# R package for Bayesian Network Structure Learning

Francesco Sambo, Alberto Franzin

September 22, 2014

## 1 Introduction

Bayesian Networks 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, also in presence of missing data, and a set of additional tools to use Bayesian Networks, such as methods to perform belief propagation.

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.

## 2 Installation

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

In order to install the package, it suffices to launch
`R CMD INSTALL path/to/bnstruct`
from a terminal, or to use R command `install_packages`.

Being hosted on GitHub, it is also possible to use Hadley Wickham's `install_github` tool from an R session:

```
> library("devtools")
> install_github("sambofra/bnstruct")
```

`bnstruct` requires R $\geq$ 2.10, and depends on `bitops`, `igraph`, `Matrix` and `methods`. Package `Rgraphviz` is requested in order to plot graphs, but it 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.

```
> dataset <- BNDataset(name="Example")
> # creates an enpty BNDataset object
```

## 3.1 Data format

`bnstruct` requires the data files to be in a format we describe in this section. 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 `0` 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`).

We provide two sample datasets, one with complete data (the `Asia` network) and one with missing values (the `Child` network), in the `extdata` subfolder; the user can refer to them as an example.

## 3.2 Importing a dataset

The preferred way to create a *BNDataset* object is by reading a dataset from a file. In order to accomplish this, we provide the `read.dataset` method.

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

The sample datasets we provide come with two custom loaders:

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

### 3.3 Creating a `BNDataset` from data

Another possible way for creating a `BNDataset` is to create it from a *data.frame* or a *matrix* and some metadata. This is useful, for example, for instantiating a dataset after the data has already been processed in some way.

In particular, it is requested to provide the data, in `data.frame` or `matrix` form, and three additional vectors of informations on the domain: one containing the names of the variables, another one containing values indicating the cardinality (for discrete variables) or the quantization domain (for continuous variables) of the variables, and the last one containing the status of the variables (`c` for continuous, `d` for discrete). Please note that all of the metadata are required when choosing this option, and it is also suggested that the slot names are specified when passing data and metadata as parameters to the constructor; when no dataset name is provided, the slot names are mandatory.

```
> data <- matrix(c(1:16), nrow = 4, ncol = 4)
> dataset <- BNDataset(name = "MyData", data = data,
+                      variables = c("a", "b", "c", "d"),
+                      node.sizes = c(4,8,12,16),
+                      discreteness = rep('d',4))
```

### 3.4 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(name="Example")
> dataset <- read.dataset(dataset,
+                         header.file = "path/to/file.header",
+                         data.file   = "path/to/file.data")
> 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. Imputation can also be performed during the loading of a dataset, as shown in the following example.

```
> dataset <- BNDataset(name="Example")
> dataset <- read.dataset(dataset,
+                         header.file = "path/to/file.header",
+                         data.file   = "path/to/file.data",
+                         imputation  = TRUE,
+                         k.impute    = 10)
```

Note that, when imputed data is present, it has higher priority over raw data when using a dataset (see section 3.6).

The sample dataset available using the `child()` method contains both raw and imputed data.

## 3.5 Bootstrap

`BNDataset` objects have also room for bootstrap samples, i.e. random samples with replacement of the original data with the same number of observations, both for raw and imputed data. We provide the `bootstrap` method for this.

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

Again, the generation of bootstrap samples can be performed while loading a dataset.

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

The sample datasets provided have no bootstrap samples in them.

## 3.6 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, but we just show how to retrieve data.

The main operation that can be done with a `BNDataset` is to get the data it contains. The main methods we provide are `get.raw.data`, `get.imputed.data` and `get.data`. `get.data` is just a proxy for one of the other two methods. As previously mentioned, imputed data (if present) has higher priority over raw data, since it is supposed to be more useful. Therefore, if imputed data is present, `get.data` will behave as `get.imputed.data`; otherwise, it will return the raw dataset just like `get.raw.data`.

```
> dataset.1 <- child()
> # if we want raw data
> get.raw.data(dataset.1)
> # if we want imputed dataset, the following are equivalent
```

```
> get.imputed.data(dataset.1)
> get.data(dataset.1)

> dataset.2 <- asia()
> # we can only get raw data, the following are equivalent
> get.raw.data(dataset.2)
> get.data(dataset.2)
```

We can check if a dataset has imputed data or not with the `has.imputed.data` method.

```
> dataset.1 <- child()
> has.imputed.data(dataset.1) # TRUE

> dataset.2 <- asia()
> has.imputed.data(dataset.2) # FALSE
```

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 imputed samples can be tested using `has.imp.boots`. We also provide the `get.boot` method to directly access a single sample. Again, imputed samples have higher priority.

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

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

# 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.

```
> net <- BN(name = "Example")
```

The method of choice to create a `BN` object is to create it from a `BNDataset`. The following code will create an empty `BN` object for the `Child` network.

```
> dataset <- child() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
```

Now, starting from the empty network and the dataset, we can proceed with the tasks of structure and parameter learning.

## 4.1 Structure learning

When constructing a network starting from a dataset, the first operation we may want to perform is to learn the structure of the network. `bnstruct` provides the `learn.structure` method for this task.

```
> dataset <- child() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net     <- learn.structure(net, dataset)
```

The `learn.structure` 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).

We provide two algorithms in order 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 [2]), 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 (and the default one) is the Max-Min Hill-Climbing heuristic (`mmhc`, see Tsamardinos, Brown, and Aliferis [3]), that performs a statistical sieving of the search space followed by a greedy evaluation. It is 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. Also for this algorithm there are parameters that may need to be tuned, mainly the confidence threshold of the statistical pruning.

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() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net.1   <- learn.structure(net, dataset,
+                            algo = "sm", scoring.func = "AIC")
> net.2   <- learn.structure(net, dataset,
+                            algo = "mmhc", scoring.func = "BDeu")
```

The `learn.structure` method by default computes the structure as a DAG. We can however use bootstrap samples to learn 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 [1]). This can be done by

providing the constructor or the `learn.structure` method a `BNDataset` with bootstrap samples, and the additional parameter `bootstrap = TRUE`.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net.1   <- learn.structure(net, dataset, algo = "mmhc",
+               scoring.func = "AIC", bootstrap = TRUE)
> # or, for explicitely learning from raw data
> net.2   <- learn.structure(net, dataset, algo = "mmhc",
+               scoring.func = "AIC", bootstrap = TRUE,
+               imputation = FALSE)
```

### 4.2 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. `bnstruct` provides the `learn.params` method for this task, performing a Maximum-A-Posteriori (MAP) estimate of the parameters.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net     <- learn.structure(net, dataset,
+                            algo = "mmhc",
+                            scoring.func = "AIC")
> net     <- learn.params(net, dataset)
```

### 4.3 Reading a network from a file

Other than starting from a dataset, it is also possible to read a network from a file, formatted in a specific way in order to describe the structure and the parameters of the network. Several of such formats exist; we provide methods for two of the most widely adopted ones, the `dsc` and the `bif` format. The method names are, respectively, `read.dsc` and `read.bif`. These methods return a full `BN` object.

```
> net <- read.dsc("path/to/network.dsc")
> plot(net)
```

## 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).

```
> dag(net)
> wpdag(net.boot)
```

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.

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

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.

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

# 6 Inference in networks

## 6.1 Belief Propagation

The `bnstruct` package provides a tool to perform belief propagation using a junction tree. This tool is the `InferenceEngine` object. It contains a copy of a network, an updated network, the adjacency matrix of the junction tree computed starting from the original network, the list of cliques of variables that form the nodes of the junction tree and the list of joint probability tables for the cliques composing the junction tree.

An `InferenceEngine` can be built from a network: in this case, the junction tree is immediately constructed.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> inf.eng <- InferenceEngine(net)
```

Belief propagation over the junction tree can be then performed using the `belief.propagation` method.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> inf.eng <- InferenceEngine(net)
> inf.eng <- belief.propagation(inf.eng)
```

Belief propagation can be fed with a list of observations. This can be done in two ways: as parameters in the method, or inserting them directly into the inference engine. Note that the two options are mutually exclusive, in the sense that the list of observations given as parameter replaces (in the whole `InferenceEngine` object returned) the observations contained in the engine. Furthermore, if a list of observations contains multiple observations of the same variable, only the last one is considered.

```
> dataset <- asia() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net     <- learn.structure(net, dataset)
> net     <- learn.params(net, dataset)
> inf.eng <- InferenceEngine(net)
> inf.eng <- belief.propagation(inf.eng,
+                               observed.vars = c("Asia","X-ray"),
+                               observed.vals = c(1,1))
> print(updated.bn(inf.eng))
> # is equivalent to
> observations(inf.eng) <- list(c("Asia","X-ray"), c(1,1))
> inf.eng <- belief.propagation(inf.eng)
> plot(updated.bn(inf.eng))
```

### 6.2 The Expectation-Maximization algorithm

`bnstruct` can also use an `InferenceEngine` and a `BNDataset` to perform the Expectation-Maximization algorithm to estimate the parameters of the network. It suffices to use the `em` method, that returns an `InferenceEngine` containing an updated network with the newly estimated conditional probability tables.

```
> dataset <- child() # or any other way to create a custom BNDataset
> net     <- BN(dataset)
> net     <- learn.structure(net, dataset)
> net     <- learn.params(net, dataset)
> inf.eng <- InferenceEngine(net)
> inf.eng <- em(inf.eng, dataset)
> plot(updated.bn(inf.eng))
```

## 7 Other utilities

### 7.1 Sample data

We provide also methods for generating a sample of data, or a complete dataset.

The two methods for this purpose are the `sample.row` and `sample.dataset`, that generate, respectively, a vector of values and a `BNDataset` object. Both the methods accept as first argument a `BN` or an `InferenceEngine` object. To generate a `BNDataset` with `sample.dataset` one should also provide the number of observations to sample, via the `n` parameter.

```
> net <- BN(...)
> eng <- InferenceEngine(net)
> sample.row(net)
> sample.row(eng)
> sample.dataset(net,1000)
> sample.dataset(eng,10000)
```

# References


[1] 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.

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

[3] 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.