PyBNesian

Release 0.1.0dev0

David Atienza

Jul 11, 2021

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 3.6, 3.7, 3.8 and 3.9.

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==3.0` with PyBNesian, first install the required version of `pyarrow`:

```
pip install pyarrow==3.0.0
```

Then, proceed with the *Building* steps.

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.
- 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.models import GaussianNetwork
>>> from pybnesian.factors.continuous import 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
```

(continues on next page)

(continued from previous page)

```
>>> print("Indices: " + str(gbn.indices()))
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
```

(continues on next page)

(continued from previous page)

```
>>> sample = gbn.sample(50)

>>> # 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.learning.algorithms import hc
>>> from pybnesian.models import 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:

```
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.factors.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 `BayesianNetwork.fit()` or `DynamicBayesianNetwork.fit()`.
- `Factor.logl()`. This method is needed for `BayesianNetwork.logl()` or `DynamicBayesianNetwork.logl()`.
- `Factor.slogl()`. This method is needed for `BayesianNetwork.slogl()` or `DynamicBayesianNetwork.slogl()`.
- `Factor.sample()`. This method is needed for `BayesianNetwork.sample()` or `DynamicBayesianNetwork.sample()`.
- `Factor.data_type()`. This method is needed for `DynamicBayesianNetwork.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.factors import FactorType, Factor
from pybnesian.factors.continuous import 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):
        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
```

(continues on next page)

(continued from previous page)

```
        self._fitted = True

    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.models 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.models 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.models import HeterogeneousBN
>>> from pybnesian.factors.continuous import CKDEType
>>> from pybnesian.models import 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.
```

(continues on next page)

(continued from previous page)

```
>>> 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.models import HeterogeneousBN
>>> from pybnesian.factors.continuous import CKDEType
>>> from pybnesian.factors.discrete import DiscreteFactorType
>>> from pybnesian.models import 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.models 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
```

(continues on next page)

(continued from previous page)

```
    # 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)

    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.models 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.models import GaussianNetworkType, BayesianNetwork
>>> g1 = BayesianNetwork(GaussianNetworkType(), ["a", "b", "c", "d"])
>>> g2 = BayesianNetwork(MyRestrictedGaussianType(), ["a", "b", "c", "d"])
```

(continues on next page)

(continued from previous page)

```
>>> 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.models import BayesianNetwork, ConditionalBayesianNetwork,\
                             DynamicBayesianNetwork

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):
```

(continues on next page)

(continued from previous page)

```
        # 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.learning.independences 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
```

(continues on next page)

(continued from previous page)

```
            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"]):
                    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.learning.algorithms 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 can not 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.learning.scores 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"]
```

(continues on next page)

(continued from previous page)

```
    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

    # 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.models import GaussianNetwork
>>> from pybnesian.learning.algorithms import GreedyHillClimbing
>>> from pybnesian.learning.operators import 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.learning.operators import Operator, RemoveArc

class MyAddArc(Operator):

    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.learning.operators 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:
```

(continues on next page)

(continued from previous page)

```
                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:
            arc = s[0]
            delta = s[1]
            op = MyAddArc(arc[0], arc[1], delta)
            # The operator can not 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.learning.algorithms import GreedyHillClimbing
>>> hc = GreedyHillClimbing()
>>> add_set = MyAddArcSet()
>>> # We will use the OracleScore: a -> c <- b, c -> d
>>> score = OracleScore()
```

(continues on next page)

(continued from previous page)

```
>>> 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.learning.algorithms.callbacks import Callback

class PrintOperator(Callback):

    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.learning.algorithms 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!
```

CHAPTER

THREE

API REFERENCE

3.1 Data Manipulation

3.1.1 DataFrame Operations

3.1.2 Dynamic Data

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

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.graph 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")
```

(continues on next page)

(continued from previous page)

```
>>> 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.graph 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()])
```

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 can not 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.

3.2.3 Bibliography

3.3 Factors module

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`.

3.3.2 Continuous Factors

The continuous factors are implemented in the submodule pybnesian.factors.continuous.

Linear Gaussian CPD

Conditional Kernel Density Estimation (CKDE)

3.3.3 Discrete Factors

The discrete factors are implemented in the submodule pybnesian.factors.discrete.

3.3.4 Other Types

This types are not factors, but are auxiliary types for other factors.

3.3.5 Bibliography

3.4 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).

3.4.2 Bayesian Network Types

3.4.3 Bayesian Networks

Concrete Bayesian Networks

These classes implements `BayesianNetwork` with an specific `BayesianNetworkType`. Thus, the constructors do not have the `type` parameter.

3.4.4 Conditional Bayesian Networks

Concrete Conditional Bayesian Networks

These classes implements `ConditionalBayesianNetwork` with an specific `BayesianNetworkType`. Thus, the constructors do not have the `type` parameter.

3.4.5 Dynamic Bayesian Networks

Concrete Dynamic Bayesian Networks

These classes implements `DynamicBayesianNetwork` with an specific `BayesianNetworkType`. Thus, the constructors do not have the `type` parameter.

3.5 Learning module

3.5.1 Parameter Learning

Currently, it only implements Maximum Likelihood Estimation (MLE) for `LinearGaussianCPD` and `DiscreteFactor`.

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

Concrete classes

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

Operator Sets

Other

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

Concrete classes

Bibliography

3.5.5 Learning Algorithms

This classes implement many different learning structure algorithms.

Learning Algorithms Components

Learning Callbacks

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.graph 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 *pybnesian.load()* can load any saved object:

```
>>> import pickle
>>> from pybnesian import load
>>> from pybnesian.graph import Dag
>>> g = Dag(["a", "b", "c", "d"], [("a", "b")])
```

(continues on next page)

(continued from previous page)

```
>>> 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 v0.2.1

- 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.

4.2 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.

4.3 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

[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

INDEX