Quick start guide for PY_BANSHEE

Paul Koot, Miguel Angel Mendoza-Lugo, Dominik Paprotny, Daniël T. H. Worm, Elisa Ragno, Oswaldo Morales-Nápoles,

Contents

Preface

This short guide was written to enable users an easier start with the PY_BANSHEE package. It has the same structure as the quick guide for BANSHEE, the MATLAB equivalent toolbox. It is not a step-by-step guide, but rather clarifies the main options for users to apply the package in their Python scripts. We recommend to read it in combination with our paper "BANSHEE–A MATLAB Toolbox for Non-Parametric Bayesian Networks" and the update paper "Update (1.2) to BANSHEE-A MATLAB toolbox for Non-Parametric Bayesian Networks and Python-based version (1.0) PY_BANSHEE".

Requirements

A Python environment in which the package can be installed using pip. The package was tested with a Python 3.9.7 virtual environment built with Anaconda, but should work with some older versions as well.

Installation

In the command prompt, activate the virtual environment (create a new one if necessary) in which the PY_BANSHEE package will be installed. The package can also be installed in the base environment. However this is discouraged particularly for Anaconda, as a 'pip install' might conflict with other packages present that were already installed using 'conda install'. Use pip install py-banshee to install PY_BANSHEE and its dependencies.

The package can be updated by running the same command. If working in a virtual environment, please make sure the package graphviz is also installed there. Graphviz is the python package that is used to visualize the NPBN (function bn_visualize). If you want to use this function, besides the python package graphviz also the Graphviz software should be installed.

For Windows:

1. Install windows package from: https://graphviz.org/download/ (Linux and Mac instructions can be found here as well)
2. Install python graphviz package
3. Add 'bin' folder to User path in environment variables manager (e.g: C:\Program Files (x86)\Graphviz2.38\bin)
4. Add location dot.exe to System Path (e.g: C:\Program Files (x86)\Graphviz2.38\bin\dot.exe)

Overview of the package

BANSHEE has two components:

- Six function scripts for implementing and analyzing non-parametric Bayesian Networks (NPBN);
- Three scripts containing example applications of the NPBN functions, intended for learning how to use the package and NPBN method in general; example_3.py is the Python equivalent of example.m in the MATLAB Toolbox BANSHEE.

Each function has an extensive description in its header, which is also accessible through Python's `help` function, e.g.:

```
help(inference) # access help for function inference
```

The ensuing help text presents all arguments and options of each PY_BANSHEE function and provides background information.

Using the package

When using the package for the first time, it is recommended to run `example_3.py`, a script that highlights the use of all five function scripts. This section contains information on how to use each function illustrated by each step of the `example_3.py` script.

Creating DAGs of new NPBN models

NPBN models are largely expert knowledge-driven models. They are intended to recreate real-life interactions in a probabilistic framework. The structure could still be learned from the data using an iterative approach. In the first step, a simple rank correlation matrix can help to identify the first arc between a variable of interest and an explanatory variable (according to the user's designation of those). It can be generated with a function from the pingouin package `pairwise_corr` for a pandas DataFrame `data`:

```
RHO=pg.pairwise_corr(data,method='spearman')
```

Further arcs are added by the user (using expert knowledge or the unconditional correlations) and then running the `bn_rankcorr` function from PY_BANSHEE. Plotting the BN rank correlation matrix and visualizing the directed acyclic graph (DAG) with the (conditional) correlations with `bn_visualize` allows identifying arcs for which the correlation is very low. These arcs could be removed and the correlation matrix recalculated to obtain new (conditional) correlations in the BN. The iterative process can be combined with the diagnostic tools of the toolbox (primarily `gaussian_distance`) and validation of the BN's predictions obtained through `inference` function, as shown in the `example_3.py` script. The most important feature of the DAG that has to be maintained is the lack of circular connections between variables (it has to be acyclic).

Example

The `example_3.py` script employs a dataset `cities.csv`, which contains data on nine quality-of-life indicators in 329 cities in the United States. This file can be downloaded from the Github repository (https://github.com/mike-mendoza/py_banshee). In the first step, data are loaded: the DataFrame `data` contains the numerical data, with nine variables in columns and a row of data per city. The column headers contain the variable names, which is useful to have for visualizing the NPBN and other analyses (default names are assigned if names are not specified). The purpose of the model is to use different indicators of quality of life and predict the level of personal safety in American cities.

The data are prepared as follows:

```
# Set random seed to obtain the same inference results every run, for
# research purposes (used by sampling in function inference from prediction.py)
np.random.seed(123)

# Define location of data file
data = pd.read_csv('cities.csv')

# Define name of output figure with BN; used by bn_visualize
fig_name = 'bn_cities'

# Select the columns to use in the NPBN
columns_used=[0, 6, 7, 8, 3]      # climate, arts, recreation, economics,
                                  # safety
data = data.iloc[:,columns_used]

# Extract the variable names
names = list(data.columns)

# Extract number of nodes from data
N = data.shape[1]
```

After the data are ready, the structure of the NPBN is defined (the DAG). The DAG is constructed through expert knowledge supported by BN rank correlation matrix and the diagnostic tools. In the code, it is defined as a list `parent_cell`. Each cell of the array refers to one node (variable) of the DAG and lists all the parent nodes (empty if there are no parents). In our example, the DAG (Fig. 3) is as follows:

```
# Defining the structure of the BN
parent_cell = [None]*N
parent_cell[0] = []               # climate (no parents)
parent_cell[1] = [2]              # arts (parent node: recreation)
parent_cell[2] = [3, 0]           # recreation (parent nodes: economics, climate)
parent_cell[3] = []               # economics (no parents)
parent_cell[4] = [1, 2, 3, 0]     # safety (parents: all other variables)
```

BN Rank Correlation Matrix (bn_rankcorr)

Purpose

`R = bn_rankcorr(parent_cell, data, var_names=[], is_data=True, plot=False)` computes the BN rank correlation matrix, which quantifies the strength of the (conditional) dependency between variables.

Input

`parent_cell` A list containing the structure of the BN (directed acyclic graph – DAG). Each list element is a node of the BN and contains a list of the node's parents, defined as a list with reference to particular cell(s). An example of the DAG is shown in the previous section, "Creating DAGs of new NPBN models".

data A pandas DataFrame containing data for quantifying the NPBN. Data for each node need to be located in columns in the same order as specified in **parent_cell**. The number of columns need to equal the number of nodes specified in **parent_cell**. Optionally, a list of rank correlations can be used, one conditional correlation per arc, following the same structure as **parent_cell**. This second option is intended for use in a User-Defined Random Model (UDRM). An example of a UDRM is shown in `example1.py` script.

is_data Specifies the input data type:

False – list of lists **data** contains rank correlations (for UDRM models);

True – pandas DataFrame **data** contains actual data.

plot (optional parameter). A plot of correlation matrix **R** can be displayed:

False - do not create a plot (default);

True - create a plot.

var_names (optional parameter). A list containing names of the nodes for the plot; should be provided if **plot==True**.

Output

R An *n*-by-*n* numpy.ndarray with the Spearman's (conditional) rank correlations, where *n* is the number of nodes, as specified in **parent_cell**.

Example

The DAG defined in `parent_cell` variable, the DataFrame `Data` and (optionally) a list of variable names `names` are inputs for the `bn_rankcorr` function. The inputs are composed as follows:

```
R = bn_rankcorr(parent_cell,     # structure of the BN
        data,                    # matrix of data
        var_names = names        # names of variables
        is_data = True,                  # matrix data contains actual data
        plot = True,                     # create a plot (False= don't create plot)
```

The value of `is_data` is `True`, as we provide a matrix of actual data to quantify the model. Further, the value of `plot` is also `True`, as we want the function to generate a plot.

Running the function returns a BN correlation matrix `R` and a plot (Fig. 1).

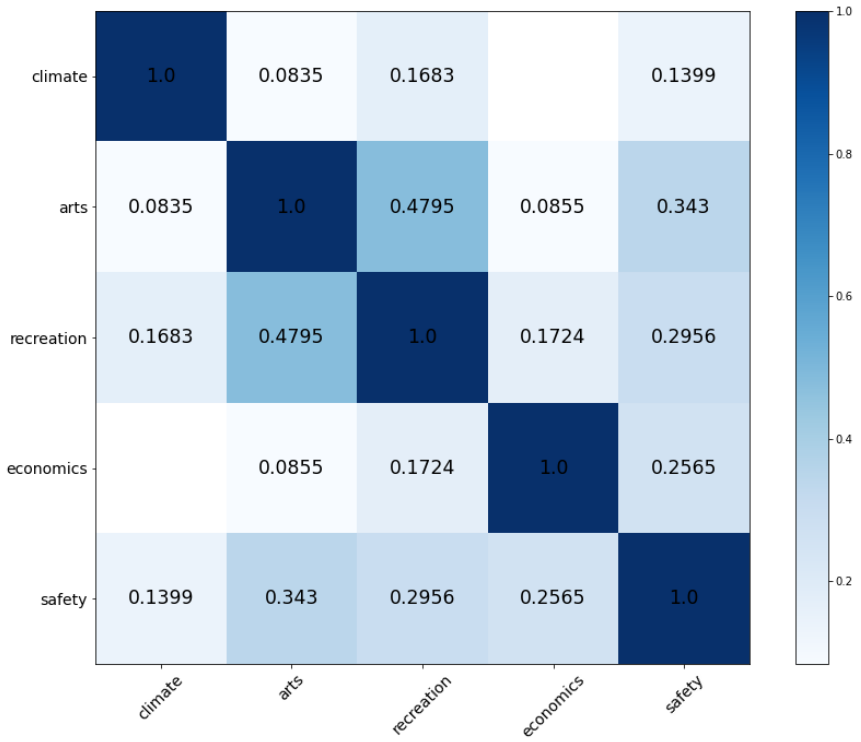

Figure 1: BN rank correlation matrix generated with bn_rankcorr after running example_3.py script.

Visualize the BN (bn_visualize)

Purpose

`bn_visualize(parent_cell, R, names, data=None, fig_name="")` creates a directed digraph presenting the structure of nodes and arcs of the Bayesian Network (BN), also displaying the (conditional) rank correlations at each arc.

Input

`parent_cell` A list containing the structure of the BN (directed acyclic graph – DAG). Each cell is a node of the BN and contains a list of the node's parents, defined as a vector with reference to particular cell(s). An example of the DAG is shown in the previous section, "Creating DAGs of new NPBN models".

`R` An *n*-by-*n* matrix with the Spearman's (conditional) rank correlations, where *n* is the number of nodes, as specified in **`parent_cell`**. This variable should be generated with **`bn_rankcorr`** function (see section "BN Rank Correlation Matrix").

`names` A list containing names of the nodes for the plot

`data` the same data that can be used as input in **`bn_rankcorr`**. When this argument is given as input, the nodes in the visualization contain the marginal distribution of the data within each node.

`fig_name` Name extension of the .png file with the NPBN that is created: BN_visualize_'fig_name'.png. The file is saved in the working directory.

Output

The function doesn't provide an output variable. Instead, it generates a plot presenting the DAG: nodes, variables and (conditional) rank correlations and, if **`data`** is specified, also the marginal distributions.

Example

The DAG from the cities example can be visualized using `bn_visualize` function. The only required arguments are `parent_cell, R` and `names` from the previous steps. Also `fig_name` is passed into the function to save the figure:

```
bn_visualize(parent_cell,        # structure of the BN
        R,                       # the rank correlation matrix (function 1)
        data.columns,            # names of variables
        fig_name = fig_name)     # figure name
```

Running the function returns a plot containing nodes (red dots with names of variables), the arcs with a defined direction, and the value of the (conditional) rank correlations on the arcs (Fig. 2).

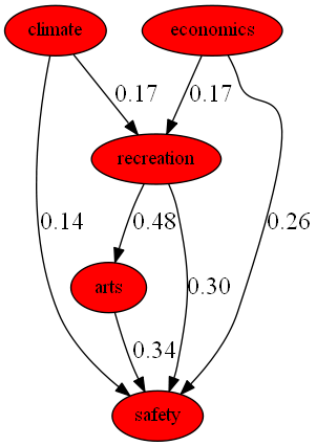

Figure 2: Quantified DAG generated with bn_visualize after running example_3.py script.

By adding and changing the slightly (so the first file is not overwritten):

```
bn_visualize(parent_cell,                        # structure of the BN
        R,                                       # the rank correlation matrix (function 1)
        data.columns,                            # names of variables
        data = data,                             # DataFrame with data
        fig_name = fig_name + '_margins')        # figure name
```

we come to a plot with marginal distributions (Fig.3).

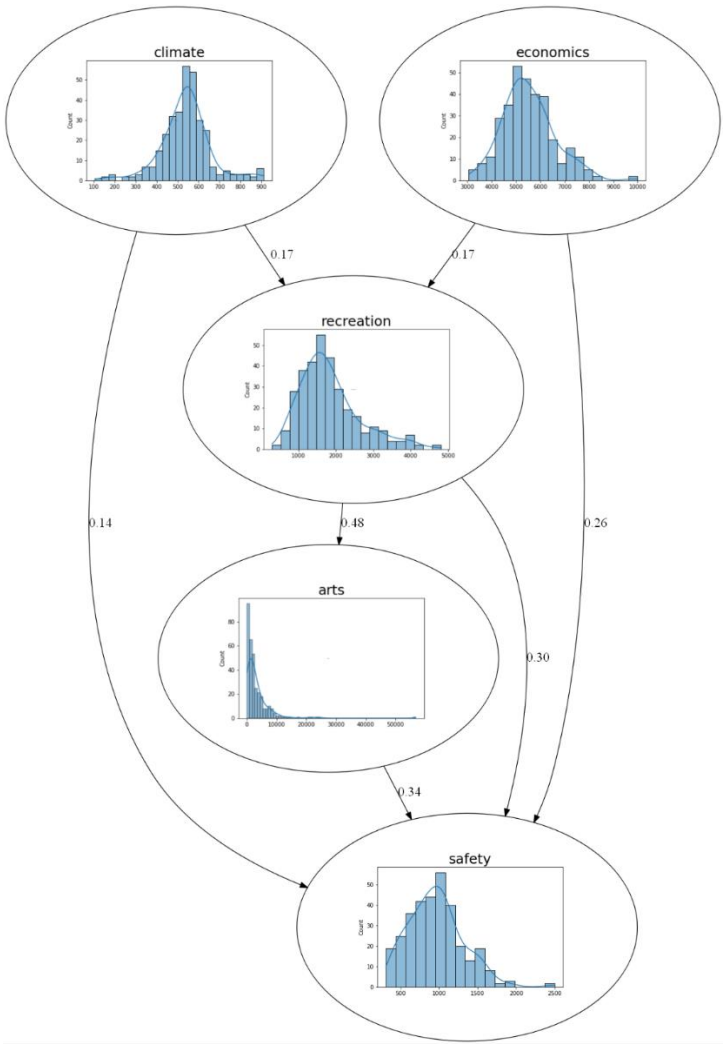

Figure 3: Quantified DAG with marginals generated with bn_visualize after running example_3.py script.

Diagnostic test 1: Goodness-of-fit test for copulas (cvm_statistic)

Purpose

`M = cvm_statistic(DATA, names, plot=False, fig_name="")` calculates the goodness-of-fit for pairs of variables, using Cramer-von Mises statistic.

Input

`DATA` A DataFrame containing data for quantifying the NPBN. Each column is one variable.

`plot` A plot of highlighting the optimal copula per pair of variables can be displayed:

False - do not create a plot;

True - create a plot.

`names` A list containing names of the nodes for the plot.

fig_name Name extension of the .png file with the statistics that are created: cvm_statistics_'fig_name'.png. The file is saved in the working directory.

Output

M A matrix containing the following columns:

[1]: first variable in the pair (column number in the matrix **DATA**);

[2]: second variable in the pair (column number in the matrix **DATA**);

[3]: Spearman's rank correlation between variables;

[4]: Cramer-von Mises statistic for Gaussian copula;

[5]: Cramer-von Mises statistic for Gumbel copula;

[6]: Cramer-von Mises statistic for Clayton copula;

[7]: Cramer-von Mises statistic for Frank copula.

The Cramer-von Mises statistic measures the sum of squared difference between the parametric and empirical copulas.

Example

The `cvm_statistic` can be used to analyse some of the underlying assumptions of the method. Firstly, the assumption that the bivariate dependencies between variables can be modelled with a Gaussian (normal) copula. The function `cvm_statistic` computes the sum of squared differences between the parametric and empirical copulas. The lower value of the resulting `M` metric, the better fit between the parametric and empirical copulas is achieved. The only required argument is the matrix of data used in the previous steps:

```
M = cvm_statistic(data,                  # DataFrame with data
                  names = data.columns,  # names of variables
                  plot = True,           # create a plot (False = don't create plot)
                  fig_name = fig_name)   # figure name
```

Running the script will result in the following graph (Fig. 4):

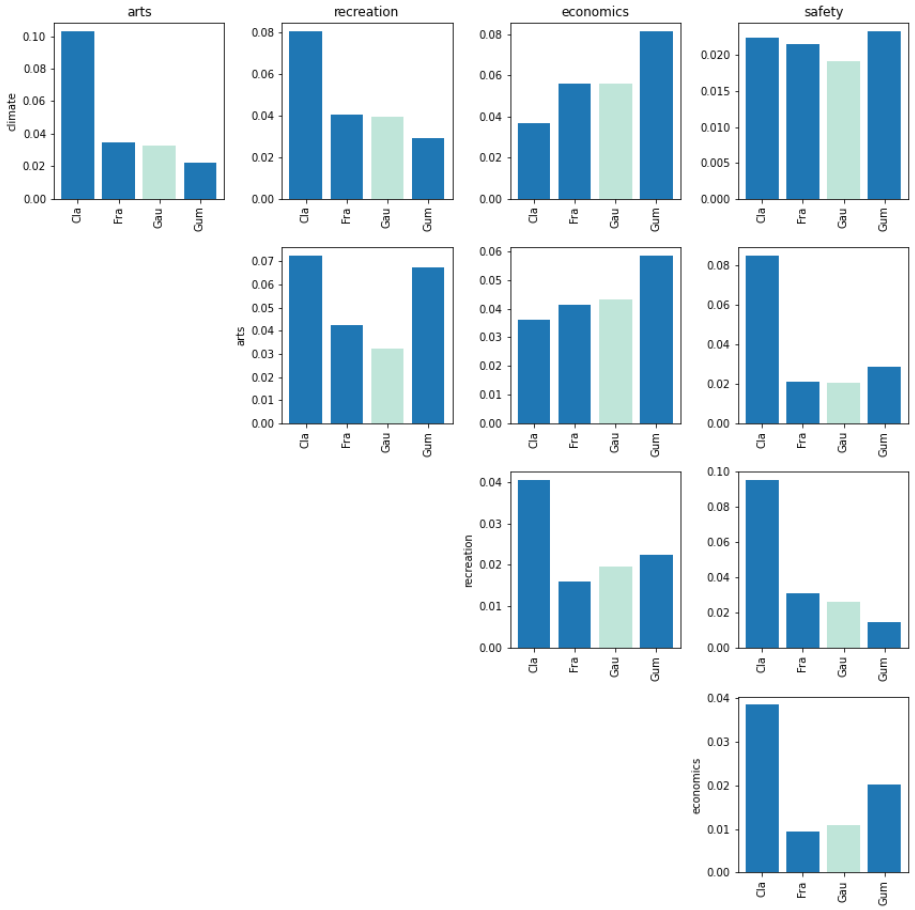

Figure 4: Diagnostic results for four copula types obtained using cvm_statistics after running example_3.py script.

The results of the diagnostic routines `M` highlight that the Gaussian copula is in the majority of cases the most suitable for representing the dependency between variables, especially for the variable of interest (safety).

Diagnostic test 2: Distance between Gaussian densities (gaussian_distance)

Purpose

`D_ERC,B_ERC,D_BNRC,B_BNRC=gaussian_distance(R,DATA,SampleSize_1=1000, SampleSize_2=1000,M=1000,Plot=False, Type="H",fig_name="")` computes the d-calibration score, which compares the distance between both the empirical and BN rank correlation matrices and the empirical normal rank correlation matrix.

Input

`R` An *n*-by-*n* numpy.ndarray with the Spearman's (conditional) rank correlations, where *n* is the number of nodes. This variable should be generated with **`bn_rankcorr`** function (see section "BN Rank Correlation Matrix").

DATA A DataFrame containing data for quantifying the NPBN. Data for each node need to be located in columns in the same order as specified in **R**. The number of columns need to be equal to the number of nodes specified in **R**.

SampleSize_1 (optional parameter). The number of samples to be drawn in the resampling of the distributions in the test d-Cal(ERC,NRC). 1000 is the default.

SampleSize_2 (optional parameter). The number of samples to be drawn in the resampling of the distributions in the test d-Cal(NRC,BNRC). 1000 is the default.

M (optional parameter). Number of iterations of calculating the d-calibration scores to compute the confidence interval of the determinant of the sampled random distribution. 1000 is the default.

Plot (optional parameter). A plot of the d-calibration scores can be displayed:

False - do not create a plot (default);

True - create a plot.

Type (optional parameter). A string that sets the type of measure used to calculate the distance. Available methods are:

'H' Hellinger distance (default);

'KL' Symmetric Kullback–Leibler divergence;

'B' Bhattacharyya distance;

'G' G distance from Abou Moustafa et al. (2010)[1].

fig_name Name extension of the .png file with the d-calibration scores that is created: gaussian_distance_'fig_name'.png. The file is saved in the working directory.

Output

D_ERC Value of the d-calibration score for the empirical rank correlation matrix of **DATA**

D_BNRC Value of the d-calibration score for the Bayesian Network rank correlation matrix **R**.

B_ERC Quantile range (5^{th} and 95^{th} percentile) of the distribution of the determinant of the empirical distribution of **DATA** transformed to standard normal.

B_BNRC Quantile range (5^{th} and 95^{th} percentile) of the distribution of the determinant of the empirical distribution of the Bayesian Network.

[1] Abou Moustafa, K.T., De La Torre, F., and Ferrie, F. P. (2010). Designing a Metric for the Difference between Gaussian Densities. In: Angeles et al., "Brain, Body and Machine. Advances in Intelligent and Soft Computing." Berlin: Springer, 57-70.

The score is 1 if the matrices are equal and 0 if one matrix contains a pair of variables perfectly correlated, and the other one does not, and the score will be "small" as the matrices differ from each other elementwise.

Example

The second diagnostic test is done with the function `gaussian_distance`, which requires two arguments from the cities example, namely the rank correlation matrix `R` and the dataset `data`.

```
D_ERC,B_ERC,D_BNRC,B_BNRC = gaussian_distance(
                                    R,      # the rank correlation matrix
                                    data,   # DataFrame with data
                                    4000,   # number of samples drawn d-Cal(ERC,NRC)
                                    400,    # number of samples drawn d-Cal(NRC,BNRC)
                                    1000,   # number of iterations to compute CI
                                    Plot = True,      # create a plot (0=don't create plot)
                                    Type = 'H',       # take Hellinger distance (default)
                                    fig_name=fig_name) # figure name
```

Three additional arguments that can be specified. In this example,

```
SampleSize_1 = 4000, SampleSize_2 = 400, M = 1000
```

indicates that the function will draw 4000 samples of the normal distribution, and then perform 1000 iterations to obtain the distribution of the d-calibration score. This option was added as the test is sensitive to the number of samples drawn as well as the number of iterations and is rather severe for large datasets. A plot can be generated (Fig. 5) and finally the distance metric can be specified out of four metrics implemented in the code. By default, Hellinger distance is used (`'H'`).

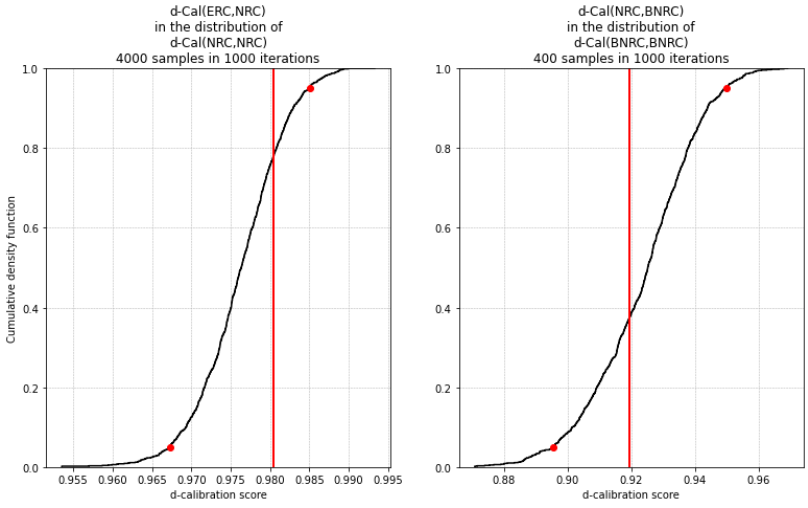

Figure 5: The d-calibration score of the BN model obtained using gaussian_distance after running example_3.py script.

The plot shows that the d-calibration score of the empirical rank correlation matrix (`D_ERC`) is inside the 90% confidence interval of the determinant of the empirical normal distribution (`B_ERC`) (Fig. 5 left). The d-calibration score of the BN's rank correlation matrix (`D_BNRC`) is well within the 90% confidence interval of the determinant of the random normal distribution

sampled for the same correlation matrix (`B_BNRC`; Fig. 5 right). This means that while the joint normal copula is not a good assumption for the whole dataset, it is valid for the particular configuration of the BN model.

Inference of the BN (inference)

Purpose

`F = inference(Nodes,Values,R,DATA,OUTPUT="full",SampleSize=1000, Interp="next", empirical_data=True,distributions=[],parameters=[])` makes inference of the BN model, i.e. conditionalizes nodes of the model in order to obtain conditional distributions, which can be used as predictions for the values of the other nodes without observed values.

Input

`Nodes` A vector defining nodes to be conditionalized. The values of the vector define the variables in the same order as in matrix R. At least one node has to be left out from **`Nodes`** in order to make inferences at least for this one node.

`Values` A DataFrame containing data on which the inference will be based upon. Data for each node need to be located in columns in the same order as specified in **`Nodes`** and **`R`**. The number of columns need to equal the number of nodes specified in **`Nodes`** and **`R`**.

`R` An *n*-by-*n* numpy.ndarray with the Spearman's (conditional) rank correlations, where *n* is the number of nodes, as specified in **`parent_cell`**. This variable should be generated with **`bn_rankcorr`** function (see section "BN Rank Correlation Matrix").

`DATA` A matrix containing data for quantifying the NPBN. Data for each node need to be located in columns in the same order as specified in **`Nodes`**, **`Values`** and **`R`**. The number of columns need to be equal the number of nodes specified in **`Nodes`**, **`Values`** and **`R`**.

`OUTPUT` (optional parameter). A string setting the type of output of the function:

- `'full'` provides a list with the conditional empirical distributions (default).
- `'mean'` provides a matrix with the mean of the conditional empirical distributions.
- `'median'` provides a matrix with the median of the conditional empirical distributions.

`SampleSize` Number of samples drawn when conditionalizing the NPBN. 1000 is the default.

`Interp` A string with the name of the interpolation method. The options are the same as in MATLAB's `interp1` function: `'linear'`, `'nearest'`, `'nearest-up'`, `'zero'`, `'slinear'`, `'quadratic'`, `'cubic'`, `'previous'`, or `'next'` (default)

`empirical_data` True if the data are empirical observations, False if data are parametric distributions

`distributions` A list with the names of the distributions for each node

`parameters` A list with the corresponding parameters of the distributions

Output

F By default, provides a list with the conditional empirical distributions for each row in **Values** and each node not specified in **Nodes**.

Example

Once the BN model was configured and analysed, it can be used to make inference. The function `inference` requires four arguments, of which two are used in the cities example: the rank correlation matrix `R` and the dataset `Data`. The argument `Nodes` defines which nodes are to be conditionalized, according to the numbering defined in the DAG by the `parent_cell` variable. Per each node conditionalized, a numeric value has to be provided in the variable `Values` in the same order as specified in `Nodes`. For example, if `R` is a 5-by-5 matrix,

```
Nodes = [0, 2, 4]
```

will conditionalize the BN using the first, third and fifth node and make inference of the second and fourth node. Notice that in contrast to MATLAB (indexing starts at one (1)), indexing in Python starts at zero (0).

Multiple rows of data (i.e. different observation records) in `Values` are possible. The configuration of those arguments in the cities example is:

```
condition = [0, 1, 2, 3]                    # conditionalized variables, all except for safety (predict)
values = data.iloc[:,condition].to_numpy() # data for conditionalization
```

The full `inference` function is written as follows:

```
F = inference(condition,        # nodes that will be conditionalized
        values,                 # information used to conditionalize the
                                # nodes of the NPBN
        R,                      # the rank correlation matrix
        data,                   # DataFrame with data
        Output = 'mean')        # type of output data
```

In this setting, the output variable `F` will contain only the mean value of the predictions of the fifth node (safety), computed based on 1,000 samples of the BN, with the empirical marginal distributions interpolated using the 'next' method.

Computations made with inference can be e.g. used to compare prediction of the model with observations. It should be noted that the function, for larger dataset, will display the progress of the calculation and the display `Calculation complete` once all data in `Values` have been processed.

Using real-world example models

We include one script (`example_2.py`) that reproduce the non-parametric Bayesian network and the sample based conditioning case number 6 of the article: "*Reliability analysis of reinforced concrete vehicle bridges columns using non-parametric Bayesian networks*" (https://doi.org/10.1016/j.engstruct.2019.03.011). Its corresponding datasets: (i)Concrete_vehicle_bridge_column.csv, (ii)Samp_based_Case_6.csv and (iii)UNINET_BN_Rank_corr_mat.txt are included.

If the user is interested in other examples, please have a look at the BANSHEE Quick Guide to get a feeling of the real life examples. The references to the papers of this real life examples can be found in *"BANSHEE–A MATLAB Toolbox for Non-Parametric Bayesian Networks"*. The examples presented there are used in the same way, similar to the `inference` function. The `predict_river_discharge` is highlighted in a wrapper script `example_hydro_simulation`. It computes and visualizes river discharges for an example dataset of Kingston river gauge in London, United Kingdom. All models have example datasets for inference provided.