PyBNesian

Release 0.4.3

David Atienza

Oct 23, 2022

CONTENTS:

CHAPTER

ONE

PYBNESIAN

- **PyBNesian** is a Python package that implements Bayesian networks. Currently, it is mainly dedicated to learning Bayesian networks.
- **PyBNesian** is implemented in C++, to achieve significant performance gains. It uses Apache Arrow to enable fast interoperability between Python and C++. In addition, some parts are implemented in OpenCL to achieve GPU acceleration.
- **PyBNesian** allows extending its functionality using Python code, so new research can be easily developed.

1.1 Dependencies

- Python 3.7, 3.8, 3.9 and 3.10.

The library has been tested on Ubuntu 16.04/20.04 and Windows 10, but should be compatible with other operating systems.

1.1.1 Libraries

The library depends on NumPy, Apache Arrow, and pybind11.

Building PyBNesian requires linking to Apache Arrow. Therefore, even though the library is compatible with `pyarrow>=3.0` each compiled binary is compatible with a specific `pyarrow` version. The pip repository provides compiled binaries for all the major operating systems (Linux, Windows, Mac OS X) targeting the last `pyarrow` version.

If you need a different version of `pyarrow` you will have to build PyBNesian from source. For example, if you need to use a `pyarrow==8.0` with PyBNesian, first install the required version of `pyarrow`:

```
pip install pyarrow==8.0.0
```

Then, proceed with the *Building* steps.

Note: Before building PyBNesian with a different version of `pyarrow`, you should check that your Python version is compatible with `pyarrow`.

1.2 Installation

PyBNesian can be installed with pip:

```
pip install pybnesian
```

1.3 Build from Source

1.3.1 Prerequisites

- Python 3.6, 3.7, 3.8 or 3.9.
- C++17 compatible compiler.
- CMake (it is needed to compile *NLopt* <*https://github.com/stevengj/nlopt*>).
- OpenCL 1.2 headers/library available.

If needed you can select a C++ compiler by setting the environment variable *CC*. For example, in Ubuntu, we can use Clang 11 with the following command before installing PyBNesian:

```
export CC=clang-11
```

1.3.2 Building

Clone the repository:

```
git clone https://github.com/davenza/PyBNesian.git
cd PyBNesian
git checkout v0.1.0 # You can checkout a specific version if you want
python setup.py install
```

1.4 Testing

The library contains tests that can be executed using pytest. They also require scipy and pandas installed. Install them using pip:

```
pip install pytest scipy pandas
```

Run the tests with:

```
pytest
```

1.5 Usage Example

```
>>> from pybnesian import GaussianNetwork, LinearGaussianCPD
>>> # Create a GaussianNetwork with 4 nodes and no arcs.
>>> gbn = GaussianNetwork(['a', 'b', 'c', 'd'])
>>> # Create a GaussianNetwork with 4 nodes and 3 arcs.
>>> gbn = GaussianNetwork(['a', 'b', 'c', 'd'], [('a', 'c'), ('b', 'c'), ('c', 'd')])

>>> # Return the nodes of the network.
>>> print("Nodes: " + str(gbn.nodes()))
Nodes: ['a', 'b', 'c', 'd']
>>> # Return the arcs of the network.
>>> print("Arcs: " + str(gbn.nodes()))
Arcs: ['a', 'b', 'c', 'd']
>>> # Return the parents of c.
>>> print("Parents of c: " + str(gbn.parents('c')))
Parents of c: ['b', 'a']
>>> # Return the children of c.
>>> print("Children of c: " + str(gbn.children('c')))
Children of c: ['d']

>>> # You can access to the graph of the network.
>>> graph = gbn.graph()
>>> # Return the roots of the graph.
>>> print("Roots: " + str(sorted(graph.roots())))
Roots: ['a', 'b']
>>> # Return the leaves of the graph.
>>> print("Leaves: " + str(sorted(graph.leaves())))
Leaves: ['d']
>>> # Return the topological sort.
>>> print("Topological sort: " + str(graph.topological_sort()))
Topological sort: ['a', 'b', 'c', 'd']

>>> # Add an arc.
>>> gbn.add_arc('a', 'b')
>>> # Flip (reverse) an arc.
>>> gbn.flip_arc('a', 'b')
>>> # Remove an arc.
>>> gbn.remove_arc('b', 'a')

>>> # We can also add nodes.
>>> gbn.add_node('e')
4
>>> # We can get the number of nodes
>>> assert gbn.num_nodes() == 5
>>> # ... and the number of arcs
>>> assert gbn.num_arcs() == 3
>>> # Remove a node.
>>> gbn.remove_node('b')

>>> # Each node has an unique index to identify it
>>> print("Indices: " + str(gbn.indices()))
```

(continues on next page)

(continued from previous page)

```
Indices: {'e': 4, 'c': 2, 'd': 3, 'a': 0}
>>> idx_a = gbn.index('a')

>>> # And we can get the node name from the index
>>> print("Node 2: " + str(gbn.name(2)))
Node 2: c

>>> # The model is not fitted right now.
>>> assert gbn.fitted() == False

>>> # Create a LinearGaussianCPD (variable, parents, betas, variance)
>>> d_cpd = LinearGaussianCPD("d", ["c"], [3, 1.2], 0.5)

>>> # Add the CPD to the GaussianNetwork
>>> gbn.add_cpds([d_cpd])

>>> # The CPD is still not fitted because there are 3 nodes without CPD.
>>> assert gbn.fitted() == False

>>> # Let's generate some random data to fit the model.
>>> import numpy as np
>>> np.random.seed(1)
>>> import pandas as pd
>>> DATA_SIZE = 100
>>> a_array = np.random.normal(3, np.sqrt(0.5), size=DATA_SIZE)
>>> c_array = -4.2 - 1.2*a_array + np.random.normal(0, np.sqrt(0.75), size=DATA_SIZE)
>>> d_array = 3 + 1.2 * c_array + np.random.normal(0, np.sqrt(0.5), size=DATA_SIZE)
>>> e_array = np.random.normal(0, 1, size=DATA_SIZE)
>>> df = pd.DataFrame({'a': a_array,
...                    'c': c_array,
...                    'd': d_array,
...                    'e': e_array
...                })

>>> # Fit the model. You can pass a pandas.DataFrame or a pyarrow.RecordBatch as␣
↪argument.
>>> # This fits the remaining CPDs
>>> gbn.fit(df)
>>> assert gbn.fitted() == True

>>> # Check the learned CPDs.
>>> print(gbn.cpd('a'))
[LinearGaussianCPD] P(a) = N(3.043, 0.396)
>>> print(gbn.cpd('c'))
[LinearGaussianCPD] P(c | a) = N(-4.423 + -1.083*a, 0.659)
>>> print(gbn.cpd('d'))
[LinearGaussianCPD] P(d | c) = N(3.000 + 1.200*c, 0.500)
>>> print(gbn.cpd('e'))
[LinearGaussianCPD] P(e) = N(-0.020, 1.144)

>>> # You can sample some data
>>> sample = gbn.sample(50)
```

(continues on next page)

(continued from previous page)

```
>>> # Compute the log-likelihood of each instance
>>> ll = gbn.logl(sample)
>>> # or the sum of log-likelihoods.
>>> sll = gbn.slogl(sample)
>>> assert np.isclose(ll.sum(), sll)

>>> # Save the model, include the CPDs in the file.
>>> gbn.save('test', include_cpd=True)

>>> # Load the model
>>> from pybnesian import load
>>> loaded_gbn = load('test.pickle')

>>> # Learn the structure using greedy hill-climbing.
>>> from pybnesian import hc, GaussianNetworkType
>>> # Learn a Gaussian network.
>>> learned = hc(df, bn_type=GaussianNetworkType())
>>> learned.num_arcs()
2
```

CHAPTER

TWO

EXTENDING PYBNESIAN FROM PYTHON

PyBNesian is completely implemented in C++ for better performance. However, some functionality might not be yet implemented.

PyBNesian allows extending its functionality easily using Python code. This extension code can interact smoothly with the C++ implementation, so that we can reuse most of the current implemented models or algorithms. Also, C++ code is usually much faster than Python, so reusing the implementation also provides performance improvements.

Almost all components of the library can be extended:

- Factors: to include new conditional probability distributions.
- Models: to include new types of Bayesian network models.
- Independence tests: to include new conditional independence tests.
- Learning scores: to include new learning scores.
- Learning operators: to include new operators.
- Learning callbacks: callback function on each iteration of *GreedyHillClimbing*.

The extended functionality can be used exactly equal to the base functionality.

Note: You should avoid re-implementing the base functionality using extensions. Extension code is usually worse in performance for two reasons:

- Usually, the Python code is slower than C++ (unless you have a really good implementation!).
- Crossing the Python<->C++ boundary has a performance cost. Reducing the transition between languages is always good for performance

For all the extensible components, the strategy is always to implement an abstract class.

Warning: All the classes that need to be inherited are developed in C++. For this reason, in the constructor of the new classes it is always necessary to explicitly call the constructor of the parent class. This should be the first line of the constructor.

For example, when inheriting from *FactorType*, **DO NOT DO this:**

```
class NewFactorType(FactorType):
    def __init__(self):
        # Some code in the constructor
```

The following code is correct:

```python
class NewFactorType(FactorType):
def __init__(self):
    FactorType.__init__(self)
    # Some code in the constructor
```

Check the constructor details of the abstract classes in the *API Reference* to make sure you call the parent constructor with the correct parameters.

If you have forgotten to call the parent constructor, the following error message will be displayed when creating a new object (for pybind11>=2.6):

```
>>> t = NewFactorType()
TypeError: pybnesian.FactorType.__init__() must be called when overriding __init__
```

2.1 Factor Extension

Implementing a new factor usually involves creating two new classes that inherit from `FactorType` and `Factor`. A `FactorType` is the representation of a `Factor` type. A `Factor` is an specific instance of a factor (a conditional probability distribution for a given variable and evidence).

These two classes are usually related: a `FactorType` can create instances of `Factor` (with `FactorType.new_factor()`), and a `Factor` returns its corresponding `FactorType` (with `Factor.type()`).

A new `FactorType` need to implement the following methods:

- `FactorType.__str__()`.
- `FactorType.new_factor()`.

A new `Factor` need to implement the following methods:

- `Factor.__str__()`.
- `Factor.type()`.
- `Factor.fitted()`.
- `Factor.fit()`. This method is needed for `BayesianNetworkBase.fit()` or `DynamicBayesianNetworkBase.fit()`.
- `Factor.logl()`. This method is needed for `BayesianNetworkBase.logl()` or `DynamicBayesianNetworkBase.logl()`.
- `Factor.slogl()`. This method is needed for `BayesianNetworkBase.slogl()` or `DynamicBayesianNetworkBase.slogl()`.
- `Factor.sample()`. This method is needed for `BayesianNetworkBase.sample()` or `DynamicBayesianNetworkBase.sample()`.
- `Factor.data_type()`. This method is needed for `DynamicBayesianNetworkBase.sample()`.

You can avoid implementing some of these methods if you do not need them. If a method is needed for a functionality but it is not implemented, an error message is shown when trying to execute that functionality:

```
Tried to call pure virtual function Class::method
```

To illustrate, we will create an alternative implementation of a linear Gaussian CPD.

```
import numpy as np
from scipy.stats import norm
import pyarrow as pa
from pybnesian import FactorType, Factor, CKDEType

# Define our Factor type
class MyLGType(FactorType):
    def __init__(self):
        # IMPORTANT: Always call the parent class to initialize the C++ object.
        FactorType.__init__(self)

    # The __str__ is also used in __repr__ by default.
    def __str__(self):
        return "MyLGType"

    # Create the factor instance defined below.
    def new_factor(self, model, variable, evidence, *args, **kwargs):
        return MyLG(variable, evidence)

class MyLG(Factor):
    def __init__(self, variable, evidence):
        # IMPORTANT: Always call the parent class to initialize the C++ object.
        # The variable and evidence are accessible through self.variable() and self.
↪evidence().
        Factor.__init__(self, variable, evidence)
        self._fitted = False
        self.beta = np.empty((1 + len(evidence),))
        self.variance = -1

    def __str__(self):
        if self._fitted:
            return "MyLG(beta: " + str(self.beta) + ", variance: " + str(self.variance)␣
↪+ ")"
        else:
            return "MyLG(unfitted)"

    def data_type(self):
        return pa.float64()

    def fit(self, df):
        pandas_df = df.to_pandas()

        # Run least squares to train the linear regression
        restricted_df = pandas_df.loc[:, [self.variable()] + self.evidence()].dropna()
        numpy_variable = restricted_df.loc[:, self.variable()].to_numpy()
        numpy_evidence =  restricted_df.loc[:, self.evidence()].to_numpy()
        linregress_data = np.column_stack((np.ones(numpy_evidence.shape[0]), numpy_
↪evidence))
        (self.beta, res, _, _) = np.linalg.lstsq(linregress_data, numpy_variable,␣
↪rcond=None)
        self.variance = res[0] / (linregress_data.shape[0] - 1)
        # Model fitted
        self._fitted = True
```

(continues on next page)

(continued from previous page)

```
    def fitted(self):
        return self._fitted

    def logl(self, df):
        pandas_df = df.to_pandas()

        expected_means = self.beta[0] + np.sum(self.beta[1:] * pandas_df.loc[:,self.
↪evidence()], axis=1)
        return norm.logpdf(pandas_df.loc[:,self.variable()], expected_means, np.
↪sqrt(self.variance))

    def sample(self, n, evidence, seed):
        pandas_df = df.to_pandas()

        expected_means = self.beta[0] + np.sum(self.beta[1:] * pandas_df.loc[:,self.
↪evidence()], axis=1)
        return np.random.normal(expected_means, np.sqrt(self.variance))

    def slogl(self, df):
        return self.logl(df).sum()

    def type(self):
        return MyLGType()
```

2.1.1 Serialization

All the factors can be saved using pickle with the method *Factor.save()*. The class *Factor* already provides a `__getstate__` and `__setstate__` implementation that saves the base information (variable name and evidence variable names). If you need to save more data in your class, there are two alternatives:

- Implement the methods `Factor.__getstate_extra__()` and `Factor.__setstate_extra__()`. These methods have the the same restrictions as the `__getstate__` and `__setstate__` methods (the returned objects must be pickleable).
- Re-implement the `Factor.__getstate__()` and `Factor.__setstate__()` methods. Note, however, that it is needed to call the parent class constructor explicitly in `Factor.__setstate__()` (as in *warning constructor*). This is needed to initialize the C++ part of the object. Also, you will need to add yourself the base information.

For example, if we want to implement serialization support for our re-implementation of linear Gaussian CPD, we can add the following code:

```
class MyLG(Factor):
    #
    # Previous code
    #

    def __getstate_extra__(self):
        return {'fitted': self._fitted,
                'beta': self.beta,
                'variance': self.variance}
```

(continues on next page)

(continued from previous page)

```
    def __setstate_extra__(self, extra):
        self._fitted = extra['fitted']
        self.beta = extra['beta']
        self.variance = extra['variance']
```

Alternatively, the following code will also work correctly:

```
class MyLG(Factor):
    #
    # Previous code
    #

    def __getstate__(self):
        # Make sure to include the variable and evidence.
        return {'variable': self.variable(),
                'evidence': self.evidence(),
                'fitted': self._fitted,
                'beta': self.beta,
                'variance': self.variance}

    def __setstate__(self, extra):
        # Call the parent constructor always in __setstate__ !
        Factor.__init__(self, extra['variable'], extra['evidence'])
        self._fitted = extra['fitted']
        self.beta = extra['beta']
        self.variance = extra['variance']
```

2.1.2 Using Extended Factors

The extended factors can not be used in some specific networks: A *GaussianNetwork* only admits *LinearGaussianCPDType*, a *SemiparametricBN* admits *LinearGaussianCPDType* or *CKDEType*, and so on…

If you try to use MyLG in a Gaussian network, a ValueError is raised.

```
>>> from pybnesian import GaussianNetwork
>>> g = GaussianNetwork(["a", "b", "c", "d"])
>>> g.set_node_type("a", MyLGType())
Traceback (most recent call last):
...
ValueError: Wrong factor type "MyLGType" for node "a" in Bayesian network type
↪"GaussianNetworkType"
```

There are two alternatives to use an extended *Factor*:

- Create an extended model (see *Model Extension*) that admits the new extended *Factor*.
- Use a generic Bayesian network like *HomogeneousBN* and *HeterogeneousBN*.

The *HomogeneousBN* and *HeterogeneousBN* Bayesian networks admit any *FactorType*. The difference between them is that *HomogeneousBN* is homogeneous (all the nodes have the same *FactorType*) and *HeterogeneousBN* is heterogeneous (each node can have a different *FactorType*).

Our extended factor MyLG can be used with an *HomogeneousBN* to create and alternative implementation of a *GaussianNetwork*:

```
>>> import pandas as pd
>>> from pybnesian import HomogeneousBN, GaussianNetwork
>>> # Create some multivariate normal sample data
>>> def generate_sample_data(size, seed=0):
...     np.random.seed(seed)
...     a_array = np.random.normal(3, 0.5, size=size)
...     b_array = np.random.normal(2.5, 2, size=size)
...     c_array = -4.2 + 1.2*a_array + 3.2*b_array + np.random.normal(0, 0.75, size=size)
...     d_array = 1.5 - 0.3 * c_array + np.random.normal(0, 0.5, size=size)
...     return pd.DataFrame({'a': a_array, 'b': b_array, 'c': c_array, 'd': d_array})
>>> df = generate_sample_data(300)
>>> df_test = generate_sample_data(20, seed=1)
>>> # Create an HomogeneousBN and fit it
>>> homo = HomogeneousBN(MyLGType(), ["a", "b", "c", "d"], [("a", "c")])
>>> homo.fit(df)
>>> # Create a GaussianNetwork and fit it
>>> gbn = GaussianNetwork(["a", "b", "c", "d"], [("a", "c")])
>>> gbn.fit(df)
>>> # Check parameters
>>> def check_parameters(cpd1, cpd2):
...     assert np.all(np.isclose(cpd1.beta, cpd2.beta))
...     assert np.isclose(cpd1.variance, cpd2.variance)
>>> # Check the parameters for all CPDs.
>>> check_parameters(homo.cpd("a"), gbn.cpd("a"))
>>> check_parameters(homo.cpd("b"), gbn.cpd("b"))
>>> check_parameters(homo.cpd("c"), gbn.cpd("c"))
>>> check_parameters(homo.cpd("d"), gbn.cpd("d"))
>>> # Check the log-likelihood.
>>> assert np.all(np.isclose(homo.logl(df_test), gbn.logl(df_test)))
>>> assert np.isclose(homo.slogl(df_test), gbn.slogl(df_test))
```

The extended factor can also be used in an heterogeneous Bayesian network. For example, we can imitate the behaviour of a *SemiparametricBN* using an *HeterogeneousBN*:

```
>>> from pybnesian import HeterogeneousBN, CKDEType, SemiparametricBN
>>> df = generate_sample_data(300)
>>> df_test = generate_sample_data(20, seed=1)
>>> # Create an heterogeneous with "MyLG" factors as default.
>>> het = HeterogeneousBN([MyLGType()],  ["a", "b", "c", "d"], [("a", "c")])
>>> het.set_node_type("a", CKDEType())
>>> het.fit(df)
>>> # Create a SemiparametricBN
>>> spbn = SemiparametricBN(["a", "b", "c", "d"], [("a", "c")], [("a", CKDEType())])
>>> spbn.fit(df)
>>> # Check the parameters of the CPDs
>>> check_parameters(het.cpd("b"), spbn.cpd("b"))
>>> check_parameters(het.cpd("c"), spbn.cpd("c"))
>>> check_parameters(het.cpd("d"), spbn.cpd("d"))
>>> # Check the log-likelihood.
>>> assert np.all(np.isclose(het.logl(df_test), spbn.logl(df_test)))
>>> assert np.isclose(het.slogl(df_test), spbn.slogl(df_test))
```

The *HeterogeneousBN* can also be instantiated using a dict to specify different default factor types for different data types. For example, we can mix the `MyLG` factor with *DiscreteFactor* for discrete data:

```
>>> import pyarrow as pa
>>> import pandas as pd
>>> from pybnesian import HeterogeneousBN, CKDEType, DiscreteFactorType, SemiparametricBN

>>> def generate_hybrid_sample_data(size, seed=0):
...     np.random.seed(seed)
...     a_array = np.random.normal(3, 0.5, size=size)
...     b_categories = np.asarray(['b1', 'b2'])
...     b_array = b_categories[np.random.choice(b_categories.size, size, p=[0.5, 0.5])]
...     c_array = -4.2 + 1.2 * a_array + np.random.normal(0, 0.75, size=size)
...     d_array = 1.5 - 0.3 * c_array + np.random.normal(0, 0.5, size=size)
...     return pd.DataFrame({'a': a_array,
...                          'b': pd.Series(b_array, dtype='category'),
...                          'c': c_array,
...                          'd': d_array})

>>> df = generate_hybrid_sample_data(20)
>>> # Create an heterogeneous with "MyLG" factors as default for continuous data and
>>> # "DiscreteFactorType" for categorical data.
>>> het = HeterogeneousBN({pa.float64(): [MyLGType()],
...                        pa.float32(): [MyLGType()],
...                        pa.dictionary(pa.int8(), pa.utf8()): [DiscreteFactorType()]},
...                        ["a", "b", "c", "d"],
...                        [("a", "c")])
>>> het.set_node_type("a", CKDEType())
>>> het.fit(df)
>>> assert het.node_type('a') == CKDEType()
>>> assert het.node_type('b') == DiscreteFactorType()
>>> assert het.node_type('c') == MyLGType()
>>> assert het.node_type('d') == MyLGType()
```

2.2 Model Extension

Implementing a new model Bayesian network model involves creating a class that inherits from *BayesianNetworkType*. Optionally, you also might want to inherit from *BayesianNetwork*, *ConditionalBayesianNetwork* and *DynamicBayesianNetwork*.

A *BayesianNetworkType* is the representation of a Bayesian network model. This is similar to the relation between *FactorType* and a factor. The *BayesianNetworkType* defines the restrictions and properties that characterise a Bayesian network model. A *BayesianNetworkType* is used by all the variants of Bayesian network models: *BayesianNetwork*, *ConditionalBayesianNetwork* and *DynamicBayesianNetwork*. For this reason, the constructors *BayesianNetwork.__init__()*, *ConditionalBayesianNetwork.__init__()* *DynamicBayesianNetwork.__init__()* take the underlying *BayesianNetworkType* as parameter. Thus, once a new *BayesianNetworkType* is implemented, you can use your new Bayesian model with the three variants automatically.

Implementing a *BayesianNetworkType* requires to implement the following methods:

- *BayesianNetworkType.__str__()*.
- *BayesianNetworkType.is_homogeneous()*.
- *BayesianNetworkType.default_node_type()*. This method is optional. It is only needed for homogeneous Bayesian networks.

- *BayesianNetworkType.data_default_node_type()*. This method is optional. It is only needed for non-homogeneous Bayesian networks.
- *BayesianNetworkType.compatible_node_type()*. This method is optional. It is only needed for non-homogeneous Bayesian networks. If not implemented, it accepts any *FactorType* for each node.
- *BayesianNetworkType.can_have_arc()*. This method is optional. If not implemented, it accepts any arc.
- *BayesianNetworkType.new_bn()*.
- *BayesianNetworkType.new_cbn()*.
- *BayesianNetworkType.alternative_node_type()*. This method is optional. This method is needed to learn a Bayesian network structure with *ChangeNodeTypeSet*. This method is only needed for non-homogeneous Bayesian networks.

To illustrate, we will create a Gaussian network that only admits arcs `source -> target` where `source` contains the letter "a". To make the example more interesting we will also use our custom implementation `MyLG` (*in the previous section*).

```
from pybnesian import BayesianNetworkType

class MyRestrictedGaussianType(BayesianNetworkType):
    def __init__(self):
        # Remember to call the parent constructor.
        BayesianNetworkType.__init__(self)

    # The __str__ is also used in __repr__ by default.
    def __str__(self):
        return "MyRestrictedGaussianType"

    def is_homogeneous(self):
        return True

    def default_node_type(self):
        return MyLGType()

    # NOT NEEDED because it is homogeneous. If heterogeneous we would return
    # the default node type for the data_type.
    # def data_default_node_type(self, data_type):
    #     if data_type.equals(pa.float64()) or data_type.equals(pa.float32()):
    #         return MyLGType()
    #     else:
    #         raise ValueError("Wrong data type for MyRestrictedGaussianType")
    #
    # NOT NEEDED because it is homogeneous. If heterogeneous we would check
    # that the node type is correct.
    # def compatible_node_type(self, model, node):
    #    return self.node_type(node) == MyLGType or self.node_type(node) == ...

    def can_have_arc(self, model, source, target):
        # Our restriction for arcs.
        return "a" in source.lower()

    def new_bn(self, nodes):
        return BayesianNetwork(MyRestrictedGaussianType(), nodes)
```

(continues on next page)

(continued from previous page)

```
    def new_cbn(self, nodes, interface_nodes):
        return ConditionalBayesianNetwork(MyRestrictedGaussianType(), nodes, interface_
↪nodes)

    # NOT NEEDED because it is homogeneous. Also, it is not needed if you do not want to␣
↪change the node type.
    # def alternative_node_type(self, node):
    #     pass
```

The arc restrictions defined by *BayesianNetworkType.can_have_arc()* can be an alternative to the blacklist lists in some learning algorithms. However, this arc restrictions are applied always:

```
>>> from pybnesian import BayesianNetwork
>>> g = BayesianNetwork(MyRestrictedGaussianType(), ["a", "b", "c", "d"])
>>> g.add_arc("a", "b") # This is OK
>>> g.add_arc("b", "c") # Not allowed
Traceback (most recent call last):
...
ValueError: Cannot add arc b -> c.
>>> g.add_arc("c", "a") # Also, not allowed
Traceback (most recent call last):
...
ValueError: Cannot add arc c -> a.
>>> g.flip_arc("a", "b") # Not allowed, because it would generate a b -> a arc.
Traceback (most recent call last):
...
ValueError: Cannot flip arc a -> b.
```

2.2.1 Creating Bayesian Network Types

BayesianNetworkType can adapt the behavior of a Bayesian network with a few lines of code. However, you may want to create your own Bayesian network class instead of directly using a *BayesianNetwork*, a *ConditionalBayesianNetwork* or a *DynamicBayesianNetwork*. This has some advantages:

- The source code can be better organized using a different class for each Bayesian network model.
- Using `type(model)` over different types of models would return a different type:

```
>>> from pybnesian import GaussianNetworkType, BayesianNetwork
>>> g1 = BayesianNetwork(GaussianNetworkType(), ["a", "b", "c", "d"])
>>> g2 = BayesianNetwork(MyRestrictedGaussianType(), ["a", "b", "c", "d"])
>>> assert type(g1) == type(g2) # The class type is the same, but the code would be
>>>                             # more obvious if it weren't.
>>> assert g1.type() != g2.type() # You have to use this.
```

- It allows more customization of the Bayesian network behavior.

To create your own Bayesian network, you have to inherit from *BayesianNetwork*, *ConditionalBayesianNetwork* or *DynamicBayesianNetwork*:

```
from pybnesian import BayesianNetwork, ConditionalBayesianNetwork,\
                       DynamicBayesianNetwork
```

(continues on next page)

(continued from previous page)

```
class MyRestrictedBN(BayesianNetwork):
    def __init__(self, nodes, arcs=None):
        # You can initialize with any BayesianNetwork.__init__ constructor.
        if arcs is None:
            BayesianNetwork.__init__(self, MyRestrictedGaussianType(), nodes)
        else:
            BayesianNetwork.__init__(self, MyRestrictedGaussianType(), nodes, arcs)

class MyConditionalRestrictedBN(ConditionalBayesianNetwork):
    def __init__(self, nodes, interface_nodes, arcs=None):
        # You can initialize with any ConditionalBayesianNetwork.__init__ constructor.
        if arcs is None:
            ConditionalBayesianNetwork.__init__(self, MyRestrictedGaussianType(), nodes,
                                                interface_nodes)
        else:
            ConditionalBayesianNetwork.__init__(self, MyRestrictedGaussianType(), nodes,
                                                interface_nodes, arcs)

class MyDynamicRestrictedBN(DynamicBayesianNetwork):
    def __init__(self, variables, markovian_order):
        # You can initialize with any DynamicBayesianNetwork.__init__ constructor.
        DynamicBayesianNetwork.__init__(self, MyRestrictedGaussianType(), variables,
                                        markovian_order)
```

Also, it is recommended to change the *BayesianNetworkType.new_bn()* and *BayesianNetworkType.new_cbn()* definitions:

```
class MyRestrictedGaussianType(BayesianNetworkType):
    #
    # Previous code
    #

    def new_bn(self, nodes):
        return MyRestrictedBN(nodes)

    def new_cbn(self, nodes, interface_nodes):
        return MyConditionalRestrictedBN(nodes, interface_nodes)
```

Creating your own Bayesian network classes allows you to overload the base functionality. Thus, you can customize completely the behavior of your Bayesian network. For example, we can print a message each time an arc is added:

```
class MyRestrictedBN(BayesianNetwork):
    #
    # Previous code
    #

    def add_arc(self, source, target):
        print("Adding arc " + source + " -> " + target)
        # Call the base functionality
        BayesianNetwork.add_arc(self, source, target)
```

```
>>> bn = MyRestrictedBN(["a", "b", "c", "d"])
>>> bn.add_arc("a", "c")
Adding arc a -> c
>>> assert bn.has_arc("a", "c")
```

Note: *BayesianNetwork*, *ConditionalBayesianNetwork* and *DynamicBayesianNetwork* are not abstract classes. These classes provide an implementation for the abstract classes *BayesianNetworkBase*, *ConditionalBayesianNetworkBase* or *DynamicBayesianNetworkBase*.

2.2.2 Serialization

The Bayesian network models can be saved using pickle with the *BayesianNetworkBase.save()* method. This method saves the structure of the Bayesian network and, optionally, the factors within the Bayesian network. When the *BayesianNetworkBase.save()* is called, *BayesianNetworkBase.include_cpd* property is first set and then `__getstate__()` is called. `__getstate__()` saves the factors within the Bayesian network model only if *BayesianNetworkBase.include_cpd* is `True`. The factors can be saved only if the *Factor* is also plickeable (see *Factor serialization*).

As with factor serialization, an implementation of `__getstate__()` and `__setstate__()` is provided when inheriting from *BayesianNetwork*, *ConditionalBayesianNetwork* or *DynamicBayesianNetwork*. This implementation saves:

- The underlying graph of the Bayesian network.
- The underlying *BayesianNetworkType*.
- The list of *FactorType* for each node.
- The list of *Factor* within the Bayesian network (if *BayesianNetworkBase.include_cpd* is `True`).

In the case of *DynamicBayesianNetwork*, it saves the above list for both the static and transition networks.

If your extended Bayesian network class need to save more data, there are two alternatives:

- Implement the methods `__getstate_extra__()` and `__setstate_extra__()`. These methods have the the same restrictions as the `__getstate__()` and `__setstate__()` methods (the returned objects must be pickleable).

```
class MyRestrictedBN(BayesianNetwork):
    #
    # Previous code
    #

    def __getstate_extra__(self):
        # Save some extra data.
        return {'extra_data': self.extra_data}

    def __setstate_extra__(self, d):
        # Here, you can access the extra data. Initialize the attributes that you need
        self.extra_data = d['extra_data']
```

- Re-implement the `__getstate__()` and `__setstate__()` methods. Note, however, that it is needed to call the parent class constructor explicitly in the `__setstate__()` method (as in *warning constructor*). This is needed to initialize the C++ part of the object. Also, you will need to add yourself the base information.

```
class MyRestrictedBN(BayesianNetwork):
    #
    # Previous code
    #

    def __getstate__(self):
    d = {'graph': self.graph(),
         'type': self.type(),
         # You can omit this line if type is homogeneous
         'factor_types': list(self.node_types().items()),
         'extra_data': self.extra_data}

    if self.include_cpd:
        factors = []

        for n in self.nodes():
            if self.cpd(n) is not None:
                factors.append(self.cpd(n))
        d['factors'] = factors

    return d

def __setstate__(self, d):
    # Call the parent constructor always in __setstate__ !
    BayesianNetwork.__init__(self, d['type'], d['graph'], d['factor_types'])

    if "factors" in d:
        self.add_cpds(d['factors'])

    # Here, you can access the extra data.
    self.extra_data = d['extra_data']
```

The same strategy is used to implement serialization in *ConditionalBayesianNetwork* and *DynamicBayesianNetwork*.

> **Warning:** Some functionalities require to make copies of Bayesian network models. Copying Bayesian network models is currently implemented using this serialization suppport. Therefore, it is highly recommended to implement `__getstate_extra__()`/`__setstate_extra__()` or `__getstate__()`/`__setstate__()`. Otherwise, the extra information defined in the extended classes would be lost.

2.3 Independence Test Extension

Implementing a new conditional independence test involves creating a class that inherits from *IndependenceTest*.

A new *IndependenceTest* needs to implement the following methods:

- *IndependenceTest.num_variables()*.
- *IndependenceTest.variable_names()*.
- *IndependenceTest.has_variables()*.
- *IndependenceTest.name()*.

- *IndependenceTest.pvalue()*.

To illustrate, we will implement a conditional independence test that has perfect information about the conditional indepencences (an oracle independence test):

```
from pybnesian import IndependenceTest

class OracleTest(IndependenceTest):

    # An Oracle class that represents the independences of this Bayesian network:
    #
    #  "a"     "b"
    #    \      /
    #     \    /
    #      \  /
    #       V
    #      "c"
    #       |
    #       |
    #       V
    #      "d"

    def __init__(self):
        # IMPORTANT: Always call the parent class to initialize the C++ object.
        IndependenceTest.__init__(self)
        self.variables = ["a", "b", "c", "d"]

    def num_variables(self):
        return len(self.variables)

    def variable_names(self):
        return self.variables

    def has_variables(self, vars):
        return set(vars).issubset(set(self.variables))

    def name(self, index):
        return self.variables[index]

    def pvalue(self, x, y, z):
        if z is None:
            # a _|_ b
            if set([x, y]) == set(["a", "b"]):
                return 1
            else:
                return 0
        else:
            z = list(z)
            if "c" in z:
                # a _|_ d | "c" in Z
                if set([x, y]) == set(["a", "d"]):
                    return 1
                # b _|_ d | "c" in Z
                if set([x, y]) == set(["b", "d"]):
```

(continues on next page)

(continued from previous page)

```
                return 1
            return 0
```

The oracle version of the PC algorithm guarantees the return of the correct network structure. We can use our new oracle independence test with the *PC* algorithm.

```
>>> from pybnesian import PC
>>> pc = PC()
>>> oracle = OracleTest()
>>> graph = pc.estimate(oracle)
>>> assert set(graph.arcs()) == {('a', 'c'), ('b', 'c'), ('c', 'd')}
>>> assert graph.num_edges() == 0
```

To learn dynamic Bayesian networks your class has to override *DynamicIndependenceTest*. A new *DynamicIndependenceTest* needs to implement the following methods:

- *DynamicIndependenceTest.num_variables()*.
- *DynamicIndependenceTest.variable_names()*.
- *DynamicIndependenceTest.has_variables()*.
- *DynamicIndependenceTest.name()*.
- *DynamicIndependenceTest.markovian_order()*.
- *DynamicIndependenceTest.static_tests()*.
- *DynamicIndependenceTest.transition_tests()*.

Usually, your extended *IndependenceTest* will use data. It is easy to implement a related *DynamicIndependenceTest* by taking a *DynamicDataFrame* as parameter and using the methods *DynamicDataFrame.static_df()* and *DynamicDataFrame.transition_df()* to implement *DynamicIndependenceTest.static_tests()* and *DynamicIndependenceTest.transition_tests()* respectively.

2.4 Learning Scores Extension

Implementing a new learning score involves creating a class that inherits from *Score* or *ValidatedScore*. The score must be decomposable.

The *ValidatedScore* is an *Score* that is evaluated in two different data sets: a training dataset and a validation dataset.

An extended *Score* class needs to implement the following methods:

- *Score.has_variables()*.
- *Score.compatible_bn()*.
- *Score.score()*. This method is optional. The default implementation sums the local score for all the nodes.
- *Score.local_score()*. Only the version with 3 arguments `score.local_score(model, variable, evidence)` needs to be implemented. The version with 2 arguments cannot be overriden.
- *Score.local_score_node_type()*. This method is optional. This method is only needed if the score is used together with *ChangeNodeTypeSet*.
- *Score.data()*. This method is optional. It is needed to infer the default node types in the *GreedyHillClimbing* algorithm.

In addition, an extended *ValidatedScore* class needs to implement the following methods to get the score in the validation dataset:

- *ValidatedScore.vscore()*. This method is optional. The default implementation sums the validation local score for all the nodes.
- *ValidatedScore.vlocal_score()*. Only the version with 3 arguments `score.vlocal_score(model, variable, evidence)` needs to be implemented. The version with 2 arguments can not be overriden.
- *ValidatedScore.vlocal_score_node_type()*. This method is optional. This method is only needed if the score is used together with *ChangeNodeTypeSet*.

To illustrate, we will implement an oracle score that only returns positive score to the arcs `a -> c`, `b -> c` and `c -> d`.

```
from pybnesian import Score

class OracleScore(Score):

    # An oracle class that returns positive scores for the arcs in the
    # following Bayesian network:
    #
    #  "a"     "b"
    #    \     /
    #     \   /
    #      \ /
    #       V
    #      "c"
    #       |
    #       |
    #       V
    #      "d"

    def __init__(self):
        Score.__init__(self)
        self.variables = ["a", "b", "c", "d"]

    def has_variables(self, vars):
        return set(vars).issubset(set(self.variables))

    def compatible_bn(self, model):
        return self.has_variables(model.nodes())

    def local_score(self, model, variable, evidence):
        if variable == "c":
            v = -1
            if "a" in evidence:
                v += 1
            if "b" in evidence:
                v += 1.5
            return v
        elif variable == "d" and evidence == ["c"]:
            return 1
        else:
            return -1
```

(continues on next page)

(continued from previous page)

```
    # NOT NEEDED because this score does not use data.
    # In that case, this method can return None or you can avoid implementing this␣
↪method.
    def data(self):
        return None
```

We can use this new score, for example, with a *GreedyHillClimbing*.

```
>>> from pybnesian import GaussianNetwork, GreedyHillClimbing, ArcOperatorSet
>>>
>>> hc = GreedyHillClimbing()
>>> start_model = GaussianNetwork(["a", "b", "c", "d"])
>>> learned_model = hc.estimate(ArcOperatorSet(), OracleScore(), start_model)
>>> assert set(learned_model.arcs()) == {('a', 'c'), ('b', 'c'), ('c', 'd')}
```

To learn dynamic Bayesian networks your class has to override *DynamicScore*. A new *DynamicScore* needs to implement the following methods:

- *DynamicScore.has_variables()*.
- *DynamicScore.static_score()*.
- *DynamicScore.transition_score()*.

Usually, your extended *Score* will use data. It is easy to implement a related *DynamicScore* by taking a *DynamicDataFrame* as parameter and using the methods *DynamicDataFrame.static_df()* and *DynamicDataFrame.transition_df()* to implement *DynamicScore.static_score()* and *DynamicScore.transition_score()* respectively.

2.5 Learning Operators Extension

Implementing a new learning score involves creating a class that inherits from *Operator* (or *ArcOperator* for operators related with a single arc). Next, a new *OperatorSet* must be defined to use the new learning operator within a learning algorithm.

An extended *Operator* class needs to implement the following methods:

- *Operator.__eq__()*. This method is optional. This method is needed if the *OperatorTabuSet* is used (in the *GreedyHillClimbing* it is used when the score is *ValidatedScore*).
- *Operator.__hash__()*. This method is optional. This method is needed if the *OperatorTabuSet* is used (in the *GreedyHillClimbing* it is used when the score is *ValidatedScore*).
- *Operator.__str__()*.
- *Operator.apply()*.
- *Operator.nodes_changed()*.
- *Operator.opposite()*. This method is optional. This method is needed if the *OperatorTabuSet* is used (in the *GreedyHillClimbing* it is used when the score is *ValidatedScore*).

To illustrate, we will create a new *AddArc* operator.

```
from pybnesian import Operator, RemoveArc

class MyAddArc(Operator):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, source, target, delta):
        # IMPORTANT: Always call the parent class to initialize the C++ object.
        Operator.__init__(self, delta)
        self.source = source
        self.target = target

    def __eq__(self, other):
        return self.source == other.source and self.target == other.target

    def __hash__(self):
        return hash((self.source, self.target))

    def __str__(self):
        return "MyAddArc(" + self.source + " -> " + self.target + ")"

    def apply(self, model):
        model.add_arc(self.source, self.target)

    def nodes_changed(self, model):
        return [self.target]

    def opposite():
        return RemoveArc(self.source, self.target, -self.delta())
```

To use this new operator, we need to define a *OperatorSet* that returns this type of operators. An extended *OperatorSet* class needs to implement the following methods:

- *OperatorSet.cache_scores()*.
- *OperatorSet.find_max()*.
- *OperatorSet.find_max_tabu()*. This method is optional. This method is needed if the *OperatorTabuSet* is used (in the *GreedyHillClimbing* it is used when the score is *ValidatedScore*).
- *OperatorSet.set_arc_blacklist()*. This method is optional. Implement it only if you need to check that an arc is blacklisted.
- *OperatorSet.set_arc_whitelist()*. This method is optional. Implement it only if you need to check that an arc is whitelisted.
- *OperatorSet.set_max_indegree()*. This method is optional. Implement it only if you need to check the maximum indegree of the graph.
- *OperatorSet.set_type_blacklist()*. This method is optional. Implement it only if you need to check that a node type is blacklisted.
- *OperatorSet.set_type_whitelist()*. This method is optional. Implement it only if you need to check that a node type is whitelisted.
- *OperatorSet.update_scores()*.
- *OperatorSet.finished()*. This method is optional. Implement it only if your class needs to clear the state.

To illustrate, we will create an operator set that only contains the `MyAddArc` operators. Therefore, this *OperatorSet* can only add arcs.

```
from pybnesian import OperatorSet

class MyAddArcSet(OperatorSet):

    def __init__(self):
        # IMPORTANT: Always call the parent class to initialize the C++ object.
        OperatorSet.__init__(self)
        self.blacklist = set()
        self.max_indegree = 0
        # Contains a dict {(source, target) : delta} of operators.
        self.set = {}

    # Auxiliary method
    def update_node(self, model, score, n):
        lc = self.local_score_cache()

        parents = model.parents(n)

        # Remove the parent operators, they will be added next.
        self.set = {p[0]: p[1] for p in self.set.items() if p[0][1] != n}

        blacklisted_parents = map(lambda op: op[0],
                                  filter(lambda bl : bl[1] == n, self.blacklist))
        # If max indegree == 0, there is no limit.
        if self.max_indegree == 0 or len(parents)  < self.max_indegree:
            possible_parents = set(model.nodes())\
                               - set(n)\
                               - set(parents)\
                               - set(blacklisted_parents)

            for p in possible_parents:
                if model.can_add_arc(p, n):
                    self.set[(p, n)] = score.local_score(model, n, parents + [p])\
                                       - lc.local_score(model, n)

    def cache_scores(self, model, score):
        for n in model.nodes():
            self.update_node(model, score, n)

    def find_max(self, model):
        sort_ops = sorted(self.set.items(), key=lambda op: op[1], reverse=True)

        for s in sort_ops:
            arc = s[0]
            delta = s[1]
            if model.can_add_arc(arc[0], arc[1]):
                return MyAddArc(arc[0], arc[1], delta)
        return None

    def find_max_tabu(self, model, tabu):
        sort_ops = sorted(self.set.items(), key=lambda op: op[1], reverse=True)

        for s in sort_ops:
```

(continues on next page)

(continued from previous page)

```
            arc = s[0]
            delta = s[1]
            op = MyAddArc(arc[0], arc[1], delta)
            # The operator cannot be in the tabu set.
            if model.can_add_arc(arc[0], arc[1]) and not tabu.contains(op):
                return op
        return None

    def update_scores(self, model, score, changed_nodes):
        for n in changed_nodes:
            self.update_node(model, score, n)

    def set_arc_blacklist(self, blacklist):
        self.blacklist = set(blacklist)

    def set_max_indegree(self, max_indegree):
        self.max_indegree = max_indegree

    def finished(self):
        self.blacklist.clear()
        self.max_indegree = 0
        self.set.clear()
```

This *OperatorSet* can be used in a *GreedyHillClimbing*:

```
>>> from pybnesian import GreedyHillClimbing
>>> hc = GreedyHillClimbing()
>>> add_set = MyAddArcSet()
>>> # We will use the OracleScore: a -> c <- b, c -> d
>>> score = OracleScore()
>>> bn = GaussianNetwork(["a", "b", "c", "d"])
>>> learned = hc.estimate(add_set, score, bn)
>>> assert set(learned_model.arcs()) == {("a", "c"), ("b", "c"), ("c", "d")}
>>> learned = hc.estimate(add_set, score, bn, arc_blacklist=[("b", "c")])
>>> assert set(learned.arcs()) == {("a", "c"), ("c", "d")}
>>> learned = hc.estimate(add_set, score, bn, max_indegree=1)
>>> assert learned.num_arcs() == 2
```

2.6 Callbacks Extension

The greedy hill-climbing algorithm admits a `callback` parameter that allows some custom functionality to be run on each iteration. To create a callback, a new class must be created that inherits from *Callback*. A new *Callback* needs to implement the following method:

- *Callback.call*.

To illustrate, we will create a callback that prints the last operator applied on each iteration:

```
from pybnesian import Callback

class PrintOperator(Callback):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        # IMPORTANT: Always call the parent class to initialize the C++ object.
        Callback.__init__(self)

    def call(self, model, operator, score, iteration):
        if operator is None:
            if iteration == 0:
                print("The algorithm starts!")
            else:
                print("The algorithm ends!")
        else:
            print("Iteration " + str(iteration) + ". Last operator: " + str(operator))
```

Now, we can use this callback in the *GreedyHillClimbing*:

```
>>> from pybnesian import GreedyHillClimbing
>>> hc = GreedyHillClimbing()
>>> add_set = MyAddArcSet()
>>> # We will use the OracleScore: a -> c <- b, c -> d
>>> score = OracleScore()
>>> bn = GaussianNetwork(["a", "b", "c", "d"])
>>> callback = PrintOperator()
>>> learned = hc.estimate(add_set, score, bn, callback=callback)
The algorithm starts!
Iteration 1. Last operator: MyAddArc(c -> d)
Iteration 2. Last operator: MyAddArc(b -> c)
Iteration 3. Last operator: MyAddArc(a -> c)
The algorithm ends!
```

2.7 Bandwidth Selection

The *KDE* *ProductKDE* and *CKDE* classes can accept an *BandwidthSelector* to estimate the bandwidth of the kernel density estimation models.

A new bandwidth selection technique can be implemented by creating a class that inherits from *BandwidthSelector* and implementing the following methods:

- *BandwidthSelector.bandwidth*. To select an unconstrained bandwidth matrix $\mathbf{H}$ for a *KDE*.
- *BandwidthSelector.diag_bandwidth*. To select a diagonal bandwidth matrix $\mathbf{h}$ for a *ProductKDE*.
- *BandwidthSelector.__str__*, which is also automatically used as __repr__.

To illustrate, we will create a bandwidth selector that always return an unitary bandwidth matrix:

```
class UnitaryBandwidth(BandwidthSelector):
    def __init__(self):
        BandwidthSelector.__init__(self)

    # For a KDE.
    def bandwidth(self, df, variables):
    return np.eye(len(variables))
```

(continues on next page)

(continued from previous page)

```
    # For a ProductKDE.
    def diag_bandwidth(self, df, variables):
        return np.ones((len(variables),))

    def __str__(self):
        return "UnitaryBandwidth"
```

CHAPTER

THREE

API REFERENCE

3.1 Data Manipulation

PyBNesian implements some useful dataset manipulation techniques such as k-fold cross validation and hold-out.

3.1.1 DataFrame

Internally, PyBNesian uses a `pyarrow.RecordBatch` to enable a zero-copy data exchange between C++ and Python.

Most of the classes and methods takes as argument, or returns a `DataFrame` type. This represents an encapsulation of `pyarrow.RecordBatch`:

- When a `DataFrame` is taken as argument in a function, both a `pyarrow.RecordBatch` or a `pandas.DataFrame` can be used as a parameter.
- When PyBNesian specifies a `DataFrame` return type, a `pyarrow.RecordBatch` is returned. This can be converted easily to a `pandas.DataFrame` using `pyarrow.RecordBatch.to_pandas()`.

DataFrame Operations

class pybnesian.**CrossValidation**

This class implements k-fold cross-validation, i.e. it splits the data into k disjoint sets of train and test data.

__init__(*self: pybnesian.CrossValidation*, *df: DataFrame*, *k: int = 10*, *seed: Optional[int] = None*, *include_null: bool = False*) → None

This constructor takes a `DataFrame` and returns a k-fold cross-validation. It shuffles the data before applying the cross-validation.

Parameters

- **df** – A `DataFrame`.
- **k** – Number of folds.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **include_null** – Whether to include the rows where some columns may be null (missing). If false, the rows with some missing values are filtered before performing the cross-validation. Else, all the rows are included.

Raises

`ValueError` – If k is greater than the number of rows.

__iter__(*self: pybnesian.CrossValidation*) → Iterator

Iterates over the k-fold cross-validation.

Returns
The iterator returns a tuple (`DataFrame`, `DataFrame`) which contains the training data and test data of each fold.

```
>>> from pybnesian import CrossValidation
>>> df = pd.DataFrame({'a': np.random.rand(20), 'b': np.random.rand(20)})
>>> for (training_data, test_data) in CrossValidation(df):
...     assert training_data.num_rows == 18
...     assert test_data.num_rows == 2
```

fold(*self: pybnesian.CrossValidation, index: int*) → Tuple[DataFrame, DataFrame]

Returns the index-th fold.

Parameters
index – Fold index.

Returns
A tuple (`DataFrame`, `DataFrame`) which contains the training data and test data of each fold.

indices(*self: pybnesian.CrossValidation*) → Iterator

Iterates over the row indices of each training and test `DataFrame`.

Returns
A tuple (list, list) containing the row indices (with respect to the original `DataFrame`) of the train and test data of each fold.

```
>>> from pybnesian import CrossValidation
>>> df = pd.DataFrame({'a': np.random.rand(20), 'b': np.random.rand(20)})
>>> for (training_indices, test_indices) in CrossValidation(df).indices():
...     assert set(range(20)) == set(list(training_indices) + list(test_
↪indices))
```

loc(*self: pybnesian.CrossValidation, columns: str or int or List[str] or List[int]*) → *CrossValidation*

Selects columns from the `CrossValidation` object.

Parameters
columns – Columns to select. The columns can be represented by their index (int or List[int]) or by their name (str or List[str]).

Returns
A `CrossValidation` object with the selected columns.

class pybnesian.HoldOut

This class implements holdout validation, i.e. it splits the data into training and test sets.

__init__(*self: pybnesian.HoldOut, df: DataFrame, test_ratio: float = 0.2, seed: Optional[int] = None, include_null: bool = False*) → None

This constructor takes a `DataFrame` and returns a split into training an test sets. It shuffles the data before applying the holdout.

Parameters

- **df** – A `DataFrame`.
- **test_ratio** – Proportion of instances left for the test data.

- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **include_null** – Whether to include the rows where some columns may be null (missing). If false, the rows with some missing values are filtered before performing the cross-validation. Else, all the rows are included.

test_data(*self*: pybnesian.HoldOut) → DataFrame

Gets the test data.

Returns
Test data.

training_data(*self*: pybnesian.HoldOut) → DataFrame

Gets the training data.

Returns
Training data.

Dynamic Data

class pybnesian.**DynamicDataFrame**

This class implements the adaptation of a *DynamicDataFrame* to a dynamic context (temporal series). This is useful to make easier to learn dynamic Bayesian networks.

A *DynamicDataFrame* creates columns with different temporal delays from the data in the static `DataFrame`. Each column in the *DynamicDataFrame* is named with the following pattern: `[variable_name]_t_[temporal_index]`. The `variable_name` is the name of each column in the static `DataFrame`. The `temporal_index` is an index with a range [0-`markovian_order`]. The index "0" is considered the "present", the index "1" delays the temporal one step into the "past", and so on…

DynamicDataFrame contains two functions *DynamicDataFrame.static_df()* and *DynamicDataFrame.transition_df()* that can be used to learn the static Bayesian network and transition Bayesian network components of a dynamic Bayesian network.

All the operations are implemented using a zero-copy strategy to avoid wasting memory.

```
>>> from pybnesian import DynamicDataFrame
>>> df = pd.DataFrame({'a': np.arange(10, dtype=float)})
>>> ddf = DynamicDataFrame(df, 2)
>>> ddf.transition_df().to_pandas()
   a_t_0  a_t_1  a_t_2
0    2.0    1.0    0.0
1    3.0    2.0    1.0
2    4.0    3.0    2.0
3    5.0    4.0    3.0
4    6.0    5.0    4.0
5    7.0    6.0    5.0
6    8.0    7.0    6.0
7    9.0    8.0    7.0
>>> ddf.static_df().to_pandas()
   a_t_1  a_t_2
0    1.0    0.0
1    2.0    1.0
2    3.0    2.0
3    4.0    3.0
4    5.0    4.0
```

(continues on next page)

(continued from previous page)

```
5     6.0     5.0
6     7.0     6.0
7     8.0     7.0
8     9.0     8.0
```

__init__(*self: pybnesian.DynamicDataFrame*, *df: DataFrame*, *markovian_order: int*) → None

Creates a *DynamicDataFrame* from an static `DataFrame` using a given markovian order.

Parameters

- **df** – A `DataFrame`.
- **markovian_order** – Markovian order of the transformation.

loc(*self: pybnesian.DynamicDataFrame*, *columns: DynamicVariable or List[DynamicVariable]*) → DataFrame

Gets a column or set of columns from the *DynamicDataFrame*. See *DynamicVariable*.

Returns

A `DataFrame` with the selected columns.

```
>>> from pybnesian import DynamicDataFrame
>>> df = pd.DataFrame({'a': np.arange(10, dtype=float),
...                    'b': np.arange(0, 100, 10, dtype=float)})
>>> ddf = DynamicDataFrame(df, 2)
>>> ddf.loc(("b", 1)).to_pandas()
   b_t_1
0   10.0
1   20.0
2   30.0
3   40.0
4   50.0
5   60.0
6   70.0
7   80.0
>>> ddf.loc([("a", 0), ("b", 1)]).to_pandas()
   a_t_0  b_t_1
0    2.0   10.0
1    3.0   20.0
2    4.0   30.0
3    5.0   40.0
4    6.0   50.0
5    7.0   60.0
6    8.0   70.0
7    9.0   80.0
```

All the DynamicVariables in the list must be of the same type, so do not mix different types:

```
>>> ddf.loc([(0, 0), ("b", 1)]) # do NOT do this!

# Either you use names or indices:
>>> ddf.loc([("a", 0), ("b", 1)]) # GOOD
>>> ddf.loc([(0, 1), (1, 1)]) # GOOD
```

markovian_order(*self:* pybnesian.DynamicDataFrame) → int

Gets the markovian order.

Returns
Markovian order of the *DynamicDataFrame*.

num_columns(*self:* pybnesian.DynamicDataFrame) → int

Gets the number of columns.

Returns
The number of columns. This is equal to the number of columns of *DynamicDataFrame.transition_df()*.

num_rows(*self:* pybnesian.DynamicDataFrame) → int

Gets the number of row.

Returns
Number of rows.

num_variables(*self:* pybnesian.DynamicDataFrame) → int

Gets the number of variables.

Returns
The number of variables. This is exactly equal to the number of columns in *DynamicDataFrame.origin_df()*.

origin_df(*self:* pybnesian.DynamicDataFrame) → DataFrame

Gets the original `DataFrame`.

Returns
The `DataFrame` passed to the constructor of *DynamicDataFrame*.

static_df(*self:* pybnesian.DynamicDataFrame) → DataFrame

Gets the `DataFrame` for the static Bayesian network. The static network estimates the probability f(`t_1`,. . . , `t_[markovian_order]`). See *DynamicDataFrame example*.

Returns
A `DataFrame` with columns from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`

temporal_slice(*self:* pybnesian.DynamicDataFrame, *indices: int or List[int]*) → DataFrame

Gets a temporal slice or a set of temporal slices. The i-th temporal slice is composed by the columns `[variable_name]_t_i`

Returns
A `DataFrame` with the selected temporal slices.

```
>>> from pybnesian import DynamicDataFrame
>>> df = pd.DataFrame({'a': np.arange(10, dtype=float), 'b': np.arange(0, 100,␣
↪10, dtype=float)})
>>> ddf = DynamicDataFrame(df, 2)
>>> ddf.temporal_slice(1).to_pandas()
   a_t_1  b_t_1
0    1.0   10.0
1    2.0   20.0
2    3.0   30.0
3    4.0   40.0
4    5.0   50.0
```

(continues on next page)

(continued from previous page)

```
5    6.0   60.0
6    7.0   70.0
7    8.0   80.0
>>> ddf.temporal_slice([0, 2]).to_pandas()
   a_t_0  b_t_0  a_t_2  b_t_2
0    2.0   20.0    0.0    0.0
1    3.0   30.0    1.0   10.0
2    4.0   40.0    2.0   20.0
3    5.0   50.0    3.0   30.0
4    6.0   60.0    4.0   40.0
5    7.0   70.0    5.0   50.0
6    8.0   80.0    6.0   60.0
7    9.0   90.0    7.0   70.0
```

transition_df(*self:* pybnesian.DynamicDataFrame) → DataFrame

Gets the `DataFrame` for the transition Bayesian network. The transition network estimates the conditional probability f(`t_0` | `t_1`, . . . , `t_[markovian_order]`). See *DynamicDataFrame example*.

Returns
A `DataFrame` with columns from `[variable_name]_t_0` to `[variable_name]_t_[markovian_order]`

class DynamicVariable

A DynamicVariable is the representation of a column in a *DynamicDataFrame*.

A DynamicVariable is a tuple (`variable_index`, `temporal_index`). `variable_index` is a `str` or `int` that represents the name or index of the variable in the original static `DataFrame`. `temporal_index` is an `int` that represents the temporal slice in the *DynamicDataFrame*. See *DynamicDataFrame.loc* for usage examples.

3.2 Graph Module

PyBNesian includes different types of graphs. There are four types of graphs:

- Undirected graphs.
- Directed graphs.
- Directed acyclic graphs (DAGs).
- Partially directed graphs.

Depending on the type of edges: directed edges (arcs) or undirected edges (edges).

Each graph type has two variants:

- Graphs. See *Graphs*.
- Conditional graphs. See *Conditional Graphs*.

3.2.1 Graphs

All the nodes in the graph are represented by a name and are associated with a non-negative unique index.

The name can be obtained from the unique index using the method `name()`, while the unique index can be obtained from the index using the method `index()`.

Removing a node invalidates the index of the removed node, while leaving the other nodes unaffected. When adding a node, the graph may reuse previously invalidated indices to avoid wasting too much memory.

If there are not removal of nodes in a graph, the unique indices are in the range [0-`num_nodes()`). The removal of nodes, can lead to some indices being greater or equal to `num_nodes()`:

```
>>> from pybnesian import UndirectedGraph
>>> g = UndirectedGraph(["a", "b", "c", "d"])
>>> g.index("a")
0
>>> g.index("b")
1
>>> g.index("c")
2
>>> g.index("d")
3
>>> g.remove_node("a")
>>> g.index("b")
1
>>> g.index("c")
2
>>> g.index("d")
3
>>> assert g.index("d") >= g.num_nodes()
```

Sometimes, this effect may be undesirable because we want to identify our nodes with a index in a range [0-`num_nodes()`). For this reason, there is a `collapsed_index()` method and other related methods `index_from_collapsed()`, `collapsed_from_index()` and `collapsed_name()`. Note that the collapsed index is not unique, because removing a node can change the collapsed index of at most one other node.

```
>>> from pybnesian import UndirectedGraph
>>> g = UndirectedGraph(["a", "b", "c", "d"])
>>> g.collapsed_index("a")
0
>>> g.collapsed_index("b")
1
>>> g.collapsed_index("c")
2
>>> g.collapsed_index("d")
3
>>> g.remove_node("a")
>>> g.collapsed_index("b")
1
>>> g.collapsed_index("c")
2
>>> g.collapsed_index("d")
0
>>> assert all([g.collapsed_index(n) < g.num_nodes() for n in g.nodes()])
```

class pybnesian.**UndirectedGraph**

Undirected graph.

static Complete(*nodes: List[str]*) → *pybnesian.UndirectedGraph*

Creates a complete `UndirectedGraph` with the specified nodes.

Parameters
nodes – Nodes of the `UndirectedGraph`.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.UndirectedGraph) -> None

Creates a `UndirectedGraph` without nodes or edges.

2. __init__(self: pybnesian.UndirectedGraph, nodes: List[str]) -> None

Creates an `UndirectedGraph` with the specified nodes and without edges.

Parameters
nodes – Nodes of the `UndirectedGraph`.

3. __init__(self: pybnesian.UndirectedGraph, edges: List[Tuple[str, str]]) -> None

Creates an `UndirectedGraph` with the specified edges (the nodes are extracted from the edges).

Parameters
edges – Edges of the `UndirectedGraph`.

4. __init__(self: pybnesian.UndirectedGraph, nodes: List[str], edges: List[Tuple[str, str]]) -> None

Creates an `UndirectedGraph` with the specified nodes and edges.

Parameters

- **nodes** – Nodes of the `UndirectedGraph`.
- **edges** – Edges of the `UndirectedGraph`.

add_edge(*self:* pybnesian.UndirectedGraph, *n1: int or str*, *n2: int or str*) → None

Adds an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

add_node(*self:* pybnesian.UndirectedGraph, *node: str*) → int

Adds a node to the graph and returns its index.

Parameters
node – Name of the new node.

Returns
Index of the new node.

collapsed_from_index(*self: pybnesian.UndirectedGraph*, *index: int*) → int

Gets the collapsed index of a node from its index.

Parameters
`index` – Index of the node.

Returns
Collapsed index of the node.

collapsed_index(*self: pybnesian.UndirectedGraph*, *node: str*) → int

Gets the collapsed index of a node from its name.

Parameters
`node` – Name of the node.

Returns
Collapsed index of the node.

collapsed_indices(*self: pybnesian.UndirectedGraph*) → Dict[str, int]

Gets the collapsed indices in the graph.

Returns
A dictionary with the collapsed index of each node.

collapsed_name(*self: pybnesian.UndirectedGraph*, *collapsed_index: int*) → str

Gets the name of a node from its collapsed index.

Parameters
`collapsed_index` – Collapsed index of the node.

Returns
Name of the node.

conditional_graph(**args*, ***kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.UndirectedGraph) -> pybnesian.ConditionalUndirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.UndirectedGraph, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalUndirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters

- `nodes` – The nodes for the new conditional graph.

- **interface_nodes** – The interface nodes for the new conditional graph.

Returns
The conditional graph transformation of `self`.

contains_node(*self: pybnesian.UndirectedGraph*, *node: str*) → bool

Tests whether the node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the node, False otherwise.

edges(*self: pybnesian.UndirectedGraph*) → List[Tuple[str, str]]

Gets the list of edges.

Returns
A list of tuples (n1, n2) representing an edge between n1 and n2.

has_edge(*self: pybnesian.UndirectedGraph*, *n1: int or str*, *n2: int or str*) → bool

Checks whether an edge between the nodes `n1` and `n2` exists.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns
True if the edge exists, False otherwise.

has_path(*self: pybnesian.UndirectedGraph*, *n1: int or str*, *n2: int or str*) → bool

Checks whether there is an undirected path between nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns
True if there is an undirected path between `n1` and `n2`, False otherwise.

index(*self: pybnesian.UndirectedGraph*, *node: str*) → int

Gets the index of a node from its name.

Parameters
node – Name of the node.

Returns
Index of the node.

index_from_collapsed(*self: pybnesian.UndirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Index of the node.

`indices`(*self: pybnesian.UndirectedGraph*) → Dict[str, int]

Gets all the indices in the graph.

Returns
A dictionary with the index of each node.

`is_valid`(*self: pybnesian.UndirectedGraph*, *index: int*) → bool

Checks whether a index is a valid index (the node is not removed). All the valid indices are always returned by `indices()`.

Parameters
index – Index of the node.

Returns
True if the index is valid, False otherwise.

`name`(*self: pybnesian.UndirectedGraph*, *index: int*) → str

Gets the name of a node from its index.

Parameters
index – Index of the node.

Returns
Name of the node.

`neighbors`(*self: pybnesian.UndirectedGraph*, *node: int or str*) → List[str]

Gets the neighbors (adjacent nodes by an edge) of a node.

Parameters
node – A node name or index.

Returns
Neighbor names.

`nodes`(*self: pybnesian.UndirectedGraph*) → List[str]

Gets the nodes of the graph.

Returns
Nodes of the graph.

`num_edges`(*self: pybnesian.UndirectedGraph*) → int

Gets the number of edges.

Returns
Number of edges.

`num_neighbors`(*self: pybnesian.UndirectedGraph*, *node: int or str*) → int

Gets the number of neighbors (adjacent nodes by an edge) of a node.

Parameters
node – A node name or index.

Returns
Number of neighbors.

`num_nodes`(*self: pybnesian.UndirectedGraph*) → int

Gets the number of nodes.

Returns
Number of nodes.

remove_edge(*self: pybnesian.UndirectedGraph*, *n1: int or str*, *n2: int or str*) → None

Removes an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, but **the type of n1 and n2 must be the same**.

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

remove_node(*self: pybnesian.UndirectedGraph*, *node: int or str*) → None

Removes a node.

Parameters
node – A node name or index.

save(*self: pybnesian.UndirectedGraph*, *filename: str*) → None

Saves the graph in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

unconditional_graph(*self: pybnesian.UndirectedGraph*) → *pybnesian.UndirectedGraph*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns
The unconditional graph transformation of `self`.

class pybnesian.**DirectedGraph**

Directed graph that may contain cycles.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DirectedGraph) -> None

Creates a `DirectedGraph` without nodes or arcs.

2. __init__(self: pybnesian.DirectedGraph, nodes: List[str]) -> None

Creates a `DirectedGraph` with the specified nodes and without arcs.

Parameters
nodes – Nodes of the `DirectedGraph`.

3. __init__(self: pybnesian.DirectedGraph, arcs: List[Tuple[str, str]]) -> None

Creates a `DirectedGraph` with the specified arcs (the nodes are extracted from the arcs).

Parameters
arcs – Arcs of the `DirectedGraph`.

4. __init__(self: pybnesian.DirectedGraph, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Creates a `DirectedGraph` with the specified nodes and arcs.

Parameters

- **nodes** – Nodes of the `DirectedGraph`.
- **arcs** – Arcs of the `DirectedGraph`.

add_arc(*self: pybnesian.DirectedGraph*, *source: int or str*, *target: int or str*) → None

Adds an arc between the nodes `source` and `target`. If the arc already exists, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

add_node(*self: pybnesian.DirectedGraph*, *node: str*) → int

Adds a node to the graph and returns its index.

Parameters
node – Name of the new node.

Returns
Index of the new node.

arcs(*self: pybnesian.DirectedGraph*) → List[Tuple[str, str]]

Gets the list of arcs.

Returns
A list of tuples (source, target) representing an arc source -> target.

children(*self: pybnesian.DirectedGraph*, *node: int or str*) → List[str]

Gets the children nodes of a node.

Parameters
node – A node name or index.

Returns
Children node names.

collapsed_from_index(*self: pybnesian.DirectedGraph*, *index: int*) → int

Gets the collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Collapsed index of the node.

collapsed_index(*self: pybnesian.DirectedGraph*, *node: str*) → int

Gets the collapsed index of a node from its name.

Parameters
node – Name of the node.

Returns
Collapsed index of the node.

collapsed_indices(*self: pybnesian.DirectedGraph*) → Dict[str, int]

Gets the collapsed indices in the graph.

Returns
A dictionary with the collapsed index of each node.

collapsed_name(*self: pybnesian.DirectedGraph, collapsed_index: int*) → str

Gets the name of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Name of the node.

conditional_graph(**args, **kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.DirectedGraph) -> pybnesian.ConditionalDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.DirectedGraph, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters

- **nodes** – The nodes for the new conditional graph.
- **interface_nodes** – The interface nodes for the new conditional graph.

Returns
The conditional graph transformation of `self`.

contains_node(*self: pybnesian.DirectedGraph, node: str*) → bool

Tests whether the node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the node, False otherwise.

flip_arc(*self: pybnesian.DirectedGraph*, *source: int or str*, *target: int or str*) → None

Flips (reverses) an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

has_arc(*self: pybnesian.DirectedGraph*, *source: int or str*, *target: int or str*) → bool

Checks whether an arc between the nodes `source` and `target` exists.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns

True if the arc exists, False otherwise.

has_path(*self: pybnesian.DirectedGraph*, *n1: int or str*, *n2: int or str*) → bool

Checks whether there is a directed path between nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns

True if there is an directed path between `n1` and `n2`, False otherwise.

index(*self: pybnesian.DirectedGraph*, *node: str*) → int

Gets the index of a node from its name.

Parameters

node – Name of the node.

Returns

Index of the node.

index_from_collapsed(*self: pybnesian.DirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from its collapsed index.

Parameters

collapsed_index – Collapsed index of the node.

Returns

Index of the node.

indices(*self: pybnesian.DirectedGraph*) → Dict[str, int]

Gets all the indices in the graph.

Returns

A dictionary with the index of each node.

`is_leaf`(*self:* pybnesian.DirectedGraph, *node: int or str*) → bool

Checks whether `node` is a leaf node. A root node do not have children nodes.

Parameters
`node` – A node name or index.

Returns
True if `node` is leaf, False otherwise.

`is_root`(*self:* pybnesian.DirectedGraph, *node: int or str*) → bool

Checks whether `node` is a root node. A root node do not have parent nodes.

Parameters
`node` – A node name or index.

Returns
True if `node` is root, False otherwise.

`is_valid`(*self:* pybnesian.DirectedGraph, *index: int*) → bool

Checks whether a index is a valid index (the node is not removed). All the valid indices are always returned by `indices()`.

Parameters
`index` – Index of the node.

Returns
True if the index is valid, False otherwise.

`leaves`(*self:* pybnesian.DirectedGraph) → Set[str]

Gets the leaf nodes of the graph. A leaf node do not have children nodes.

Returns
The set of leaf nodes.

`name`(*self:* pybnesian.DirectedGraph, *index: int*) → str

Gets the name of a node from its index.

Parameters
`index` – Index of the node.

Returns
Name of the node.

`nodes`(*self:* pybnesian.DirectedGraph) → List[str]

Gets the nodes of the graph.

Returns
Nodes of the graph.

`num_arcs`(*self:* pybnesian.DirectedGraph) → int

Gets the number of arcs.

Returns
Number of arcs.

`num_children`(*self:* pybnesian.DirectedGraph, *node: int or str*) → int

Gets the number of children nodes of a node.

Parameters
`node` – A node name or index.

Returns
Number of children nodes.

num_nodes(*self*: pybnesian.DirectedGraph) → int

Gets the number of nodes.

Returns
Number of nodes.

num_parents(*self*: pybnesian.DirectedGraph, *node*: *int or str*) → int

Gets the number of parent nodes of a node.

Parameters
node – A node name or index.

Returns
Number of parent nodes.

parents(*self*: pybnesian.DirectedGraph, *node*: *int or str*) → List[str]

Gets the parent nodes of a node.

Parameters
node – A node name or index.

Returns
Parent node names.

remove_arc(*self*: pybnesian.DirectedGraph, *source*: *int or str*, *target*: *int or str*) → None

Removes an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

remove_node(*self*: pybnesian.DirectedGraph, *node*: *int or str*) → None

Removes a node.

Parameters
node – A node name or index.

roots(*self*: pybnesian.DirectedGraph) → Set[str]

Gets the root nodes of the graph. A root node do not have parent nodes.

Returns
The set of root nodes.

save(*self*: pybnesian.DirectedGraph, *filename*: *str*) → None

Saves the graph in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

unconditional_graph(*self*: pybnesian.DirectedGraph) → *pybnesian.DirectedGraph*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns
The unconditional graph transformation of `self`.

`class pybnesian.Dag`

Bases: *DirectedGraph*

Directed acyclic graph.

`__init__(*args, **kwargs)`

Overloaded function.

1. __init__(self: pybnesian.Dag) -> None

Creates a *Dag* without nodes or arcs.

2. __init__(self: pybnesian.Dag, nodes: List[str]) -> None

Creates a *Dag* with the specified nodes and without arcs.

Parameters
nodes – Nodes of the *Dag*.

3. __init__(self: pybnesian.Dag, arcs: List[Tuple[str, str]]) -> None

Creates a *Dag* with the specified arcs (the nodes are extracted from the arcs).

Parameters
arcs – Arcs of the *Dag*.

4. __init__(self: pybnesian.Dag, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Creates a *Dag* with the specified nodes and arcs.

Parameters

- **nodes** – Nodes of the *Dag*.
- **arcs** – Arcs of the *Dag*.

`add_arc`(*self: pybnesian.Dag*, *source: int or str*, *target: int or str*) → None

Adds an arc between the nodes `source` and `target`. If the arc already exists, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

`can_add_arc`(*self: pybnesian.Dag*, *source: int or str*, *target: int or str*) → bool

Checks whether an arc between the nodes `source` and `target` can be added. That is, the arc is valid and do not generate a cycle.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns
True if the arc can be added, False otherwise.

can_flip_arc(*self: pybnesian.Dag*, *source: int or str*, *target: int or str*) → bool

Checks whether an arc between the nodes `source` and `target` can be flipped. That is, the flipped arc is valid and do not generate a cycle. If the arc `source -> target` do not exist, it will return *Dag.can_add_arc()*.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns

True if the arc can be flipped, False otherwise.

conditional_graph(**args*, ***kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.Dag) -> pybnesian.ConditionalDag

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns

The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.Dag, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalDag

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters

- **nodes** – The nodes for the new conditional graph.
- **interface_nodes** – The interface nodes for the new conditional graph.

Returns

The conditional graph transformation of `self`.

flip_arc(*self: pybnesian.Dag*, *source: int or str*, *target: int or str*) → None

Flips (reverses) an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

save(*self: pybnesian.Dag*, *filename: str*) → None

Saves the graph in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

to_pdag(*self: pybnesian.Dag*) → *pybnesian.PartiallyDirectedGraph*

Gets the *PartiallyDirectedGraph* (PDAG) that represents the equivalence class of this *Dag*.

It implements the DAG-to-PDAG algorithm in [dag2pdag]. See also [dag2pdag_extra].

Returns
A *PartiallyDirectedGraph* that represents the equivalence class of this *Dag*.

topological_sort(*self: pybnesian.Dag*) → List[str]

Gets the topological sort of the DAG.

Returns
Topological sort as a list of nodes.

unconditional_graph(*self: pybnesian.Dag*) → *pybnesian.Dag*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns
The unconditional graph transformation of `self`.

class pybnesian.**PartiallyDirectedGraph**

Partially directed graph. This graph can have edges and arcs.

static CompleteUndirected(*nodes: List[str]*) → *pybnesian.PartiallyDirectedGraph*

Creates a *PartiallyDirectedGraph* that is a complete undirected graph.

Parameters
nodes – Nodes of the *PartiallyDirectedGraph*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.PartiallyDirectedGraph) -> None

Creates a *PartiallyDirectedGraph* without nodes, arcs and edges.

2. __init__(self: pybnesian.PartiallyDirectedGraph, nodes: List[str]) -> None

Creates a *PartiallyDirectedGraph* with the specified nodes and without arcs and edges.

Parameters
nodes – Nodes of the *PartiallyDirectedGraph*.

3. __init__(self: pybnesian.PartiallyDirectedGraph, arcs: List[Tuple[str, str]], edges: List[Tuple[str, str]]) -> None

Creates a *PartiallyDirectedGraph* with the specified arcs and edges (the nodes are extracted from the arcs and edges).

Parameters
- **arcs** – Arcs of the *PartiallyDirectedGraph*.

- **edges** – Edges of the *PartiallyDirectedGraph*.

4. __init__(self: pybnesian.PartiallyDirectedGraph, nodes: List[str], arcs: List[Tuple[str, str]], edges: List[Tuple[str, str]]) -> None

Creates a *PartiallyDirectedGraph* with the specified nodes and arcs.

Parameters

- **nodes** – Nodes of the *PartiallyDirectedGraph*.
- **arcs** – Arcs of the *PartiallyDirectedGraph*.
- **edges** – Edges of the *PartiallyDirectedGraph*.

add_arc(*self: pybnesian.PartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None

Adds an arc between the nodes `source` and `target`. If the arc already exists, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

add_edge(*self: pybnesian.PartiallyDirectedGraph*, *n1: int or str*, *n2: int or str*) → None

Adds an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

add_node(*self: pybnesian.PartiallyDirectedGraph*, *node: str*) → int

Adds a node to the graph and returns its index.

Parameters
node – Name of the new node.

Returns
Index of the new node.

arcs(*self: pybnesian.PartiallyDirectedGraph*) → List[Tuple[str, str]]

Gets the list of arcs.

Returns
A list of tuples (source, target) representing an arc source -> target.

children(*self: pybnesian.PartiallyDirectedGraph*, *node: int or str*) → List[str]

Gets the children nodes of a node.

Parameters
node – A node name or index.

Returns
Children node names.

collapsed_from_index(*self: pybnesian.PartiallyDirectedGraph*, *index: int*) → int

Gets the collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Collapsed index of the node.

collapsed_index(*self: pybnesian.PartiallyDirectedGraph*, *node: str*) → int

Gets the collapsed index of a node from its name.

Parameters
node – Name of the node.

Returns
Collapsed index of the node.

collapsed_indices(*self: pybnesian.PartiallyDirectedGraph*) → Dict[str, int]

Gets the collapsed indices in the graph.

Returns
A dictionary with the collapsed index of each node.

collapsed_name(*self: pybnesian.PartiallyDirectedGraph*, *collapsed_index: int*) → str

Gets the name of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Name of the node.

conditional_graph(**args*, ***kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.PartiallyDirectedGraph) -> pybnesian.ConditionalPartiallyDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.PartiallyDirectedGraph, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalPartiallyDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters
- **nodes** – The nodes for the new conditional graph.

- **interface_nodes** – The interface nodes for the new conditional graph.

Returns
The conditional graph transformation of `self`.

contains_node(*self: pybnesian.PartiallyDirectedGraph*, *node: str*) → bool
Tests whether the node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the node, False otherwise.

direct(*self: pybnesian.PartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None
Transformation to create the arc `source -> target` when possible.

- If there is an edge `source` – `target`, it is transformed into an arc `source -> target`.
- If there is an arc `target -> source`, it is flipped into an arc `source -> target`.
- Else, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters
- **source** – A node name or index.
- **target** – A node name or index.

edges(*self: pybnesian.PartiallyDirectedGraph*) → List[Tuple[str, str]]
Gets the list of edges.

Returns
A list of tuples (n1, n2) representing an edge between n1 and n2.

flip_arc(*self: pybnesian.PartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None
Flips (reverses) an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters
- **source** – A node name or index.
- **target** – A node name or index.

has_arc(*self: pybnesian.PartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → bool
Checks whether an arc between the nodes `source` and `target` exists.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters
- **source** – A node name or index.
- **target** – A node name or index.

Returns
True if the arc exists, False otherwise.

has_connection(*self: pybnesian.PartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → bool

Checks whether two nodes `source` and `target` are connected.

Two nodes `source` and `target` are connected if there is an edge `source – target`, or an arc `source -> target` or an arc `target -> source`.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns

True if `source` and `target` are connected, False otherwise.

has_edge(*self: pybnesian.PartiallyDirectedGraph*, *n1: int or str*, *n2: int or str*) → bool

Checks whether an edge between the nodes `n1` and `n2` exists.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns

True if the edge exists, False otherwise.

index(*self: pybnesian.PartiallyDirectedGraph*, *node: str*) → int

Gets the index of a node from its name.

Parameters

node – Name of the node.

Returns

Index of the node.

index_from_collapsed(*self: pybnesian.PartiallyDirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from its collapsed index.

Parameters

collapsed_index – Collapsed index of the node.

Returns

Index of the node.

indices(*self: pybnesian.PartiallyDirectedGraph*) → Dict[str, int]

Gets all the indices in the graph.

Returns

A dictionary with the index of each node.

is_leaf(*self: pybnesian.PartiallyDirectedGraph*, *node: int or str*) → bool

Checks whether `node` is a leaf node. A root node do not have children nodes.

Parameters

node – A node name or index.

Returns

True if `node` is leaf, False otherwise.

is_root(*self: pybnesian.PartiallyDirectedGraph*, *node: int or str*) → bool

Checks whether `node` is a root node. A root node do not have parent nodes.

Parameters
node – A node name or index.

Returns
True if `node` is root, False otherwise.

is_valid(*self: pybnesian.PartiallyDirectedGraph*, *index: int*) → bool

Checks whether a index is a valid index (the node is not removed). All the valid indices are always returned by `indices()`.

Parameters
index – Index of the node.

Returns
True if the index is valid, False otherwise.

leaves(*self: pybnesian.PartiallyDirectedGraph*) → Set[str]

Gets the leaf nodes of the graph. A leaf node do not have children nodes.

Returns
The set of leaf nodes.

name(*self: pybnesian.PartiallyDirectedGraph*, *index: int*) → str

Gets the name of a node from its index.

Parameters
index – Index of the node.

Returns
Name of the node.

neighbors(*self: pybnesian.PartiallyDirectedGraph*, *node: int or str*) → List[str]

Gets the neighbors (adjacent nodes by an edge) of a node.

Parameters
node – A node name or index.

Returns
Neighbor names.

nodes(*self: pybnesian.PartiallyDirectedGraph*) → List[str]

Gets the nodes of the graph.

Returns
Nodes of the graph.

num_arcs(*self: pybnesian.PartiallyDirectedGraph*) → int

Gets the number of arcs.

Returns
Number of arcs.

num_children(*self: pybnesian.PartiallyDirectedGraph*, *node: int or str*) → int

Gets the number of children nodes of a node.

Parameters
node – A node name or index.

Returns
Number of children nodes.

num_edges(*self:* pybnesian.PartiallyDirectedGraph) → int

Gets the number of edges.

Returns
Number of edges.

num_neighbors(*self:* pybnesian.PartiallyDirectedGraph, *node: int or str*) → int

Gets the number of neighbors (adjacent nodes by an edge) of a node.

Parameters
node – A node name or index.

Returns
Number of neighbors.

num_nodes(*self:* pybnesian.PartiallyDirectedGraph) → int

Gets the number of nodes.

Returns
Number of nodes.

num_parents(*self:* pybnesian.PartiallyDirectedGraph, *node: int or str*) → int

Gets the number of parent nodes of a node.

Parameters
node – A node name or index.

Returns
Number of parent nodes.

parents(*self:* pybnesian.PartiallyDirectedGraph, *node: int or str*) → List[str]

Gets the parent nodes of a node.

Parameters
node – A node name or index.

Returns
Parent node names.

remove_arc(*self:* pybnesian.PartiallyDirectedGraph, *source: int or str*, *target: int or str*) → None

Removes an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

remove_edge(*self:* pybnesian.PartiallyDirectedGraph, *n1: int or str*, *n2: int or str*) → None

Removes an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, but **the type of n1 and n2 must be the same**.

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

remove_node(*self: pybnesian.PartiallyDirectedGraph*, *node: int or str*) → None

Removes a node.

Parameters
node – A node name or index.

roots(*self: pybnesian.PartiallyDirectedGraph*) → Set[str]

Gets the root nodes of the graph. A root node do not have parent nodes.

Returns
The set of root nodes.

save(*self: pybnesian.PartiallyDirectedGraph*, *filename: str*) → None

Saves the graph in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

to_approximate_dag(*self: pybnesian.PartiallyDirectedGraph*) → *pybnesian.Dag*

Gets a `Dag` approximate extension of `self`. This method can be useful when `PartiallyDirectedGraph.to_dag()` cannot return a valid extension.

The algorithm is based on generating a topological sort which tries to preserve a similar structure.

Returns
A `Dag` approximate extension of `self`.

to_dag(*self: pybnesian.PartiallyDirectedGraph*) → *pybnesian.Dag*

Gets a `Dag` which belongs to the equivalence class of `self`.

It implements the algorithm in [pdag2dag].

Returns
A `Dag` which belongs to the equivalence class of `self`.

Raises
ValueError – If `self` do not have a valid DAG extension.

unconditional_graph(*self: pybnesian.PartiallyDirectedGraph*) → *pybnesian.PartiallyDirectedGraph*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns
The unconditional graph transformation of `self`.

undirect(*self: pybnesian.PartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None

Transformation to create the edge `source` – `target` when possible.

- If there is not an arc `target -> source`, converts the arc `source -> target` into an edge `source` – `target`. If there is not an arc `source -> target`, it adds the edge `source` – `target`.
- Else, the graph is left unaffected

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

3.2.2 Conditional Graphs

A conditional graph is the underlying graph in a conditional Bayesian networks ([PGM], Section 5.6). In a conditional Bayesian network, only the normal nodes can have a conditional probability density, while the interface nodes are always observed. A conditional graph splits all the nodes in two subsets: normal nodes and interface nodes. In a conditional graph, the interface nodes cannot have parents.

In a conditional graph, normal nodes can be returned with `nodes()`, the interface nodes with `interface_nodes()` and the joint set of nodes with `joint_nodes()`. Also, there are many other functions that have the prefix `interface` and `joint` to denote the interface and joint sets of nodes. Among them, there is a collapsed index version for only interface nodes, `interface_collapsed_index()`, and the joint set of nodes, `joint_collapsed_index()`. Note that the collapsed index for each set of nodes is independent.

class pybnesian.**ConditionalUndirectedGraph**

Conditional undirected graph.

static Complete(*nodes: List[str]*, *interface_nodes: List[str]*) → *pybnesian.ConditionalUndirectedGraph*

Creates a complete `ConditionalUndirectedGraph` with the specified nodes. A complete conditional undirected graph connects every pair of nodes with an edge, except for pairs of interface nodes.

Parameters

- **nodes** – Nodes of the `ConditionalUndirectedGraph`.
- **interface_nodes** – Interface nodes of the `ConditionalUndirectedGraph`.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalUndirectedGraph) -> None

Creates a `ConditionalUndirectedGraph` without nodes or edges.

2. __init__(self: pybnesian.ConditionalUndirectedGraph, nodes: List[str], interface_nodes: List[str]) -> None

Creates a `ConditionalUndirectedGraph` with the specified nodes, interface_nodes and without edges.

Parameters

- **nodes** – Nodes of the `ConditionalUndirectedGraph`.
- **interface_nodes** – Interface nodes of the `ConditionalUndirectedGraph`.

3. __init__(self: pybnesian.ConditionalUndirectedGraph, nodes: List[str], interface_nodes: List[str], edges: List[Tuple[str, str]]) -> None

Creates a `ConditionalUndirectedGraph` with the specified nodes, interface_nodes and edges.

Parameters

- **nodes** – Nodes of the `ConditionalUndirectedGraph`.
- **interface_nodes** – Interface nodes of the `ConditionalUndirectedGraph`.
- **edges** – Edges of the `ConditionalUndirectedGraph`.

add_edge(*self: pybnesian.ConditionalUndirectedGraph*, *n1: int or str*, *n2: int or str*) → None

Adds an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- `n1` – A node name or index.
- `n2` – A node name or index.

`add_interface_node`(*self:* pybnesian.ConditionalUndirectedGraph, *node:* *str*) → int

Adds an interface node to the graph and returns its index.

Parameters
`node` – Name of the new interface node.

Returns
Index of the new interface node.

`add_node`(*self:* pybnesian.ConditionalUndirectedGraph, *node:* *str*) → int

Adds a node to the graph and returns its index.

Parameters
`node` – Name of the new node.

Returns
Index of the new node.

`collapsed_from_index`(*self:* pybnesian.ConditionalUndirectedGraph, *index:* *int*) → int

Gets the collapsed index of a node from its index.

Parameters
`index` – Index of the node.

Returns
Collapsed index of the node.

`collapsed_index`(*self:* pybnesian.ConditionalUndirectedGraph, *node:* *str*) → int

Gets the collapsed index of a node from its name.

Parameters
`node` – Name of the node.

Returns
Collapsed index of the node.

`collapsed_indices`(*self:* pybnesian.ConditionalUndirectedGraph) → Dict[str, int]

Gets all the collapsed indices for the nodes in the graph.

Returns
A dictionary with the collapsed index of each node.

`collapsed_name`(*self:* pybnesian.ConditionalUndirectedGraph, *collapsed_index:* *int*) → str

Gets the name of a node from its collapsed index.

Parameters
`collapsed_index` – Collapsed index of the node.

Returns
Name of the node.

`conditional_graph`(**args*, ***kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.ConditionalUndirectedGraph) -> pybnesian.ConditionalUndirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.ConditionalUndirectedGraph, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalUndirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters

- **nodes** – The nodes for the new conditional graph.
- **interface_nodes** – The interface nodes for the new conditional graph.

Returns
The conditional graph transformation of `self`.

contains_interface_node(*self: pybnesian.ConditionalUndirectedGraph*, *node: str*) → bool

Tests whether the interface node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the interface node, False otherwise.

contains_joint_node(*self: pybnesian.ConditionalUndirectedGraph*, *node: str*) → bool

Tests whether the node is in the joint set of nodes or not.

Parameters
node – Name of the node.

Returns
True if the node is in the joint set of nodes, False otherwise.

contains_node(*self: pybnesian.ConditionalUndirectedGraph*, *node: str*) → bool

Tests whether the node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the node, False otherwise.

edges(*self: pybnesian.ConditionalUndirectedGraph*) → List[Tuple[str, str]]

Gets the list of edges.

Returns
A list of tuples (n1, n2) representing an edge between n1 and n2.

has_edge(*self:* pybnesian.ConditionalUndirectedGraph, *n1: int or str*, *n2: int or str*) → bool

Checks whether an edge between the nodes `n1` and `n2` exists.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns
True if the edge exists, False otherwise.

has_path(*self:* pybnesian.ConditionalUndirectedGraph, *n1: int or str*, *n2: int or str*) → bool

Checks whether there is an undirected path between nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns
True if there is an undirected path between `n1` and `n2`, False otherwise.

index(*self:* pybnesian.ConditionalUndirectedGraph, *node: str*) → int

Gets the index of a node from its name.

Parameters
node – Name of the node.

Returns
Index of the node.

index_from_collapsed(*self:* pybnesian.ConditionalUndirectedGraph, *collapsed_index: int*) → int

Gets the index of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Index of the node.

index_from_interface_collapsed(*self:* pybnesian.ConditionalUndirectedGraph, *collapsed_index: int*) → int

Gets the index of a node from the interface collapsed index.

Parameters
collapsed_index – Interface collapsed index of the node.

Returns
Index of the node.

index_from_joint_collapsed(*self:* pybnesian.ConditionalUndirectedGraph, *collapsed_index: int*) → int

Gets the index of a node from the joint collapsed index.

Parameters
collapsed_index – Joint collapsed index of the node.

Returns
Index of the node.

indices(*self: pybnesian.ConditionalUndirectedGraph*) → Dict[str, int]

Gets all the indices for the nodes in the graph.

Returns
A dictionary with the index of each node.

interface_collapsed_from_index(*self: pybnesian.ConditionalUndirectedGraph, index: int*) → int

Gets the interface collapsed index of a node from its index.

Parameters
`index` – Index of the node.

Returns
Interface collapsed index of the node.

interface_collapsed_index(*self: pybnesian.ConditionalUndirectedGraph, node: str*) → int

Gets the interface collapsed index of an interface node from its name.

Parameters
`node` – Name of the interface node.

Returns
Interface collapsed index of the interface node.

interface_collapsed_indices(*self: pybnesian.ConditionalUndirectedGraph*) → Dict[str, int]

Gets all the interface collapsed indices for the interface nodes in the graph.

Returns
A dictionary with the interface collapsed index of each interface node.

interface_collapsed_name(*self: pybnesian.ConditionalUndirectedGraph, collapsed_index: int*) → str

Gets the name of an interface node from its collapsed index.

Parameters
`collapsed_index` – Collapsed index of the interface node.

Returns
Name of the interface node.

interface_edges(*self: pybnesian.ConditionalUndirectedGraph*) → List[Tuple[str, str]]

Gets the edges where one of the nodes is an interface node.

Returns
edges as a list of tuples (inode, node), where `inode` is an interface node and `node` is a normal node.

interface_nodes(*self: pybnesian.ConditionalUndirectedGraph*) → List[str]

Gets the interface nodes of the graph.

Returns
Interface nodes of the graph.

is_interface(*self: pybnesian.ConditionalUndirectedGraph, node: int or str*) → bool

Checks whether the `node` is an interface node.

Parameters
`node` – A node name or index.

Returns
True if `node` is interface node, False, otherwise.

is_valid(*self: pybnesian.ConditionalUndirectedGraph*, *index: int*) → bool

Checks whether a index is a valid index (the node is not removed). All the valid indices are always returned by `indices()`.

Parameters
index – Index of the node.

Returns
True if the index is valid, False otherwise.

joint_collapsed_from_index(*self: pybnesian.ConditionalUndirectedGraph*, *index: int*) → int

Gets the joint collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Joint collapsed index of the node.

joint_collapsed_index(*self: pybnesian.ConditionalUndirectedGraph*, *node: str*) → int

Gets the joint collapsed index of a node from its name.

Parameters
node – Name of the node.

Returns
Joint collapsed index of the node.

joint_collapsed_indices(*self: pybnesian.ConditionalUndirectedGraph*) → Dict[str, int]

Gets all the joint collapsed indices for the joint set of nodes in the graph.

Returns
A dictionary with the joint collapsed index of each joint node.

joint_collapsed_name(*self: pybnesian.ConditionalUndirectedGraph*, *collapsed_index: int*) → str

Gets the name of a node from its joint collapsed index.

Parameters
collapsed_index – Joint collapsed index of the node.

Returns
Name of the node.

joint_nodes(*self: pybnesian.ConditionalUndirectedGraph*) → List[str]

Gets the joint set of nodes of the graph.

Returns
Joint set of nodes of the graph.

name(*self: pybnesian.ConditionalUndirectedGraph*, *index: int*) → str

Gets the name of a node from its index.

Parameters
index – Index of the node.

Returns
Name of the node.

neighbors(*self: pybnesian.ConditionalUndirectedGraph*, *node: int or str*) → List[str]

Gets the neighbors (adjacent nodes by an edge) of a node.

Parameters
node – A node name or index.

Returns
Neighbor names.

nodes(*self:* pybnesian.ConditionalUndirectedGraph) → List[str]

Gets the nodes of the graph.

Returns
Nodes of the graph.

num_edges(*self:* pybnesian.ConditionalUndirectedGraph) → int

Gets the number of edges.

Returns
Number of edges.

num_interface_nodes(*self:* pybnesian.ConditionalUndirectedGraph) → int

Gets the number of interface nodes.

Returns
Number of interface nodes.

num_joint_nodes(*self:* pybnesian.ConditionalUndirectedGraph) → int

Gets the number of joint nodes. That is, `num_nodes() + num_interface_nodes()`

Returns
Number of joint nodes.

num_neighbors(*self:* pybnesian.ConditionalUndirectedGraph, *node: int or str*) → int

Gets the number of neighbors (adjacent nodes by an edge) of a node.

Parameters
node – A node name or index.

Returns
Number of neighbors.

num_nodes(*self:* pybnesian.ConditionalUndirectedGraph) → int

Gets the number of nodes.

Returns
Number of nodes.

remove_edge(*self:* pybnesian.ConditionalUndirectedGraph, *n1: int or str*, *n2: int or str*) → None

Removes an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, but **the type of n1 and n2 must be the same**.

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

remove_interface_node(*self:* pybnesian.ConditionalUndirectedGraph, *node: int or str*) → None

Removes an interface node.

Parameters
node – A node name or index.

remove_node(*self:* pybnesian.ConditionalUndirectedGraph, *node: int or str*) → None

Removes a node.

Parameters

node – A node name or index.

save(*self:* pybnesian.ConditionalUndirectedGraph, *filename: str*) → None

Saves the graph in a pickle file with the given name.

Parameters

filename – File name of the saved graph.

set_interface(*self:* pybnesian.ConditionalUndirectedGraph, *node: int or str*) → None

Converts a normal node into an interface node.

Parameters

node – A node name or index.

set_node(*self:* pybnesian.ConditionalUndirectedGraph, *node: int or str*) → None

Converts an interface node into a normal node.

Parameters

node – A node name or index.

unconditional_graph(*self:* pybnesian.ConditionalUndirectedGraph) → *pybnesian.UndirectedGraph*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns

The unconditional graph transformation of `self`.

class pybnesian.**ConditionalDirectedGraph**

Conditional directed graph.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalDirectedGraph) -> None

Creates a *ConditionalDirectedGraph* without nodes or arcs.

2. __init__(self: pybnesian.ConditionalDirectedGraph, nodes: List[str], interface_nodes: List[str]) -> None

Creates a *ConditionalDirectedGraph* with the specified nodes, interface_nodes and without arcs.

Parameters

- **nodes** – Nodes of the *ConditionalDirectedGraph*.
- **interface_nodes** – Interface nodes of the *ConditionalDirectedGraph*.

3. __init__(self: pybnesian.ConditionalDirectedGraph, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Creates a *ConditionalDirectedGraph* with the specified nodes and arcs.

Parameters

- **nodes** – Nodes of the *ConditionalDirectedGraph*.
- **interface_nodes** – Interface nodes of the *ConditionalDirectedGraph*.
- **arcs** – Arcs of the *ConditionalDirectedGraph*.

add_arc(*self: pybnesian.ConditionalDirectedGraph*, *source: int or str*, *target: int or str*) → None

Adds an arc between the nodes `source` and `target`. If the arc already exists, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

add_interface_node(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → int

Adds an interface node to the graph and returns its index.

Parameters
node – Name of the new interface node.

Returns
Index of the new interface node.

add_node(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → int

Adds a node to the graph and returns its index.

Parameters
node – Name of the new node.

Returns
Index of the new node.

arcs(*self: pybnesian.ConditionalDirectedGraph*) → List[Tuple[str, str]]

Gets the list of arcs.

Returns
A list of tuples (source, target) representing an arc source -> target.

children(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → List[str]

Gets the children nodes of a node.

Parameters
node – A node name or index.

Returns
Children node names.

collapsed_from_index(*self: pybnesian.ConditionalDirectedGraph*, *index: int*) → int

Gets the collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Collapsed index of the node.

collapsed_index(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → int

Gets the collapsed index of a node from its name.

Parameters
node – Name of the node.

Returns
Collapsed index of the node.

collapsed_indices(*self: pybnesian.ConditionalDirectedGraph*) → Dict[str, int]

Gets all the collapsed indices for the nodes in the graph.

Returns
A dictionary with the collapsed index of each node.

collapsed_name(*self: pybnesian.ConditionalDirectedGraph*, *collapsed_index: int*) → str

Gets the name of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Name of the node.

conditional_graph(**args*, ***kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.ConditionalDirectedGraph) -> pybnesian.ConditionalDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.ConditionalDirectedGraph, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters

- **nodes** – The nodes for the new conditional graph.
- **interface_nodes** – The interface nodes for the new conditional graph.

Returns
The conditional graph transformation of `self`.

contains_interface_node(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → bool

Tests whether the interface node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the interface node, False otherwise.

contains_joint_node(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → bool

Tests whether the node is in the joint set of nodes or not.

Parameters
node – Name of the node.

Returns
True if the node is in the joint set of nodes, False otherwise.

contains_node(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → bool

Tests whether the node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the node, False otherwise.

flip_arc(*self: pybnesian.ConditionalDirectedGraph*, *source: int or str*, *target: int or str*) → None

Flips (reverses) an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

has_arc(*self: pybnesian.ConditionalDirectedGraph*, *source: int or str*, *target: int or str*) → bool

Checks whether an arc between the nodes `source` and `target` exists.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns
True if the arc exists, False otherwise.

has_path(*self: pybnesian.ConditionalDirectedGraph*, *n1: int or str*, *n2: int or str*) → bool

Checks whether there is a directed path between nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns
True if there is an directed path between `n1` and `n2`, False otherwise.

index(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → int

Gets the index of a node from its name.

Parameters
node – Name of the node.

Returns
Index of the node.

index_from_collapsed(*self: pybnesian.ConditionalDirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Index of the node.

index_from_interface_collapsed(*self: pybnesian.ConditionalDirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from the interface collapsed index.

Parameters
collapsed_index – Interface collapsed index of the node.

Returns
Index of the node.

index_from_joint_collapsed(*self: pybnesian.ConditionalDirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from the joint collapsed index.

Parameters
collapsed_index – Joint collapsed index of the node.

Returns
Index of the node.

indices(*self: pybnesian.ConditionalDirectedGraph*) → Dict[str, int]

Gets all the indices for the nodes in the graph.

Returns
A dictionary with the index of each node.

interface_arcs(*self: pybnesian.ConditionalDirectedGraph*) → List[Tuple[str, str]]

Gets the arcs where the source node is an interface node.

Returns
arcs with an interface node as source node.

interface_collapsed_from_index(*self: pybnesian.ConditionalDirectedGraph*, *index: int*) → int

Gets the interface collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Interface collapsed index of the node.

interface_collapsed_index(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → int

Gets the interface collapsed index of an interface node from its name.

Parameters
node – Name of the interface node.

Returns
Interface collapsed index of the interface node.

interface_collapsed_indices(*self: pybnesian.ConditionalDirectedGraph*) → Dict[str, int]

Gets all the interface collapsed indices for the interface nodes in the graph.

Returns
A dictionary with the interface collapsed index of each interface node.

interface_collapsed_name(*self: pybnesian.ConditionalDirectedGraph*, *collapsed_index: int*) → str

Gets the name of an interface node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the interface node.

Returns
Name of the interface node.

interface_nodes(*self: pybnesian.ConditionalDirectedGraph*) → List[str]

Gets the interface nodes of the graph.

Returns
Interface nodes of the graph.

is_interface(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → bool

Checks whether the `node` is an interface node.

Parameters
node – A node name or index.

Returns
True if `node` is interface node, False, otherwise.

is_leaf(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → bool

Checks whether `node` is a leaf node. A root node do not have children nodes.

Parameters
node – A node name or index.

Returns
True if `node` is leaf, False otherwise.

is_root(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → bool

Checks whether `node` is a root node. A root node do not have parent nodes.

This implementation do not take into account the interface arcs. That is, if a node only have interface nodes as parents, it is considered a root.

Parameters
node – A node name or index.

Returns
True if `node` is root, False otherwise.

is_valid(*self: pybnesian.ConditionalDirectedGraph*, *index: int*) → bool

Checks whether a index is a valid index (the node is not removed). All the valid indices are always returned by `indices()`.

Parameters
index – Index of the node.

Returns
True if the index is valid, False otherwise.

joint_collapsed_from_index(*self: pybnesian.ConditionalDirectedGraph*, *index: int*) → int

Gets the joint collapsed index of a node from its index.

Parameters
`index` – Index of the node.

Returns
Joint collapsed index of the node.

joint_collapsed_index(*self: pybnesian.ConditionalDirectedGraph*, *node: str*) → int

Gets the joint collapsed index of a node from its name.

Parameters
`node` – Name of the node.

Returns
Joint collapsed index of the node.

joint_collapsed_indices(*self: pybnesian.ConditionalDirectedGraph*) → Dict[str, int]

Gets all the joint collapsed indices for the joint set of nodes in the graph.

Returns
A dictionary with the joint collapsed index of each joint node.

joint_collapsed_name(*self: pybnesian.ConditionalDirectedGraph*, *collapsed_index: int*) → str

Gets the name of a node from its joint collapsed index.

Parameters
`collapsed_index` – Joint collapsed index of the node.

Returns
Name of the node.

joint_nodes(*self: pybnesian.ConditionalDirectedGraph*) → List[str]

Gets the joint set of nodes of the graph.

Returns
Joint set of nodes of the graph.

leaves(*self: pybnesian.ConditionalDirectedGraph*) → Set[str]

Gets the leaf nodes of the graph. A leaf node do not have children nodes.

This implementation do not include the interface nodes in the result. Thus, this returns the same result as an unconditional graph without the interface nodes.

Returns
The set of leaf nodes.

name(*self: pybnesian.ConditionalDirectedGraph*, *index: int*) → str

Gets the name of a node from its index.

Parameters
`index` – Index of the node.

Returns
Name of the node.

nodes(*self: pybnesian.ConditionalDirectedGraph*) → List[str]

Gets the nodes of the graph.

Returns
Nodes of the graph.

num_arcs(*self: pybnesian.ConditionalDirectedGraph*) → int

Gets the number of arcs.

Returns
Number of arcs.

num_children(*self: pybnesian.ConditionalDirectedGraph, node: int or str*) → int

Gets the number of children nodes of a node.

Parameters
node – A node name or index.

Returns
Number of children nodes.

num_interface_nodes(*self: pybnesian.ConditionalDirectedGraph*) → int

Gets the number of interface nodes.

Returns
Number of interface nodes.

num_joint_nodes(*self: pybnesian.ConditionalDirectedGraph*) → int

Gets the number of joint nodes. That is, `num_nodes() + num_interface_nodes()`

Returns
Number of joint nodes.

num_nodes(*self: pybnesian.ConditionalDirectedGraph*) → int

Gets the number of nodes.

Returns
Number of nodes.

num_parents(*self: pybnesian.ConditionalDirectedGraph, node: int or str*) → int

Gets the number of parent nodes of a node.

Parameters
node – A node name or index.

Returns
Number of parent nodes.

parents(*self: pybnesian.ConditionalDirectedGraph, node: int or str*) → List[str]

Gets the parent nodes of a node.

Parameters
node – A node name or index.

Returns
Parent node names.

remove_arc(*self: pybnesian.ConditionalDirectedGraph, source: int or str, target: int or str*) → None

Removes an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

remove_interface_node(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → None

Removes an interface node.

Parameters
node – A node name or index.

remove_node(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → None

Removes a node.

Parameters
node – A node name or index.

roots(*self: pybnesian.ConditionalDirectedGraph*) → Set[str]

Gets the root nodes of the graph. A root node do not have parent nodes.

This implementation do not include the interface nodes in the result. Also, do not take into account the interface arcs. That is, if a node only have interface nodes as parents, it is considered a root. Thus, this returns the same result as an unconditional graph without the interface nodes.

Returns
The set of root nodes.

save(*self: pybnesian.ConditionalDirectedGraph*, *filename: str*) → None

Saves the graph in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

set_interface(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → None

Converts a normal node into an interface node.

Parameters
node – A node name or index.

set_node(*self: pybnesian.ConditionalDirectedGraph*, *node: int or str*) → None

Converts an interface node into a normal node.

Parameters
node – A node name or index.

unconditional_graph(*self: pybnesian.ConditionalDirectedGraph*) → *pybnesian.DirectedGraph*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns
The unconditional graph transformation of `self`.

class pybnesian.**ConditionalDag**

Bases: *ConditionalDirectedGraph*

Conditional directed acyclic graph.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalDag) -> None

Creates a *ConditionalDag* without nodes or arcs.

2. __init__(self: pybnesian.ConditionalDag, nodes: List[str], interface_nodes: List[str]) -> None

Creates a *ConditionalDag* with the specified nodes, interface_nodes and without arcs.

Parameters

- **nodes** – Nodes of the *ConditionalDag*.
- **interface_nodes** – Interface nodes of the *ConditionalDag*.

3. __init__(self: pybnesian.ConditionalDag, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Creates a *ConditionalDag* with the specified nodes, interface_nodes and arcs.

Parameters

- **nodes** – Nodes of the *ConditionalDag*.
- **interface_nodes** – Interface nod"es of the *ConditionalDag*.
- **arcs** – Arcs of the *ConditionalDag*.

add_arc(*self: pybnesian.ConditionalDag*, *source: int or str*, *target: int or str*) → None

Adds an arc between the nodes `source` and `target`. If the arc already exists, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

can_add_arc(*self: pybnesian.ConditionalDag*, *source: int or str*, *target: int or str*) → bool

Checks whether an arc between the nodes `source` and `target` can be added. That is, the arc is valid and do not generate a cycle or connects two interface nodes.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns

True if the arc can be added, False otherwise.

can_flip_arc(*self: pybnesian.ConditionalDag*, *source: int or str*, *target: int or str*) → bool

Checks whether an arc between the nodes `source` and `target` can be flipped. That is, the flipped arc is valid and do not generate a cycle. If the arc `source -> target` do not exist, it will return *ConditionalDag.can_add_arc()*.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns

True if the arc can be flipped, False otherwise.

conditional_graph(**args*, ***kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.ConditionalDag) -> pybnesian.ConditionalDag

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.ConditionalDag, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalDag

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters

- **nodes** – The nodes for the new conditional graph.
- **interface_nodes** – The interface nodes for the new conditional graph.

Returns
The conditional graph transformation of `self`.

flip_arc(*self: pybnesian.ConditionalDag*, *source: int or str*, *target: int or str*) → None

Flips (reverses) an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

save(*self: pybnesian.ConditionalDag*, *filename: str*) → None

Saves the graph in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

to_pdag(*self: pybnesian.ConditionalDag*) → *pybnesian.ConditionalPartiallyDirectedGraph*

Gets the `ConditionalPartiallyDirectedGraph` (PDAG) that represents the equivalence class of this `ConditionalDag`.

It implements the DAG-to-PDAG algorithm in [dag2pdag]. See also [dag2pdag_extra].

Returns
A `ConditionalPartiallyDirectedGraph` that represents the equivalence class of this `ConditionalDag`.

topological_sort(*self:* pybnesian.ConditionalDag) → List[str]

Gets the topological sort of the conditional DAG. This topological sort does not include the interface nodes, since they are known to be always roots (they can be included at the very beginning of the topological sort).

Returns
Topological sort as a list of nodes.

unconditional_graph(*self:* pybnesian.ConditionalDag) → *pybnesian.Dag*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns
The unconditional graph transformation of `self`.

class pybnesian.**ConditionalPartiallyDirectedGraph**

Conditional partially directed graph. This graph can have edges and arcs, except between pairs of interface nodes.

static **CompleteUndirected**(*nodes: List[str], interface_nodes: List[str]*) → *pybnesian.ConditionalPartiallyDirectedGraph*

Creates a `ConditionalPartiallyDirectedGraph` that is a complete undirected graph. A complete conditional undirected graph connects every pair of nodes with an edge, except for pairs of interface nodes.

Parameters

- **nodes** – Nodes of the `ConditionalPartiallyDirectedGraph`.
- **interface_nodes** – Interface nodes of the `ConditionalPartiallyDirectedGraph`.

__init__(**args, **kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalPartiallyDirectedGraph) -> None

Creates a `ConditionalPartiallyDirectedGraph` without nodes or arcs.

2. __init__(self: pybnesian.ConditionalPartiallyDirectedGraph, nodes: List[str], interface_nodes: List[str]) -> None

Creates a `ConditionalPartiallyDirectedGraph` with the specified nodes, interface_nodes and without edges.

Parameters

- **nodes** – Nodes of the `ConditionalPartiallyDirectedGraph`.
- **interface_nodes** – Interface nodes of the `ConditionalPartiallyDirectedGraph`.

3. __init__(self: pybnesian.ConditionalPartiallyDirectedGraph, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]], edges: List[Tuple[str, str]]) -> None

Creates a `ConditionalPartiallyDirectedGraph` with the specified nodes and arcs.

Parameters

- **nodes** – Nodes of the `ConditionalPartiallyDirectedGraph`.
- **interface_nodes** – Interface nodes of the `ConditionalPartiallyDirectedGraph`.
- **arcs** – Arcs of the `ConditionalPartiallyDirectedGraph`.

- **edges** – Edges of the *ConditionalPartiallyDirectedGraph*.

add_arc(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None

Adds an arc between the nodes `source` and `target`. If the arc already exists, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

add_edge(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *n1: int or str*, *n2: int or str*) → None

Adds an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

add_interface_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → int

Adds an interface node to the graph and returns its index.

Parameters
node – Name of the new interface node.

Returns
Index of the new interface node.

add_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → int

Adds a node to the graph and returns its index.

Parameters
node – Name of the new node.

Returns
Index of the new node.

arcs(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → List[Tuple[str, str]]

Gets the list of arcs.

Returns
A list of tuples (source, target) representing an arc source -> target.

children(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → List[str]

Gets the children nodes of a node.

Parameters
node – A node name or index.

Returns
Children node names.

collapsed_from_index(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *index: int*) → int

Gets the collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Collapsed index of the node.

`collapsed_index`(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → int

Gets the collapsed index of a node from its name.

Parameters
node – Name of the node.

Returns
Collapsed index of the node.

`collapsed_indices`(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → Dict[str, int]

Gets all the collapsed indices for the nodes in the graph.

Returns
A dictionary with the collapsed index of each node.

`collapsed_name`(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *collapsed_index: int*) → str

Gets the name of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Name of the node.

`conditional_graph`(**args*, ***kwargs*)

Overloaded function.

1. conditional_graph(self: pybnesian.ConditionalPartiallyDirectedGraph) -> pybnesian.ConditionalPartiallyDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the same nodes and without interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_graph(self: pybnesian.ConditionalPartiallyDirectedGraph, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalPartiallyDirectedGraph

Transforms the graph to a conditional graph.

- If `self` is not conditional, it returns a conditional version of the graph with the given nodes and interface nodes.
- If `self` is conditional, it returns the same graph type with the given nodes and interface nodes.

Parameters
- **nodes** – The nodes for the new conditional graph.
- **interface_nodes** – The interface nodes for the new conditional graph.

Returns
The conditional graph transformation of `self`.

contains_interface_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → bool

Tests whether the interface node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the interface node, False otherwise.

contains_joint_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → bool

Tests whether the node is in the joint set of nodes or not.

Parameters
node – Name of the node.

Returns
True if the node is in the joint set of nodes, False otherwise.

contains_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → bool

Tests whether the node is in the graph or not.

Parameters
node – Name of the node.

Returns
True if the graph contains the node, False otherwise.

direct(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None

Transformation to create the arc `source -> target` when possible.

- If there is an edge `source` – `target`, it is transformed into an arc `source -> target`.
- If there is an arc `target -> source`, it is flipped into an arc `source -> target`.
- Else, the graph is left unaffected.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

edges(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → List[Tuple[str, str]]

Gets the list of edges.

Returns
A list of tuples (n1, n2) representing an edge between n1 and n2.

flip_arc(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None

Flips (reverses) an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

has_arc(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → bool

Checks whether an arc between the nodes `source` and `target` exists.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns

True if the arc exists, False otherwise.

has_connection(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → bool

Checks whether two nodes `source` and `target` are connected.

Two nodes `source` and `target` are connected if there is an edge `source – target`, or an arc `source -> target` or an arc `target -> source`.

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

Returns

True if `source` and `target` are connected, False otherwise.

has_edge(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *n1: int or str*, *n2: int or str*) → bool

Checks whether an edge between the nodes `n1` and `n2` exists.

`n1` and `n2` can be the name or the index, **but the type of n1 and n2 must be the same.**

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

Returns

True if the edge exists, False otherwise.

index(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → int

Gets the index of a node from its name.

Parameters

node – Name of the node.

Returns

Index of the node.

index_from_collapsed(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from its collapsed index.

Parameters

collapsed_index – Collapsed index of the node.

Returns

Index of the node.

index_from_interface_collapsed(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from the interface collapsed index.

Parameters
collapsed_index – Interface collapsed index of the node.

Returns
Index of the node.

index_from_joint_collapsed(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *collapsed_index: int*) → int

Gets the index of a node from the joint collapsed index.

Parameters
collapsed_index – Joint collapsed index of the node.

Returns
Index of the node.

indices(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → Dict[str, int]

Gets all the indices for the nodes in the graph.

Returns
A dictionary with the index of each node.

interface_arcs(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → List[Tuple[str, str]]

Gets the arcs where the source node is an interface node.

Returns
arcs with an interface node as source node.

interface_collapsed_from_index(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *index: int*) → int

Gets the interface collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Interface collapsed index of the node.

interface_collapsed_index(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: str*) → int

Gets the interface collapsed index of an interface node from its name.

Parameters
node – Name of the interface node.

Returns
Interface collapsed index of the interface node.

interface_collapsed_indices(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → Dict[str, int]

Gets all the interface collapsed indices for the interface nodes in the graph.

Returns
A dictionary with the interface collapsed index of each interface node.

interface_collapsed_name(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *collapsed_index: int*) → str

Gets the name of an interface node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the interface node.

Returns
Name of the interface node.

interface_edges(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → List[Tuple[str, str]]

Gets the edges where one of the nodes is an interface node.

Returns
edges as a list of tuples (inode, node), where `inode` is an interface node and `node` is a normal node.

interface_nodes(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → List[str]

Gets the interface nodes of the graph.

Returns
Interface nodes of the graph.

is_interface(*self: pybnesian.ConditionalPartiallyDirectedGraph, node: int or str*) → bool

Checks whether the `node` is an interface node.

Parameters
node – A node name or index.

Returns
True if `node` is interface node, False, otherwise.

is_leaf(*self: pybnesian.ConditionalPartiallyDirectedGraph, node: int or str*) → bool

Checks whether `node` is a leaf node. A root node do not have children nodes.

Parameters
node – A node name or index.

Returns
True if `node` is leaf, False otherwise.

is_root(*self: pybnesian.ConditionalPartiallyDirectedGraph, node: int or str*) → bool

Checks whether `node` is a root node. A root node do not have parent nodes.

This implementation do not take into account the interface arcs. That is, if a node only have interface nodes as parents, it is considered a root.

Parameters
node – A node name or index.

Returns
True if `node` is root, False otherwise.

is_valid(*self: pybnesian.ConditionalPartiallyDirectedGraph, index: int*) → bool

Checks whether a index is a valid index (the node is not removed). All the valid indices are always returned by `indices()`.

Parameters
index – Index of the node.

Returns
True if the index is valid, False otherwise.

joint_collapsed_from_index(*self: pybnesian.ConditionalPartiallyDirectedGraph, index: int*) → int

Gets the joint collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Joint collapsed index of the node.

`joint_collapsed_index`(*self*: pybnesian.ConditionalPartiallyDirectedGraph, *node*: *str*) → int

Gets the joint collapsed index of a node from its name.

Parameters
`node` – Name of the node.

Returns
Joint collapsed index of the node.

`joint_collapsed_indices`(*self*: pybnesian.ConditionalPartiallyDirectedGraph) → Dict[str, int]

Gets all the joint collapsed indices for the joint set of nodes in the graph.

Returns
A dictionary with the joint collapsed index of each joint node.

`joint_collapsed_name`(*self*: pybnesian.ConditionalPartiallyDirectedGraph, *collapsed_index*: *int*) → str

Gets the name of a node from its joint collapsed index.

Parameters
`collapsed_index` – Joint collapsed index of the node.

Returns
Name of the node.

`joint_nodes`(*self*: pybnesian.ConditionalPartiallyDirectedGraph) → List[str]

Gets the joint set of nodes of the graph.

Returns
Joint set of nodes of the graph.

`leaves`(*self*: pybnesian.ConditionalPartiallyDirectedGraph) → Set[str]

Gets the leaf nodes of the graph. A leaf node do not have children nodes.

This implementation do not include the interface nodes in the result. Thus, this returns the same result as an unconditional graph without the interface nodes.

Returns
The set of leaf nodes.

`name`(*self*: pybnesian.ConditionalPartiallyDirectedGraph, *index*: *int*) → str

Gets the name of a node from its index.

Parameters
`index` – Index of the node.

Returns
Name of the node.

`neighbors`(*self*: pybnesian.ConditionalPartiallyDirectedGraph, *node*: *int or str*) → List[str]

Gets the neighbors (adjacent nodes by an edge) of a node.

Parameters
`node` – A node name or index.

Returns
Neighbor names.

nodes(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → List[str]

Gets the nodes of the graph.

Returns
Nodes of the graph.

num_arcs(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → int

Gets the number of arcs.

Returns
Number of arcs.

num_children(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → int

Gets the number of children nodes of a node.

Parameters
node – A node name or index.

Returns
Number of children nodes.

num_edges(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → int

Gets the number of edges.

Returns
Number of edges.

num_interface_nodes(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → int

Gets the number of interface nodes.

Returns
Number of interface nodes.

num_joint_nodes(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → int

Gets the number of joint nodes. That is, `num_nodes() + num_interface_nodes()`

Returns
Number of joint nodes.

num_neighbors(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → int

Gets the number of neighbors (adjacent nodes by an edge) of a node.

Parameters
node – A node name or index.

Returns
Number of neighbors.

num_nodes(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → int

Gets the number of nodes.

Returns
Number of nodes.

num_parents(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → int

Gets the number of parent nodes of a node.

Parameters
node – A node name or index.

Returns
Number of parent nodes.

parents(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → List[str]

Gets the parent nodes of a node.

Parameters

node – A node name or index.

Returns

Parent node names.

remove_arc(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None

Removes an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

`source` and `target` can be the name or the index, but **the type of source and target must be the same**.

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

remove_edge(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *n1: int or str*, *n2: int or str*) → None

Removes an edge between the nodes `n1` and `n2`.

`n1` and `n2` can be the name or the index, but **the type of n1 and n2 must be the same**.

Parameters

- **n1** – A node name or index.
- **n2** – A node name or index.

remove_interface_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → None

Removes an interface node.

Parameters

node – A node name or index.

remove_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → None

Removes a node.

Parameters

node – A node name or index.

roots(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → Set[str]

Gets the root nodes of the graph. A root node do not have parent nodes.

This implementation do not include the interface nodes in the result. Also, do not take into account the interface arcs. That is, if a node only have interface nodes as parents, it is considered a root. Thus, this returns the same result as an unconditional graph without the interface nodes.

Returns

The set of root nodes.

save(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *filename: str*) → None

Saves the graph in a pickle file with the given name.

Parameters

filename – File name of the saved graph.

set_interface(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → None

Converts a normal node into an interface node.

Parameters

node – A node name or index.

set_node(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *node: int or str*) → None

Converts an interface node into a normal node.

Parameters

node – A node name or index.

to_approximate_dag(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → *pybnesian.ConditionalDag*

Gets a `Dag` approximate extension of `self`. This method can be useful when `ConditionalPartiallyDirectedGraph.to_dag()` cannot return a valid extension.

The algorithm is based on generating a topological sort which tries to preserve a similar structure.

Returns

A `Dag` approximate extension of `self`.

to_dag(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → *pybnesian.ConditionalDag*

Gets a `Dag` which belongs to the equivalence class of `self`.

It implements the algorithm in [pdag2dag].

Returns

A `Dag` which belongs to the equivalence class of `self`.

Raises

ValueError – If `self` do not have a valid DAG extension.

unconditional_graph(*self: pybnesian.ConditionalPartiallyDirectedGraph*) → *pybnesian.PartiallyDirectedGraph*

Transforms the graph to an unconditional graph.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned graph.

Returns

The unconditional graph transformation of `self`.

undirect(*self: pybnesian.ConditionalPartiallyDirectedGraph*, *source: int or str*, *target: int or str*) → None

Transformation to create the edge `source` – `target` when possible.

- If there is not an arc `target -> source`, converts the arc `source -> target` into an edge `source` – `target`. If there is not an arc `source -> target`, it adds the edge `source` – `target`.
- Else, the graph is left unaffected

`source` and `target` can be the name or the index, **but the type of source and target must be the same.**

Parameters

- **source** – A node name or index.
- **target** – A node name or index.

3.2.3 Bibliography

3.3 Factors module

The factors are usually represented as conditional probability functions and are a component of a Bayesian network.

3.3.1 Abstract Types

The `FactorType` and `Factor` classes are abstract and both of them need to be implemented to create a new factor type. Each `Factor` is always associated with a specific `FactorType`.

class pybnesian.**FactorType**

A representation of a `Factor` type.

__init__(*self: pybnesian.FactorType*) → None

Initializes a new `FactorType`

__str__(*self: pybnesian.FactorType*) → str

new_factor(*self: pybnesian.FactorType, model: BayesianNetworkBase or ConditionalBayesianNetworkBase, variable: str, evidence: List[str], *args, **kwargs*) → pybnesian.Factor

Create a new corresponding `Factor` for a `model` with the given `variable` and `evidence`.

Note that `evidence` might be different from `model.parents(variable)`.

Parameters

- **model** – The model that will contain the `Factor`.
- **variable** – Variable name.
- **evidence** – List of evidence variable names.
- **args** – Additional arguments to construct the `Factor`.
- **kwargs** – Additional keyword arguments used to construct the `Factor`.

Returns

A corresponding `Factor` with the given `variable` and `evidence`.

class pybnesian.**Factor**

__init__(*self: pybnesian.Factor, variable: str, evidence: List[str]*) → None

Initializes a new `Factor` with a given `variable` and `evidence`.

Parameters

- **variable** – Variable name.
- **evidence** – List of evidence variable names.

__str__(*self: pybnesian.Factor*) → str

data_type(*self: pybnesian.Factor*) → pyarrow.DataType

Returns the `pyarrow.DataType` that represents the type of data handled by the `Factor`.

For a continuous Factor, this usually returns `pyarrow.float64()` or `pyarrow.float32()`. The discrete factor is usually a `pyarrow.dictionary()`.

Returns

the `pyarrow.DataType` physical data type representation of the `Factor`.

evidence(*self: pybnesian.Factor*) → List[str]

Gets the evidence variable list.

Returns
Evidence variable list.

fit(*self: pybnesian.Factor*, *df: DataFrame*) → None

Fits the `Factor` with the data in `df`.

Parameters
df – DataFrame to fit the `Factor`.

fitted(*self: pybnesian.Factor*) → bool

Checks whether the factor is fitted.

Returns
True if the factor is fitted, False otherwise.

logl(*self: pybnesian.Factor*, *df: DataFrame*) → numpy.ndarray[numpy.float64[m, 1]]

Returns the log-likelihood of each instance in the DataFrame `df`.

Parameters
df – DataFrame to compute the log-likelihood.

Returns
A `numpy.ndarray` vector with dtype `numpy.float64`, where the i-th value is the log-likelihod of the i-th instance of `df`.

sample(*self: pybnesian.Factor*, *n: int*, *evidence_values: Optional[DataFrame] = None*, *seed: Optional[int] = None*) → pyarrow.Array

Samples `n` values from this `Factor`. This method returns a `pyarrow.Array` with `n` values with the same type returned by `Factor.data_type()`.

If this `Factor` has evidence variables, the DataFrame `evidence_values` contains `n` instances for each evidence variable. Each sampled instance must be conditioned on `evidence_values`.

Parameters

- **n** – Number of instances to sample.
- **evidence_values** – DataFrame of evidence values to condition the sampling.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.

save(*self: pybnesian.Factor*, *filename: str*) → None

Saves the `Factor` in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

slogl(*self: pybnesian.Factor*, *df: DataFrame*) → float

Returns the sum of the log-likelihood of each instance in the DataFrame `df`. That is, the sum of the result of `Factor.logl()`.

Parameters
df – DataFrame to compute the sum of the log-likelihood.

Returns
The sum of log-likelihood for DataFrame `df`.

type(*self*: pybnesian.Factor) → *pybnesian.FactorType*

Returns the corresponding *FactorType* of this *Factor*.

Returns

FactorType corresponding to this *Factor*.

variable(*self*: pybnesian.Factor) → str

Gets the variable modelled by this *Factor*.

Returns

Variable name.

3.3.2 Continuous Factors

Linear Gaussian CPD

class pybnesian.**LinearGaussianCPDType**

Bases: *FactorType*

LinearGaussianCPDType is the corresponding CPD type of *LinearGaussianCPD*.

__init__(*self*: pybnesian.LinearGaussianCPDType) → None

Instantiates a *LinearGaussianCPDType*.

class pybnesian.**LinearGaussianCPD**

Bases: *Factor*

This is a linear Gaussian CPD:

$$\hat{f}(\text{variable} \mid \text{evidence}) = \mathcal{N}(\text{variable}; \text{beta}_0 + \sum_{i=1}^{|\text{evidence}|} \text{beta}_i \cdot \text{evidence}_i, \text{variance})$$

It is parametrized by the following attributes:

Variables

- **beta** – The beta vector.
- **variance** – The variance.

```
>>> from pybnesian import LinearGaussianCPD
>>> cpd = LinearGaussianCPD("a", ["b"])
>>> assert not cpd.fitted()
>>> cpd.beta
array([], dtype=float64)
>>> cpd.beta = np.asarray([1., 2.])
>>> assert not cpd.fitted()
>>> cpd.variance = 0.5
>>> assert cpd.fitted()
>>> cpd.beta
array([1., 2.])
>>> cpd.variance
0.5
```

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.LinearGaussianCPD, variable: str, evidence: List[str]) -> None

Initializes a new *LinearGaussianCPD* with a given `variable` and `evidence`.

The *LinearGaussianCPD* is left unfitted.

Parameters

- **variable** – Variable name.
- **evidence** – List of evidence variable names.

2. __init__(self: pybnesian.LinearGaussianCPD, variable: str, evidence: List[str], beta: numpy.ndarray[numpy.float64[m, 1]], variance: float) -> None

Initializes a new *LinearGaussianCPD* with a given `variable` and `evidence`.

The *LinearGaussianCPD* is fitted with `beta` and `variance`.

Parameters

- **variable** – Variable name.
- **evidence** – List of evidence variable names.
- **beta** – Vector of parameters.
- **variance** – Variance of the linear Gaussian CPD.

property beta

The beta vector of parameters. The beta vector is a `numpy.ndarray` vector of type `numpy.float64` with size `len(evidence) + 1`.

`beta[0]` is always the intercept coefficient and `beta[i]` is the corresponding coefficient for the variable `evidence[i-1]` for `i > 0`.

cdf(*self: pybnesian.LinearGaussianCPD*, *df: DataFrame*) → numpy.ndarray[numpy.float64[m, 1]]

Returns the cumulative distribution function values of each instance in the DataFrame `df`.

Parameters
df – DataFrame to compute the log-likelihood.

Returns
A `numpy.ndarray` vector with dtype `numpy.float64`, where the i-th value is the cumulative distribution function value of the i-th instance of `df`.

property variance

The variance of the linear Gaussian CPD. This is a `float` value.

Conditional Kernel Density Estimation (CKDE)

class pybnesian.**CKDEType**

Bases: *FactorType*

CKDEType is the corresponding CPD type of *CKDE*.

__init__(*self: pybnesian.CKDEType*) → None

Instantiates a *CKDEType*.

class pybnesian.**CKDE**

Bases: *Factor*

A conditional kernel density estimator (CKDE) is the ratio of two KDE models:

$$\hat{f}(\text{variable} \mid \text{evidence}) = \frac{\hat{f}_K(\text{variable}, \text{evidence})}{\hat{f}_K(\text{evidence})}$$

where $\hat{f}_K$ is a *KDE* estimation.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.CKDE, variable: str, evidence: List[str]) -> None

Initializes a new *CKDE* with a given `variable` and `evidence`.

Parameters

- **variable** – Variable name.
- **evidence** – List of evidence variable names.

2. __init__(self: pybnesian.CKDE, variable: str, evidence: List[str], bandwidth_selector: pybnesian.BandwidthSelector) -> None

Initializes a new *CKDE* with a given `variable` and `evidence`.

Parameters

- **variable** – Variable name.
- **evidence** – List of evidence variable names.
- **bandwidth_selector** – Procedure to fit the bandwidth.

cdf(*self: pybnesian.CKDE*, *df: DataFrame*) → numpy.ndarray[numpy.float64[m, 1]]

Returns the cumulative distribution function values of each instance in the DataFrame `df`.

Parameters
df – DataFrame to compute the log-likelihood.

Returns
A `numpy.ndarray` vector with dtype `numpy.float64`, where the i-th value is the cumulative distribution function value of the i-th instance of `df`.

kde_joint(*self: pybnesian.CKDE*) → *pybnesian.KDE*

Gets the joint $\hat{f}_K(\text{variable}, \text{evidence})$ *KDE* model.

Returns
Joint KDE model.

kde_marg(*self: pybnesian.CKDE*) → *pybnesian.KDE*

Gets the marginalized $\hat{f}_K(\text{evidence})$ *KDE* model.

Returns
Marginalized KDE model.

num_instances(*self: pybnesian.CKDE*) → int

Gets the number of training instances (N).

Returns
Number of training instances.

3.3.3 Discrete Factors

class pybnesian.**DiscreteFactorType**

Bases: *FactorType*

DiscreteFactorType is the corresponding CPD type of *DiscreteFactor*.

__init__(*self: pybnesian.DiscreteFactorType*) → None

Instantiates a *DiscreteFactorType*.

class pybnesian.**DiscreteFactor**

Bases: *Factor*

This is a discrete factor implemented as a conditional probability table (CPT).

__init__(*self: pybnesian.DiscreteFactor*, *variable: str*, *evidence: List[str]*) → None

Initializes a new *DiscreteFactor* with a given `variable` and `evidence`.

Parameters

- **variable** – Variable name.
- **evidence** – List of evidence variable names.

3.3.4 Hybrid Factors

class pybnesian.**CLinearGaussianCPD**

Bases: *Factor*

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.CLinearGaussianCPD, arg0: str, arg1: List[str]) -> None
2. __init__(self: pybnesian.CLinearGaussianCPD, arg0: str, arg1: List[str], arg2: numpy.ndarray[numpy.float64[m, 1]], arg3: float) -> None
3. __init__(self: pybnesian.CLinearGaussianCPD, arg0: str, arg1: List[str], arg2: Dict[pybnesian.Assignment, Tuple[numpy.ndarray[numpy.float64[m, 1]], float]]) -> None

conditional_factor(*self: pybnesian.CLinearGaussianCPD*, *arg0: pybnesian.Assignment*) → *pybnesian.Factor*

class pybnesian.**HCKDE**

Bases: *Factor*

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.HCKDE, arg0: str, arg1: List[str]) -> None
2. __init__(self: pybnesian.HCKDE, variable: str, evidence: List[str], bandwidth_selector: pybnesian.BandwidthSelector) -> None
3. __init__(self: pybnesian.HCKDE, variable: str, evidence: List[str], bandwidth_selector: Dict[pybnesian.Assignment, Tuple[pybnesian.BandwidthSelector]]) -> None

conditional_factor(*self: pybnesian.HCKDE*, *arg0: pybnesian.Assignment*) → *pybnesian.Factor*

3.3.5 Other Types

This types are not factors, but are auxiliary types for other factors.

Kernel Density Estimation

class pybnesian.**BandwidthSelector**

A *BandwidthSelector* estimates the bandwidth of a kernel density estimation (KDE) model.

If the bandwidth matrix cannot be calculated because the data has a singular covariance matrix, you should raise a *SingularCovarianceData*.

__init__(*self: pybnesian.BandwidthSelector*) → None

Initializes a *BandwidthSelector*.

__str__(*self: pybnesian.BandwidthSelector*) → str

bandwidth(*self: pybnesian.BandwidthSelector, df: DataFrame, variables: List[str]*) → numpy.ndarray[numpy.float64[m, n]]

Selects the bandwidth of a set of variables for a *KDE* with a given data `df`.

Parameters

- **df** – DataFrame to select the bandwidth.
- **variables** – A list of variables.

Returns

A float or numpy matrix of floats representing the bandwidth matrix.

diag_bandwidth(*self: pybnesian.BandwidthSelector, df: DataFrame, variables: List[str]*) → numpy.ndarray[numpy.float64[m, 1]]

Selects the bandwidth vector of a set of variables for a *ProductKDE* with a given data `df`.

Parameters

- **df** – DataFrame to select the bandwidth.
- **variables** – A list of variables.

Returns

A numpy vector of floats. The i-th entry is the bandwidth h_i^2 for the `variables[i]`.

class pybnesian.**ScottsBandwidth**

Bases: *BandwidthSelector*

Selects the bandwidth using the Scott's rule [Scott]:

$$\hat{h}_i = \hat{\sigma}_i \cdot N^{-1/(d+4)}.$$

This is a simplification of the normal reference rule.

__init__(*self: pybnesian.ScottsBandwidth*) → None

Initializes a *ScottsBandwidth*.

class pybnesian.**NormalReferenceRule**

Bases: *BandwidthSelector*

Selects the bandwidth using the normal reference rule:

$$\hat{h}_i = \left(\frac{4}{d+2}\right)^{1/(d+4)} \hat{\sigma}_i \cdot N^{-1/(d+4)}.$$

__init__(*self: pybnesian.NormalReferenceRule*) → None

Initializes a *NormalReferenceRule*.

class pybnesian.UCV

Bases: *BandwidthSelector*

Selects the bandwidth using the Unbiased Cross Validation (UCV) criterion (also known as least-squares cross validation).

See Equation (3.8) in [MVKSA]:

$$\text{UCV}(\mathbf{H}) = N^{-1}|\mathbf{H}|^{-1/2}(4\pi)^{-d/2} + \{N(N-1)\}^{-1} \sum_{i,j:\ i \neq j}^{N} \{(1 - N^{-1})\phi_{2\mathbf{H}} - \phi_{\mathbf{H}}\}(\mathbf{t}_i - \mathbf{t}_j)$$

where N is the number of training instances, ϕ_Σ is the multivariate Gaussian kernel function with covariance Σ, $\mathbf{t}_i$ is the i-th training instance, and $\mathbf{H}$ is the bandwidth matrix.

__init__(*self: pybnesian.UCV*) → None

Initializes a *UCV*.

class pybnesian.KDE

This class implements Kernel Density Estimation (KDE) for a set of variables:

$$\hat{f}(\text{variables}) = \frac{1}{N|\mathbf{H}|} \sum_{i=1}^{N} K(\mathbf{H}^{-1}(\text{variables} - \mathbf{t}_i))$$

where N is the number of training instances, $K()$ is the multivariate Gaussian kernel function, $\mathbf{t}_i$ is the i-th training instance, and $\mathbf{H}$ is the bandwidth matrix.

__init__(**args, **kwargs*)

Overloaded function.

1. __init__(self: pybnesian.KDE, variables: List[str]) -> None

Initializes a KDE with the given `variables`. It uses the *NormalReferenceRule* as the default bandwidth selector.

Parameters
`variables` – List of variable names.

2. __init__(self: pybnesian.KDE, variables: List[str], bandwidth_selector: pybnesian.BandwidthSelector) -> None

Initializes a KDE with the given `variables` and `bandwidth_selector` procedure to fit the bandwidth.

Parameters

- **variables** – List of variable names.
- **bandwidth_selector** – Procedure to fit the bandwidth.

property bandwidth

Bandwidth matrix ($\mathbf{H}$)

data_type(*self*: pybnesian.KDE) → pyarrow.DataType

Returns the `pyarrow.DataType` that represents the type of data handled by the `KDE`.

It can return `pyarrow.float64` or `pyarrow.float32`.

Returns
the `pyarrow.DataType` physical data type representation of the `KDE`.

dataset(*self*: pybnesian.KDE) → DataFrame

Gets the training dataset for this KDE (the $\mathbf{t}_i$ instances).

Returns
Training instance.

fit(*self*: pybnesian.KDE, *df*: *DataFrame*) → None

Fits the `KDE` with the data in `df`. It estimates the bandwidth $\mathbf{H}$ automatically using the provided bandwidth selector.

Parameters
df – DataFrame to fit the `KDE`.

fitted(*self*: pybnesian.KDE) → bool

Checks whether the model is fitted.

Returns
True if the model is fitted, False otherwise.

logl(*self*: pybnesian.KDE, *df*: *DataFrame*) → numpy.ndarray[numpy.float64[m, 1]]

Returns the log-likelihood of each instance in the DataFrame `df`.

Parameters
df – DataFrame to compute the log-likelihood.

Returns
A `numpy.ndarray` vector with dtype `numpy.float64`, where the i-th value is the log-likelihod of the i-th instance of `df`.

num_instances(*self*: pybnesian.KDE) → int

Gets the number of training instances (N).

Returns
Number of training instances.

num_variables(*self*: pybnesian.KDE) → int

Gets the number of variables.

Returns
Number of variables.

save(*self*: pybnesian.KDE, *filename*: *str*) → None

Saves the `KDE` in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

slogl(*self*: pybnesian.KDE, *df*: *DataFrame*) → float

Returns the sum of the log-likelihood of each instance in the DataFrame `df`. That is, the sum of the result of `KDE.logl`.

Parameters
df – DataFrame to compute the sum of the log-likelihood.

Returns
The sum of log-likelihood for DataFrame `df`.

variables(*self: pybnesian.KDE*) → List[str]

Gets the variable names:

Returns
List of variable names.

class pybnesian.ProductKDE

This class implements a product Kernel Density Estimation (KDE) for a set of variables:

$$\hat{f}(x_1, \ldots, x_d) = \frac{1}{N \cdot h_1 \cdot \ldots \cdot h_d} \sum_{i=1}^{N} \prod_{j=1}^{d} K\left(\frac{(x_j - t_{ji})}{h_j}\right)$$

where N is the number of training instances, d is the dimensionality of the product KDE, $K()$ is the multivariate Gaussian kernel function, t_{ji} is the value of the j-th variable in the i-th training instance, and h_j is the bandwidth parameter for the j-th variable.

__init__(**args, **kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ProductKDE, variables: List[str]) -> None

Initializes a ProductKDE with the given `variables`.

Parameters
variables – List of variable names.

2. __init__(self: pybnesian.ProductKDE, variables: List[str], bandwidth_selector: pybnesian.BandwidthSelector) -> None

Initializes a ProductKDE with the given `variables` and `bandwidth_selector` procedure to fit the bandwidth.

Parameters

- **variables** – List of variable names.
- **bandwidth_selector** – Procedure to fit the bandwidth.

property bandwidth

Vector of bandwidth values (h_j^2).

data_type(*self: pybnesian.ProductKDE*) → pyarrow.DataType

Returns the `pyarrow.DataType` that represents the type of data handled by the *ProductKDE*.

It can return `pyarrow.float64` or `pyarrow.float32`.

Returns
the `pyarrow.DataType` physical data type representation of the *ProductKDE*.

dataset(*self: pybnesian.ProductKDE*) → DataFrame

Gets the training dataset for this ProductKDE (the $\mathbf{t}_i$ instances).

Returns
Training instance.

fit(*self: pybnesian.ProductKDE*, *df: DataFrame*) → None

Fits the *ProductKDE* with the data in `df`. It estimates the bandwidth vector h_j automatically using the provided bandwidth selector.

Parameters
`df` – DataFrame to fit the *ProductKDE*.

fitted(*self: pybnesian.ProductKDE*) → bool

Checks whether the model is fitted.

Returns
True if the model is fitted, False otherwise.

logl(*self: pybnesian.ProductKDE*, *df: DataFrame*) → numpy.ndarray[numpy.float64[m, 1]]

Returns the log-likelihood of each instance in the DataFrame `df`.

Parameters
`df` – DataFrame to compute the log-likelihood.

Returns
A `numpy.ndarray` vector with dtype `numpy.float64`, where the i-th value is the log-likelihod of the i-th instance of `df`.

num_instances(*self: pybnesian.ProductKDE*) → int

Gets the number of training instances (N).

Returns
Number of training instances.

num_variables(*self: pybnesian.ProductKDE*) → int

Gets the number of variables.

Returns
Number of variables.

save(*self: pybnesian.ProductKDE*, *filename: str*) → None

Saves the *ProductKDE* in a pickle file with the given name.

Parameters
filename – File name of the saved graph.

slogl(*self: pybnesian.ProductKDE*, *df: DataFrame*) → float

Returns the sum of the log-likelihood of each instance in the DataFrame `df`. That is, the sum of the result of *ProductKDE.logl*.

Parameters
`df` – DataFrame to compute the sum of the log-likelihood.

Returns
The sum of log-likelihood for DataFrame `df`.

variables(*self: pybnesian.ProductKDE*) → List[str]

Gets the variable names:

Returns
List of variable names.

exception pybnesian.**SingularCovarianceData**

Bases: `ValueError`

This exception signals that the data has a singular covariance matrix.

Other

class pybnesian.**UnknownFactorType**

UnknownFactorType is the representation of an unknown *FactorType*. This factor type is assigned by default to each node in an heterogeneous Bayesian network.

__init__(*self: pybnesian.UnknownFactorType*) → None

Instantiates an *UnknownFactorType*.

class pybnesian.**Assignment**

Assignment represents the assignment of values to a set of variables.

__init__(*self: pybnesian.Assignment, assignments: Dict[str, AssignmentValue]*) → None

Initializes an *Assignment* from a dict that contains the value for each variable. The key of the dict is the name of the variable, and the value of the dict can be an `str` or a `float` value.

Parameters
assignments – Value assignments for each variable.

empty(*self: pybnesian.Assignment*) → bool

Checks whether the *Assignment* does not have assignments.

Returns
True if the *Assignment* does not have assignments, False otherwise.

has_variables(*self: pybnesian.Assignment, variables: List[str]*) → bool

Checks whether the *Assignment* contains assignments for all the `variables`.

Parameters
variables – Variable names.

Returns
True if the *Assignment* contains values for all the given variables, False otherwise.

insert(*self: pybnesian.Assignment, variable: str, value: AssignmentValue*) → None

Inserts a new assignment for a `variable` with a `value`.

Parameters

- **variable** – Variable name.
- **value** – Value (`str` or `float`) for the variable.

remove(*self: pybnesian.Assignment, variable: str*) → None

Removes the assignment for the `variable`.

Parameters
variable – Variable name.

size(*self: pybnesian.Assignment*) → int

Gets the number of assignments in the *Assignment*.

Returns
The number of assignments.

value(*self: pybnesian.Assignment, variable: str*) → AssignmentValue

Returns the assignment value for a given `variable`.

Parameters
variable – Variable name.

Returns
Value assignment of the variable.

class pybnesian.**Args**

__init__(*self: pybnesian.Args*, **args*) → None

The *Args* defines a wrapper over **args*. This class allows to distinguish between a tuple representing **args* or a tuple parameter while using *Arguments*.

Example:

```
Arguments({ 'a' : ((1, 2), {'param': 3}) })
# or
Arguments({ 'a' : Args((1, 2), {'param': 3}) })
```

defines an **args* with 2 arguments: a tuple (1, 2) and a dict {'param': 3}. No ***kwargs* is defined.

```
Arguments({ 'a' : (Args(1, 2), Kwargs(param = 3)) })
```

defines an **args* with 2 arguments: 1 and 2. It also defines a ***kwargs* with param = 3.

class pybnesian.**Kwargs**

__init__(*self: pybnesian.Kwargs*, ***kwargs*) → None

The *Kwargs* defines a wrapper over ***kwargs*. This class allows to distinguish between a dict representing ***kwargs* or a dict parameter while using *Arguments*.

See *Example Args/Kwargs*.

class pybnesian.**Arguments**

The *Arguments* class collects different arguments to construct *Factor*.

The *Arguments* object is constructed from a dictionary that associates each *Factor* configuration with a set of arguments.

The keys of the dictionary can be:

- A 2-tuple (`name`, `factor_type`) defines arguments for a *Factor* of variable `name` with *FactorType* `factor_type`.
- An str defines arguments for a *Factor* of variable `name`.
- A *FactorType* defines arguments for a *Factor* with *FactorType* `factor_type`.

The values of the dictionary can be:

- A 2-tuple (*Args*, *Kwargs*) defines **args* and ***kwargs*.
- An *Args* or tuple (…) defines only **args*.
- A *Kwargs* or dict { … }: defines only ***kwargs*.

When searching for the defined arguments in *Arguments* for a given factor with `name` and `factor_type`, the most specific configurations have preference over more general ones.

- If a 2-tuple (`name`, `factor_type`) configuration exists, the corresponding arguments are returned.
- Else, if a `name` configuration exists, the corresponding arguments are returned.
- Else, if a `factor_type` configuration exists, the corresponding arguments are returned.
- Else, empty **args* and ***kwargs* are returned.

`__init__`(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.Arguments) -> None

Initializes an empty *`Arguments`*.

2. __init__(self: pybnesian.Arguments, dict_arguments: dict) -> None

Initializes a new *`Arguments`* with the given configurations and arguments.

Parameters

`dict_arguments` – A dictionary { configurations : arguments} that associates each *`Factor`* configuration with a set of arguments.

`args`(*self: pybnesian.Arguments*, *node: str*, *node_type: factors::FactorType*) → Tuple[*args, **kwargs]

Returns the *args and **kwargs defined for a `node` with a given `node_type`.

Parameters

- **`node`** – A node name.
- **`node_type`** – *`FactorType`* for node.

Returns

2-tuple containing `(*args, **kwargs)`

3.3.6 Bibliography

3.4 Bayesian Networks

PyBNesian includes many different types of Bayesian networks.

3.4.1 Abstract Classes

This classes are abstract and define the interface for Bayesian network objects. The *`BayesianNetworkType`* specifies the type of Bayesian networks.

Each *`BayesianNetworkType`* can be used in many multiple variants of Bayesian networks: *`BayesianNetworkBase`* (a normal Bayesian network), *`ConditionalBayesianNetworkBase`* (a conditional Bayesian network) and *`DynamicBayesianNetworkBase`* (a dynamic Bayesian network).

class `pybnesian.`**`BayesianNetworkType`**

A representation of a *`BayesianNetwork`* that defines its behaviour.

`__init__`(*self: pybnesian.BayesianNetworkType*) → None

Initializes a new *`BayesianNetworkType`*

`__str__`(*self: pybnesian.BayesianNetworkType*) → str

`alternative_node_type`(*model: BayesianNetworkBase or ConditionalBayesianNetworkBase*, *source: str*) → List[*pybnesian.FactorType*]

Returns all feasible alternative *`FactorType`* for `node`.

Parameters

- **`model`** – BayesianNetwork model.
- **`node`** – Name of the node.

Returns
A list of alternative *FactorType*. If you implement this method in a Python-derived class, you can return an empty list or None to specify that no changes are possible.

can_have_arc(*model: BayesianNetworkBase or ConditionalBayesianNetworkBase*, *source: str*, *target: str*) → bool

Checks whether the *BayesianNetworkType* allows an arc `source -> target` in the Bayesian network `model`.

Parameters

- **model** – BayesianNetwork model.
- **source** – Name of the source node.
- **target** – Name of the target node.

Returns
True if the arc `source -> target` is allowed in `model`, False otherwise.

compatible_node_type(*model: BayesianNetworkBase or ConditionalBayesianNetworkBase*, *node: str*, *node_type:* pybnesian.FactorType) → bool

Checks whether the *FactorType* `node_type` is allowed for `node` by this *BayesianNetworkType*.

Parameters

- **model** – BayesianNetwork model.
- **node** – Name of the node to check.
- **node_type** – *FactorType* for `node`.

Returns
True if the current *FactorType* is allowed, False otherwise.

data_default_node_type(*self:* pybnesian.BayesianNetworkType, *datatype: pyarrow.DataType*) → List[*pybnesian.FactorType*]

Returns a list of default *FactorType* for the nodes of this Bayesian network type with data type `datatype`. This method is only needed for non-homogeneous Bayesian networks and defines the priority of use of the different *FactorType* for the given `datatype`. If a *FactorType* is blacklisted for a given node, the next element in the list is used as the default *FactorType*. See also *BayesianNetworkBase.set_unknown_node_types()*.

Parameters
datatype – `pyarrow.DataType` defining the type of data for a node.

Returns
List of default *FactorType* for a node given the `datatype`.

default_node_type(*self:* pybnesian.BayesianNetworkType) → *pybnesian.FactorType*

Returns the default *FactorType* of each node in this Bayesian network type. This method is only needed for homogeneous Bayesian networks and returns the unique possible *FactorType*.

Returns
default *FactorType* for the nodes.

is_homogeneous(*self:* pybnesian.BayesianNetworkType) → bool

Checks whether the Bayesian network is homogeneous.

A Bayesian network is homogeneous if the *FactorType* of all the nodes are forced to be the same: for example, a Gaussian network is homogeneous because the *FactorType* type of each node is always *LinearGaussianCPDType*.

Returns
True if the Bayesian network is homogeneous, False otherwise.

new_bn(*self: pybnesian.BayesianNetworkType*, *nodes: List[str]*) → *pybnesian.BayesianNetworkBase*

Returns an empty unconditional Bayesian network of this type with the given `nodes`.

Parameters
nodes – Nodes of the new Bayesian network.

Returns
A new empty unconditional Bayesian network.

new_cbn(*self: pybnesian.BayesianNetworkType*, *nodes: List[str]*, *interface_nodes: List[str]*) → *pybnesian.ConditionalBayesianNetworkBase*

Returns an empty conditional Bayesian network of this type with the given `nodes` and `interface_nodes`.

Parameters

- **nodes** – Nodes of the new Bayesian network.
- **nodes** – Interface nodes of the new Bayesian network.

Returns
A new empty conditional Bayesian network.

class pybnesian.**BayesianNetworkBase**

This class defines an interface of base operations for all the Bayesian networks.

It reproduces many of the methods in the underlying graph to perform additional initializations and simplify the access. See *Graph Module*.

__str__(*self: pybnesian.BayesianNetworkBase*) → str

add_arc(*self: pybnesian.BayesianNetworkBase*, *source: str*, *target: str*) → None

Adds an arc between the nodes `source` and `target`. If the arc already exists, the graph is left unaffected.

Parameters

- **source** – A node name.
- **target** – A node name.

add_cpds(*self: pybnesian.BayesianNetworkBase*, *cpds: List[pybnesian.Factor]*) → None

Adds a list of CPDs to the Bayesian network. The list may be complete (for all the nodes all the Bayesian network) or partial (just some a subset of the nodes).

Parameters
cpds – List of `Factor`.

add_node(*self: pybnesian.BayesianNetworkBase*, *node: str*) → int

Adds a node to the Bayesian network and returns its index.

Parameters
node – Name of the new node.

Returns
Index of the new node.

arcs(*self: pybnesian.BayesianNetworkBase*) → List[Tuple[str, str]]

Gets the list of arcs.

Returns
A list of tuples (source, target) representing an arc source -> target.

can_add_arc(*self:* pybnesian.BayesianNetworkBase, *source:* *str*, *target:* *str*) → bool

Checks whether an arc between the nodes `source` and `target` can be added.

An arc addition can be not allowed for multiple reasons:

- It generates a cycle.
- It is a conditional BN and both source and target are interface nodes.
- It is not allowed by the *BayesianNetworkType*.

Parameters

- **source** – A node name.
- **target** – A node name.

Returns

True if the arc can be added, False otherwise.

can_flip_arc(*self:* pybnesian.BayesianNetworkBase, *source:* *str*, *target:* *str*) → bool

Checks whether an arc between the nodes `source` and `target` can be flipped.

An arc flip can be not allowed for multiple reasons:

- It generates a cycle.
- It is not allowed by the *BayesianNetworkType*.

Parameters

- **source** – A node name.
- **target** – A node name.

Returns

True if the arc can be added, False otherwise.

children(*self:* pybnesian.BayesianNetworkBase, *node:* *str*) → List[str]

Gets the children nodes of a node.

Parameters

node – A node name.

Returns

Children node names.

clone(*self:* pybnesian.BayesianNetworkBase) → *pybnesian.BayesianNetworkBase*

Clones (copies) this Bayesian network.

Returns

A copy of `self`.

collapsed_from_index(*self:* pybnesian.BayesianNetworkBase, *index:* *int*) → int

Gets the collapsed index of a node from its index.

Parameters

index – Index of the node.

Returns

Collapsed index of the node.

collapsed_index(*self: pybnesian.BayesianNetworkBase*, *node: str*) → int

Gets the collapsed index of a node from its name.

Parameters
node – Name of the node.

Returns
Collapsed index of the node.

collapsed_indices(*self: pybnesian.BayesianNetworkBase*) → Dict[str, int]

Gets all the collapsed indices for the nodes in the graph.

Returns
A dictionary with the collapsed index of each node.

collapsed_name(*self: pybnesian.BayesianNetworkBase*, *collapsed_index: int*) → str

Gets the name of a node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the node.

Returns
Name of the node.

conditional_bn(**args*, ***kwargs*)

Overloaded function.

1. conditional_bn(self: pybnesian.BayesianNetworkBase) -> pybnesian.ConditionalBayesianNetworkBase

Returns the conditional Bayesian network version of this Bayesian network.

- If `self` is not conditional, it returns a conditional version of the Bayesian network where the graph is transformed using `Dag.conditional_graph`.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

2. conditional_bn(self: pybnesian.BayesianNetworkBase, nodes: List[str], interface_nodes: List[str]) -> pybnesian.ConditionalBayesianNetworkBase

Returns the conditional Bayesian network version of this Bayesian network.

- If `self` is not conditional, it returns a conditional version of the Bayesian network where the graph is transformed using `Dag.conditional_graph` using the given set of nodes and interface nodes.
- If `self` is conditional, it returns a copy of `self`.

Returns
The conditional graph transformation of `self`.

contains_node(*self: pybnesian.BayesianNetworkBase*, *node: str*) → bool

Tests whether the node is in the Bayesian network or not.

Parameters
node – Name of the node.

Returns
True if the Bayesian network contains the node, False otherwise.

cpd(*self: pybnesian.BayesianNetworkBase*, *node: str*) → *pybnesian.Factor*

Returns the conditional probability distribution (CPD) associated to `node`. This is a *Factor* type.

Parameters
`node` – A node name.

Returns
The *Factor* associated to `node`

Raises
`ValueError` – If `node` do not have an associated *Factor* yet.

fit(*self: pybnesian.BayesianNetworkBase*, *df: DataFrame*, *construction_args: pybnesian.Arguments = Arguments*) → None

Fit all the unfitted *Factor* with the data `df`.

Parameters

- **df** – DataFrame to fit the Bayesian network.
- **construction_args** – Additional arguments provided to construct the *Factor*.

fitted(*self: pybnesian.BayesianNetworkBase*) → bool

Checks whether the model is fitted.

Returns
True if the model is fitted, False otherwise.

flip_arc(*self: pybnesian.BayesianNetworkBase*, *source: str*, *target: str*) → None

Flips (reverses) an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

Parameters

- **source** – A node name.
- **target** – A node name.

force_type_whitelist(*self: pybnesian.BayesianNetworkBase*, *type_whitelist: List[Tuple[str, pybnesian.FactorType]]*) → None

Forces the Bayesian network to have the given whitelisted node types.

Parameters
type_whitelist – List of node type tuples (`node`, *FactorType*) that specifies the whitelisted type for each node.

force_whitelist(*self: pybnesian.BayesianNetworkBase*, *arc_whitelist: List[Tuple[str, str]]*) → None

Include the given whitelisted arcs. It checks the validity of the graph after including the arc whitelist.

Parameters
arc_whitelist – List of arcs tuples (`source`, `target`) that must be added to the graph.

has_arc(*self: pybnesian.BayesianNetworkBase*, *source: str*, *target: str*) → bool

Checks whether an arc between the nodes `source` and `target` exists.

Parameters

- **source** – A node name.
- **target** – A node name.

Returns
True if the arc exists, False otherwise.

has_path(*self: pybnesian.BayesianNetworkBase*, *n1: str*, *n2: str*) → bool

Checks whether there is a directed path between nodes `n1` and `n2`.

Parameters

- **n1** – A node name.
- **n2** – A node name.

Returns

True if there is an directed path between `n1` and `n2`, False otherwise.

has_unknown_node_types(*self: pybnesian.BayesianNetworkBase*) → bool

Checks whether there are nodes with an unknown node type (i.e. `UnknownFactorType`).

Returns

True if there are nodes with an unkown node type, False otherwise.

property include_cpd

This property indicates if the factors of the Bayesian network model should be saved when `__getstate__` is called.

index(*self: pybnesian.BayesianNetworkBase*, *node: str*) → int

Gets the index of a node from its name.

Parameters

node – Name of the node.

Returns

Index of the node.

index_from_collapsed(*self: pybnesian.BayesianNetworkBase*, *collapsed_index: int*) → int

Gets the index of a node from its collapsed index.

Parameters

collapsed_index – Collapsed index of the node.

Returns

Index of the node.

indices(*self: pybnesian.BayesianNetworkBase*) → Dict[str, int]

Gets all the indices in the graph.

Returns

A dictionary with the index of each node.

is_valid(*self: pybnesian.BayesianNetworkBase*, *node: str*) → bool

Checks whether a node is valid (the node is not removed).

Parameters

node – Node name.

Returns

True if the node is valid, False otherwise.

logl(*self: pybnesian.BayesianNetworkBase*, *df: DataFrame*) → numpy.ndarray[numpy.float64[m, 1]]

Returns the log-likelihood of each instance in the DataFrame `df`. This returns the sum of the log-likelihood for all the factors in the Bayesian network.

Parameters

df – DataFrame to compute the log-likelihood.

Returns
A `numpy.ndarray` vector with dtype `numpy.float64`, where the i-th value is the log-likelihod of the i-th instance of `df`.

name(*self: pybnesian.BayesianNetworkBase*, *index: int*) → str

Gets the name of a node from its index.

Parameters
index – Index of the node.

Returns
Name of the node.

node_type(*self: pybnesian.BayesianNetworkBase*, *node: str*) → *pybnesian.FactorType*

Gets the corresponding `FactorType` for `node`.

Parameters
node – A node name.

Returns
The `FactorType` of `node`.

node_types(*self: pybnesian.BayesianNetworkBase*) → Dict[str, *pybnesian.FactorType*]

Gets the `FactorType` for all the nodes.

Returns
The corresponding `FactorType` for each node.

nodes(*self: pybnesian.BayesianNetworkBase*) → List[str]

Gets the nodes of the Bayesian network.

Returns
Nodes of the Bayesian network.

num_arcs(*self: pybnesian.BayesianNetworkBase*) → int

Gets the number of arcs.

Returns
Number of arcs.

num_children(*self: pybnesian.BayesianNetworkBase*, *node: str*) → int

Gets the number of children nodes of a node.

Parameters
node – A node name.

Returns
Number of children nodes.

num_nodes(*self: pybnesian.BayesianNetworkBase*) → int

Gets the number of nodes.

Returns
Number of nodes.

num_parents(*self: pybnesian.BayesianNetworkBase*, *node: str*) → int

Gets the number of parent nodes of a node.

Parameters
node – A node name.

Returns
Number of parent nodes.

parents(*self: pybnesian.BayesianNetworkBase*, *node: str*) → List[str]

Gets the parent nodes of a node.

Parameters
node – A node name.

Returns
Parent node names.

remove_arc(*self: pybnesian.BayesianNetworkBase*, *source: str*, *target: str*) → None

Removes an arc between the nodes `source` and `target`. If the arc do not exist, the graph is left unaffected.

Parameters

- **source** – A node name.
- **target** – A node name.

remove_node(*self: pybnesian.BayesianNetworkBase*, *node: str*) → None

Removes a node.

Parameters
node – A node name.

sample(*self: pybnesian.BayesianNetworkBase*, *n: int*, *seed: Optional[int] = None*, *ordered: bool = False*) → DataFrame

Samples `n` values from this BayesianNetwork. This method returns a `pyarrow.RecordBatch` with `n` instances.

If `ordered` is True, it orders the columns according to the list *`BayesianNetworkBase.nodes()`*. Else, it orders the columns according to a topological sort.

Parameters

- **n** – Number of instances to sample.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **ordered** – If True, order the columns according to *`BayesianNetworkBase.nodes()`*.

Returns
A DataFrame with `n` instances that contains the sampled data.

save(*self: pybnesian.BayesianNetworkBase*, *filename: str*, *include_cpd: bool = False*) → None

Saves the Bayesian network in a pickle file with the given name. If `include_cpd` is True, it also saves the conditional probability distributions (CPDs) in the Bayesian network.

Parameters

- **filename** – File name of the saved Bayesian network.
- **include_cpd** – Include the CPDs.

set_node_type(*self: pybnesian.BayesianNetworkBase*, *node: str*, *new_type: pybnesian.FactorType*) → None

Sets the `new_type` *`FactorType`* for `node`.

Parameters

- **node** – A node name.

- **new_type** – The new `FactorType` for `node`.

`set_unknown_node_types`(*self:* pybnesian.BayesianNetworkBase, *df: DataFrame*, *type_blacklist: List[Tuple[*str*,* pybnesian.FactorType*]] = []*) → None

Changes the unknown node types (i.e. the nodes with `UnknownFactorType`) to the default node types specified by the `BayesianNetworkType`. If a `FactorType` is blacklisted for a given node, the next element in the `BayesianNetworkType.data_default_node_type()` list is used as the default `FactorType`.

Parameters

- **df** – DataFrame to get the default node type for each unknown node type.
- **type_blacklist** – List of type blacklist (forbidden `FactorType`).

`slogl`(*self:* pybnesian.BayesianNetworkBase, *df: DataFrame*) → float

Returns the sum of the log-likelihood of each instance in the DataFrame `df`. That is, the sum of the result of `BayesianNetworkBase.logl()`.

Parameters
df – DataFrame to compute the sum of the log-likelihood.

Returns
The sum of log-likelihood for DataFrame `df`.

`type`(*self:* pybnesian.BayesianNetworkBase) → *pybnesian.BayesianNetworkType*

Gets the underlying `BayesianNetworkType`.

Returns
The `BayesianNetworkType` of `self`.

`unconditional_bn`(*self:* pybnesian.BayesianNetworkBase) → *pybnesian.BayesianNetworkBase*

Returns the unconditional Bayesian network version of this Bayesian network.

- If `self` is not conditional, it returns a copy of `self`.
- If `self` is conditional, the interface nodes are included as nodes in the returned Bayesian network.

Returns
The unconditional graph transformation of `self`.

`underlying_node_type`(*self:* pybnesian.BayesianNetworkBase, *df: DataFrame*, *node:* str) → *pybnesian.FactorType*

Gets the underlying `FactorType` for a given node type.

1) If the node has a node type different from `UnknownFactorType`, it returns it.
2) Else, it returns the first default node type from `BayesianNetworkType.data_default_node_type`.

Parameters

- **df** – Data to extract the underlying node type (if 2) is required).
- **node** – A node name.

Returns
The underlying `FactorType` for each node.

class pybnesian.**ConditionalBayesianNetworkBase**

Bases: *BayesianNetworkBase*

This class defines an interface of base operations for the conditional Bayesian networks.

It includes some methods of the *ConditionalDag* to simplify the access to the graph.

add_interface_node(*self: pybnesian.ConditionalBayesianNetworkBase*, *node: str*) → int

Adds an interface node to the Bayesian network and returns its index.

Parameters
node – Name of the new interface node.

Returns
Index of the new interface node.

clone(*self: pybnesian.ConditionalBayesianNetworkBase*) → *pybnesian.ConditionalBayesianNetworkBase*

Clones (copies) this Bayesian network.

Returns
A copy of `self`.

contains_interface_node(*self: pybnesian.ConditionalBayesianNetworkBase*, *node: str*) → bool

Tests whether the interface node is in the Bayesian network or not.

Parameters
node – Name of the node.

Returns
True if the Bayesian network contains the interface node, False otherwise.

contains_joint_node(*self: pybnesian.ConditionalBayesianNetworkBase*, *node: str*) → bool

Tests whether the node is in the joint set of nodes or not.

Parameters
node – Name of the node.

Returns
True if the node is in the joint set of nodes, False otherwise.

index_from_interface_collapsed(*self: pybnesian.ConditionalBayesianNetworkBase*, *collapsed_index: int*) → int

Gets the index of a node from the interface collapsed index.

Parameters
collapsed_index – Interface collapsed index of the node.

Returns
Index of the node.

index_from_joint_collapsed(*self: pybnesian.ConditionalBayesianNetworkBase*, *collapsed_index: int*) → int

Gets the index of a node from the joint collapsed index.

Parameters
collapsed_index – Joint collapsed index of the node.

Returns
Index of the node.

interface_arcs(*self:* pybnesian.ConditionalBayesianNetworkBase) → List[Tuple[str, str]]

Gets the arcs where the source node is an interface node.

Returns
arcs with an interface node as source node.

interface_collapsed_from_index(*self:* pybnesian.ConditionalBayesianNetworkBase, *index: int*) → int

Gets the interface collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Interface collapsed index of the node.

interface_collapsed_index(*self:* pybnesian.ConditionalBayesianNetworkBase, *node: str*) → int

Gets the interface collapsed index of an interface node from its name.

Parameters
node – Name of the interface node.

Returns
Interface collapsed index of the interface node.

interface_collapsed_indices(*self:* pybnesian.ConditionalBayesianNetworkBase) → Dict[str, int]

Gets all the interface collapsed indices for the interface nodes in the graph.

Returns
A dictionary with the interface collapsed index of each interface node.

interface_collapsed_name(*self:* pybnesian.ConditionalBayesianNetworkBase, *collapsed_index: int*) → str

Gets the name of an interface node from its collapsed index.

Parameters
collapsed_index – Collapsed index of the interface node.

Returns
Name of the interface node.

interface_nodes(*self:* pybnesian.ConditionalBayesianNetworkBase) → List[str]

Gets the interface nodes of the Bayesian network.

Returns
Interface nodes of the Bayesian network.

is_interface(*self:* pybnesian.ConditionalBayesianNetworkBase, *node: str*) → bool

Checks whether the `node` is an interface node.

Parameters
node – A node name.

Returns
True if `node` is interface node, False, otherwise.

joint_collapsed_from_index(*self:* pybnesian.ConditionalBayesianNetworkBase, *index: int*) → int

Gets the joint collapsed index of a node from its index.

Parameters
index – Index of the node.

Returns
Joint collapsed index of the node.

joint_collapsed_index(*self: pybnesian.ConditionalBayesianNetworkBase*, *node: str*) → int

Gets the joint collapsed index of a node from its name.

Parameters
node – Name of the node.

Returns
Joint collapsed index of the node.

joint_collapsed_indices(*self: pybnesian.ConditionalBayesianNetworkBase*) → Dict[str, int]

Gets all the joint collapsed indices for the joint set of nodes in the graph.

Returns
A dictionary with the joint collapsed index of each joint node.

joint_collapsed_name(*self: pybnesian.ConditionalBayesianNetworkBase*, *collapsed_index: int*) → str

Gets the name of a node from its joint collapsed index.

Parameters
collapsed_index – Joint collapsed index of the node.

Returns
Name of the node.

joint_nodes(*self: pybnesian.ConditionalBayesianNetworkBase*) → List[str]

Gets the joint set of nodes of the Bayesian network.

Returns
Joint set of nodes of the Bayesian network.

num_interface_nodes(*self: pybnesian.ConditionalBayesianNetworkBase*) → int

Gets the number of interface nodes.

Returns
Number of interface nodes.

num_joint_nodes(*self: pybnesian.ConditionalBayesianNetworkBase*) → int

Gets the number of joint nodes. That is, `num_nodes() + num_interface_nodes()`

Returns
Number of joint nodes.

remove_interface_node(*self: pybnesian.ConditionalBayesianNetworkBase*, *node: str*) → None

Removes an interface node.

Parameters
node – A node name.

sample(*self: pybnesian.ConditionalBayesianNetworkBase*, *evidence: DataFrame*, *seed: Optional[int] = None*, *concat_evidence: bool = False*, *ordered: bool = False*) → DataFrame

Samples `n` values from this conditional BayesianNetwork conditioned on `evidence`. `evidence` must contain a column for each interface node. This method returns a `pyarrow.RecordBatch` with `n` instances.

If `concat` is True, it concatenates `evidence` in the result.

If `ordered` is True, it orders the columns according to the list `BayesianNetworkBase.nodes()`. Else, it orders the columns according to a topological sort.

Parameters

- **n** – Number of instances to sample.
- **evidence** – A DataFrame of `n` instances to condition the sampling.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **ordered** – If True, order the columns according to `BayesianNetworkBase.nodes()`.

Returns
A DataFrame with `n` instances that contains the sampled data.

set_interface(*self: pybnesian.ConditionalBayesianNetworkBase*, *node: str*) → None

Converts a normal node into an interface node.

Parameters
node – A node name.

set_node(*self: pybnesian.ConditionalBayesianNetworkBase*, *node: str*) → None

Converts an interface node into a normal node.

Parameters
node – A node name.

class pybnesian.**DynamicBayesianNetworkBase**

This class defines an interface of a dynamic Bayesian network.

A dynamic Bayesian network is defined over a set of variables. Each variable is replicated in different nodes (one for each temporal slice). Thus, we differentiate in this documentation between the terms "variable" and "node". To create the nodes, we suffix the variable names using the structure `[variable_name]_t_[temporal_index]`. The `variable_name` is the name of each variable, and `temporal_index` is an index with a range [0-`markovian_order`]. The index "0" is considered the "present", the index "1" delays the temporal one step into the "past", and so on... This is related with the way `DynamicDataFrame` generates the columns.

The dynamic Bayesian is composed of two Bayesian networks:

- a static Bayesian network that defines the probability distribution of the first `markovian_order` instances. It estimates the probability f(`t_1`,..., `t_[markovian_order]`). This Bayesian network is represented with a normal Bayesian network.
- a transition Bayesian network that defines the probability distribution of the i-th instance given the previous `markovian_order` instances. It estimates the probability f(`t_0` | `t_1`, ..., `t_[markovian_order]`), where `t_0` (the present) is the i-th instance. Once the probability of the i-th instance is estimated, the transition network moves a step forward, to estimate the (i+1)-th instance, and so on. This transition Bayesian network is represented with a conditional Bayesian network.

Both Bayesian networks must be of the same `BayesianNetworkType`.

__str__(*self: pybnesian.DynamicBayesianNetworkBase*) → str

add_variable(*self: pybnesian.DynamicBayesianNetworkBase*, *variable: str*) → None

Adds a variable to the dynamic Bayesian network. It adds a node for each temporal slice in the static and transition Bayesian networks.

Parameters
variable – Name of the new variable.

contains_variable(*self: pybnesian.DynamicBayesianNetworkBase*, *variable: str*) → bool

Tests whether the variable is in the dynamic Bayesian network or not.

Parameters
variable – Name of the variable.

Returns
True if the dynamic Bayesian network contains the variable, False otherwise.

fit(*self: pybnesian.DynamicBayesianNetworkBase*, *df: DataFrame*, *construction_args: pybnesian.Arguments = Arguments*) → None

Fit all the unfitted `Factor` with the data `df` in both the static and transition Bayesian networks.

Parameters

- **df** – DataFrame to fit the dynamic Bayesian network.
- **construction_args** – Additional arguments provided to construct the `Factor`.

fitted(*self: pybnesian.DynamicBayesianNetworkBase*) → bool

Checks whether the model is fitted.

Returns
True if the model is fitted, False otherwise.

logl(*self: pybnesian.DynamicBayesianNetworkBase*, *df: DataFrame*) → numpy.ndarray[numpy.float64[m, 1]]

Returns the log-likelihood of each instance in the DataFrame `df`.

Parameters
df – DataFrame to compute the log-likelihood.

Returns
A `numpy.ndarray` vector with dtype `numpy.float64`, where the i-th value is the log-likelihood of the i-th instance of `df`.

markovian_order(*self: pybnesian.DynamicBayesianNetworkBase*) → int

Gets the markovian order of the dynamic Bayesian network.

Returns
markovian order of this dynamic Bayesian network.

num_variables(*self: pybnesian.DynamicBayesianNetworkBase*) → int

Gets the number of variables.

Returns
Number of variables.

remove_variable(*self: pybnesian.DynamicBayesianNetworkBase*, *variable: str*) → None

Removes a variable. It removes all the corresponding nodes in the static and transition Bayesian networks.

Parameters
variable – A variable name.

sample(*self: pybnesian.DynamicBayesianNetworkBase*, *n: int*, *seed: Optional[int] = None*) → DataFrame

Samples `n` values from this dynamic Bayesian network. This method returns a `pyarrow.RecordBatch` with `n` instances.

Parameters

- **n** – Number of instances to sample.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.

save(*self: pybnesian.DynamicBayesianNetworkBase*, *filename: str*, *include_cpd: bool = False*) → None

Saves the dynamic Bayesian network in a pickle file with the given name. If `include_cpd` is True, it also saves the conditional probability distributions (CPDs) in the dynamic Bayesian network.

Parameters

- **filename** – File name of the saved dynamic Bayesian network.
- **include_cpd** – Include the CPDs.

slogl(*self*: pybnesian.DynamicBayesianNetworkBase, *df*: *DataFrame*) → float

Returns the sum of the log-likelihood of each instance in the DataFrame df. That is, the sum of the result of *DynamicBayesianNetworkBase.logl()*.

Parameters
df – DataFrame to compute the sum of the log-likelihood.

Returns
The sum of log-likelihood for DataFrame df.

static_bn(*self*: pybnesian.DynamicBayesianNetworkBase) → *pybnesian.BayesianNetworkBase*

Returns the static Bayesian network.

Returns
Static Bayesian network.

transition_bn(*self*: pybnesian.DynamicBayesianNetworkBase) → *pybnesian.ConditionalBayesianNetworkBase*

Returns the transition Bayesian network.

Returns
Transition Bayesian network.

type(*self*: pybnesian.DynamicBayesianNetworkBase) → *pybnesian.BayesianNetworkType*

Gets the underlying *BayesianNetworkType*.

Returns
The *BayesianNetworkType* of self.

variables(*self*: pybnesian.DynamicBayesianNetworkBase) → List[str]

Gets the variables of the dynamic Bayesian network.

Returns
Variables of the dynamic Bayesian network.

3.4.2 Bayesian Network Types

class pybnesian.**GaussianNetworkType**

Bases: *BayesianNetworkType*

This *BayesianNetworkType* represents a Gaussian network: homogeneous with *LinearGaussianCPD* factors.

__init__(*self*: pybnesian.GaussianNetworkType) → None

class pybnesian.**SemiparametricBNType**

Bases: *BayesianNetworkType*

This *BayesianNetworkType* represents a semiparametric Bayesian network: non-homogeneous with *LinearGaussianCPD* and *CKDE* factors for continuous data. The default is *LinearGaussianCPD*. It also supports discrete data using *DiscreteFactor*.

In a SemiparametricBN network, the discrete nodes can only have discrete parents.

__init__(*self:* pybnesian.SemiparametricBNType) → None

class pybnesian.**KDENetworkType**

Bases: `BayesianNetworkType`

This `BayesianNetworkType` represents a KDE Bayesian network: homogeneous with `CKDE` factors.

__init__(*self:* pybnesian.KDENetworkType) → None

class pybnesian.**DiscreteBNType**

Bases: `BayesianNetworkType`

This `BayesianNetworkType` represents a discrete Bayesian network: homogeneous with `DiscreteFactor` factors.

__init__(*self:* pybnesian.DiscreteBNType) → None

class pybnesian.**HomogeneousBNType**

Bases: `BayesianNetworkType`

__init__(*self:* pybnesian.HomogeneousBNType, *default_factor_type:* pybnesian.FactorType) → None

Initializes an `HomogeneousBNType` with a default node type.

Parameters

default_factor_type – Default factor type for all the nodes in the Bayesian network.

class pybnesian.**HeterogeneousBNType**

Bases: `BayesianNetworkType`

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.HeterogeneousBNType, default_factor_type: List[pybnesian.FactorType]) -> None

Initializes an `HeterogeneousBNType` with a list of default node types for all the data types.

Parameters

default_factor_type – Default factor type for all the nodes in the Bayesian network.

2. __init__(self: pybnesian.HeterogeneousBNType, default_factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]]) -> None

Initializes an `HeterogeneousBNType` with a default node type for a set of data types.

Parameters

default_factor_type – Default factor type depending on the factor type.

default_node_types(*self:* pybnesian.HeterogeneousBNType) → Dict[pyarrow.DataType, List[*pybnesian.FactorType*]]

Returns the dict of default `FactorType` for each data type.

Returns

dict of default `FactorType` for each data type.

single_default(*self:* pybnesian.HeterogeneousBNType) → bool

Checks whether the `HeterogeneousBNType` defines only a default `FactorType` for all the data types.

Returns

True if it defines a single `FactorType` for all the data types. False if different default `FactorType` is defined for different data types.

class pybnesian.**CLGNetworkType**

Bases: *BayesianNetworkType*

This *BayesianNetworkType* represents a conditional linear Gaussian (CLG) network: heterogeneous with *LinearGaussianCPD* factors for the continuous data and *DiscreteFactor* for the categorical data.

In a CLG network, the discrete nodes can only have discrete parents, while the continuous nodes can have discrete and continuous parents.

__init__(*self*: pybnesian.CLGNetworkType) → None

3.4.3 Bayesian Networks

class pybnesian.**BayesianNetwork**

Bases: *BayesianNetworkBase*

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str]) -> None

Initializes the *BayesianNetwork* with a given `type` and `nodes`.

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **nodes** – List of node names.

2. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *BayesianNetwork* with a given `type` and `nodes`. It specifies the `node_types` for the nodes.

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **nodes** – List of node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

3. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, arcs: List[Tuple[str, str]]) -> None

Initializes the *BayesianNetwork* with a given `type` and `arcs` (the nodes are extracted from the arcs).

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **arcs** – Arcs of the Bayesian network.

4. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *BayesianNetwork* with a given `type` and `arcs` (the nodes are extracted from the arcs). It specifies the `node_types` for the nodes.

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **arcs** – Arcs of the Bayesian network.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

5. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *BayesianNetwork* with a given `type`, `nodes` and `arcs`.

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **nodes** – List of node names.
- **arcs** – Arcs of the Bayesian network.

6. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *BayesianNetwork* with a given `type`, `nodes` and `arcs`. It specifies the `node_types` for the nodes.

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **nodes** – List of node names.
- **arcs** – Arcs of the Bayesian network.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

7. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, graph: pybnesian.Dag) -> None

Initializes the *BayesianNetwork* with a given `type`, and `graph`

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **graph** – *Dag* of the Bayesian network.

8. __init__(self: pybnesian.BayesianNetwork, type: pybnesian.BayesianNetworkType, graph: pybnesian.Dag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *BayesianNetwork* with a given `type`, and `graph`. It specifies the `node_types` for the nodes.

Parameters

- **type** – *BayesianNetworkType* of this Bayesian network.
- **graph** – *Dag* of the Bayesian network.

- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

can_have_cpd(*self:* pybnesian.BayesianNetwork, *node:* *str*) → bool

Checks whether a given node name can have an associated CPD. For

Parameters
node – A node name.

Returns
True if the given node can have a CPD, False otherwise.

check_compatible_cpd(*self:* pybnesian.BayesianNetwork, *cpd:* pybnesian.Factor) → None

Checks whether the given CPD is compatible with this Bayesian network.

Parameters
cpd – A *Factor*.

Returns
True if cpd is compatible with this Bayesian network, False otherwise.

graph(*self:* pybnesian.BayesianNetwork) → *pybnesian.Dag*

Gets the underlying graph of the Bayesian network.

Returns
Graph of the Bayesian network.

Concrete Bayesian Networks

These classes implements *BayesianNetwork* with an specific *BayesianNetworkType*. Thus, the constructors do not have the type parameter.

class pybnesian.**GaussianNetwork**

Bases: *BayesianNetwork*

This class implements a *BayesianNetwork* with the type *GaussianNetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.GaussianNetwork, nodes: List[str]) -> None

Initializes the *GaussianNetwork* with the given nodes.

Parameters
nodes – List of node names.

2. __init__(self: pybnesian.GaussianNetwork, arcs: List[Tuple[str, str]]) -> None

Initializes the *GaussianNetwork* with the given arcs (the nodes are extracted from the arcs).

Parameters
arcs – Arcs of the *GaussianNetwork*.

3. __init__(self: pybnesian.GaussianNetwork, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *GaussianNetwork* with the given nodes and arcs.

Parameters

- **nodes** – List of node names.

- **arcs** – Arcs of the *GaussianNetwork*.

4. __init__(self: pybnesian.GaussianNetwork, graph: pybnesian.Dag) -> None

Initializes the *GaussianNetwork* with the given graph.

Parameters
graph – *Dag* of the Bayesian network.

class pybnesian.**SemiparametricBN**

Bases: *BayesianNetwork*

This class implements a *BayesianNetwork* with the type *SemiparametricBNType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.SemiparametricBN, nodes: List[str]) -> None

Initializes the *SemiparametricBN* with the given nodes.

Parameters
nodes – List of node names.

2. __init__(self: pybnesian.SemiparametricBN, arcs: List[Tuple[str, str]]) -> None

Initializes the *SemiparametricBN* with the given arcs (the nodes are extracted from the arcs).

Parameters
arcs – Arcs of the *SemiparametricBN*.

3. __init__(self: pybnesian.SemiparametricBN, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *SemiparametricBN* with the given nodes and arcs.

Parameters

- **nodes** – List of node names.
- **arcs** – Arcs of the *SemiparametricBN*.

4. __init__(self: pybnesian.SemiparametricBN, graph: pybnesian.Dag) -> None

Initializes the *SemiparametricBN* with the given graph.

Parameters
graph – *Dag* of the Bayesian network.

5. __init__(self: pybnesian.SemiparametricBN, nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *SemiparametricBN* with the given nodes. It specifies the node_types for the nodes.

Parameters

- **nodes** – List of node names.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

6. __init__(self: pybnesian.SemiparametricBN, arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *SemiparametricBN* with the given `arcs` (the nodes are extracted from the arcs). It specifies the `node_types` for the nodes.

Parameters

- **arcs** – Arcs of the SemiparametricBN.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

7. __init__(self: pybnesian.SemiparametricBN, nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *SemiparametricBN* with the given `nodes` and `arcs`. It specifies the `node_types` for the nodes.

Parameters

- **nodes** – List of node names.
- **arcs** – Arcs of the *SemiparametricBN*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

8. __init__(self: pybnesian.SemiparametricBN, graph: pybnesian.Dag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *SemiparametricBN* with the given `graph`. It specifies the `node_types` for the nodes.

Parameters

- **graph** – *Dag* of the Bayesian network.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

`class pybnesian.`**`KDENetwork`**

Bases: *BayesianNetwork*

This class implements a *BayesianNetwork* with the type *KDENetworkType*.

`__init__`(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.KDENetwork, nodes: List[str]) -> None

Initializes the *KDENetwork* with the given `nodes`.

Parameters
nodes – List of node names.

2. __init__(self: pybnesian.KDENetwork, arcs: List[Tuple[str, str]]) -> None

Initializes the *KDENetwork* with the given `arcs` (the nodes are extracted from the arcs).

Parameters
arcs – Arcs of the *KDENetwork*.

3. __init__(self: pybnesian.KDENetwork, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *KDENetwork* with the given `nodes` and `arcs`.

Parameters

- **nodes** – List of node names.
- **arcs** – Arcs of the *KDENetwork*.

4. __init__(self: pybnesian.KDENetwork, graph: pybnesian.Dag) -> None

Initializes the *KDENetwork* with the given `graph`.

Parameters
graph – *Dag* of the Bayesian network.

class pybnesian.**DiscreteBN**

Bases: *BayesianNetwork*

This class implements a *BayesianNetwork* with the type *DiscreteBNType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DiscreteBN, nodes: List[str]) -> None

Initializes the *DiscreteBN* with the given `nodes`.

Parameters
nodes – List of node names.

2. __init__(self: pybnesian.DiscreteBN, arcs: List[Tuple[str, str]]) -> None

Initializes the *DiscreteBN* with the given `arcs` (the nodes are extracted from the arcs).

Parameters
arcs – Arcs of the *DiscreteBN*.

3. __init__(self: pybnesian.DiscreteBN, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *DiscreteBN* with the given `nodes` and `arcs`.

Parameters

- **nodes** – List of node names.
- **arcs** – Arcs of the *DiscreteBN*.

4. __init__(self: pybnesian.DiscreteBN, graph: pybnesian.Dag) -> None

Initializes the *DiscreteBN* with the given `graph`.

Parameters
graph – *Dag* of the Bayesian network.

class pybnesian.**HomogeneousBN**

Bases: *BayesianNetwork*

This class implements an homogeneous Bayesian network. This Bayesian network can be used with any *FactorType*. You can set the *FactorType* in the constructor.

`__init__`(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.HomogeneousBN, factor_type: pybnesian.FactorType, nodes: List[str]) -> None

Initializes the *`HomogeneousBN`* of `factor_type` with the given `nodes`.

Parameters

- **`factor_type`** – *`FactorType`* for all the nodes.
- **`nodes`** – List of node names.

2. __init__(self: pybnesian.HomogeneousBN, factor_type: pybnesian.FactorType, arcs: List[Tuple[str, str]]) -> None

Initializes the *`HomogeneousBN`* of `factor_type` with the given `arcs` (the nodes are extracted from the arcs).

Parameters

- **`factor_type`** – *`FactorType`* for all the nodes.
- **`arcs`** – Arcs of the *`HomogeneousBN`*.

3. __init__(self: pybnesian.HomogeneousBN, factor_type: pybnesian.FactorType, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *`HomogeneousBN`* of `factor_type` with the given `nodes` and `arcs`.

Parameters

- **`factor_type`** – *`FactorType`* for all the nodes.
- **`nodes`** – List of node names.
- **`arcs`** – Arcs of the *`HomogeneousBN`*.

4. __init__(self: pybnesian.HomogeneousBN, factor_type: pybnesian.FactorType, graph: pybnesian.Dag) -> None

Initializes the *`HomogeneousBN`* of `factor_type` with the given `graph`.

Parameters

- **`factor_type`** – *`FactorType`* for all the nodes.
- **`graph`** – *`Dag`* of the Bayesian network.

`class pybnesian.`**`HeterogeneousBN`**

Bases: *`BayesianNetwork`*

This class implements an heterogeneous Bayesian network. This Bayesian network accepts a different *`FactorType`* for each node. You can set the default *`FactorType`* in the constructor.

`__init__`(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str]) -> None

Initializes the *`HeterogeneousBN`* of default `factor_type` with the given `nodes`.

Parameters

- **factor_type** – List of default *FactorType* for the Bayesian network.
- **nodes** – List of node names.

2. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of default `factor_type` with the given `nodes` and `node_types`.

Parameters

- **factor_type** – List of default *FactorType* for the Bayesian network.
- **nodes** – List of node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

3. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], arcs: List[Tuple[str, str]]) -> None

Initializes the *HeterogeneousBN* of default `factor_type` with the given `arcs` (the nodes are extracted from the arcs).

Parameters

- **factor_type** – List of default *FactorType* for the Bayesian network.
- **arcs** – Arcs of the *HeterogeneousBN*.

4. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of default `factor_type` with the given `arcs` (the nodes are extracted from the arcs) and `node_types`.

Parameters

- **factor_type** – List of default *FactorType* for the Bayesian network.
- **arcs** – Arcs of the *HeterogeneousBN*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

5. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *HeterogeneousBN* of default `factor_type` with the given `nodes` and `arcs`.

Parameters

- **factor_type** – List of default *FactorType* for the Bayesian network.
- **nodes** – List of node names.
- **arcs** – Arcs of the *HeterogeneousBN*.

6. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of default `factor_type` with the given `nodes`, `arcs` and `node_types`.

Parameters

- **factor_type** – List of default *FactorType* for the Bayesian network.
- **nodes** – List of node names.
- **arcs** – Arcs of the *HeterogeneousBN*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

7. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], graph: pybnesian.Dag) -> None

Initializes the *HeterogeneousBN* of default `factor_type` with the given `graph`.

Parameters

- **factor_type** – Default *FactorType* for the Bayesian network.
- **graph** – *Dag* of the Bayesian network.

8. __init__(self: pybnesian.HeterogeneousBN, factor_type: List[pybnesian.FactorType], graph: pybnesian.Dag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of default `factor_type` with the given `graph` and `node_types`.

Parameters

- **factor_type** – Default *FactorType* for the Bayesian network.
- **graph** – *Dag* of the Bayesian network.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

9. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str]) -> None

Initializes the *HeterogeneousBN* of different default `factor_types`, with the given `nodes`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.

10. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of different default `factor_types`, with the given `nodes` and `node_types`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.

- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

11. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], arcs: List[Tuple[str, str]]) -> None

Initializes the *HeterogeneousBN* of different default `factor_types` with the given `arcs` (the nodes are extracted from the arcs).

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **arcs** – Arcs of the *HeterogeneousBN*.

12. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of different default `factor_types` with the given `arcs` (the nodes are extracted from the arcs) and `node_types`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **arcs** – Arcs of the *HeterogeneousBN*.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

13. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *HeterogeneousBN* of different default `factor_types` with the given `nodes` and `arcs`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.
- **arcs** – Arcs of the *HeterogeneousBN*.

14. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of different default `factor_types` with the given `nodes`, `arcs` and `node_types`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.

- **arcs** – Arcs of the *HeterogeneousBN*.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

15. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], graph: pybnesian.Dag) -> None

Initializes the *HeterogeneousBN* of different default factor_types with the given graph.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **graph** – *Dag* of the Bayesian network.

16. __init__(self: pybnesian.HeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], graph: pybnesian.Dag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *HeterogeneousBN* of different default factor_types with the given graph and node_types.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **graph** – *Dag* of the Bayesian network.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

class pybnesian.**CLGNetwork**

Bases: *BayesianNetwork*

This class implements a *BayesianNetwork* with the type *CLGNetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.CLGNetwork, nodes: List[str]) -> None

Initializes the *CLGNetwork* with the given nodes.

Parameters

nodes – List of node names.

2. __init__(self: pybnesian.CLGNetwork, arcs: List[Tuple[str, str]]) -> None

Initializes the *CLGNetwork* with the given arcs (the nodes are extracted from the arcs).

Parameters

arcs – Arcs of the *CLGNetwork*.

3. __init__(self: pybnesian.CLGNetwork, nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *CLGNetwork* with the given nodes and arcs.

Parameters

- **nodes** – List of node names.
- **arcs** – Arcs of the *CLGNetwork*.

4. __init__(self: pybnesian.CLGNetwork, graph: pybnesian.Dag) -> None

Initializes the *CLGNetwork* with the given `graph`.

Parameters
graph – *Dag* of the Bayesian network.

5. __init__(self: pybnesian.CLGNetwork, nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *CLGNetwork* with the given `nodes`. It specifies the `node_types` for the nodes.

Parameters

- **nodes** – List of node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

6. __init__(self: pybnesian.CLGNetwork, arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *CLGNetwork* with the given `arcs` (the nodes are extracted from the arcs). It specifies the `node_types` for the nodes.

Parameters

- **arcs** – Arcs of the *CLGNetwork*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

7. __init__(self: pybnesian.CLGNetwork, nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *CLGNetwork* with the given `nodes` and `arcs`. It specifies the `node_types` for the nodes.

Parameters

- **nodes** – List of node names.
- **arcs** – Arcs of the *CLGNetwork*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

8. __init__(self: pybnesian.CLGNetwork, graph: pybnesian.Dag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *CLGNetwork* with the given `graph`. It specifies the `node_types` for the nodes.

Parameters

- **graph** – *Dag* of the Bayesian network.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

3.4.4 Conditional Bayesian Networks

class pybnesian.**ConditionalBayesianNetwork**

Bases: *ConditionalBayesianNetworkBase*

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalBayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalBayesianNetwork* with a given `type`, `nodes` and `interface_nodes`.

Parameters

- **type** – *BayesianNetworkType* of this conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalBayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str], interface_nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalBayesianNetwork* with a given `type`, `nodes` and `interface_nodes`. It specifies the `node_types` for the nodes.

Parameters

- **type** – *BayesianNetworkType* of this conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

3. __init__(self: pybnesian.ConditionalBayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalBayesianNetwork* with a given `type`, `nodes`, `interface_nodes` and `arcs`.

Parameters

- **type** – *BayesianNetworkType* of this conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the conditional Bayesian network.

4. __init__(self: pybnesian.ConditionalBayesianNetwork, type: pybnesian.BayesianNetworkType, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalBayesianNetwork* with a given `type`, `nodes`, `interface_nodes` and `arcs`. It specifies the `node_types` for the nodes.

Parameters

- **type** – *BayesianNetworkType* of this conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the conditional Bayesian network.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

5. __init__(self: pybnesian.ConditionalBayesianNetwork, type: pybnesian.BayesianNetworkType, graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalBayesianNetwork* with a given `type`, and `graph`

Parameters

- **type** – *BayesianNetworkType* of this conditional Bayesian network.
- **graph** – *ConditionalDag* of the conditional Bayesian network.

6. __init__(self: pybnesian.ConditionalBayesianNetwork, type: pybnesian.BayesianNetworkType, graph: pybnesian.ConditionalDag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalBayesianNetwork* with a given `type`, and `graph`. It specifies the `node_types` for the nodes.

Parameters

- **type** – *BayesianNetworkType* of this conditional Bayesian network.
- **graph** – *ConditionalDag* of the conditional Bayesian network.
- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

can_have_cpd(*self: pybnesian.ConditionalBayesianNetwork*, *node: str*) → bool

Checks whether a given node name can have an associated CPD. For

Parameters
node – A node name.

Returns
True if the given node can have a CPD, False otherwise.

check_compatible_cpd(*self: pybnesian.ConditionalBayesianNetwork*, *cpd: pybnesian.Factor*) → None

Checks whether the given CPD is compatible with this Bayesian network.

Parameters
cpd – A *Factor*.

Returns
True if `cpd` is compatible with this Bayesian network, False otherwise.

graph(*self: pybnesian.ConditionalBayesianNetwork*) → *pybnesian.ConditionalDag*

Gets the underlying graph of the Bayesian network.

Returns
Graph of the Bayesian network.

Concrete Conditional Bayesian Networks

These classes implements *ConditionalBayesianNetwork* with an specific *BayesianNetworkType*. Thus, the constructors do not have the `type` parameter.

class pybnesian.**ConditionalGaussianNetwork**

Bases: *ConditionalBayesianNetwork*

This class implements a *ConditionalBayesianNetwork* with the type *GaussianNetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalGaussianNetwork, nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalGaussianNetwork* with the given `nodes` and `interface_nodes`.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalGaussianNetwork, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalGaussianNetwork* with the given `nodes`, `interface_nodes` and `arcs`.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalGaussianNetwork*.

3. __init__(self: pybnesian.ConditionalGaussianNetwork, graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalGaussianNetwork* with the given `graph`.

Parameters

graph – *ConditionalDag* of the conditional Bayesian network.

class pybnesian.**ConditionalSemiparametricBN**

Bases: *ConditionalBayesianNetwork*

This class implements a *ConditionalBayesianNetwork* with the type *SemiparametricBNType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalSemiparametricBN, nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalSemiparametricBN* with the given `nodes` and `interface_nodes`.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalSemiparametricBN, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalSemiparametricBN* with the given `nodes`, `interface_nodes` and `arcs`.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalSemiparametricBN*.

3. __init__(self: pybnesian.ConditionalSemiparametricBN, graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalSemiparametricBN* with the given `graph`.

Parameters
graph – *ConditionalDag* of the conditional Bayesian network.

4. __init__(self: pybnesian.ConditionalSemiparametricBN, nodes: List[str], interface_nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalSemiparametricBN* with the given `nodes` and `interface_nodes`. It specifies the `node_types` for the nodes.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

5. __init__(self: pybnesian.ConditionalSemiparametricBN, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalSemiparametricBN* with the given `nodes`, `interface_nodes` and `arcs`. It specifies the `node_types` for the nodes.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalSemiparametricBN*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

6. __init__(self: pybnesian.ConditionalSemiparametricBN, graph: pybnesian.ConditionalDag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalSemiparametricBN* with the given `graph`. It specifies the `node_types` for the nodes.

Parameters

- **graph** – *ConditionalDag* of the conditional Bayesian network.

- **node_types** – List of node type tuples (node, *FactorType*) that specifies the type for each node.

class pybnesian.**ConditionalKDENetwork**

Bases: *ConditionalBayesianNetwork*

This class implements a *ConditionalBayesianNetwork* with the type *KDENetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalKDENetwork, nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalKDENetwork* with the given nodes and interface_nodes.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalKDENetwork, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalKDENetwork* with the given nodes, interface_nodes and arcs.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalKDENetwork*.

3. __init__(self: pybnesian.ConditionalKDENetwork, graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalKDENetwork* with the given graph.

Parameters

graph – *ConditionalDag* of the conditional Bayesian network.

class pybnesian.**ConditionalDiscreteBN**

Bases: *ConditionalBayesianNetwork*

This class implements a *ConditionalBayesianNetwork* with the type *DiscreteBNType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalDiscreteBN, nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalDiscreteBN* with the given nodes and interface_nodes.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalDiscreteBN, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalDiscreteBN* with the given `nodes`, `interface_nodes` and `arcs`.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalDiscreteBN*.

3. __init__(self: pybnesian.ConditionalDiscreteBN, graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalDiscreteBN* with the given `graph`.

Parameters
graph – *ConditionalDag* of the conditional Bayesian network.

class pybnesian.**ConditionalHomogeneousBN**

Bases: *ConditionalBayesianNetwork*

This class implements an homogeneous conditional Bayesian network. This conditional Bayesian network can be used with any *FactorType*. You can set the *FactorType* in the constructor.

__init__(**args, **kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalHomogeneousBN, factor_type: pybnesian.FactorType, nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalHomogeneousBN* of `factor_type` with the given `nodes` and `interface_nodes`.

Parameters

- **factor_type** – *FactorType* for all the nodes.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalHomogeneousBN, factor_type: pybnesian.FactorType, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalHomogeneousBN* of `factor_type` with the given `nodes`, `interface_nodes` and `arcs`.

Parameters

- **factor_type** – *FactorType* for all the nodes.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalHomogeneousBN*.

3. __init__(self: pybnesian.ConditionalHomogeneousBN, factor_type: pybnesian.FactorType, graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalHomogeneousBN* of `factor_type` with the given `graph`.

Parameters

- **factor_type** – *FactorType* for all the nodes.

- **graph** – *ConditionalDag* of the conditional Bayesian network.

class pybnesian.**ConditionalHeterogeneousBN**

Bases: *ConditionalBayesianNetwork*

This class implements an heterogeneous conditional Bayesian network. This conditional Bayesian network accepts a different *FactorType* for each node. You can set the default *FactorType* in the constructor.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalHeterogeneousBN* of default `factor_type` with the given `nodes` and `interface_nodes`.

Parameters

- **factor_type** – List of default *FactorType* for the conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str], interface_nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalHeterogeneousBN* of default `factor_type` with the given `nodes`, `interface_nodes` and `node_types`.

Parameters

- **factor_type** – List of default *FactorType* for the conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

3. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalHeterogeneousBN* of default `factor_type` with the given `nodes`, `interface_nodes` and `arcs`.

Parameters

- **factor_type** – List of default *FactorType* for the conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalHeterogeneousBN*.

4. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_type: List[pybnesian.FactorType], nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalHeterogeneousBN* of default `factor_type` with the given `nodes`, `interface_nodes`, `arcs` and `node_types`.

Parameters

- **factor_type** – List of default *FactorType* for the conditional Bayesian network.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalHeterogeneousBN*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

5. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_type: List[pybnesian.FactorType], graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalHeterogeneousBN* of default `factor_type` with the given `graph`.

Parameters

- **factor_type** – List of default *FactorType* for the conditional Bayesian network.
- **graph** – *ConditionalDag* of the conditional Bayesian network.

6. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_type: List[pybnesian.FactorType], graph: pybnesian.ConditionalDag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalHeterogeneousBN* of default `factor_type` with the given `graph` and `node_types`.

Parameters

- **factor_type** – List of default *FactorType* for the conditional Bayesian network.
- **graph** – *ConditionalDag* of the conditional Bayesian network.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

7. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalHeterogeneousBN* of different default `factor_types` with the given `nodes` and `interface_nodes`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

8. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str], interface_nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalHeterogeneousBN* of different default `factor_types` with the given `nodes`, `interface_nodes` and `node_types`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

9. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalHeterogeneousBN* of different default `factor_types` with the given `nodes`, `interface_nodes` and `arcs`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalHeterogeneousBN*.

10. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalHeterogeneousBN* of different default `factor_types` with the given `nodes`, `interface_nodes`, `arcs` and `node_types`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalHeterogeneousBN*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

11. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalHeterogeneousBN* of different default `factor_types` with the given `graph`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.

- **graph** – *ConditionalDag* of the conditional Bayesian network.

12. __init__(self: pybnesian.ConditionalHeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], graph: pybnesian.ConditionalDag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalHeterogeneousBN* of different default `factor_types` with the given `graph` and `node_types`.

Parameters

- **factor_types** – Default *FactorType* for the Bayesian network for each different data type.
- **graph** – *ConditionalDag* of the conditional Bayesian network.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

class pybnesian.**ConditionalCLGNetwork**

Bases: *ConditionalBayesianNetwork*

This class implements a *ConditionalBayesianNetwork* with the type *CLGNetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.ConditionalCLGNetwork, nodes: List[str], interface_nodes: List[str]) -> None

Initializes the *ConditionalCLGNetwork* with the given `nodes` and `interface_nodes`.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.

2. __init__(self: pybnesian.ConditionalCLGNetwork, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]]) -> None

Initializes the *ConditionalCLGNetwork* with the given `nodes`, `interface_nodes` and `arcs`.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalCLGNetwork*.

3. __init__(self: pybnesian.ConditionalCLGNetwork, graph: pybnesian.ConditionalDag) -> None

Initializes the *ConditionalCLGNetwork* with the given `graph`.

Parameters

graph – *ConditionalDag* of the conditional Bayesian network.

4. __init__(self: pybnesian.ConditionalCLGNetwork, nodes: List[str], interface_nodes: List[str], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalCLGNetwork* with the given `nodes` and `interface_nodes`. It specifies the `node_types` for the nodes.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

5. __init__(self: pybnesian.ConditionalCLGNetwork, nodes: List[str], interface_nodes: List[str], arcs: List[Tuple[str, str]], node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalCLGNetwork* with the given `nodes`, `interface_nodes` and `arcs`. It specifies the `node_types` for the nodes.

Parameters

- **nodes** – List of node names.
- **interface_nodes** – List of interface node names.
- **arcs** – Arcs of the *ConditionalCLGNetwork*.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

6. __init__(self: pybnesian.ConditionalCLGNetwork, graph: pybnesian.ConditionalDag, node_types: List[Tuple[str, pybnesian.FactorType]]) -> None

Initializes the *ConditionalCLGNetwork* with the given `graph`. It specifies the `node_types` for the nodes.

Parameters

- **graph** – *ConditionalDag* of the conditional Bayesian network.
- **node_types** – List of node type tuples (`node`, *FactorType*) that specifies the type for each node.

3.4.5 Dynamic Bayesian Networks

class pybnesian.**DynamicBayesianNetwork**

Bases: *DynamicBayesianNetworkBase*

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicBayesianNetwork, type: pybnesian.BayesianNetworkType, variables: List[str], markovian_order: int) -> None

Initializes the *DynamicBayesianNetwork* with the given `variables` and `markovian_order`. It creates empty the static and transition Bayesian networks with the given `type`.

Parameters

- **type** – *BayesianNetworkType* of the static and transition Bayesian networks.
- **variables** – List of node names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicBayesianNetwork, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicBayesianNetwork* with the given `variables` and `markovian_order`. The static and transition Bayesian networks are initialized with `static_bn` and `transition_bn` respectively.

Both `static_bn` and `transition` must contain the expected nodes:

- For the static network, it must contain the nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.
- For the transition network, it must contain the nodes `[variable_name]_t_0`, and the interface nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.

The static and transition networks must have the same *BayesianNetworkType*.

Parameters

- **variables** – List of node names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

Concrete Dynamic Bayesian Networks

These classes implements *DynamicBayesianNetwork* with an specific *BayesianNetworkType*. Thus, the constructors do not have the `type` parameter.

class pybnesian.**DynamicGaussianNetwork**

Bases: *DynamicBayesianNetwork*

This class implements a *DynamicBayesianNetwork* with the type *GaussianNetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicGaussianNetwork, variables: List[str], markovian_order: int) -> None

Initializes the *DynamicGaussianNetwork* with the given `variables` and `markovian_order`. It creates empty static and transition Bayesian networks.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicGaussianNetwork, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicGaussianNetwork* with the given `variables` and `markovian_order`. The static and transition Bayesian networks are initialized with `static_bn` and `transition_bn` respectively.

Both `static_bn` and `transition_bn` must contain the expected nodes:

- For the static network, it must contain the nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.
- For the transition network, it must contain the nodes `[variable_name]_t_0`, and the interface nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

class pybnesian.**DynamicSemiparametricBN**

Bases: *DynamicBayesianNetwork*

This class implements a *DynamicBayesianNetwork* with the type *SemiparametricBNType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicSemiparametricBN, variables: List[str], markovian_order: int) -> None

Initializes the *DynamicSemiparametricBN* with the given `variables` and `markovian_order`. It creates empty static and transition Bayesian networks.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicSemiparametricBN, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicSemiparametricBN* with the given `variables` and `markovian_order`. The static and transition Bayesian networks are initialized with `static_bn` and `transition_bn` respectively.

Both `static_bn` and `transition_bn` must contain the expected nodes:

- For the static network, it must contain the nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.
- For the transition network, it must contain the nodes `[variable_name]_t_0`, and the interface nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

class pybnesian.**DynamicKDENetwork**

Bases: *DynamicBayesianNetwork*

This class implements a *DynamicBayesianNetwork* with the type *KDENetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicKDENetwork, variables: List[str], markovian_order: int) -> None

Initializes the *DynamicKDENetwork* with the given `variables` and `markovian_order`. It creates empty static and transition Bayesian networks.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicKDENetwork, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicKDENetwork* with the given `variables` and `markovian_order`. The static and transition Bayesian networks are initialized with `static_bn` and `transition_bn` respectively.

Both `static_bn` and `transition_bn` must contain the expected nodes:

- For the static network, it must contain the nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.
- For the transition network, it must contain the nodes `[variable_name]_t_0`, and the interface nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

class pybnesian.**DynamicDiscreteBN**

Bases: *DynamicBayesianNetwork*

This class implements a *DynamicBayesianNetwork* with the type *DiscreteBN*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicDiscreteBN, variables: List[str], markovian_order: int) -> None

Initializes the *DynamicDiscreteBN* with the given `variables` and `markovian_order`. It creates empty static and transition Bayesian networks.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicDiscreteBN, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicDiscreteBN* with the given `variables` and `markovian_order`. The static and transition Bayesian networks are initialized with `static_bn` and `transition_bn` respectively.

Both `static_bn` and `transition_bn` must contain the expected nodes:

- For the static network, it must contain the nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.
- For the transition network, it must contain the nodes `[variable_name]_t_0`, and the interface nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

class pybnesian.**DynamicHomogeneousBN**

Bases: *DynamicBayesianNetwork*

This class implements an homogeneous dynamic Bayesian network. This dynamic Bayesian network can be used with any *FactorType*. You can set the *FactorType* in the constructor.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicHomogeneousBN, factor_type: pybnesian.FactorType, variables: List[str], markovian_order: int) -> None

Initializes the *DynamicHomogeneousBN* of `factor_type` with the given `variables` and `markovian_order`. It creates empty static and transition Bayesian networks.

Parameters

- **factor_type** – *FactorType* for all the nodes.
- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicHomogeneousBN, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicHomogeneousBN* with the given `variables` and `markovian_order`. The static and transition Bayesian networks are initialized with `static_bn` and `transition_bn` respectively.

Both `static_bn` and `transition_bn` must contain the expected nodes:

- For the static network, it must contain the nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.
- For the transition network, it must contain the nodes `[variable_name]_t_0`, and the interface nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.

The type of `static_bn` and `transition_bn` must be *HomogeneousBNType*.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

class pybnesian.**DynamicHeterogeneousBN**

Bases: *DynamicBayesianNetwork*

This class implements an heterogeneous dynamic Bayesian network. This dynamic Bayesian network accepts a different `FactorType` for each node. You can set the default `FactorType` in the constructor.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicHeterogeneousBN, factor_type: List[pybnesian.FactorType], variables: List[str], markovian_order: int) -> None

Initializes the *DynamicHeterogeneousBN* of default `factor_type` with the given `variables` and `markovian_order`. It creates empty static and transition Bayesian networks.

Parameters

- **factor_type** – Default `FactorType` for the dynamic Bayesian network.
- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicHeterogeneousBN, factor_types: Dict[pyarrow.DataType, List[pybnesian.FactorType]], variables: List[str], markovian_order: int) -> None

Initializes the *DynamicHeterogeneousBN* of different default `factor_types` with the given `variables` and `markovian_order`. It creates empty static and transition Bayesian networks.

Parameters

- **factor_types** – Default `FactorType` for the Bayesian network for each different data type.
- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

3. __init__(self: pybnesian.DynamicHeterogeneousBN, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicHeterogeneousBN* with the given `variables` and `markovian_order`. The static and transition Bayesian networks are initialized with `static_bn` and `transition_bn` respectively.

Both `static_bn` and `transition_bn` must contain the expected nodes:

- For the static network, it must contain the nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.
- For the transition network, it must contain the nodes `[variable_name]_t_0`, and the interface nodes from `[variable_name]_t_1` to `[variable_name]_t_[markovian_order]`.

The type of static_bn and transition_bn must be *HeterogeneousBNType*.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

class pybnesian.**DynamicCLGNetwork**

Bases: *DynamicBayesianNetwork*

This class implements a *DynamicBayesianNetwork* with the type *CLGNetworkType*.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.DynamicCLGNetwork, variables: List[str], markovian_order: int) -> None

Initializes the *DynamicCLGNetwork* with the given variables and markovian_order. It creates empty static and transition Bayesian networks.

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.

2. __init__(self: pybnesian.DynamicCLGNetwork, variables: List[str], markovian_order: int, static_bn: pybnesian.BayesianNetworkBase, transition_bn: pybnesian.ConditionalBayesianNetworkBase) -> None

Initializes the *DynamicCLGNetwork* with the given variables and markovian_order. The static and transition Bayesian networks are initialized with static_bn and transition_bn respectively.

Both static_bn and transition_bn must contain the expected nodes:

- For the static network, it must contain the nodes from [variable_name]_t_1 to [variable_name]_t_[markovian_order].
- For the transition network, it must contain the nodes [variable_name]_t_0, and the interface nodes from [variable_name]_t_1 to [variable_name]_t_[markovian_order].

Parameters

- **variables** – List of variable names.
- **markovian_order** – Markovian order of the dynamic Bayesian network.
- **static_bn** – Static Bayesian network.
- **transition_bn** – Transition Bayesian network.

3.5 Learning module

PyBNesian implements different algorithms to learn Bayesian networks from data. It includes the parameter learning and the structure learning.

3.5.1 Parameter Learning

PyBNesian implements learning parameter learning for `Factor` from data.

Currently, it only implements Maximum Likelihood Estimation (MLE) for `LinearGaussianCPD` and `DiscreteFactor`.

`pybnesian.MLE`(*factor_type:* pybnesian.FactorType) → object

Generates an MLE estimator for the given `factor_type`.

Parameters
`factor_type` – A `FactorType`.

Returns
An MLE estimator.

`class pybnesian.LinearGaussianParams`

`__init__`(*self:* pybnesian.LinearGaussianParams, *beta: numpy.ndarray[numpy.float64[m, 1]]*, *variance: float*) → None

Initializes `MLELinearGaussianParams` with the given `beta` and `variance`.

`property beta`

The beta vector of parameters. The beta vector is a `numpy.ndarray` vector of type `numpy.float64` with size `len(evidence) + 1`.

`beta[0]` is always the intercept coefficient and `beta[i]` is the corresponding coefficient for the variable `evidence[i-1]` for `i > 0`.

`property variance`

The variance of the linear Gaussian CPD. This is a `float` value.

`class pybnesian.MLELinearGaussianCPD`

Maximum Likelihood Estimator (MLE) for `LinearGaussianCPD`.

This class is created using the function `MLE()`.

```
>>> from pybnesian import LinearGaussianCPDType, MLE
>>> mle = MLE(LinearGaussianCPDType())
```

`estimate`(*self:* pybnesian.MLELinearGaussianCPD, *df: DataFrame*, *variable: str*, *evidence: List[str]*) → *pybnesian.LinearGaussianParams*

Estimate the parameters of a `LinearGaussianCPD` with the given `variable` and `evidence`. The parameters are estimated with maximum likelihood estimation on the data `df`.

Parameters

- `df` – DataFrame to estimate the parameters.
- `variable` – Variable of the `LinearGaussianCPD`.
- `evidence` – Evidence of the `LinearGaussianCPD`.

class pybnesian.**DiscreteFactorParams**

__init__(*self: pybnesian.DiscreteFactorParams*, *logprob: numpy.ndarray[numpy.float64]*) → None

Initializes *DiscreteFactorParams* with a given `logprob` (see *DiscreteFactorParams.logprob*).

property logprob

A conditional probability table (in log domain). This is a `numpy.ndarray` with `(len(evidence) + 1)` dimensions. The first dimension corresponds to the variable being modelled, while the rest corresponds to the evidence variables.

Each dimension have a shape equal to the cardinality of the corresponding variable and each value is equal to the log-probability of the assignments for all the variables.

For example, if we are modelling the parameters for the *DiscreteFactor* of a variable with two evidence variables:

$$\text{logprob}[i, j, k] = \log P(\text{variable} = i \mid \text{evidence}_1 = j, \text{evidence}_2 = k)$$

As logprob defines a conditional probability table, the sum of conditional probabilities must sum 1.

```
>>> from pybnesian import DiscreteFactorType, MLE
>>> variable = np.random.choice(["a1", "a2", "a3"], size=50, p=[0.5, 0.3, 0.2])
>>> evidence = np.random.choice(["b1", "b2"], size=50, p=[0.5, 0.5])
>>> df = pd.DataFrame({'variable': variable, 'evidence': evidence}, dtype=
↪"category")
>>> mle = MLE(DiscreteFactorType())
>>> params = mle.estimate(df, "variable", ["evidence"])
>>> assert params.logprob.ndim == 2
>>> assert params.logprob.shape == (3, 2)
>>> ss = np.exp(params.logprob).sum(axis=0)
>>> assert np.all(np.isclose(ss, np.ones(2)))
```

3.5.2 Structure Scores

This section includes different learning scores that evaluate the goodness of a Bayesian network. This is used for the score-and-search learning algorithms such as *GreedyHillClimbing*, *MMHC* and *DMMHC*.

Abstract classes

class pybnesian.**Score**

A *Score* scores Bayesian network structures.

__init__(*self: pybnesian.Score*) → None

Initializes a *Score*.

__str__(*self: pybnesian.Score*) → str

compatible_bn(*self: pybnesian.Score*, *model: BayesianNetworkBase or ConditionalBayesianNetworkBase*) → bool

Checks whether the `model` is compatible (can be used) with this *Score*.

Parameters

model – A Bayesian network model.

Returns
True if the Bayesian network model is compatible with this *Score*, False otherwise.

data(*self: pybnesian.Score*) → DataFrame

Returns the DataFrame used to calculate the score and local scores.

Returns
DataFrame used to calculate the score. If the score do not use data, it returns None.

has_variables(*self: pybnesian.Score, variables: str or List[str]*) → bool

Checks whether this *Score* has the given `variables`.

Parameters
variables – Name or list of variables.

Returns
True if the *Score* is defined over the set of `variables`, False otherwise.

local_score(**args, **kwargs*)

Overloaded function.

1. local_score(self: pybnesian.Score, model: pybnesian.ConditionalBayesianNetworkBase, variable: str) -> float
2. local_score(self: pybnesian.Score, model: pybnesian.BayesianNetworkBase, variable: str) -> float

Returns the local score value of a node `variable` in the `model`.

For example:

```
>>> score.local_score(m, "a")
```

returns the local score of node `"a"` in the model `m`. This method assumes that the parents in the score are `m.parents("a")` and its node type is `m.node_type("a")`.

Parameters

- **model** – Bayesian network model.
- **variable** – A variable name.

Returns
Local score value of `node` in the `model`.

3. local_score(self: pybnesian.Score, model: pybnesian.ConditionalBayesianNetworkBase, variable: str, evidence: List[str]) -> float
4. local_score(self: pybnesian.Score, model: pybnesian.BayesianNetworkBase, variable: str, evidence: List[str]) -> float

Returns the local score value of a node `variable` in the `model` if it had `evidence` as parents.

For example:

```
>>> score.local_score(m, "a", ["b"])
```

returns the local score of node `"a"` in the model `m`, with `["b"]` as parents. This method assumes that the node type of `"a"` is `m.node_type("a")`.

Parameters

- **model** – Bayesian network model.

- **variable** – A variable name.
- **evidence** – A list of parent names.

Returns
Local score value of `node` in the `model` with `evidence` as parents.

local_score_node_type(*self: pybnesian.Score*, *model: pybnesian.BayesianNetworkBase*, *variable_type: pybnesian.FactorType*, *variable: str*, *evidence: List[str]*) → float

Returns the local score value of a node `variable` in the `model` if its conditional distribution were a `variable_type` factor and it had `evidence` as parents.

For example:

```
>>> score.local_score(m, LinearGaussianCPDType(), "a", ["b"])
```

returns the local score of node `"a"` in the model `m`, with `["b"]` as parents assuming the conditional distribution of `"a"` is a *LinearGaussianCPD*.

Parameters

- **model** – Bayesian network model.
- **variable_type** – The *FactorType* of the node `variable`.
- **variable** – A variable name.
- **evidence** – A list of parent names.

Returns
Local score value of `node` in the `model` with `evidence` as parents and `variable_type` as conditional distribution.

score(*self: pybnesian.Score*, *model: BayesianNetworkBase or ConditionalBayesianNetworkBase*) → float

Returns the score value of the `model`.

Parameters
model – Bayesian network model.

Returns
Score value of `model`.

class pybnesian.ValidatedScore

Bases: *Score*

A *ValidatedScore* is a score with training and validation scores. In a *ValidatedScore*, the training is driven by the training score through the functions *Score.score()*, `Score.local_score_variable()`, *Score.local_score()* and *Score.local_score_node_type()*). The convergence of the structure is evaluated using a validation likelihood (usually defined over different data) through the functions *ValidatedScore.vscore()*, `ValidatedScore.vlocal_score_variable()`, *ValidatedScore.vlocal_score()* and *ValidatedScore.vlocal_score_node_type()*.

__init__(*self: pybnesian.ValidatedScore*) → None

vlocal_score(**args*, ***kwargs*)

Overloaded function.

1. vlocal_score(self: pybnesian.ValidatedScore, model: pybnesian.ConditionalBayesianNetworkBase, variable: str) -> float
2. vlocal_score(self: pybnesian.ValidatedScore, model: pybnesian.BayesianNetworkBase, variable: str) -> float

vlocal_score(self: pybnesian.ValidatedScore, model: BayesianNetworkBase or ConditionalBayesianNetworkBase, variable: str) -> float

Returns the validated local score value of a node `variable` in the `model`.

For example:

```
>>> score.local_score(m, "a")
```

returns the validated local score of node `"a"` in the model `m`. This method assumes that the parents of `"a"` are `m.parents("a")` and its node type is `m.node_type("a")`.

Parameters

- **model** – Bayesian network model.
- **variable** – A variable name.

Returns
Validated local score value of `node` in the `model`.

3. vlocal_score(self: pybnesian.ValidatedScore, arg0: pybnesian.ConditionalBayesianNetworkBase, arg1: str, arg2: List[str]) -> float
4. vlocal_score(self: pybnesian.ValidatedScore, model: pybnesian.BayesianNetworkBase, variable: str, evidence: List[str]) -> float

vlocal_score(self: pybnesian.ValidatedScore, model: BayesianNetworkBase or ConditionalBayesianNetworkBase, variable: str, evidence: List[str]) -> float

Returns the validated local score value of a node `variable` in the `model` if it had `evidence` as parents.

For example:

```
>>> score.local_score(m, "a", ["b"])
```

returns the validated local score of node `"a"` in the model `m`, with `["b"]` as parents. This method assumes that the node type of `"a"` is `m.node_type("a")`.

Parameters

- **model** – Bayesian network model.
- **variable** – A variable name.
- **evidence** – A list of parent names.

Returns
Validated local score value of `node` in the `model` with `evidence` as parents.

vlocal_score_node_type(*self:* pybnesian.ValidatedScore, *model:* pybnesian.BayesianNetworkBase, *variable_type:* pybnesian.FactorType, *variable:* *str*, *evidence: List[str]*) → float

Returns the validated local score value of a node `variable` in the `model` if its conditional distribution were a `variable_type` factor and it had `evidence` as parents.

For example:

```
>>> score.vlocal_score(m, LinearGaussianCPDType(), "a", ["b"])
```

returns the validated local score of node `"a"` in the model `m`, with `["b"]` as parents assuming the conditional distribution of `"a"` is a *LinearGaussianCPD*.

Parameters

- **model** – Bayesian network model.
- **variable_type** – The *FactorType* of the node variable.
- **variable** – A variable name.
- **evidence** – A list of parent names.

Returns
Validated local score value of node in the model with evidence as parents and variable_type as conditional distribution.

vscore(*self: pybnesian.ValidatedScore*, *model: BayesianNetworkBase or ConditionalBayesianNetworkBase*) → float

Returns the validated score value of the model.

Parameters
model – Bayesian network model.

Returns
Validated score value of model.

class pybnesian.**DynamicScore**

A *DynamicScore* adapts the static *Score* to learn dynamic Bayesian networks. It generates a static and a transition score to learn the static and transition components of the dynamic Bayesian network.

The dynamic scores are usually implemented using a *DynamicDataFrame* with the methods *DynamicDataFrame.static_df* and *DynamicDataFrame.transition_df*.

__init__(*self: pybnesian.DynamicScore*) → None

Initializes a *DynamicScore*.

has_variables(*self: pybnesian.DynamicScore*, *variables: str or List[str]*) → bool

Checks whether this *DynamicScore* has the given variables.

Parameters
variables – Name or list of variables.

Returns
True if the *DynamicScore* is defined over the set of variables, False otherwise.

static_score(*self: pybnesian.DynamicScore*) → *pybnesian.Score*

It returns the static score component of the *DynamicScore*.

Returns
The static score component.

transition_score(*self: pybnesian.DynamicScore*) → *pybnesian.Score*

It returns the transition score component of the *DynamicScore*.

Returns
The transition score component.

Concrete classes

class `pybnesian.BIC`

Bases: *Score*

This class implements the Bayesian Information Criterion (BIC).

`__init__`(*self: pybnesian.BIC, df: DataFrame*) → None

Initializes a *BIC* with the given DataFrame `df`.

Parameters

df – DataFrame to compute the BIC score.

class `pybnesian.BGe`

Bases: *Score*

This class implements the Bayesian Gaussian equivalent (BGe).

`__init__`(*self: pybnesian.BGe, df: DataFrame, iss_mu: float = 1, iss_w: Optional[float] = None, nu: Optional[numpy.ndarray[numpy.float64[m, 1]]] = None*) → None

Initializes a *BGe* with the given DataFrame `df`.

Parameters

- **df** – DataFrame to compute the BGe score.
- **iss_mu** – Imaginary sample size for the normal component of the normal-Wishart prior.
- **iss_w** – Imaginary sample size for the Wishart component of the normal-Wishart prior.
- **nu** – Mean vector of the normal-Wishart prior.

class `pybnesian.BDe`

Bases: *Score*

This class implements the Bayesian Dirichlet equivalent (BDe).

`__init__`(*self: pybnesian.BDe, df: DataFrame, iss: float = 1*) → None

Initializes a *BDe* with the given DataFrame `df`.

Parameters

- **df** – DataFrame to compute the BDe score.
- **iss** – Imaginary sample size of the Dirichlet prior.

class `pybnesian.CVLikelihood`

Bases: *Score*

This class implements an estimation of the log-likelihood on unseen data using k-fold cross validation over the data.

`__init__`(*self: pybnesian.CVLikelihood, df: DataFrame, k: int = 10, seed: Optional[int] = None, construction_args: pybnesian.Arguments = Arguments*) → None

Initializes a *CVLikelihood* with the given DataFrame `df`. It uses a *CrossValidation* with `k` folds and the given `seed`.

Parameters

- **df** – DataFrame to compute the score.
- **k** – Number of folds of the cross validation.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.

- **construction_args** – Additional arguments provided to construct the *Factor*.

property cv

The underlying *CrossValidation* object to compute the score.

class pybnesian.**HoldoutLikelihood**

Bases: *Score*

This class implements an estimation of the log-likelihood on unseen data using a holdout dataset. Thus, the parameters are estimated using training data, and the score is estimated in the holdout data.

__init__(*self: pybnesian.HoldoutLikelihood*, *df: DataFrame*, *test_ratio: float = 0.2*, *seed: Optional[int] = None*, *construction_args: pybnesian.Arguments = Arguments*) → None

Initializes a *HoldoutLikelihood* with the given DataFrame `df`. It uses a *HoldOut* with the given `test_ratio` and `seed`.

Parameters

- **df** – DataFrame to compute the score.
- **test_ratio** – Proportion of instances left for the holdout data.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **construction_args** – Additional arguments provided to construct the *Factor*.

property holdout

The underlying *HoldOut* object to compute the score.

test_data(*self: pybnesian.HoldoutLikelihood*) → DataFrame

Gets the holdout data of the *HoldOut* object.

training_data(*self: pybnesian.HoldoutLikelihood*) → DataFrame

Gets the training data of the *HoldOut* object.

class pybnesian.**ValidatedLikelihood**

Bases: *ValidatedScore*

This class mixes the functionality of *CVLikelihood* and *HoldoutLikelihood*. First, it applies a *HoldOut* split over the data. Then:

- It estimates the training score using a *CVLikelihood* over the training data.
- It estimates the validation score using the training data to estimate the parameters and calculating the log-likelihood on the holdout data.

__init__(*self: pybnesian.ValidatedLikelihood*, *df: DataFrame*, *test_ratio: float = 0.2*, *k: int = 10*, *seed: Optional[int] = None*, *construction_args: pybnesian.Arguments = Arguments*) → None

Initializes a *ValidatedLikelihood* with the given DataFrame `df`. The *HoldOut* is initialized with `test_ratio` and `seed`. The `CVLikelihood` is initialized with `k` and `seed` over the training data of the holdout *HoldOut*.

Parameters

- **df** – DataFrame to compute the score.
- **test_ratio** – Proportion of instances left for the holdout data.
- **k** – Number of folds of the cross validation.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **construction_args** – Additional arguments provided to construct the *Factor*.

property cv_lik

The underlying *CVLikelihood* to compute the training score.

property holdout_lik

The underlying *HoldoutLikelihood* to compute the validation score.

training_data(*self*: pybnesian.ValidatedLikelihood) → DataFrame

The underlying training data of the *HoldOut*.

validation_data(*self*: pybnesian.ValidatedLikelihood) → DataFrame

The underlying holdout data of the *HoldOut*.

class pybnesian.DynamicBIC

Bases: *DynamicScore*

The dynamic adaptation of the *BIC* score.

__init__(*self*: pybnesian.DynamicBIC, *ddf*: pybnesian.DynamicDataFrame) → None

Initializes a *DynamicBIC* with the given *DynamicDataFrame* ddf.

Parameters

ddf – *DynamicDataFrame* to compute the *DynamicBIC* score.

class pybnesian.DynamicBGe

Bases: *DynamicScore*

The dynamic adaptation of the *BGe* score.

__init__(*self*: pybnesian.DynamicBGe, *ddf*: pybnesian.DynamicDataFrame, *iss_mu*: *float* = 1, *iss_w*: *Optional[float] = None*, *nu*: *Optional[numpy.ndarray[numpy.float64[m, 1]]] = None*) → None

Initializes a *DynamicBGe* with the given *DynamicDataFrame* ddf.

Parameters

- **ddf** – *DynamicDataFrame* to compute the *DynamicBGe* score.
- **iss_mu** – Imaginary sample size for the normal component of the normal-Wishart prior.
- **iss_w** – Imaginary sample size for the Wishart component of the normal-Wishart prior.
- **nu** – Mean vector of the normal-Wishart prior.

class pybnesian.DynamicBDe

Bases: *DynamicScore*

The dynamic adaptation of the *BDe* score.

__init__(*self*: pybnesian.DynamicBDe, *ddf*: pybnesian.DynamicDataFrame, *iss*: *float* = 1) → None

Initializes a *DynamicBDe* with the given *DynamicDataFrame* ddf.

Parameters

- **ddf** – *DynamicDataFrame* to compute the *DynamicBDe* score.
- **iss** – Imaginary sample size of the Dirichlet prior.

class pybnesian.DynamicCVLikelihood

Bases: *DynamicScore*

The dynamic adaptation of the *CVLikelihood* score.

__init__(*self: pybnesian.DynamicCVLikelihood*, *df: pybnesian.DynamicDataFrame*, *k: int = 10*, *seed: Optional[int] = None*) → None

Initializes a `DynamicCVLikelihood` with the given `DynamicDataFrame` `df`. The `k` and `seed` parameters are passed to the static and transition components of `CVLikelihood`.

Parameters

- **df** – `DynamicDataFrame` to compute the score.
- **k** – Number of folds of the cross validation.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.

class pybnesian.**DynamicHoldoutLikelihood**

Bases: `DynamicScore`

The dynamic adaptation of the `HoldoutLikelihood` score.

__init__(*self: pybnesian.DynamicHoldoutLikelihood*, *df: pybnesian.DynamicDataFrame*, *test_ratio: float = 0.2*, *seed: Optional[int] = None*) → None

Initializes a `DynamicHoldoutLikelihood` with the given `DynamicDataFrame` `df`. The `test_ratio` and `seed` parameters are passed to the static and transition components of `HoldoutLikelihood`.

Parameters

- **df** – `DynamicDataFrame` to compute the score.
- **test_ratio** – Proportion of instances left for the holdout data.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.

class pybnesian.**DynamicValidatedLikelihood**

Bases: `DynamicScore`

The dynamic adaptation of the `ValidatedLikelihood` score.

__init__(*self: pybnesian.DynamicValidatedLikelihood*, *df: pybnesian.DynamicDataFrame*, *test_ratio: float = 0.2*, *k: int = 10*, *seed: Optional[int] = None*) → None

Initializes a `DynamicValidatedLikelihood` with the given `DynamicDataFrame` `df`. The `test_ratio`, `k` and `seed` parameters are passed to the static and transition components of `ValidatedLikelihood`.

Parameters

- **df** – `DynamicDataFrame` to compute the score.
- **test_ratio** – Proportion of instances left for the holdout data.
- **k** – Number of folds of the cross validation.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.

3.5.3 Learning Operators

This section includes learning operators that are used to make small, local changes to a given Bayesian network structure. This is used for the score-and-search learning algorithms such as `GreedyHillClimbing`, `MMHC` and `DMMHC`.

There are two type of classes in this section: operators and operator sets:

- The operators are the representation of a change in a Bayesian network structure.
- The operator sets coordinate sets of operators. They can find the best operator over the set and update the score and availability of each operator in the set.

Operators

class pybnesian.**Operator**

An operator is the representation of a change in a Bayesian network structure. Each operator has a delta score associated that measures the difference in score when the operator is applied to the Bayesian network.

__eq__(*self:* pybnesian.Operator, *other:* pybnesian.Operator) → bool

__hash__(*self:* pybnesian.Operator) → int

Returns the hash value of this operator. **Two equal operators (without taking into account the delta value) must return the same hash value.**

Returns
Hash value of `self` operator.

__init__(*self:* pybnesian.Operator, *delta: float*) → None

Initializes an *Operator* with a given `delta`.

Parameters
delta – Delta score of the operator.

__str__(*self:* pybnesian.Operator) → str

apply(*self:* pybnesian.Operator, *model:* pybnesian.BayesianNetworkBase) → None

Apply the operator to the `model`.

Parameters
model – Bayesian network model.

delta(*self:* pybnesian.Operator) → float

Gets the delta score of the operator.

Returns
Delta score of the operator.

nodes_changed(*self:* pybnesian.Operator, *model: BayesianNetworkBase or ConditionalBayesianNetworkBase*) → List[str]

Gets the list of nodes whose local score changes when the operator is applied.

Parameters
model – Bayesian network model.

Returns
List of nodes whose local score changes when the operator is applied.

opposite(*self:* pybnesian.Operator, *model: BayesianNetworkBase or ConditionalBayesianNetworkBase*) → *Operator*

Returns an operator that reverses this *Operator* given the `model`. For example:

```
>>> from pybnesian import AddArc, RemoveArc, GaussianNetwork
>>> gbn = GaussianNetwork(["a", "b"])
>>> add = AddArc("a", "b", 1)
>>> assert add.opposite(gbn) == RemoveArc("a", "b", -1)
```

Parameters
model – The model where the `self` operator would be applied.

Returns
The opposite operator of `self`.

class pybnesian.**ArcOperator**

Bases: *Operator*

This class implements an operator that perfoms a change in a single arc.

__init__(*self: pybnesian.ArcOperator*, *source: str*, *target: str*, *delta: float*) → None

Initializes an *ArcOperator* of the arc `source -> target` with delta score `delta`.

Parameters

- **source** – Name of the source node.
- **target** – Name of the target node.
- **delta** – Delta score of the operator.

source(*self: pybnesian.ArcOperator*) → str

Gets the source of the *ArcOperator*.

Returns

Name of the source node.

target(*self: pybnesian.ArcOperator*) → str

Gets the target of the *ArcOperator*.

Returns

Name of the target node.

class pybnesian.**AddArc**

Bases: *ArcOperator*

This operator adds the arc `source -> target`.

__init__(*self: pybnesian.AddArc*, *source: str*, *target: str*, *delta: float*) → None

Initializes the *AddArc* operator of the arc `source -> target` with delta score `delta`.

Parameters

- **source** – Name of the source node.
- **target** – Name of the target node.
- **delta** – Delta score of the operator.

class pybnesian.**RemoveArc**

Bases: *ArcOperator*

This operator removes the arc `source -> target`.

__init__(*self: pybnesian.RemoveArc*, *source: str*, *target: str*, *delta: float*) → None

Initializes the *RemoveArc* operator of the arc `source -> target` with delta score `delta`.

Parameters

- **source** – Name of the source node.
- **target** – Name of the target node.
- **delta** – Delta score of the operator.

class pybnesian.**FlipArc**

Bases: *ArcOperator*

This operator flips (reverses) the arc `source -> target`.

__init__(*self:* pybnesian.FlipArc, *source:* *str*, *target:* *str*, *delta:* *float*) → None

Initializes the *FlipArc* operator of the arc `source -> target` with delta score `delta`.

Parameters

- **source** – Name of the source node.
- **target** – Name of the target node.
- **delta** – Delta score of the operator.

class pybnesian.**ChangeNodeType**

Bases: *Operator*

This operator changes the *FactorType* of a node.

__init__(*self:* pybnesian.ChangeNodeType, *node:* *str*, *node_type:* pybnesian.FactorType, *delta:* *float*) → None

Initializes the *ChangeNodeType* operator to change the type of the `node` to a new `node_type`.

Parameters

- **node** – Name of the source node.
- **node_type** – The new *FactorType* of the `node`.
- **delta** – Delta score of the operator.

node(*self:* pybnesian.ChangeNodeType) → str

Gets the node of the *ChangeNodeType*.

Returns

Node of the operator.

node_type(*self:* pybnesian.ChangeNodeType) → *pybnesian.FactorType*

Gets the new *FactorType* of the *ChangeNodeType*.

Returns

New *FactorType* of the node.

Operator Sets

class pybnesian.**OperatorSet**

The *OperatorSet* coordinates a set of operators. It caches/updates the score of each operator in the set and finds the operator with the best score.

__init__(*self:* pybnesian.OperatorSet, *calculate_local_cache:* *bool* = *True*) → None

Initializes an *OperatorSet*.

If `calculate_local_cache` is True, a *LocalScoreCache* is automatically initialized when *OperatorSet.cache_scores()* is called. Also, the local score cache is automatically updated on each *OperatorSet.update_scores()* call. Therefore, the local score cache is always updated. You can always get the local score cache using *OperatorSet.local_score_cache()*. The local score values can be accessed using *LocalScoreCache.local_score()*.

If `calculate_local_cache` is False, there is no local cache.

Parameters

calculate_local_cache – If True automatically initializes and updates a *LocalScoreCache*.

cache_scores(*self: pybnesian.OperatorSet*, *model: pybnesian.BayesianNetworkBase*, *score: pybnesian.Score*) → None

Caches the delta score values of each operator in the set.

Parameters

- **model** – Bayesian network model.
- **score** – The *Score* object to cache the scores.

find_max(*self: pybnesian.OperatorSet*, *model: pybnesian.BayesianNetworkBase*) → *pybnesian.Operator*

Finds the best operator in the set to apply to the `model`. This function must not return an invalid operator:

- An operator that creates cycles.
- An operator that contradicts blacklists, whitelists or max indegree.

If no valid operator is available in the set, it returns `None`.

Parameters
model – Bayesian network model.

Returns
The best valid operator, or `None` if there is no valid operator.

find_max_tabu(*self: pybnesian.OperatorSet*, *model: pybnesian.BayesianNetworkBase*, *tabu_set: pybnesian.OperatorTabuSet*) → *pybnesian.Operator*

This method is similar to *OperatorSet.find_max()*, but it also receives a `tabu_set` of operators.

This method must not return an operator in the `tabu_set` in addition to the restrictions of *OperatorSet.find_max()*.

Parameters

- **model** – Bayesian network model.
- **tabu_set** – Tabu set of operators.

Returns
The best valid operator, or `None` if there is no valid operator.

finished(*self: pybnesian.OperatorSet*) → None

Marks the finalization of the algorithm. It clears the state of the object, so *OperatorSet.cache_scores()* can be called again.

local_score_cache(*self: pybnesian.OperatorSet*) → *pybnesian.LocalScoreCache*

Returns the current *LocalScoreCache* of this *OperatorSet*.

Returns
LocalScoreCache of this operator set.

set_arc_blacklist(*self: pybnesian.OperatorSet*, *arc_blacklist: List[Tuple[str, str]]*) → None

Sets the arc blacklist (a list of arcs that cannot be added).

Parameters
arc_blacklist – The list of blacklisted arcs.

set_arc_whitelist(*self: pybnesian.OperatorSet*, *arc_whitelist: List[Tuple[str, str]]*) → None

Sets the arc whitelist (a list of arcs that are forced).

Parameters
arc_whitelist – The list of whitelisted arcs.

set_max_indegree(*self: pybnesian.OperatorSet*, *max_indegree: int*) → None

Sets the max indegree allowed. This may change the set of valid operators.

Parameters

max_indegree – Max indegree allowed.

set_type_blacklist(*self: pybnesian.OperatorSet*, *type_blacklist: List[Tuple[str, pybnesian.FactorType]]*) → None

Sets the type blacklist (a list of `FactorType` that are not allowed).

Parameters

type_blacklist – The list of blacklisted `FactorType`.

set_type_whitelist(*self: pybnesian.OperatorSet*, *type_whitelist: List[Tuple[str, pybnesian.FactorType]]*) → None

Sets the type whitelist (a list of `FactorType` that are forced).

Parameters

type_whitelist – The list of whitelisted `FactorType`.

update_scores(*self: pybnesian.OperatorSet*, *model: pybnesian.BayesianNetworkBase*, *score: pybnesian.Score*, *changed_nodes: List[str]*) → None

Updates the delta score values of the operators in the set after applying an operator in the `model`. `changed_nodes` determines the nodes whose local score has changed after applying the operator.

Parameters

- **model** – Bayesian network model.
- **score** – The `Score` object to cache the scores.
- **changed_nodes** – The nodes whose local score has changed.

class pybnesian.**ArcOperatorSet**

Bases: `OperatorSet`

This set of operators contains all the operators related with arc changes (`AddArc`, `RemoveArc`, `FlipArc`)

__init__(*self: pybnesian.ArcOperatorSet*, *blacklist: List[Tuple[str, str]] = []*, *whitelist: List[Tuple[str, str]] = []*, *max_indegree: int = 0*) → None

Initializes an `ArcOperatorSet` with optional sets of arc blacklists/whitelists and maximum indegree.

Parameters

- **blacklist** – List of blacklisted arcs.
- **whitelist** – List of whitelisted arcs.
- **max_indegree** – Max indegree allowed.

class pybnesian.**ChangeNodeTypeSet**

Bases: `OperatorSet`

This set of operators contains all the possible operators of type `ChangeNodeType`.

__init__(*self: pybnesian.ChangeNodeTypeSet*, *type_blacklist: List[Tuple[str, pybnesian.FactorType]] = []*, *type_whitelist: List[Tuple[str, pybnesian.FactorType]] = []*) → None

Initializes a `ChangeNodeTypeSet` with blacklisted and whitelisted `FactorType`.

Parameters

- **type_blacklist** – The list of blacklisted `FactorType`.

- **type_whitelist** – The list of whitelisted *FactorType*.

class pybnesian.**OperatorPool**

Bases: *OperatorSet*

This set of operators can join a list of *OperatorSet*, so that they can act as a single *OperatorSet*.

__init__(*self: pybnesian.OperatorPool*, *opsets: List[pybnesian.OperatorSet]*) → None

Initializes an *OperatorPool* with a list of *OperatorSet*.

Parameters
opsets – List of *OperatorSet*.

Other

class pybnesian.**OperatorTabuSet**

An *OperatorTabuSet* that contains forbidden operators.

__init__(*self: pybnesian.OperatorTabuSet*) → None

Creates an empty *OperatorTabuSet*.

clear(*self: pybnesian.OperatorTabuSet*) → None

Erases all the operators from the set.

contains(*self: pybnesian.OperatorTabuSet*, *operator: pybnesian.Operator*) → bool

Checks whether this tabu set contains `operator`.

Parameters
operator – The operator to be checked.

Returns
True if the tabu set contains the `operator`, False otherwise.

empty(*self: pybnesian.OperatorTabuSet*) → bool

Checks if the set has no operators

Returns
True if the set is empty, False otherwise.

insert(*self: pybnesian.OperatorTabuSet*, *operator: pybnesian.Operator*) → None

Inserts an operator into the tabu set.

Parameters
operator – Operator to insert.

class pybnesian.**LocalScoreCache**

This class implements a cache for the local score of each node.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pybnesian.LocalScoreCache) -> None

Initializes an empty *LocalScoreCache*.

2. __init__(self: pybnesian.LocalScoreCache, model: pybnesian.BayesianNetworkBase) -> None

Initializes a *LocalScoreCache* for the given `model`.

Parameters
model – A Bayesian network model.

`cache_local_scores`(*self:* pybnesian.LocalScoreCache, *model:* pybnesian.BayesianNetworkBase, *score:* pybnesian.Score) → None

Caches the local score for all the nodes.

Parameters

- **model** – A Bayesian network model.
- **score** – A *Score* object to calculate the score.

`cache_vlocal_scores`(*self:* pybnesian.LocalScoreCache, *model:* pybnesian.BayesianNetworkBase, *score:* pybnesian.ValidatedScore) → None

Caches the validation local score for all the nodes.

Parameters

- **model** – A Bayesian network model.
- **score** – A *ValidatedScore* object to calculate the score.

`local_score`(*self:* pybnesian.LocalScoreCache, *model:* pybnesian.BayesianNetworkBase, *node:* *str*) → float

Returns the local score of the `node` in the `model`.

Parameters

- **model** – A Bayesian network model.
- **node** – A node name.

Returns

Local score of `node` in `model`.

`sum`(*self:* pybnesian.LocalScoreCache) → float

Sums the local score for all the variables.

Returns

Total score.

`update_local_score`(*self:* pybnesian.LocalScoreCache, *model:* pybnesian.BayesianNetworkBase, *score:* pybnesian.Score, *node:* *str*) → None

Updates the local score of the `node` in the `model`.

Parameters

- **model** – A Bayesian network model.
- **score** – A *Score* object to calculate the score.
- **node** – A node name.

`update_vlocal_score`(*self:* pybnesian.LocalScoreCache, *model:* pybnesian.BayesianNetworkBase, *score:* pybnesian.ValidatedScore, *node:* *str*) → None

Updates the validation local score of the `node` in the `model`.

Parameters

- **model** – A Bayesian network model.
- **score** – A *ValidatedScore* object to calculate the score.
- **node** – A node name.

3.5.4 Independence Tests

This section includes conditional tests of independence. These tests are used in many constraint-based learning algorithms such as *PC*, *MMPC*, *MMHC* and *DMMHC*.

Abstract classes

class pybnesian.**IndependenceTest**

The *IndependenceTest* is an abstract class defining an interface for a conditional test of independence.

An *IndependenceTest* is defined over a set of variables and can calculate the p-value of any conditional test on these variables.

__init__(*self:* pybnesian.IndependenceTest) → None

Initializes an *IndependenceTest*.

has_variables(*self:* pybnesian.IndependenceTest, *variables: str or List[str]*) → bool

Checks whether this *IndependenceTest* has the given `variables`.

Parameters
variables – Name or list of variables.

Returns
True if the *IndependenceTest* is defined over the set of `variables`, False otherwise.

name(*self:* pybnesian.IndependenceTest, *index: int*) → str

Gets the variable name of the index-th variable.

Parameters
index – Index of the variable.

Returns
Variable name at the `index` position.

num_variables(*self:* pybnesian.IndependenceTest) → int

Gets the number of variables of the *IndependenceTest*.

Returns
Number of variables of the *IndependenceTest*.

pvalue(**args*, ***kwargs*)

Overloaded function.

1. pvalue(self: pybnesian.IndependenceTest, x: str, y: str) -> float

Calculates the p-value of the unconditional test of independence $x \perp y$.

Parameters

- **x** – A variable name.
- **y** – A variable name.

Returns
The p-value of the unconditional test of independence $x \perp y$.

2. pvalue(self: pybnesian.IndependenceTest, x: str, y: str, z: str) -> float

Calculates the p-value of an univariate conditional test of independence $x \perp y \mid z$.

Parameters

- **x** – A variable name.
- **y** – A variable name.
- **z** – A variable name.

Returns
The p-value of an univariate conditional test of independence $x \perp y \mid z$.

3. pvalue(self: pybnesian.IndependenceTest, x: str, y: str, z: List[str]) -> float

Calculates the p-value of a multivariate conditional test of independence $x \perp y \mid \mathbf{z}$.

Parameters

- **x** – A variable name.
- **y** – A variable name.
- **z** – A list of variable names.

Returns
The p-value of a multivariate conditional test of independence $x \perp y \mid \mathbf{z}$.

variable_names(*self: pybnesian.IndependenceTest*) → List[str]

Gets the list of variable names of the *IndependenceTest*.

Returns
List of variable names of the *IndependenceTest*.

class pybnesian.**DynamicIndependenceTest**

A *DynamicIndependenceTest* adapts the static *IndependenceTest* to learn dynamic Bayesian networks. It generates a static and a transition independence test to learn the static and transition components of the dynamic Bayesian network.

The dynamic independence tests are usually implemented using a *DynamicDataFrame* with the methods *DynamicDataFrame.static_df* and *DynamicDataFrame.transition_df*.

has_variables(*self: pybnesian.DynamicScore, variables: str or List[str]*) → bool

Checks whether this *DynamicScore* has the given `variables`.

Parameters
variables – Name or list of variables.

Returns
True if the *DynamicScore* is defined over the set of `variables`, False otherwise.

markovian_order(*self: pybnesian.DynamicIndependenceTest*) → int

Gets the markovian order used in this *DynamicIndependenceTest*.

Returns
Markovian order of the *DynamicIndependenceTest*.

name(*self: pybnesian.DynamicIndependenceTest, index: int*) → str

Gets the variable name of the index-th variable.

Parameters
index – Index of the variable.

Returns
Variable name at the `index` position.

num_variables(*self:* pybnesian.DynamicIndependenceTest) → int

Gets the number of variables of the *DynamicIndependenceTest*.

Returns
Number of variables of the *DynamicIndependenceTest*.

static_tests(*self:* pybnesian.DynamicIndependenceTest) → *pybnesian.IndependenceTest*

It returns the static independence test component of the *DynamicIndependenceTest*.

Returns
The static independence test component.

transition_tests(*self:* pybnesian.DynamicIndependenceTest) → *pybnesian.IndependenceTest*

It returns the transition independence test component of the *DynamicIndependenceTest*.

Returns
The transition independence test component.

variable_names(*self:* pybnesian.DynamicIndependenceTest) → List[str]

Gets the list of variable names of the *DynamicIndependenceTest*.

Returns
List of variable names of the *DynamicIndependenceTest*.

Concrete classes

class pybnesian.**LinearCorrelation**

Bases: *IndependenceTest*

This class implements a partial linear correlation independence test. This independence is only valid for continuous data.

__init__(*self:* pybnesian.LinearCorrelation, *df: DataFrame*) → None

Initializes a *LinearCorrelation* for the continuous variables in the DataFrame `df`.

Parameters
df – DataFrame on which to calculate the independence tests.

class pybnesian.**MutualInformation**

Bases: *IndependenceTest*

This class implements a hypothesis test based on mutual information. This independence is implemented for a mix of categorical and continuous data. The estimation of the mutual information assumes that the continuous data has a Gaussian probability distribution. To compute the p-value, we use the relation between the Likelihood-ratio test and the mutual information, so it is known that the null distribution has a chi-square distribution.

The theory behind this implementation is described with more detail in the following `document`.

__init__(*self:* pybnesian.MutualInformation, *df: DataFrame*, *asymptotic_df: bool = True*) → None

Initializes a *MutualInformation* for data `df`. The degrees of freedom for the chi-square null distribution can be calculated with the with the asymptotic (if `asymptotic_df` is true) or empirical (if `asymptotic_df` is false) expressions.

Parameters

- **df** – DataFrame on which to calculate the independence tests.
- **asymptotic_df** – Whether to calculate the degrees of freedom with the asympototic or empirical expression. See the `theory document`.

mi(**args*, ***kwargs*)

Overloaded function.

1. mi(self: pybnesian.MutualInformation, x: str, y: str) -> float

Estimates the unconditional mutual information $\text{MI}(x, y)$.

Parameters

- **x** – A variable name.
- **y** – A variable name.

Returns

The unconditional mutual information $\text{MI}(x, y)$.

2. mi(self: pybnesian.MutualInformation, x: str, y: str, z: str) -> float

Estimates the univariate conditional mutual information $\text{MI}(x, y \mid z)$.

Parameters

- **x** – A variable name.
- **y** – A variable name.
- **z** – A variable name.

Returns

The univariate conditional mutual information $\text{MI}(x, y \mid z)$.

3. mi(self: pybnesian.MutualInformation, x: str, y: str, z: List[str]) -> float

Estimates the multivariate conditional mutual information $\text{MI}(x, y \mid \mathbf{z})$.

Parameters

- **x** – A variable name.
- **y** – A variable name.
- **z** – A list of variable names.

Returns

The multivariate conditional mutual information $\text{MI}(x, y \mid \mathbf{z})$.

class pybnesian.**KMutualInformation**

Bases: *IndependenceTest*

This class implements a non-parametric independence test that is based on the estimation of the mutual information using k-nearest neighbors. This independence is only implemented for continuous data.

This independence test is based on [CMIknn].

__init__(*self: pybnesian.KMutualInformation*, *df: DataFrame*, *k: int*, *seed: Optional[int] = None*, *shuffle_neighbors: int = 5*, *samples: int = 1000*) → None

Initializes a *KMutualInformation* for data `df`. `k` is the number of neighbors in the k-nn model used to estimate the mutual information.

This is a permutation independence test, so `samples` defines the number of permutations. `shuffle neighbors` (k_{perm} in the original paper [CMIknn]) defines how many neighbors are used to perform the conditional permutations.

Parameters

- **df** – DataFrame on which to calculate the independence tests.
- **k** – number of neighbors in the k-nn model used to estimate the mutual information.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **shuffle_neighbors** – Number of neighbors used to perform the conditional permutation.
- **samples** – Number of permutations for the *KMutualInformation*.

mi(**args*, ***kwargs*)

Overloaded function.

1. mi(self: pybnesian.KMutualInformation, x: str, y: str) -> float

Estimates the unconditional mutual information $\text{MI}(x, y)$.

Parameters

- **x** – A variable name.
- **y** – A variable name.

Returns

The unconditional mutual information $\text{MI}(x, y)$.

2. mi(self: pybnesian.KMutualInformation, x: str, y: str, z: str) -> float

Estimates the univariate conditional mutual information $\text{MI}(x, y \mid z)$.

Parameters

- **x** – A variable name.
- **y** – A variable name.
- **z** – A variable name.

Returns

The univariate conditional mutual information $\text{MI}(x, y \mid z)$.

3. mi(self: pybnesian.KMutualInformation, x: str, y: str, z: List[str]) -> float

Estimates the multivariate conditional mutual information $\text{MI}(x, y \mid \mathbf{z})$.

Parameters

- **x** – A variable name.
- **y** – A variable name.
- **z** – A list of variable names.

Returns

The multivariate conditional mutual information $\text{MI}(x, y \mid \mathbf{z})$.

class pybnesian.**RCoT**

Bases: *IndependenceTest*

This class implements a non-parametric independence test called Randomized Conditional Correlation Test (RCoT). This method is described in [RCoT]. This independence is only implemented for continuous data.

This method uses random fourier features and is designed to be a fast non-parametric independence test.

`__init__`(*self: pybnesian.RCoT*, *df: DataFrame*, *random_fourier_xy: int = 5*, *random_fourier_z: int = 100*) → None

Initializes a *RCoT* for data `df`. The number of random fourier features used for the `x` and `y` variables in *IndependenceTest.pvalue* is `random_fourier_xy`. The number of random features used for `z` is equal to `random_fourier_z`.

Parameters

- **df** – DataFrame on which to calculate the independence tests.
- **random_fourier_xy** – Number of random fourier features for the variables of the independence test.
- **randoum_fourier_z** – Number of random fourier features for the conditioning variables of the independence test.

`class pybnesian.ChiSquare`

Bases: *IndependenceTest*

Initializes a *ChiSquare* for data `df`. This independence test is only valid for categorical data.

It implements the Pearson's X^2 test.

Parameters

df – DataFrame on which to calculate the independence tests.

`__init__`(*self: pybnesian.ChiSquare*, *df: DataFrame*) → None

`class pybnesian.DynamicLinearCorrelation`

Bases: *DynamicIndependenceTest*

The dynamic adaptation of the *LinearCorrelation* independence test.

`__init__`(*self: pybnesian.DynamicLinearCorrelation*, *ddf: pybnesian.DynamicDataFrame*) → None

Initializes a *DynamicLinearCorrelation* with the given *DynamicDataFrame* `ddf`.

Parameters

ddf – *DynamicDataFrame* to create the *DynamicLinearCorrelation*.

`class pybnesian.DynamicMutualInformation`

Bases: *DynamicIndependenceTest*

The dynamic adaptation of the *MutualInformation* independence test.

`__init__`(*self: pybnesian.DynamicMutualInformation*, *ddf: pybnesian.DynamicDataFrame*, *asymptotic_df: bool = True*) → None

Initializes a *DynamicMutualInformation* with the given *DynamicDataFrame* `df`. The `asymptotic_df` parameter is passed to the static and transition components of *MutualInformation*.

Parameters

- **ddf** – *DynamicDataFrame* to create the *DynamicMutualInformation*.
- **asymptotic_df** – Whether to calculate the asymptotic or empirical degrees of freedom of the chi-square null distribution.

`class pybnesian.DynamicKMutualInformation`

Bases: *DynamicIndependenceTest*

The dynamic adaptation of the *KMutualInformation* independence test.

__init__(*self: pybnesian.DynamicKMutualInformation*, *ddf: pybnesian.DynamicDataFrame*, *k: int*, *seed: Optional[int] = None*, *shuffle_neighbors: int = 5*, *samples: int = 1000*) → None

Initializes a `DynamicKMutualInformation` with the given `DynamicDataFrame` `df`. The `k`, `seed`, `shuffle_neighbors` and `samples` parameters are passed to the static and transition components of `KMutualInformation`.

Parameters

- **ddf** – `DynamicDataFrame` to create the `DynamicKMutualInformation`.
- **k** – number of neighbors in the k-nn model used to estimate the mutual information.
- **seed** – A random seed number. If not specified or `None`, a random seed is generated.
- **shuffle_neighbors** – Number of neighbors used to perform the conditional permutation.
- **samples** – Number of permutations for the `KMutualInformation`.

class pybnesian.**DynamicRCoT**

Bases: `DynamicIndependenceTest`

The dynamic adaptation of the `RCoT` independence test.

__init__(*self: pybnesian.DynamicRCoT*, *ddf: pybnesian.DynamicDataFrame*, *random_fourier_xy: int = 5*, *random_fourier_z: int = 100*) → None

Initializes a `DynamicRCoT` with the given `DynamicDataFrame` `df`. The `random_fourier_xy` and `random_fourier_z` parameters are passed to the static and transition components of `RCoT`.

Parameters

- **ddf** – `DynamicDataFrame` to create the `DynamicRCoT`.
- **random_fourier_xy** – Number of random fourier features for the variables of the independence test.
- **randoum_fourier_z** – Number of random fourier features for the conditioning variables of the independence test.

class pybnesian.**DynamicChiSquare**

Bases: `DynamicIndependenceTest`

The dynamic adaptation of the `ChiSquare` independence test.

__init__(*self: pybnesian.DynamicChiSquare*, *ddf: pybnesian.DynamicDataFrame*) → None

Initializes a `DynamicChiSquare` with the given `DynamicDataFrame` `df`.

Parameters

ddf – `DynamicDataFrame` to create the `DynamicChiSquare`.

Bibliography

3.5.5 Learning Algorithms

pybnesian.**hc**(*df: DataFrame*, *bn_type: pybnesian.BayesianNetworkType = None*, *start: pybnesian.BayesianNetworkBase = None*, *score: Optional[str] = None*, *operators: Optional[List[str]] = None*, *arc_blacklist: List[Tuple[str, str]] = []*, *arc_whitelist: List[Tuple[str, str]] = []*, *type_blacklist: List[Tuple[str, pybnesian.FactorType]] = []*, *type_whitelist: List[Tuple[str, pybnesian.FactorType]] = []*, *callback: pybnesian.Callback = None*, *max_indegree: int = 0*, *max_iters: int = 2147483647*, *epsilon: float = 0*, *patience: int = 0*, *seed: Optional[int] = None*, *num_folds: int = 10*, *test_holdout_ratio: float = 0.2*, *verbose: int = 0*) → *pybnesian.BayesianNetworkBase*

Executes a greedy hill-climbing algorithm. This calls *GreedyHillClimbing.estimate()*.

Parameters

- **df** – DataFrame used to learn a Bayesian network model.
- **bn_type** – *BayesianNetworkType* of the returned model. If start is given, bn_type is ignored.
- **start** – Initial structure of the *GreedyHillClimbing*. If None, a new Bayesian network model is created.
- **score** – A string representing the score used to drive the search. The possible options are: "bic" for *BIC*, "bge" for *BGe*, "cv-lik" for *CVLikelihood*, "holdout-lik" for *HoldoutLikelihood*, "validated-lik for *ValidatedLikelihood*.
- **operators** – Set of operators in the search process.
- **arc_blacklist** – List of arcs blacklist (forbidden arcs).
- **arc_whitelist** – List of arcs whitelist (forced arcs).
- **type_blacklist** – List of type blacklist (forbidden *FactorType*).
- **type_whitelist** – List of type whitelist (forced *FactorType*).
- **callback** – Callback object that is called after each iteration.
- **max_indegree** – Maximum indegree allowed in the graph.
- **max_iters** – Maximum number of search iterations
- **epsilon** – Minimum delta score allowed for each operator. If the new operator is less than epsilon, the search process is stopped.
- **patience** – The patience parameter (only used with *ValidatedScore*). See *patience*.
- **seed** – Seed parameter of the score (if neeeded).
- **num_folds** – Number of folds for the *CVLikelihood* and *ValidatedLikelihood* scores.
- **test_holdout_ratio** – Parameter for the *HoldoutLikelihood* and *ValidatedLikelihood* scores.
- **verbose** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

The estimated Bayesian network structure.

This classes implement many different learning structure algorithms.

class pybnesian.**GreedyHillClimbing**

This class implements a greedy hill-climbing algorithm. It finds the best structure applying small local changes iteratively. The best operator is found using a delta score.

Patience parameter:

When the score is a *ValidatedScore*, a tabu set is used to improve the exploration during the search process if the score does not improve. This is because it is allowed to continue the search process even if the training delta score of the *ValidatedScore* is negative. The existence of the validation delta score in the *ValidatedScore* can help to control the uncertainty of the training score (the training delta score can be negative because it is a bad operator or because there is uncertainty in the data). Thus, only if both the training and validation delta scores are negative for patience iterations, the search is stopped and the best found model is returned.

`__init__`(*self: pybnesian.GreedyHillClimbing*) → None

Initializes a *`GreedyHillClimbing`*.

`estimate`(*self: pybnesian.GreedyHillClimbing, operators: pybnesian.OperatorSet, score: pybnesian.Score, start: BayesianNetworkBase or ConditionalBayesianNetworkBase, arc_blacklist: List[Tuple[str, str]] = [], arc_whitelist: List[Tuple[str, str]] = [], type_blacklist: List[Tuple[str, pybnesian.FactorType]] = [], type_whitelist: List[Tuple[str, pybnesian.FactorType]] = [], callback: pybnesian.Callback = None, max_indegree: int = 0, max_iters: int = 2147483647, epsilon: float = 0, patience: int = 0, verbose: int = 0*) → type[start]

Estimates the structure of a Bayesian network. The estimated Bayesian network is of the same type as `start`. The set of operators allowed in the search is `operators`. The delta score of each operator is evaluated using the `score`. The initial structure of the algorithm is the model `start`.

There are many optional parameters that restricts to the learning process.

Parameters

- **`operators`** – Set of operators in the search process.
- **`score`** – *`Score`* that drives the search.
- **`start`** – Initial structure. A *`BayesianNetworkBase`* or *`ConditionalBayesianNetworkBase`*
- **`arc_blacklist`** – List of arcs blacklist (forbidden arcs).
- **`arc_whitelist`** – List of arcs whitelist (forced arcs)
- **`type_blacklist`** – List of type blacklist (forbidden *`FactorType`*).
- **`type_whitelist`** – List of type whitelist (forced *`FactorType`*).
- **`callback`** – Callback object that is called after each iteration.
- **`max_indegree`** – Maximum indegree allowed in the graph.
- **`max_iters`** – Maximum number of search iterations
- **`epsilon`** – Minimum delta score allowed for each operator. If the new operator is less than epsilon, the search process is stopped.
- **`patience`** – The patience parameter (only used with *`ValidatedScore`*). See *patience*.
- **`verbose`** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

The estimated Bayesian network structure of the same type as `start`.

class `pybnesian.PC`

This class implements the PC learning algorithm. The PC algorithm finds the best partially directed graph that expresses the conditional independences in the data.

It implements the PC-stable version of [pc-stable]. This implementation is parametrized to execute the conservative PC (CPC) or the majority PC (MPC) variant.

This class can return an unconditional partially directed graph (using *`PC.estimate()`*) and a conditional partially directed graph (using *`PC.estimate_conditional()`*).

`__init__`(*self: pybnesian.PC*) → None

Initializes a *`PC`*.

estimate(*self: pybnesian.PC*, *hypot_test: pybnesian.IndependenceTest*, *nodes: List[str] = []*, *arc_blacklist: List[Tuple[str, str]] = []*, *arc_whitelist: List[Tuple[str, str]] = []*, *edge_blacklist: List[Tuple[str, str]] = []*, *edge_whitelist: List[Tuple[str, str]] = []*, *alpha: float = 0.05*, *use_sepsets: bool = False*, *ambiguous_threshold: float = 0.5*, *allow_bidirected: bool = True*, *verbose: int = 0*) → *pybnesian.PartiallyDirectedGraph*

Estimates the skeleton (the partially directed graph) using the PC algorithm.

Parameters

- **hypot_test** – The `IndependenceTest` object used to execute the conditional independence tests.
- **nodes** – The list of nodes of the returned skeleton. If empty (the default value), the node names are extracted from `IndependenceTest.variable_names()`.
- **arc_blacklist** – List of arcs blacklist (forbidden arcs).
- **arc_whitelist** – List of arcs whitelist (forced arcs).
- **edge_blacklist** – List of edge blacklist (forbidden edges). This also implicitly applies a double arc blacklist.
- **edge_whitelist** – List of edge whitelist (forced edges).
- **alpha** – The type I error of each independence test.
- **use_sepsets** – If True, it detects the v-structures using the cached sepsets in Algorithm 4.1 of [pc-stable]. Otherwise, it searches among all the possible sepsets (as in CPC and MPC).
- **ambiguous_threshold** – If `use_sepsets` is `False`, the `ambiguous_threshold` sets the threshold on the ratio of sepsets needed to declare a v-structure. If `ambiguous_threshold = 0`, it is equivalent to CPC (the v-structure is detected if no sepset contains the v-node). If `ambiguous_threshold = 0.5`, it is equivalent to MPC (the v-structure is detected if less than half of the sepsets contain the v-node).
- **allow_bidirected** – If True, it allows bi-directed arcs. This ensures that the result of the algorithm is order-independent while applying v-structures (as in LCPC and LMPC in [pc-stable]). Otherwise, it does not return bi-directed arcs.
- **verbose** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

A `PartiallyDirectedGraph` trained by PC that represents the conditional independences in `hypot_test`.

estimate_conditional(*self: pybnesian.PC*, *hypot_test: pybnesian.IndependenceTest*, *nodes: List[str]*, *interface_nodes: List[str] = []*, *arc_blacklist: List[Tuple[str, str]] = []*, *arc_whitelist: List[Tuple[str, str]] = []*, *edge_blacklist: List[Tuple[str, str]] = []*, *edge_whitelist: List[Tuple[str, str]] = []*, *alpha: float = 0.05*, *use_sepsets: bool = False*, *ambiguous_threshold: float = 0.5*, *allow_bidirected: bool = True*, *verbose: int = 0*) → *pybnesian.ConditionalPartiallyDirectedGraph*

Estimates the conditional skeleton (the conditional partially directed graph) using the PC algorithm.

Parameters

- **hypot_test** – The `IndependenceTest` object used to execute the conditional independence tests.
- **nodes** – The list of nodes of the returned skeleton.
- **interface_nodes** – The list of interface nodes of the returned skeleton.

- **arc_blacklist** – List of arcs blacklist (forbidden arcs).
- **arc_whitelist** – List of arcs whitelist (forced arcs).
- **edge_blacklist** – List of edge blacklist (forbidden edges). This also implicitly applies a double arc blacklist.
- **edge_whitelist** – List of edge whitelist (forced edges).
- **alpha** – The type I error of each independence test.
- **use_sepsets** – If True, it detects the v-structures using the cached sepsets in Algorithm 4.1 of [pc-stable]. Otherwise, it searches among all the possible sepsets (as in CPC and MPC).
- **ambiguous_threshold** – If `use_sepsets` is `False`, the `ambiguous_threshold` sets the threshold on the ratio of sepsets needed to declare a v-structure. If `ambiguous_threshold = 0`, it is equivalent to CPC (the v-structure is detected if no sepset contains the v-node). If `ambiguous_threshold = 0.5`, it is equivalent to MPC (the v-structure is detected if less than half of the sepsets contain the v-node).
- **allow_bidirected** – If True, it allows bi-directed arcs. This ensures that the result of the algorithm is order-independent while applying v-structures (as in LCPC and LMPC in [pc-stable]). Otherwise, it does not return bi-directed arcs.
- **verbose** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

A `ConditionalPartiallyDirectedGraph` trained by PC that represents the conditional independences in `hypot_test`.

class pybnesian.**MMPC**

This class implements Max-Min Parent Children (MMPC) [mmhc]. The MMPC algorithm finds the sets of parents and children of each node using a measure of association. With this estimate, it constructs a skeleton (an undirected graph). Then, this algorithm searches for v-structures as in `PC`. The final product of this algorithm is a partially directed graph.

This implementation uses the p-value as a measure of association. A lower p-value is a higher association value and viceversa.

__init__(*self: pybnesian.MMPC*) → None

Initializes a `MMPC`.

estimate(*self: pybnesian.MMPC, hypot_test: pybnesian.IndependenceTest, nodes: List[str] = [], arc_blacklist: List[Tuple[str, str]] = [], arc_whitelist: List[Tuple[str, str]] = [], edge_blacklist: List[Tuple[str, str]] = [], edge_whitelist: List[Tuple[str, str]] = [], alpha: float = 0.05, ambiguous_threshold: float = 0.5, allow_bidirected: bool = True, verbose: int = 0*) → *pybnesian.PartiallyDirectedGraph*

Estimates the skeleton (the partially directed graph) using the MMPC algorithm.

Parameters

- **hypot_test** – The `IndependenceTest` object used to execute the conditional independence tests.
- **nodes** – The list of nodes of the returned skeleton. If empty (the default value), the node names are extracted from `IndependenceTest.variable_names()`.
- **arc_blacklist** – List of arcs blacklist (forbidden arcs).
- **arc_whitelist** – List of arcs whitelist (forced arcs).

- **edge_blacklist** – List of edge blacklist (forbidden edges). This also implicitly applies a double arc blacklist.
- **edge_whitelist** – List of edge whitelist (forced edges).
- **alpha** – The type I error of each independence test.
- **ambiguous_threshold** – The `ambiguous_threshold` sets the threshold on the ratio of sepsets needed to declare a v-structure. This is equal to `ambiguous_threshold` in `PC.estimate()`.
- **allow_bidirected** – If True, it allows bi-directed arcs. This ensures that the result of the algorithm is order-independent while applying v-structures (as in LCPC and LMPC in [pc-stable]). Otherwise, it does not return bi-directed arcs.
- **verbose** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

A `PartiallyDirectedGraph` trained by MMPC.

estimate_conditional(*self: pybnesian.MMPC, hypot_test: pybnesian.IndependenceTest, nodes: List[str], interface_nodes: List[str] = [], arc_blacklist: List[Tuple[str, str]] = [], arc_whitelist: List[Tuple[str, str]] = [], edge_blacklist: List[Tuple[str, str]] = [], edge_whitelist: List[Tuple[str, str]] = [], alpha: float = 0.05, ambiguous_threshold: float = 0.5, allow_bidirected: bool = True, verbose: int = 0*) → *pybnesian.ConditionalPartiallyDirectedGraph*

Estimates the conditional skeleton (the conditional partially directed graph) using the MMPC algorithm.

Parameters

- **hypot_test** – The `IndependenceTest` object used to execute the conditional independence tests.
- **nodes** – The list of nodes of the returned skeleton.
- **interface_nodes** – The list of interface nodes of the returned skeleton.
- **arc_blacklist** – List of arcs blacklist (forbidden arcs).
- **arc_whitelist** – List of arcs whitelist (forced arcs).
- **edge_blacklist** – List of edge blacklist (forbidden edges). This also implicitly applies a double arc blacklist.
- **edge_whitelist** – List of edge whitelist (forced edges).
- **alpha** – The type I error of each independence test.
- **ambiguous_threshold** – The `ambiguous_threshold` sets the threshold on the ratio of sepsets needed to declare a v-structure. This is equal to `ambiguous_threshold` in `PC.estimate_conditional()`.
- **allow_bidirected** – If True, it allows bi-directed arcs. This ensures that the result of the algorithm is order-independent while applying v-structures (as in LCPC and LMPC in [pc-stable]). Otherwise, it does not return bi-directed arcs.
- **verbose** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

A `PartiallyDirectedGraph` trained by MMPC.

class pybnesian.**MMHC**

This class implements Max-Min Hill-Climbing (MMHC) [mmhc]. The MMHC algorithm finds the sets of possible arcs using the *MMPC* algorithm. Then, it trains the structure using a greedy hill-climbing algorithm (*GreedyHillClimbing*) blacklisting all the possible arcs not found by MMPC.

__init__(*self: pybnesian.MMHC*) → None

estimate(*self: pybnesian.MMHC, hypot_test: pybnesian.IndependenceTest, operators: pybnesian.OperatorSet, score: pybnesian.Score, nodes: List[str] = [], bn_type: pybnesian.BayesianNetworkType = GaussianNetworkType, arc_blacklist: List[Tuple[str, str]] = [], arc_whitelist: List[Tuple[str, str]] = [], edge_blacklist: List[Tuple[str, str]] = [], edge_whitelist: List[Tuple[str, str]] = [], type_blacklist: List[Tuple[str, pybnesian.FactorType]] = [], type_whitelist: List[Tuple[str, pybnesian.FactorType]] = [], callback: pybnesian.Callback = None, max_indegree: int = 0, max_iters: int = 2147483647, epsilon: float = 0, patience: int = 0, alpha: float = 0.05, verbose: int = 0*) → *pybnesian.BayesianNetworkBase*

Estimates the structure of a Bayesian network. This implementation calls *MMPC* and *GreedyHillClimbing* with the set of parameters provided.

Parameters

- **hypot_test** – The *IndependenceTest* object used to execute the conditional independence tests (for *MMPC*).
- **operators** – Set of operators in the search process (for *GreedyHillClimbing*).
- **score** – *Score* that drives the search (for *GreedyHillClimbing*).
- **nodes** – The list of nodes of the returned skeleton. If empty (the default value), the node names are extracted from *IndependenceTest.variable_names()*.
- **bn_type** – A *BayesianNetworkType*.
- **arc_blacklist** – List of arcs blacklist (forbidden arcs).
- **arc_whitelist** – List of arcs whitelist (forced arcs).
- **edge_blacklist** – List of edge blacklist (forbidden edges). This also implicitly applies a double arc blacklist.
- **edge_whitelist** – List of edge whitelist (forced edges).
- **type_blacklist** – List of type blacklist (forbidden *FactorType*).
- **type_whitelist** – List of type whitelist (forced *FactorType*).
- **callback** – Callback object that is called after each iteration of *GreedyHillClimbing*.
- **max_indegree** – Maximum indegree allowed in the graph (for *GreedyHillClimbing*).
- **max_iters** – Maximum number of search iterations (for *GreedyHillClimbing*).
- **epsilon** – Minimum delta score allowed for each operator. If the new operator is less than epsilon, the search process is stopped (for *GreedyHillClimbing*).
- **patience** – The patience parameter (only used with *ValidatedScore*). See *patience* (for *GreedyHillClimbing*).
- **alpha** – The type I error of each independence test (for *MMPC*).
- **verbose** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

The Bayesian network structure learned by MMHC.

estimate_conditional(*self: pybnesian.MMHC*, *hypot_test: pybnesian.IndependenceTest*, *operators: pybnesian.OperatorSet*, *score: pybnesian.Score*, *nodes: List[str] = []*, *interface_nodes: List[str] = []*, *bn_type: pybnesian.BayesianNetworkType = GaussianNetworkType*, *arc_blacklist: List[Tuple[str, str]] = []*, *arc_whitelist: List[Tuple[str, str]] = []*, *edge_blacklist: List[Tuple[str, str]] = []*, *edge_whitelist: List[Tuple[str, str]] = []*, *type_blacklist: List[Tuple[str, pybnesian.FactorType]] = []*, *type_whitelist: List[Tuple[str, pybnesian.FactorType]] = []*, *callback: pybnesian.Callback = None*, *max_indegree: int = 0*, *max_iters: int = 2147483647*, *epsilon: float = 0*, *patience: int = 0*, *alpha: float = 0.05*, *verbose: int = 0*) → *pybnesian.ConditionalBayesianNetworkBase*

Estimates the structure of a conditional Bayesian network. This implementation calls `MMPC` and `GreedyHillClimbing` with the set of parameters provided.

Parameters

- **hypot_test** – The `IndependenceTest` object used to execute the conditional independence tests (for `MMPC`).
- **operators** – Set of operators in the search process (for `GreedyHillClimbing`).
- **score** – `Score` that drives the search (for `GreedyHillClimbing`).
- **nodes** – The list of nodes of the returned skeleton.
- **interface_nodes** – The list of interface nodes of the returned skeleton.
- **bn_type** – A `BayesianNetworkType`.
- **arc_blacklist** – List of arcs blacklist (forbidden arcs).
- **arc_whitelist** – List of arcs whitelist (forced arcs).
- **edge_blacklist** – List of edge blacklist (forbidden edges). This also implicitly applies a double arc blacklist.
- **edge_whitelist** – List of edge whitelist (forced edges).
- **type_blacklist** – List of type blacklist (forbidden `FactorType`).
- **type_whitelist** – List of type whitelist (forced `FactorType`).
- **callback** – Callback object that is called after each iteration of `GreedyHillClimbing`.
- **max_indegree** – Maximum indegree allowed in the graph (for `GreedyHillClimbing`).
- **max_iters** – Maximum number of search iterations (for `GreedyHillClimbing`).
- **epsilon** – Minimum delta score allowed for each operator. If the new operator is less than epsilon, the search process is stopped (for `GreedyHillClimbing`).
- **patience** – The patience parameter (only used with `ValidatedScore`). See *patience* (for `GreedyHillClimbing`).
- **alpha** – The type I error of each independence test (for `MMPC`).
- **verbose** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

The conditional Bayesian network structure learned by MMHC.

class pybnesian.DMMHC

This class implements the Dynamic Max-Min Hill-Climbing (DMMHC) [dmmhc]. This algorithm uses the `MMHC` to train the static and transition components of the dynamic Bayesian network.

`__init__`(*self: pybnesian.DMMHC*) → None

`estimate`(*self: pybnesian.DMMHC, hypot_test: pybnesian.DynamicIndependenceTest, operators: pybnesian.OperatorSet, score: pybnesian.DynamicScore, variables: List[str] = [], bn_type: pybnesian.BayesianNetworkType = GaussianNetworkType, markovian_order: int = 1, static_callback: pybnesian.Callback = None, transition_callback: pybnesian.Callback = None, max_indegree: int = 0, max_iters: int = 2147483647, epsilon: float = 0, patience: int = 0, alpha: float = 0.05, verbose: int = 0*) → *pybnesian.DynamicBayesianNetworkBase*

Estimates a dynamic Bayesian network. This implementation uses `MMHC` to estimate both the static and transition Bayesian networks. This set of parameters are provided to the functions `MMHC.estimate()` and `MMHC.estimate_conditional()`.

Parameters

- **`hypot_test`** – The `DynamicIndependenceTest` object used to execute the conditional independence tests (for `MMPC`).
- **`operators`** – Set of operators in the search process (for `GreedyHillClimbing`).
- **`score`** – `DynamicScore` that drives the search (for `GreedyHillClimbing`).
- **`variables`** – The list of variables of the dynamic Bayesian network. If empty (the default value), the variable names are extracted from `DynamicIndependenceTest.variable_names()`.
- **`bn_type`** – A `BayesianNetworkType`.
- **`markovian_order`** – The markovian order of the dynamic Bayesian network.
- **`static_callback`** – Callback object that is called after each iteration of `GreedyHillClimbing` to learn the static component of the dynamic Bayesian network.
- **`transition_callback`** – Callback object that is called after each iteration of `GreedyHillClimbing` to learn the transition component of the dynamic Bayesian network.
- **`max_indegree`** – Maximum indegree allowed in the graph (for `GreedyHillClimbing`).
- **`max_iters`** – Maximum number of search iterations (for `GreedyHillClimbing`).
- **`epsilon`** – Minimum delta score allowed for each operator. If the new operator is less than epsilon, the search process is stopped (for `GreedyHillClimbing`).
- **`patience`** – The patience parameter (only used with `ValidatedScore`). See *patience* (for `GreedyHillClimbing`).
- **`alpha`** – The type I error of each independence test (for `MMPC`).
- **`verbose`** – If True the progress will be displayed, otherwise nothing will be displayed.

Returns

The dynamic Bayesian network structure learned by DMMHC.

Learning Algorithms Components

class pybnesian.**MeekRules**

This class implements the Meek rules [meek]. These rules direct some edges in a partially directed graph to create an equivalence class of Bayesian networks.

static rule1(*graph: pybnesian.PartiallyDirectedGraph or pybnesian.ConditionalPartiallyDirectedGraph*) → bool

Applies the rule 1 to `graph`.

Parameters
graph – Graph to apply the rule 1.

Returns
True if the rule changed the graph, False otherwise.

static rule2(*graph: pybnesian.PartiallyDirectedGraph or pybnesian.ConditionalPartiallyDirectedGraph*) → bool

Applies the rule 2 to `graph`.

Parameters
graph – Graph to apply the rule 2.

Returns
True if the rule changed the graph, False otherwise.

static rule3(*graph: pybnesian.PartiallyDirectedGraph or pybnesian.ConditionalPartiallyDirectedGraph*) → bool

Applies the rule 3 to `graph`.

Parameters
graph – Graph to apply the rule 3.

Returns
True if the rule changed the graph, False otherwise.

Learning Callbacks

class pybnesian.**Callback**

A *Callback* object is called after each iteration of a *GreedyHillClimbing*.

__init__(*self:* pybnesian.Callback) → None

Initializes a *Callback*.

call(*self:* pybnesian.Callback, *model:* pybnesian.BayesianNetworkBase, *operator:* pybnesian.Operator, *score:* pybnesian.Score, *iteration: int*) → None

This method is called after each iteration of *GreedyHillClimbing*.

Parameters

- **model** – The model in the current `iteration` of the *GreedyHillClimbing*.
- **operator** – The last operator applied to the `model`. It is `None` at the start and at the end of the algorithm.
- **score** – The score used in the *GreedyHillClimbing*.
- **iteration** – Iteration number of the *GreedyHillClimbing*. It is 0 at the start.

class pybnesian.**SaveModel**

Bases: *Callback*

Saves the model on each iteration of *GreedyHillClimbing* using *BayesianNetworkBase.save()*. Each model is named after the iteration number.

__init__(*self:* pybnesian.SaveModel, *folder_name: str*) → None

Initializes a *SaveModel*. It saves all the models in the folder `folder_name`.

Parameters

folder_name – Name of the folder where the models will be saved.

Bibliography

3.6 Serialization

All the relevant objects (graphs, factors, Bayesian networks, etc) can be saved/loaded using the pickle format.

These objects can be saved using directly `pickle.dump` and `pickle.load`. For example:

```
>>> import pickle
>>> from pybnesian import Dag
>>> g = Dag(["a", "b", "c", "d"], [("a", "b")])
>>> with open("saved_graph.pickle", "wb") as f:
...     pickle.dump(g, f)
>>> with open("saved_graph.pickle", "rb") as f:
...     lg = pickle.load(f)
>>> assert lg.nodes() == ["a", "b", "c", "d"]
>>> assert lg.arcs() == [("a", "b")]
```

We can reduce some boilerplate code using the `save` methods: *Factor.save()*, *UndirectedGraph.save()*, *DirectedGraph.save()*, *BayesianNetworkBase.save()*, etc… Also, the *load* can load any saved object:

```
>>> import pickle
>>> from pybnesian import load, Dag
>>> g = Dag(["a", "b", "c", "d"], [("a", "b")])
>>> g.save("saved_graph")
>>> lg = load("saved_graph.pickle")
>>> assert lg.nodes() == ["a", "b", "c", "d"]
>>> assert lg.arcs() == [("a", "b")]
```

pybnesian.**load**(*filename: str*) → object

Load the saved object (a *Factor*, a graph, a *BayesianNetworkBase*, etc…) in `filename`.

Parameters

filename – File name.

Returns

The object saved in the file.

CHAPTER FOUR

CHANGELOG

4.1 v0.4.3

- Fixed a bug in `DiscreteFactor` and others hybrid factors, such as `CLinearGaussianCPD` and `HCKDE`, where categorical data would not be correctly validated. This could lead to erroneous results or undefined behavior (often leading to segmentation fault). Thanks to Carlos Li for reporting this bug.
- Support for Python 3.10 and `pyarrow>=9.0` has been added. Support for Python 3.6 has been deprecated, as `pyarrow` no longer supports it.
- manylinux2014 wheels are now used instead of manylinux2010, since `pyarrow` no longer provides manylinux2010 wheels.

4.2 v0.4.2

- Fixed important bug in OpenCL for NVIDIA GPUs, as they define small OpenCL constant memory. See https://stackoverflow.com/questions/63080816/opencl-small-constant-memory-size-on-nvidia-gpu.

4.3 v0.4.1

- Added support for Apache Arrow 7.0.0.

4.4 v0.4.0

- Added method `ConditionalBayesianNetworkBase.interface_arcs`.
- `GreedyHillClimbing` and `MMHC` now accepts a blacklist of `FactorType`.
- `BayesianNetworkType.data_default_node_type` now returns a list of `FactorType` indicating the priority of each `FactorType` for each data type.
- `BayesianNetworkBase.set_unknown_node_types` now accepts an argument of `FactorType` blacklist.
- Change `HeterogeneousBN` constructor and `HeterogeneousBNType.default_node_types` to accept lists of default `FactorType`.
- Adds constructors for `HeterogeneousBN` and `CLGNetwork` that can set the `FactorType` for each node.
- Bug Fixes:
  - An overflow error in `ChiSquare` hypothesis test was raised when the statistic were close to 0.

- Arc blacklists/whitelists with repeated arcs were not correctly processed.
- Fixed an error in the use of the patience parameter. Previously, the algorithm was executed as with a `patience - 1` value.
- Improve the validation of objects returned from Python class extensions, so it errors when the extensions are not correctly implemented.
- Fixed many serialization bugs. In particular, there were multiple bugs related with the serialization of models with Python extensions.
- Included a fix for the Windows build (by setting a correct `__cplusplus` value).
- Fixed a bug in `LinearGaussianCPD.fit` with 2 parents. In some cases, it was detecting a linear dependence between the parents that did not exist.
- Fixes a bug which causes that the Python-class extension functionality is removed. Related to: https://github.com/pybind/pybind11/issues/1333.

4.5 v0.3.4

- Improvements on the code that checks that a matrix positive definite.
- A bug affecting the learning of conditional Bayesian networks with `MMHC` has been fixed. This bug also affected `DMMHC`.
- Fixed a bug that affected the type of the parameter `bn_type` of `MMHC.estimate`, `MMHC.estimate_conditional` and `DMMHC.estimate`.

4.6 v0.3.3

- Adds support for pyarrow 5.0.0 in the PyPi wheels.
- Added `Arguments.args` to access the `args` and `kwargs` for a node.
- Added `BayesianNetworkBase.underlying_node_type` to get the underlying node type of a node given some data.
- Improves the fitting of hybrid factors. Now, an specific discrete configuration can be left unfitted if the base continuous factor raises `SingularCovarianceData`.
- Improves the `LinearGaussianCPD` fit when the covariance matrix of the data is singular.
- Improves the `NormalReferenceRule`, `ScottsBandwidth`, and `UCV` estimation when the covariance of the data is singular.
- Fixes a bug loading an heterogeneous Bayesian network from a file.
- Introduces a check that a needed category exists in discrete data.
- `Assignment` now supports integer numbers converting them automatically to float.
- Fix a bug in `GreedyHillClimbing` that caused the return of Bayesian networks with `UnknownFactorType`.
- Reduces memory usage when fitting and printing an hybrid `Factor`.
- Fixes a precision bug in `GreedyHillClimbing`.
- Improves `CrossValidation` parameter checking.

4.7 v0.3.2

- Fixed a bug in the *UCV* bandwidth selector that may cause segmentation fault.
- Added some checks to ensure that the categorical data is of type string.
- Fixed the *GreedyHillClimbing* iteration counter, which was begin increased twice per iteration.
- Added a default parameter value for `include_cpd` in *BayesianNetworkBase.save* and *DynamicBayesianNetworkBase.save*.
- Added more checks to detect ill-conditioned regression problems. The *BIC* score returns `-infinity` for ill-conditioned regression problems.

4.8 v0.3.1

- Fixed the build process to support CMake versions older than 3.13.
- Fixed a bug that might raise an error with a call to *FactorType.new_factor* with **args* and ***kwargs* arguments . This bug was only reproducible if the library was compiled with gcc.
- Added CMake as prerequisite to compile the library in the docs.

4.9 v0.3.0

- Removed all the submodules to simplify the imports. Now, all the classes are accessible directly from the pybnesian root module.
- Added a *ProductKDE* class that implements *KDE* with diagonal bandwidth matrix.
- Added an abstract class *BandwidthSelector* to implement bandwidth selection for *KDE* and *ProductKDE*. Three concrete implementations of bandwidth selection are included: *ScottsBandwidth*, *NormalReferenceRule* and *UCV*.
- Added *Arguments*, *Args* and *Kwargs* to store a set of arguments to be used to create new factors through *FactorType.new_factor*. The *Arguments* are accepted by *BayesianNetworkBase.fit* and the constructors of *CVLikelihood*, *HoldoutLikelihood* and *ValidatedLikelihood*.

4.10 v0.2.1

- An error related to the processing of categorical data with too many categories has been corrected.
- Removed `-march=native` flag in the build script to avoid the use of instruction sets not available on some CPUs.

4.11 v0.2.0

- Added conditional linear Gaussian networks (`CLGNetworkType`, `CLGNetwork`, `ConditionalCLGNetwork` and `DynamicCLGNetwork`).
- Implemented `ChiSquare` (and `DynamicChiSquare`) indepencence test.
- Implemented `MutualInformation` (and `DynamicMutualInformation`) indepencence test. This independence test is valid for hybrid data.
- Implemented `BDe` (Bayesian Dirichlet equivalent) score (and `DynamicBDe`).
- Added `UnknownFactorType` as default `FactorType` for Bayesian networks when the node type could not be deduced.
- Added `Assignment` class to represent the assignment of values to variables.

API changes:

- Added method `Score.data()`.
- Added `BayesianNetworkType.data_default_node_type()` for non-homogeneous `BayesianNetworkType`.
- Added constructor for `HeterogeneousBN` to specify a default `FactorType` for each data type. Also, it adds `HeterogeneousBNType.default_node_types()` and `HeterogeneousBNType.single_default()`.
- Added `BayesianNetworkBase.has_unknown_node_types()` and `BayesianNetworkBase.set_unknown_node_types()`.
- Changed signature of `BayesianNetworkType.compatible_node_type()` to include the new node type as argument.
- Removed `FactorType.opposite_semiparametric()`. This functionality has been replaced by `BayesianNetworkType.alternative_node_type()`.
- Included model as argument of `Operator.opposite()`.
- Added method `OperatorSet.set_type_blacklist()`. Added a type blacklist argument to `ChangeNodeTypeSet` constructor.

4.12 v0.1.0

- First release! =).

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[dag2pdag] Chickering, M. (2002). Learning Equivalence Classes of Bayesian-Network Structures. Journal of Machine Learning Research, 2, 445-498.

[dag2pdag_extra] Chickering, M. (1995). A Transformational Characterization of Equivalent Bayesian Network Structures. Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI'95), Montreal.

[pdag2dag] Dorit, D. and Tarsi, M. (1992). A simple algorithm to construct a consistent extension of a partially oriented graph (Report No: R-185).

[PGM] Koller, D. and Friedman, N. (2009). Probabilistic Graphical Models. MIT press.

[Scott] Scott, D. W. (2015). Multivariate Density Estimation: Theory, Practice and Visualization. 2nd Edition. Wiley

[MVKSA] José E. Chacón and Tarn Duong. (2018). Multivariate Kernel Smoothing and Its Applications. CRC Press.

[Semiparametric] David Atienza and Concha Bielza and Pedro Larrañaga. Semiparametric Bayesian networks. Information Sciences, vol. 584, pp. 564-582, 2022.

[HybridSemiparametric] David Atienza and Pedro Larrañaga and Concha Bielza. Hybrid semiparametric Bayesian networks. TEST, vol. 31, pp. 299-327, 2022.

[CMIknn] Runge, J. (2018). Conditional independence testing based on a nearest-neighbor estimator of conditional mutual information. International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 84, 938–947.

[RCoT] Strobl, E. V., Zhang, K., & Visweswaran, S. (2019). Approximate kernel-based conditional independence tests for fast non-parametric causal discovery. Journal of Causal Inference, 7(1).

[pc-stable] Colombo, D., & Maathuis, M. H. (2014). Order-independent constraint-based causal structure learning. Journal of Machine Learning Research, 15, 3921–3962.

[mmhc] Tsamardinos, I., Brown, L. E., & Aliferis, C. F. (2006). The max-min hill-climbing Bayesian network structure learning algorithm. Machine Learning, 65(1), 31–78.

[dmmhc] Trabelsi, G., Leray, P., Ben Ayed, M., & Alimi, A. M. (2013). Dynamic MMHC: A local search algorithm for dynamic Bayesian network structure learning. Advances in Intelligent Data Analysis XII, 8207 LNCS, 392–403.

[meek] Meek, C. (1995). Causal Inference and Causal Explanation with Background Knowledge. In Eleventh Conference on Uncertainty in Artificial Intelligence (UAI'95), 403–410.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

O

P

R

S