variables. Data files can have a first row containing the names of the corresponding variables.

In addition to the data file, a header file containing additional informations can also be provided. An header file has to be composed by three rows of tab-delimited values:

1. list of names of the variables, in the same order of the data file;
2. a list of integers representing the cardinality of the variables, in case of discrete variables, or the number of levels each variable has to be quantized in, in case of continuous variables;
3. a list that indicates, for each variable, if the variable is continuous (`c` or `C`), and thus has to be quantized before learning, or discrete (`d` or `D`).

In case of need of more advanced options when reading a dataset from files, please refer to the documentation of the **`read.dataset`** method. Imputation and bootstrap are also available as separate routines (**`impute`** and **`bootstrap`**, respectively).

We provide two sample datasets, one with complete data (the **`Asia`** network, Lauritzen and Spiegelhalter [7]) and one with missing values (the **`Child`** network, Spiegelhalter, Dawid, Lauritzen, and Cowell [10]), in the **`extdata`** subfolder; the user can refer to them as an example. The two datasets have been created with

```
> asia <- BNDataset("asia_10000.data",
+                   "asia_10000.header",
+                   starts.from=0)
> child <- BNDataset("Child_data_na_5000.data",
+                    "Child_data_na_5000.header",
+                    starts.from=0)
```

and are also available with

```
> asia  <- asia()
> child <- child()
```

3.2 Imputation

A dataset may contain various kinds of missing data, namely unobserved variables, and unobserved values for otherwise observed variables. We currently deal only with this second kind of missing data. The process of guessing the missing values is called *imputation*.

We provide the **`impute`** function to perform imputation.

```
> dataset <- BNDataset(data.file   = "path/to/file.data",
+                      header.file = "path/to/file.header")
> dataset <- impute(dataset)
```

Imputation is accomplished with the k-Nearest Neighbour algorithm. The number of neighbours to be used can be chosen specifying the `k.impute` parameter (default is `k.impute = 10`). Given that the parameter is highly dataset-dependant, we also include the `tune.knn.impute` function to assist the user while choosing the best value for `k`.

3.3 Bootstrap

`BNDataset` objects have also room for bootstrap samples (Efron and Tibshirani [2]), i.e. random samples with replacement of the original data with the same number of observations, both for raw and imputed data. Samples for imputed data are generated by imputing the corresponding sample of raw data. Therefore, by requesting imputed samples, also the raw samples will be generated.

We provide the `bootstrap` method for this.

```
> dataset <- BNDataset("path/to/file.data",
+                      "path/to/file.header")
> dataset <- bootstrap(dataset, num.boots = 100)
> dataset.with.imputed.samples <- bootstrap(dataset,
+                                           num.boots = 100,
+                                           imputation = TRUE)
```

3.4 Using data

After a `BNDataset` has been created, it is ready to be used. The complete list of methods available for a `BNDataset` object is available in the package documentation; we are not going to cover all of the methods in this brief series of examples.

For example, one may want to see the dataset.

```
> # the following are equivalent:
> print(dataset)
> show(dataset)
> dataset # from inside an R session
```

The `show()` method is an alias for the `print()` method, but allows to print the state of an instance of an object just by typing its name in an `R` session.

The main operation that can be done with a `BNDataset` is to get the data it contains. The main methods we provide are `raw.data` and `imputed.data`, which provide the raw and the imputed data, respectively. The data must be present in the object; conversely, an error will be raised. To avoid an abrupt

termination of the execution in case of error, one may run these methods in a `tryCatch()` construct and manage the errors in case they happen. Another alternative is to test the presence of data before attempting to retrieve it, using the tester methods `has.raw.data` and `has.imputed.data`.

```
> options(max.print = 200, width = 60)
>
> dataset <- child()
> # if we want raw data
> raw.data(dataset)

      V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15
 [1,]  2  3  3 NA  1 NA  1  1  2   1   1   1  NA   2   2
 [2,] NA NA  2  1  1  2  1  2  2   1   2   1   2   2  NA
 [3,]  2  3  1  2  1 NA NA  2  2   1   2   2   2   2   2
 [4,]  2  4  1  1  1  3 NA  1  2  NA   3  NA   1   2   1
 [5,]  2  2  1 NA  2  4  1  1  1   1  NA   1  NA   2  NA
 [6,] NA  2 NA  2  1  4 NA  3  2   1   3   1   3   2   2
 [7,]  2  2  1  2 NA  4 NA  3 NA   1  NA   1   1  NA  NA
 [8,] NA  1  1 NA  3  1  1  1  2   2   2   1   4   2   2
 [9,]  2  3  2  2  1  3  1  2  2   1   2   1   2   2   2
[10,]  2  4  1  1  1  3 NA NA  2  NA   3   3   2   2   1
      V16 V17 V18 V19 V20
 [1,]   2   3   2   1   2
 [2,]   2   2   1   2   2
 [3,]   1   2   1   2   2
 [4,]   3   1  NA   1  NA
 [5,]  NA   1   1   2   2
 [6,]   2   1   1   3   2
 [7,]  NA   1   1   1  NA
 [8,]   1   1  NA   4  NA
 [9,]  NA   1   1   2   2
[10,]   1   1   2   2  NA
 [ reached getOption("max.print") -- omitted 4990 rows ]

> # if we want imputed dataset: this raises an error
> imputed.data(dataset)

Error in imputed.data(dataset):  The dataset contains no imputed data.
Please impute data before learning.
See > ?impute for help.

> # with tryCatch we manage the error
> tryCatch(
+   imp.data <- imputed.data(dataset),
+   error = function(e) {
+     cat("Hey! Something went wrong. No imputed data present maybe?")
```

```
+     imp.data <<- NULL
+   }
+ )

Hey! Something went wrong. No imputed data present maybe?

> imp.data

NULL

> # test before trying
> if (has.imputed.data(dataset)) {
+   imp.data <- imputed.data(dataset)
+ } else {
+   imp.data <- NULL
+ }
> imp.data

NULL

> # now perform imputation on the dataset
> dataset <- impute(dataset)

bnstruct ::  performing imputation ...
bnstruct ::  imputation finished.

> imputed.data(dataset)

      V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15
 [1,]  2  3  3  2  1  3  1  1  2   1   1   1   1   2   2
 [2,]  2  4  2  1  1  2  1  2  2   1   2   1   2   2   1
 [3,]  2  3  1  2  1  3  1  2  2   1   2   2   2   2   2
 [4,]  2  4  1  1  1  3  1  1  2   1   3   1   1   2   1
 [5,]  2  2  1  2  2  4  1  1  1   1   3   1   1   2   2
 [6,]  2  2  1  2  1  4  1  3  2   1   3   1   3   2   2
 [7,]  2  2  1  2  2  4  1  3  2   1   3   1   1   2   2
 [8,]  2  1  1  2  3  1  1  1  2   2   2   1   4   2   2
 [9,]  2  3  2  2  1  3  1  2  2   1   2   1   2   2   2
[10,]  2  4  1  1  1  3  1  2  2   1   3   3   2   2   1
      V16 V17 V18 V19 V20
 [1,]   2   3   2   1   2
 [2,]   2   2   1   2   2
 [3,]   1   2   1   2   2
 [4,]   3   1   1   1   2
 [5,]   1   1   1   2   2
 [6,]   2   1   1   3   2
 [7,]   1   1   1   1   2
```

```
   [8,]   1   1   1   4   2
   [9,]   1   1   1   2   2
  [10,]   1   1   2   2   2
 [ reached getOption("max.print") -- omitted 4990 rows ]
```

Complete cases of the raw dataset, that is, rows that have no missing data, can be selected with the `complete` method. Please note that this method returns a new copy of the original `BNDataset` with no imputed data or bootstrap samples (if previously generated), as it is not possible to ensure consistence among data. It is possible to restrict the completeness requirement only to a subset of variables.

By default, learning operations on the raw dataset operate with available cases.

```
> complete.subset <- complete(dataset)
>
> # require completeness only on a subset of variables
> complete.subset <- complete(dataset, c(1,4,7))
```

In order to retrieve bootstrap samples, one can use the `boots` and `imp.boots` methods for samples made of raw and imputed data. The presence of raw and imputed samples can be tested using `has.boots` and `has.imputed.boots`. Trying to access a non-existent sample (e.g. imputed sample when no imputation has been performed, or sample index out of range) will raise an error. The method `num.boots` returns the number of samples.

We also provide the `boot` method to directly access a single sample.

```
> # get raw samples
> for (i in 1:num.boots(dataset))
+   print( boot(dataset, i) )
```

```
> # get imputed samples
> for (i in 1:num.boots(dataset))
+   print( boot(dataset, i, use.imputed.data = TRUE) )
```

3.5 More advanced functions

It is also possible to manage the single fields of a `BNDataset`. See `?BNDataset` for more details on the structure of the object. Please note that manually filling in a `BNDataset` may result in inconsistent instances, and therefore errors during the execution.

It is also possible to fill in an empty BNDataset using the `read.dataset` method.

4 Bayesian Networks

Bayesian Network are represented using the `BN` object. It contains information regarding the variables in the network, the directed acyclic graph (DAG) representing the structure of the network, the conditional probability tables entailed by the network, and the weighted partially DAG representing the structure as learnt using bootstrap samples.

The following code will create a `BN` object for the `Child` network, with no structure nor parameters.

```
> dataset <- child()
> net     <- BN(dataset)
```

Then we can fill in the fields of `net` by hand. See the inline help for more details.

The method of choice to create a `BN` object is, however, to create it from a `BNDataset` using the `learn.network` method.

4.1 Network learning

When constructing a network starting from a dataset, the first operation we may want to perform is to learn a network that may have generated that dataset, in particular its structure and its parameters. `bnstruct` provides the `learn.network` method for this task.

```
> dataset <- child()
> net     <- learn.network(dataset)
```

The `learn.network` method returns a new `BN` object, with a new DAG (or WPDAG, if the structure learning has been performed using bootstrap – more on this later).

Here we briefly describe the two tasks performed by the method, along with the main options.

4.1.1 Structure learning

We provide five algorithms to learn the structure of the network, that can be chosen with the `algo` parameter. The first is the Silander-Myllymäki (`sm`) exact search-and-score algorithm (see Silander and Myllymaki [9]), that performs a complete evaluation of the search space in order to discover the best network; this algorithm may take a very long time, and can be inapplicable when discovering networks with more than 25–30 nodes. Even for small networks, users are strongly encouraged to provide meaningful parameters such as the layering of the nodes, or the maximum number of parents – refer to the documentation in package manual for more details on the method parameters.

The second algorithm is the Max-Min Parent-and-Children (`mmpc`, see Tsamardinos, Brown, and Aliferis [11]), a constraint-based heuristic approach that discovers the *skeleton* of the network, that is, the set of edges connecting the variables without discovering their directionality.

Another heuristic algorithm included is a Hill Climbing method (`hc`, see Tsamardinos, Brown, and Aliferis [11]).

The fourth algorithm (and the default one) is the Max-Min Hill-Climbing heuristic (`mmhc`, Tsamardinos, Brown, and Aliferis [11]), based on the combination of the previous two algorithms, that performs a statistical sieving of the search space followed by a greedy evaluation.

The heuristic algorithms provided are considerably faster than the complete method, at the cost of a (likely) lower quality. Also note that in the case of a very dense network and lots of obsevations, the statistical evaluation of the search space may take a long time.

The last method is the Structural Expectation-Maximization (`sem`) algorithm (Friedman [3, 4]), for learning a network from a dataset with missing values. It iterates a sequence of Expectation-Maximization (in order to "fill in" the holes in the dataset) and structure learning from the guessed dataset, until convergence. The structure learning used inside SEM, due to computational reasons, is MMHC. Convergence of SEM can be controlled with the parameters `struct.threshold`, `param.threshold`, `max.sem.iterations` and `max.em.iterations`, for the structure and the parameter convergence and the maximum number of iterations of SEM and EM, respectively.

Search-and-score methods also need a scoring function to compute an estimated measure of each configuration of nodes. We provide three of the most popular scoring functions, `BDeu` (Bayesian-Dirichlet equivalent uniform, default), `AIC` (Akaike Information Criterion) and `BIC` (Bayesian Information Criterion). The scoring function can be chosen using the `scoring.func` parameter.

```
> dataset <- child()
> net.1   <- learn.network(dataset,
+                          algo = "sem",
+                          scoring.func = "AIC")
> dataset <- impute(dataset)
> net.2   <- learn.network(dataset,
+                          algo = "mmhc",
+                          scoring.func = "BDeu",
+                          use.imputed.data = TRUE)
```

It is also possible to provide an initial network as starting point for the structure search. This can be done using the `initial.network` argument, which accepts three kinds of inputs:

- a `BN` object (with a structure);
- a `matrix` containing the adjacency matrix representing the structure of a network;

- the string `random.chain` for starting from a randomly sampled chain-like network.

In order to obtain reproducible results, in case a random chain is used it is possible to provide an initial random seed.

```
> dataset <- child()
> net.1   <- learn.network(dataset,
+                          initial.network = "random.chain",
+                          seed = 12345)
> net.2   <- learn.network(dataset,
+                          algo = "sem",
+                          initial.network = net.1)
```

Prior knowledge can be given to the learning algorithm, by providing a *layering* of the variables. Variables can be grouped in (numbered) layers, and a variable can only have parents in upper (lower-numbered) layers. In order to specify this in the learning step, one supplementary argument has to be provided: `layering`, a vector containing the indices of the layers of each variable. By default, the first layer contains variables with no parents, and variables in layer j can have parents only in layers $i \leq j$. Layering is useful also in case of a Naive Bayes network, or when learning a Dynamic Bayesian Network, where each time frame can be associated to a layer. An example of learning a Naive Bayes network is given in Section 6.3.

In case of more sophisticated requirements, some other optional arguments can be provided: `max.fanin.layers` and `max.fanin` for `sm`, `layer.struct` for `mmhc` (see method documentation).

```
> layers <- c(1,2,3,3,3,3,3,3,4,4,4,4,4,4,4,5,5,5,5,5)
> net.layer <- learn.network(dataset, layering = layers)
```

The structure learning task by default computes the structure as a DAG. We can however use bootstrap samples to learn what we call a *weighted partially DAG*, in order to get a weighted confidence on the presence or absence of an edge in the structure (Friedman, Goldszmidt, and Wyner [5]). This can be done by providing the constructor or the `learn.network` method a `BNDataset` with bootstrap samples, and the additional parameter `bootstrap = TRUE`.

```
> dataset <- child()
> dataset <- bootstrap(dataset, 100, imputation = TRUE)
> net.1   <- learn.network(dataset,
+                          algo = "mmhc",
+                          scoring.func = "AIC",
+                          bootstrap = TRUE)
> # or, for learning from imputed data
> net.2   <- learn.network(dataset,
```

```
+                             algo = "mmhc",
+                             scoring.func = "AIC",
+                             bootstrap = TRUE,
+                             use.imputed.data = TRUE)
```

Structure learning can be performed also using the `learn.structure` method, which has a similar syntax, only requiring as first parameter an already initialized network for the dataset. More details can be found in the inline helper.

It is also possible to learn a WPDAG starting from a network with a DAG, using the `wpdag.from.dag` method.

4.1.2 Learning Dynamic Bayesian Networks

Dynamic Bayesian Networks (DBNs) are Bayesian Networks that represent the evolution in time of a system. The set of variables $v_1, v_2, \ldots v_N$ of a DBN is repeated over a certain number T of time slots, and variables in time slot j can have descendents in successive time slots $k > j$ but not in the previous time slots $i < j$. From a structure learning point of view we can therefore consider the task of learning a DBN as learning a layered larger BN.

In `bnstruct` we provide the wrapper method `learn.dynamic.network` for assisting the user in learning DBNs. It works similarly to `learn.network`, but an additional parameter `num.time.slots` can be provided, indicating the number of time slots observed. It is assumed that the dataset is composed by the variables $v_1^1, v_2^1, v_3^1, \ldots, v_N^1, v_1^2, v_2^2, \ldots, v_N^T$ over T time slots.

By default, the DBN is divided into T layers, each layer corresponding to a time slot, in which each variable can be parent of any other variable in the same layer and in the following ones, while it is forbidden for any variable to be parent of any other variable in the previous layers. It is possible to provide a layering for the whole DBN using the `layering` parameter as explained in the structure learning section, assigning each one of the $N \times T$ variables to a layer.

It is also possible to exploit knowledge about the layering in each time slot by specifying the layering for just N variables; such layering will be unfolded over the T time slots. However, in this case, any variable v_x in time slot i will still be considered a potential parent for variable v_y in time slot $j > i$, even if such relationship is forbidden in time slot i. The same applies for `layer.struct`.

```
> # Assume a system with 8 nodes is observed over 4 consecutive
> # time slots, of which we know the layering in each time slot.
> # The dataset contains therefore 32 variables observed.
> # We provide a layering for the single time slot, that will be
> # expanded throughout the whole DBN.
> layers <- c(1,2,2,3,3,3,3,3)
> dbn <- learn.dynamic.network(dataset,
+                              num.time.slots = 4,
+                              layering = layers)
```

4.1.3 Empirical evaluation of structure learning algorithms

In this Section we show the results obtained by MMHC on available cases, MMHC on imputed data and SEM, with the BDeu and BIC scoring functions. Our testbed is composed by three networks available in the literature, Child (20 nodes), Alarm (37 nodes) and Hepar2 (70 nodes). For each network we consider 20 datasets of 1000 observations each, with 20% missingness. The results for the three networks are reported, respectively, in Figures 1, 2 and 3. The datasets have been generated using our package. All the numerical parameters are set at their default value, and no prior knowledge of the network is assumed (e.g. layering).

Each plot shows the distributions of both the Structural Hamming Distance (SHD, the distance in terms of edges, on the vertical axis) from the original network, and the time needed to obtain to learn the network (in seconds, on the horizontal axis).

For all the three networks the overall best results are obtained using MMHC with BDeu after dataset imputation. MMHC converges in few seconds even for the large Hepar2 network; conversely, SEM takes from few minutes up to few hours, often with worse results. SEM paired with BIC failed to converge in 24 hours for all the 20 datasets.

4.1.4 Parameter learning

Parameter learning is the operation that learns the conditional probabilities entailed by a network, given the data and the structure of the network. In `bnstruct` this is done by `learn.network` performing a Maximum-A-Posteriori (MAP) estimate of the parameters. It is possible to choose if using the raw or the impute dataset (`use.imputed.data` parameter), and to configure the Equivalent Sample Size (`ess` parameter).

In case of using bootstrap samples, `learn.network` will not perform parameter learning.

`bnstruct` also provides the `learn.params` method for this task alone.

The package also provides a method for learning the parameters from a dataset with missing values using the Expectation-Maximization algorithm. Instructions to do so are provided in section 5.1.

5 Using a network

Once a network is created, it can be used. Here we briefly mention some of the basic methods provided in order to manipulate a network and access its components.

First of all, it is surely of interest to obtain the structure of a network. The `bnstruct` package provides the `dag()` and `wpdag()` methods in order to access the structure of a network learnt without and with bootstrap (respectively).

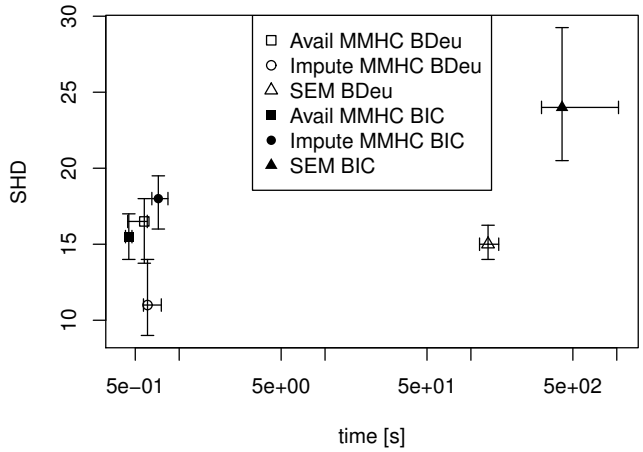


Figure 1: SHD vs. running time of i) available case analysis with MMHC, ii) kNN imputation followed by MMHC and iii) SEM, with both the BDeu and the BIC scoring functions, on 20 datasets with 1000 observations and 20% missingness sampled from the 20-nodes Child network (25 edges in the original network).

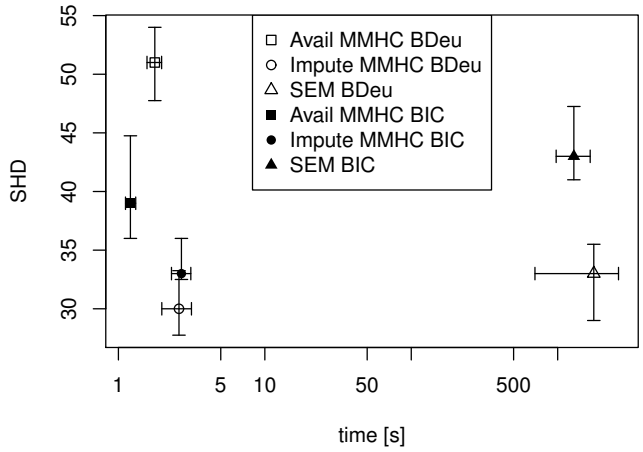


Figure 2: SHD vs. running time of i) available case analysis with MMHC, ii) kNN imputation followed by MMHC and iii) SEM, with both the BDeu and the BIC scoring functions, on 20 datasets with 1000 observations and 20% missingness sampled from the 37-nodes Alarm network (46 edges in the original network).

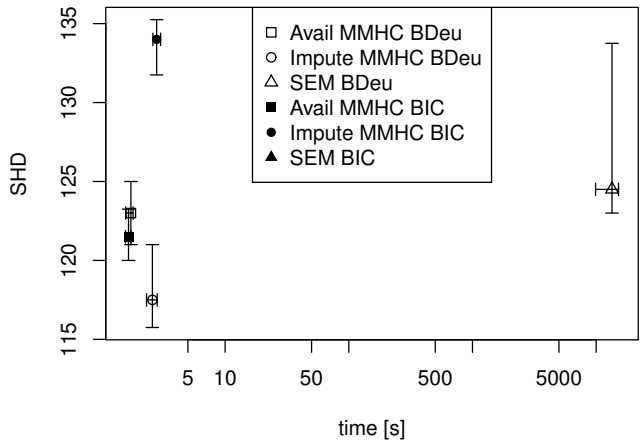


Figure 3: SHD vs. running time of i) available case analysis with MMHC, ii) kNN imputation followed by MMHC and iii) SEM, with both the BDeu and the BIC scoring functions, on 20 datasets with 1000 observations and 20% missingness sampled from the 70-nodes Hepar2 network (123 edges in the original network).

```
> dag(net)
> wpdag(net)
```

Then we may want to retrieve the parameters, using the `cpts()` method.

```
> cpts(net)
```

Another common operation that we may want to perform is displaying the network, or printing its main informations, using the `plot()`, `print()` and `show()` methods. Note that the `plot()` method is flexible enough to allow some custom settings such as the choice of the colors of the nodes, and, more importantly, some threshold settings for the networks learnt with bootstrap. As default, the DAG of a network is selected for plotting, if available, otherwise the WPDAG is used. In case of presence of both the DAG and the WPDAG, in order to specify the latter as structure to be plotted, the `plot.wpdag` logical parameter is provided. As usual, more details are available in the inline documentation of the method.

```
> plot(net) # regular DAG
> plot(net, plot.wpdag=T) # wpdag
```

As it is for `BNDatasets`, we have several equivalent options to print a network.

```
> # TFAE
> print(net)
> show(net)
> net
```

For large biological networks it might be convenient to use some external specific tool for visualization. We provide the `write.xgmml` method to export a network in the `XGMML` format, that can be then used for example with Cytoscape.

```
> write.xgmml(net)
```

5.1 Inference

Inference is performed in `bnstruct` using an `InferenceEngine` object. An `InferenceEngine` is created directly from a network.

```
> dataset <- child()
> net     <- learn.network(dataset)
> engine  <- InferenceEngine(net)
```

Optionally, a list of observations can be provided to the `InferenceEngine`, at its creation or later on. The list of observations is a list of two vector, one for the observed variables (variable indices or names can be provided, not necessarily in order - better is to list them in order of observation), and one for the observed values for the corresponding variables. In case of multiple observations of the same variable, the last one (the most recent one) is considered.

```
> dataset <- child()
> net     <- learn.network(dataset)
>
> # suppose we have observed variable 1 taking value 2
> # and variable 4 taking value 1:
> obs <- list("observed.vars" = c(1,4),
+             "observed.vals" = c(2,1))
>
> # the following are equivalent:
> engine  <- InferenceEngine(net, obs)
>
> # and
> engine  <- InferenceEngine(net)
> observations(engine) <- obs
```

The `InferenceEngine` class provides methods for belief propagation, that is, updating the conditional probabilities according to observed values, and for the Expectation-Maximization (EM) algorithm ([1]), which learns the parameters of a network from a dataset with missing values trying at the same time to guess the missing values.

Belief propagation can be done using the `belief.propagation` method. It takes an `InferenceEngine` and an optional list of observations. If no observations are provided, the engine will use the ones it already contains. The `belief.propagation` method returns an `InferenceEngine` with an `updated.bn` updated network.

```
> obs <- list("observed.vars" = c(1,4),
+             "observed.vals" = c(2,1))
> engine  <- InferenceEngine(net)
> engine  <- belief.propagation(engine, obs)
> new.net <- updated.bn(engine)
```

The EM algorithm is instead performed by the `em` method. Its arguments are an `InferenceEngine` and a `BNDataset` (optionally: a convergence `threshold`, the Equivalent Sample Size `ess` and the maximum number of iterations `max.em.iterations`), and it returns a list consisting in an updated `InferenceEngine` and an updated `BNDataset`.

```
> dataset <- child()
> net     <- learn.network(dataset)
> engine  <- InferenceEngine(net)
> results <- em(engine, dataset)
> updated.engine  <- results$InferenceEngine
> updated.dataset <- results$BNDataset
```

6 Three small but complete examples

Here we show two small but complete examples, in order to highlight how the package can provide significant results with few instructions.

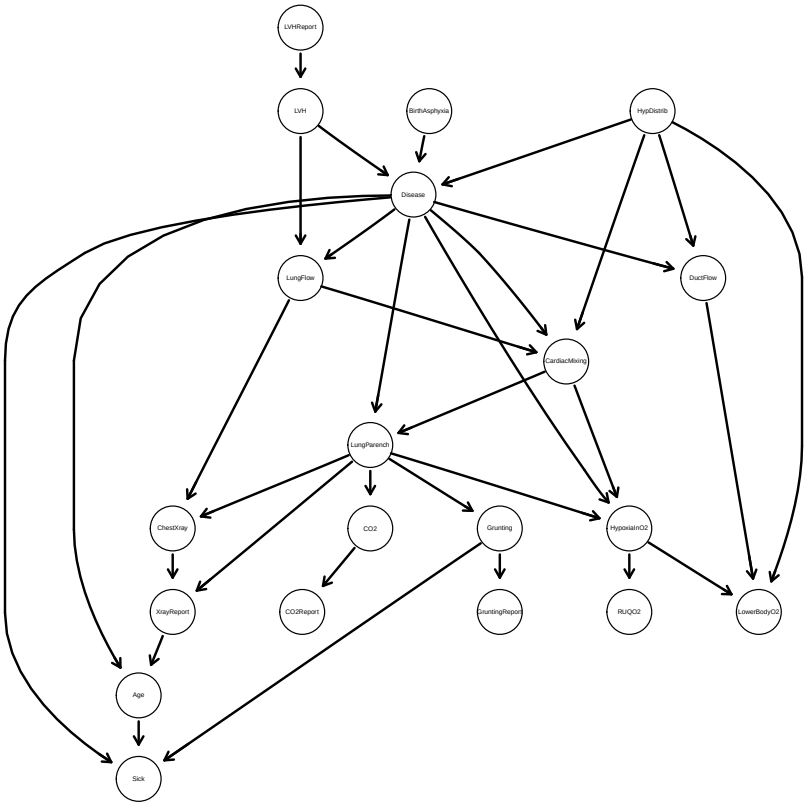
6.1 Basic Learning

First we show how some different learning setups perform on the `Child` dataset. We compare the default `mmhc-BDeu` pair on available case analysis (raw data with missing values) and on imputed data, and the `sem-BDeu` pair.

```
> dataset <- child()
>
> # learning with available cases analysis, MMHC, BDeu
> net <- learn.network(dataset)
> plot(net)
>
> # learning with imputed data, MMHC, BDeu
> imp.dataset <- impute(dataset)
> net <- learn.network(imp.dataset, use.imputed.data = TRUE)
> plot(net)
```

```
> # SEM, BDeu using previous network as starting point
> net <- learn.network(dataset, algo = "sem",
+                      scoring.func = "BDeu",
+                      initial.network = net,
+                      struct.threshold = 10,
+                      param.threshold = 0.001)
> plot(net)
```

```
> # we update the probabilities with EM from the raw dataset,
> # starting from the first network
> engine  <- InferenceEngine(net)
> results <- em(engine, dataset)
> updated.engine  <- results$InferenceEngine
> updated.dataset <- results$BNDataset
```



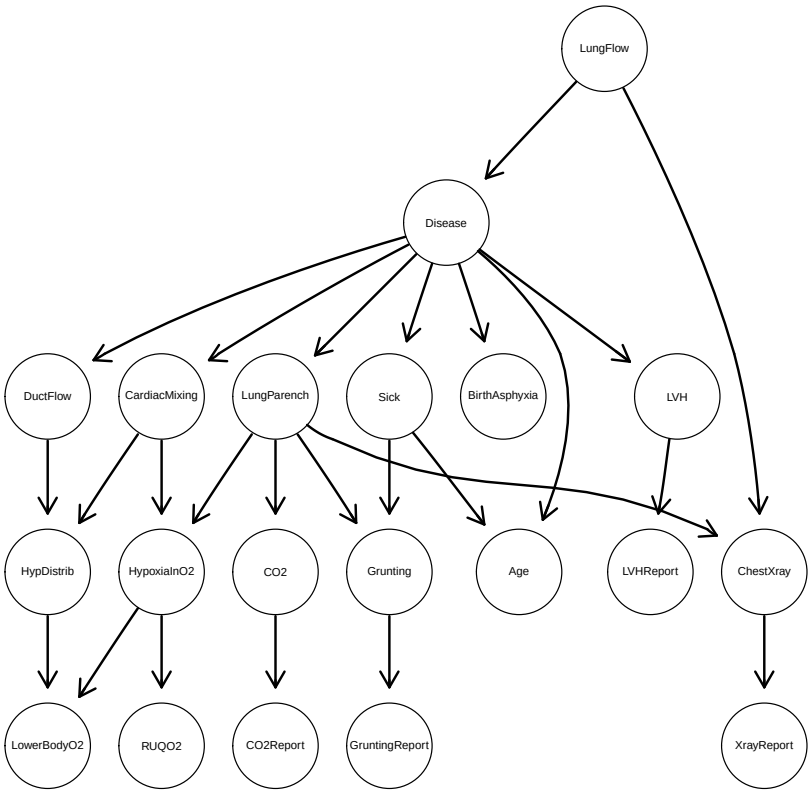
6.2 Learning with bootstrap

The second example is about learning with bootstrap. This time we use the `Asia` dataset.

```
> dataset <- asia()
> dataset <- bootstrap(dataset)
> net <- learn.network(dataset, bootstrap = TRUE)
>
> plot(net)
```

6.3 Naive Bayes

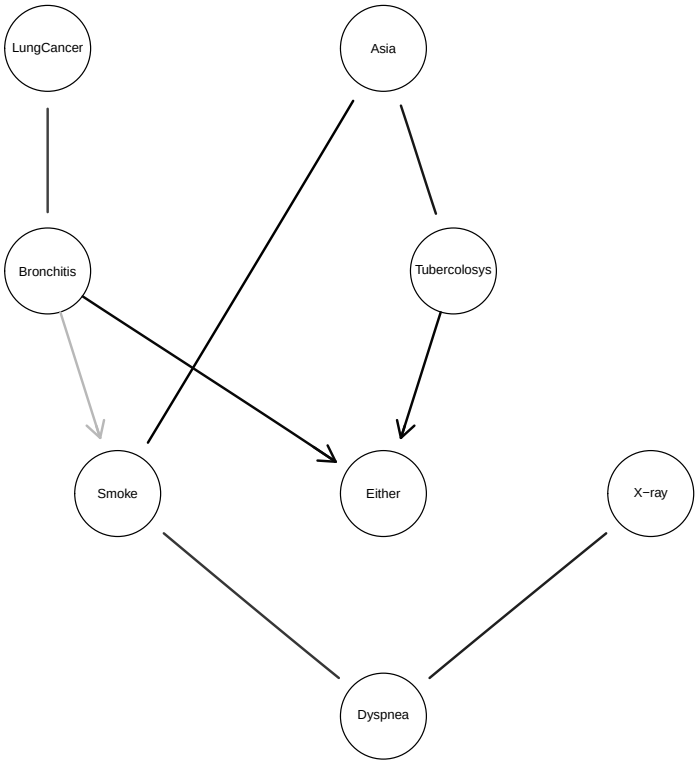
Finally, we show an example of the learning task of a Naive Bayes network. Suppose we have a mail dataset, equally divided in spam and legitimate mails. We consider four words: `buy` and `med` mainly associated to spam mails, and `bnstruct` and `learning`, observed in legitimate mails. We can divide the variables in two layers: one containing only the target variable, and the second one



with the words. We then specify the presence or absence of edges in each layer, and among the layers. Edges from lower layers to upper layers are forbidden. For this we need to define a binary squared matrix with as many rows and columns as the number of layers, so in our example we need a 2x2 matrix. Each entry $m_{i,j}$ of the matrix contains 0 if no edges can go from variables in layer i to variables in layer j, and 1 if the presence of such edges is allowed; the matrix should be upper triangular, and it will be transformed as such if it is not.

As our Naive Bayes network has edges only from the target variable to the word variables, and not between variables in layer 2, we want edges only in $m_{1,2}$, so we set that cell to 1 and all the others to 0.

```
> # artificial dataset generation
> spam <- sample(c(0,1), 1000, prob=c(0.5, 0.5), replace=T)
> buy <- sapply(spam, function(x) {
+                         if (x == 0) {
+                             sample(c(0,1),1,prob=c(0.8,0.2),replace=T)
+                         } else {
+                             sample(c(0,1),1,prob=c(0.2,0.8))}
```




```
+                       })
> med <- sapply(spam, function(x) {
+                           if (x == 0) {
+                               sample(c(0,1),1,prob=c(0.95,0.05),replace=T)
+                           } else {
+                               sample(c(0,1),1,prob=c(0.05,0.95))}
+                       })
> bns <- sapply(spam, function(x) {
+                           if (x == 0) {
+                               sample(c(0,1),1,prob=c(0.01,0.99),replace=T)
+                           } else {
+                               sample(c(0,1),1,prob=c(0.01,0.99))}
+                           })
> lea <- sapply(spam, function(x) {
+                           if (x == 0) {
+                               sample(c(0,1),1,prob=c(0.05,0.95),replace=T)
+                           } else {
```

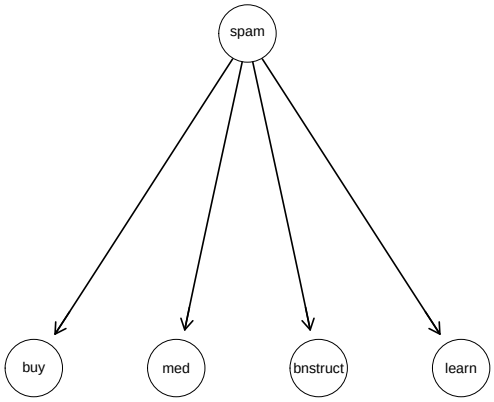


Figure 4: Naive Bayes for our spam example.

```
+                              sample(c(0,1),1,prob=c(0.95,0.05))}
+                           })
> d <- as.matrix(cbind(spam,buy,med,bns,lea))
> colnames(d) <- c("spam","buy","med","bnstruct","learn")
> library(bnstruct)
> spamdataset <- BNDataset(d, c(T,T,T,T,T),
+                           c("spam","buy","med","bnstruct","learn"),
+                           c(2,2,2,2,2), starts.from=0)
> n <- learn.network(spamdataset,
+                    algo="mmhc",
+                    layering=c(1,2,2,2,2),
+                    layer.struct=matrix(c(0,0,1,0),
+                                        c(2,2)))
> plot(n)
```

References

[1] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

[2] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[3] Nir Friedman. Learning belief networks in the presence of missing values and hidden variables. In *ICML*, volume 97, pages 125–133, 1997.

[4] Nir Friedman. The bayesian structural em algorithm. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 129–138. Morgan Kaufmann Publishers Inc., 1998.

[5] Nir Friedman, Moises Goldszmidt, and Abraham Wyner. Data analysis with bayesian networks: A bootstrap approach. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 196–205. Morgan Kaufmann Publishers Inc., 1999.

[6] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[7] Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 157–224, 1988.

[8] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[9] Tomi Silander and Petri Myllymaki. A simple approach for finding the globally optimal bayesian network structure. *arXiv preprint arXiv:1206.6875*, 2012.

[10] David J Spiegelhalter, A Philip Dawid, Steffen L Lauritzen, and Robert G Cowell. Bayesian analysis in expert systems. *Statistical science*, pages 219–247, 1993.

[11] Ioannis Tsamardinos, Laura E Brown, and Constantin F Aliferis. The max-min hill-climbing bayesian network structure learning algorithm. *Machine learning*, 65(1):31–78, 2006.