pgmpy Documentation

Release 0.1.0

pgmpy Developers

September 02, 2016

`pgmpy` is a Python library for creation, manipulation and implementation of Probablistic Graphical Models (PGM).

- Uses SciPy stack and NetworkX for mathematical and graph operations respectively.
- Provides interface to existing PGM algorithms.

Installation

1.1 Getting the dependencies

Installing from source requires you to have installed

- `Python3`
- `networkx` (==1.8.1)
- `numpy` (==1.9.1)
- `scipy` (==0.14.0)
- `cython` (==0.21)
- `pandas` (==0.15.1)
- `setuptools`
- working `C` and `C++` compiler

You can install all these requirements by issuing

```
$ [sudo] apt-get install build-essential python3-dev python3-pip
$ [sudo] pip3 install -r requirements.txt       # use requirements-dev.txt if you want to run tests
```

On Red Hat and clones (e.g CentOS), install the dependencies using:

```
$ [sudo] yum -y install gcc gcc-c++ python3-devel python3-pip
$ [sudo] pip3 install -r requirements.txt       # use requirements-dev.txt if you want to run tests
```

Or use some cross-platform binary package manager such as conda (it is recommended as well as the most easiest and hastle-free way)

Setup a virtual environment in `conda` by

```
$ conda create -n pgmpy-env python=3.4
$ source activate pgmpy-env
```

Once you have the virtual environment setup, install the depenedencies using:

```
$ conda install -f requirements.txt     # use requirements-dev.txt if you want to run tests
```

Note: In order to build the documentation you will need `sphinx` and to run the tests you will need `nose`

```
$ [sudo] pip3 install sphinx nose
```

1.2 Installing from source

You can install from source by downloading a source archive file (zip) or by checking out the source files from `git` source repository.

1. Download the source (zip file) from https://github.com/pgmpy/pgmpy or clone the pgmpy repository:

```
    $ git clone https://github.com/pgmpy/pgmpy
    $ git checkout dev
```

2. Unpack (if necessary) and change directory to the source directory.
3. Run:

```
    $ [sudo] python3 setup.py install
```

1.3 Testing

Testing requires having the `nose` library. After installation, the package can be tested by executing *from* the source directory:

```
$ nosetests3
```

This would give you a lot of output (and some warnings) but eventually should finish without errors. Otherwise, please consider posting an issue into the bug tracker or the Mailing List pgmpy@googlegroups.com .

CHAPTER 2

pgmpy API Reference

2.1 models module

2.1.1 Directed Graphical Models

2.1.2 Undirected Graphical Models

2.2 factors module

class `pgmpy.factors.`**`FactorSet`**`(`**factors_list*`)`

Base class of *DiscreteFactor Sets*.

A factor set provides a compact representation of higher dimensional factor $\phi_1 \cdot \phi_2 \cdots \phi_n$

For example the factor set corresponding to factor $\phi_1 \cdot \phi_2$ would be the union of the factors ϕ_1 and ϕ_2 i.e. factor set $\vec{\phi} = \phi_1 \cup \phi_2$.

`add_factors``(`**factors*`)`

Adds factors to the factor set.

Parameters **factors: Factor1, Factor2, ..., Factorn** :

factors to be added into the factor set

Examples

```
>>> from pgmpy.factors import FactorSet
>>> from pgmpy.factors import DiscreteFactor
>>> phi1 = DiscreteFactor(['x1', 'x2', 'x3'], [2, 3, 2], range(12))
>>> phi2 = DiscreteFactor(['x3', 'x4', 'x1'], [2, 2, 2], range(8))
>>> factor_set1 = FactorSet(phi1, phi2)
>>> phi3 = DiscreteFactor(['x5', 'x6', 'x7'], [2, 2, 2], range(8))
>>> phi4 = DiscreteFactor(['x5', 'x7', 'x8'], [2, 2, 2], range(8))
>>> factor_set1.add_factors(phi3, phi4)
>>> print(factor_set1)
set([<DiscreteFactor representing phi(x1:2, x2:3, x3:2) at 0x7f8e32b4ca10>,
     <DiscreteFactor representing phi(x5:2, x7:2, x8:2) at 0x7f8e4c393690>,
     <DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f8e32b4c750>,
     <DiscreteFactor representing phi(x3:2, x4:2, x1:2) at 0x7f8e32b4cb50>])
```

copy()

Create a copy of factor set.

Examples

```
>>> from pgmpy.factors import FactorSet
>>> from pgmpy.factors import DiscreteFactor
>>> phi1 = DiscreteFactor(['x1', 'x2', 'x3'], [2, 3, 2], range(12))
>>> phi2 = DiscreteFactor(['x3', 'x4', 'x1'], [2, 2, 2], range(8))
>>> factor_set = FactorSet(phi1, phi2)
>>> factor_set
<pgmpy.factors.FactorSet.FactorSet at 0x7fa68f390320>
>>> factor_set_copy = factor_set.copy()
>>> factor_set_copy
<pgmpy.factors.FactorSet.FactorSet at 0x7f91a0031160>
```

divide(*factorset*, *inplace=True*)

Returns a new factor set instance after division by the factor set

Division of two factor sets $\frac{\vec\phi_1}{\vec\phi_2}$ basically translates to union of all the factors present in $\vec\phi_2$ and $\frac{1}{\phi_i}$ of all the factors present in $\vec\phi_2$.

Parameters **factorset: FactorSet** :

The divisor

inplace: A boolean (Default value True) :

If inplace = True ,then it will modify the FactorSet object, if False then will return a new FactorSet object.

Returns **If inplace = False, will return a new FactorSet Object which is division of** :

given factors. :

Examples

```
>>> from pgmpy.factors import FactorSet
>>> from pgmpy.factors import DiscreteFactor
>>> phi1 = DiscreteFactor(['x1', 'x2', 'x3'], [2, 3, 2], range(12))
>>> phi2 = DiscreteFactor(['x3', 'x4', 'x1'], [2, 2, 2], range(8))
>>> factor_set1 = FactorSet(phi1, phi2)
>>> phi3 = DiscreteFactor(['x5', 'x6', 'x7'], [2, 2, 2], range(8))
>>> phi4 = DiscreteFactor(['x5', 'x7', 'x8'], [2, 2, 2], range(8))
>>> factor_set2 = FactorSet(phi3, phi4)
>>> factor_set3 = factor_set2.divide(factor_set1)
>>> print(factor_set3)
set([<DiscreteFactor representing phi(x3:2, x4:2, x1:2) at 0x7f8e32b5ba10>,
     <DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f8e32b5b650>,
     <DiscreteFactor representing phi(x1:2, x2:3, x3:2) at 0x7f8e32b5b050>,
     <DiscreteFactor representing phi(x5:2, x7:2, x8:2) at 0x7f8e32b5b8d0>])
```

get_factors()

Returns all the factors present in factor set.

Examples

```
>>> from pgmpy.factors import FactorSet
>>> from pgmpy.factors import DiscreteFactor
>>> phi1 = DiscreteFactor(['x1', 'x2', 'x3'], [2, 3, 2], range(12))
>>> phi2 = DiscreteFactor(['x3', 'x4', 'x1'], [2, 2, 2], range(8))
>>> factor_set1 = FactorSet(phi1, phi2)
>>> phi3 = DiscreteFactor(['x5', 'x6', 'x7'], [2, 2, 2], range(8))
>>> factor_set1.add_factors(phi3)
>>> factor_set1.get_factors()
{<DiscreteFactor representing phi(x1:2, x2:3, x3:2) at 0x7f827c0a23c8>,
 <DiscreteFactor representing phi(x3:2, x4:2, x1:2) at 0x7f827c0a2358>,
 <DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f825243f9e8>}
```

marginalize (*variables*, *inplace=True*)

Marginalizes the factors present in the factor sets with respect to the given variables.

Parameters variables: list, array-like :

List of the variables to be marginalized.

inplace: boolean (Default value True) :

If inplace=True it will modify the factor set itself, would create a new factor set

Returns If inplace = False, will return a new marginalized FactorSet object. :

Examples

```
>>> from pgmpy.factors import FactorSet
>>> from pgmpy.factors import DiscreteFactor
>>> phi1 = DiscreteFactor(['x1', 'x2', 'x3'], [2, 3, 2], range(12))
>>> phi2 = DiscreteFactor(['x3', 'x4', 'x1'], [2, 2, 2], range(8))
>>> factor_set1 = FactorSet(phi1, phi2)
>>> factor_set1.marginalize('x1')
>>> print(factor_set1)
set([<DiscreteFactor representing phi(x2:3, x3:2) at 0x7f8e32b4cc10>,
     <DiscreteFactor representing phi(x3:2, x4:2) at 0x7f8e32b4cf90>])
```

product (*factorset*, *inplace=True*)

Return the factor sets product with the given factor sets

Suppose $\vec\phi_1$ and $\vec\phi_2$ are two factor sets then their product is a another factors set $\vec\phi_3 = \vec\phi_1 \cup \vec\phi_2$.

Parameters factorsets: FactorSet1, FactorSet2, ..., FactorSetn :

FactorSets to be multiplied

inplace: A boolean (Default value True) :

If inplace = True , then it will modify the FactorSet object, if False, it will return a new FactorSet object.

Returns If inpalce = False, will return a new FactorSet object, which is product of two factors :

Examples

```
>>> from pgmpy.factors import FactorSet
>>> from pgmpy.factors import DiscreteFactor
>>> phi1 = DiscreteFactor(['x1', 'x2', 'x3'], [2, 3, 2], range(12))
>>> phi2 = DiscreteFactor(['x3', 'x4', 'x1'], [2, 2, 2], range(8))
>>> factor_set1 = FactorSet(phi1, phi2)
>>> phi3 = DiscreteFactor(['x5', 'x6', 'x7'], [2, 2, 2], range(8))
>>> phi4 = DiscreteFactor(['x5', 'x7', 'x8'], [2, 2, 2], range(8))
>>> factor_set2 = FactorSet(phi3, phi4)
>>> print(factor_set2)
set([<DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f8e32b5b050>,
     <DiscreteFactor representing phi(x5:2, x7:2, x8:2) at 0x7f8e32b5b690>])
>>> factor_set2.product(factor_set1)
>>> print(factor_set2)
set([<DiscreteFactor representing phi(x1:2, x2:3, x3:2) at 0x7f8e32b4c910>,
     <DiscreteFactor representing phi(x3:2, x4:2, x1:2) at 0x7f8e32b4cc50>,
     <DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f8e32b5b050>,
     <DiscreteFactor representing phi(x5:2, x7:2, x8:2) at 0x7f8e32b5b690>])
>>> factor_set2 = FactorSet(phi3, phi4)
>>> factor_set3 = factor_set2.product(factor_set1, inplace=False)
>>> print(factor_set2)
set([<DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f8e32b5b060>,
     <DiscreteFactor representing phi(x5:2, x7:2, x8:2) at 0x7f8e32b5b790>])
```

remove_factors(**factors*)

Removes factors from the factor set.

Parameters factors: Factor1, Factor2, ..., Factorn :

factors to be removed from the factor set

Examples

```
>>> from pgmpy.factors import FactorSet
>>> from pgmpy.factors import DiscreteFactor
>>> phi1 = DiscreteFactor(['x1', 'x2', 'x3'], [2, 3, 2], range(12))
>>> phi2 = DiscreteFactor(['x3', 'x4', 'x1'], [2, 2, 2], range(8))
>>> factor_set1 = FactorSet(phi1, phi2)
>>> phi3 = DiscreteFactor(['x5', 'x6', 'x7'], [2, 2, 2], range(8))
>>> factor_set1.add_factors(phi3)
>>> print(factor_set1)
set([<DiscreteFactor representing phi(x1:2, x2:3, x3:2) at 0x7f8e32b5b050>,
     <DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f8e32b5b250>,
     <DiscreteFactor representing phi(x3:2, x4:2, x1:2) at 0x7f8e32b5b150>])
>>> factor_set1.remove_factors(phi1, phi2)
>>> print(factor_set1)
set([<DiscreteFactor representing phi(x5:2, x6:2, x7:2) at 0x7f8e32b4cb10>])
```

2.3 inference module

2.4 independencies module

class `pgmpy.independencies.`**`Independencies`**`(`**assertions*`)`

Base class for independencies. independencies class represents a set of Conditional Independence assertions (eg: "X is independent of Y given Z" where X, Y and Z are random variables) or Independence assertions (eg: "X is independent of Y" where X and Y are random variables). Initialize the independencies Class with Conditional Independence assertions or Independence assertions.

Parameters assertions: Lists or Tuples :

Each assertion is a list or tuple of the form: [event1, event2 and event3] eg: assertion ['X', 'Y', 'Z'] would be X is independent of Y given Z.

Examples

Creating an independencies object with one independence assertion: Random Variable X is independent of Y

```
>>> independencies = independencies(['X', 'Y'])
```

Creating an independencies object with three conditional independence assertions: First assertion is Random Variable X is independent of Y given Z.

```
>>> independencies = independencies(['X', 'Y', 'Z'],
...             ['a', ['b', 'c'], 'd'],
...             ['l', ['m', 'n'], 'o'])
```

`add_assertions``(`**assertions*`)`

Adds assertions to independencies.

Parameters assertions: Lists or Tuples :

Each assertion is a list or tuple of variable, independent_of and given.

Examples

```
>>> from pgmpy.independencies import Independencies
>>> independencies = Independencies()
>>> independencies.add_assertions(['X', 'Y', 'Z'])
>>> independencies.add_assertions(['a', ['b', 'c'], 'd'])
```

`closure``()`

Returns a new *Independencies()*-object that additionally contains those *IndependenceAssertions* that are implied by the the current independencies (using with the semi-graphoid axioms; see (Pearl, 1989, Conditional Independence and its representations)).

Might be very slow if more than six variables are involved.

Examples

```
>>> from pgmpy.independencies import Independencies
>>> ind1 = Independencies(('A', ['B', 'C'], 'D'))
>>> ind1.closure()
(A _|_ B | D, C)
(A _|_ B, C | D)
(A _|_ B | D)
(A _|_ C | D, B)
(A _|_ C | D)
```

```
>>> ind2 = Independencies(('W', ['X', 'Y', 'Z']))
>>> ind2.closure()
(W _|_ Y)
(W _|_ Y | X)
(W _|_ Z | Y)
(W _|_ Z, X, Y)
(W _|_ Z)
(W _|_ Z, X)
(W _|_ X, Y)
(W _|_ Z | X)
(W _|_ Z, Y | X)
[..]
```

contains(*assertion*)

Returns *True* if *assertion* is contained in this *Independencies*-object, otherwise *False*.

Parameters assertion: IndependenceAssertion()-object :

Examples

```
>>> from pgmpy.independencies import Independencies, IndependenceAssertion
>>> ind = Independencies(['A', 'B', ['C', 'D']])
>>> IndependenceAssertion('A', 'B', ['C', 'D']) in ind
True
>>> # does not depend on variable order:
>>> IndependenceAssertion('B', 'A', ['D', 'C']) in ind
True
>>> # but does not check entailment:
>>> IndependenceAssertion('X', 'Y', 'Z') in Independencies(['X', 'Y'])
False
```

entails(*entailed_independencies*)

Returns *True* if the *entailed_independencies* are implied by this *Independencies*-object, otherwise *False*. Entailment is checked using the semi-graphoid axioms.

Might be very slow if more than six variables are involved.

Parameters entailed_independencies: Independencies()-object :

Examples

```
>>> from pgmpy.independencies import Independencies
>>> ind1 = Independencies([['A', 'B'], ['C', 'D'], 'E'])
>>> ind2 = Independencies(['A', 'C', 'E'])
>>> ind1.entails(ind2)
True
```

```
>>> ind2.entails(ind1)
False
```

get_assertions()

Returns the independencies object which is a set of IndependenceAssertion objects.

Examples

```
>>> from pgmpy.independencies import Independencies
>>> independencies = Independencies(['X', 'Y', 'Z'])
>>> independencies.get_assertions()
```

is_equivalent(*other*)

Returns True if the two Independencies-objects are equivalent, otherwise False. (i.e. any Bayesian Network that satisfies the one set of conditional independencies also satisfies the other).

Might be very slow if more than six variables are involved.

Parameters other: Independencies()-object :

Examples

```
>>> from pgmpy.independencies import Independencies
>>> ind1 = Independencies(['X', ['Y', 'W'], 'Z'])
>>> ind2 = Independencies(['X', 'Y', 'Z'], ['X', 'W', 'Z'])
>>> ind3 = Independencies(['X', 'Y', 'Z'], ['X', 'W', 'Z'], ['X', 'Y', ['W','Z']])
>>> ind1.is_equivalent(ind2)
False
>>> ind1.is_equivalent(ind3)
True
```

latex_string()

Returns a list of string. Each string represents the IndependenceAssertion in latex.

reduce()

Add function to remove duplicate Independence Assertions

class pgmpy.independencies.**IndependenceAssertion**(*event1=[]*, *event2=[]*, *event3=[]*)

Represents Conditional Independence or Independence assertion.

Each assertion has 3 attributes: event1, event2, event3. The attributes for

$$U \perp X, Y | Z$$

is read as: Random Variable U is independent of X and Y given Z would be:

event1 = {U}

event2 = {X, Y}

event3 = {Z}

Parameters event1: String or List of strings :

Random Variable which is independent.

event2: String or list of strings. :

Random Variables from which event1 is independent

event3: String or list of strings. :

Random Variables given which event1 is independent of event2.

Examples

```
>>> from pgmpy.independencies import IndependenceAssertion
>>> assertion = IndependenceAssertion('U', 'X')
>>> assertion = IndependenceAssertion('U', ['X', 'Y'])
>>> assertion = IndependenceAssertion('U', ['X', 'Y'], 'Z')
>>> assertion = IndependenceAssertion(['U', 'V'], ['X', 'Y'], ['Z', 'A'])
```

get_assertion()

Returns a tuple of the attributes: variable, independent_of, given.

Examples

```
>>> from pgmpy.independencies import IndependenceAssertion
>>> asser = IndependenceAssertion('X', 'Y', 'Z')
>>> asser.get_assertion()
```

2.5 readwrite module

2.6 base module

class pgmpy.base.**DirectedGraph**(*ebunch=None*)

Base class for directed graphs.

Directed graph assumes that all the nodes in graph are either random variables, factors or clusters of random variables and edges in the graph are dependencies between these random variables.

Parameters data: input graph :

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list or any Networkx graph object.

Examples

Create an empty DirectedGraph with no nodes and no edges

```
>>> from pgmpy.base import DirectedGraph
>>> G = DirectedGraph()
```

G can be grown in several ways

Nodes:

Add one node at a time:

```
>>> G.add_node('a')
```

Add the nodes from any container (a list, set or tuple or the nodes from another graph).

```
>>> G.add_nodes_from(['a', 'b'])
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge('a', 'b')
```

a list of edges,

```
>>> G.add_edges_from([('a', 'b'), ('b', 'c')])
```

If some edges connect nodes not yet in the model, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Shortcuts:

Many common graph features allow python syntax for speed reporting.

```
>>> 'a' in G     # check if node in graph
True
>>> len(G)   # number of nodes in graph
3
```

add_edge(*u*, *v*, ***kwargs*)

Add an edge between u and v.

The nodes u and v will be automatically added if they are not already in the graph

Parameters u,v : nodes

Nodes can be any hashable Python object.

Examples

```
>>> from pgmpy.base import DirectedGraph
>>> G = DirectedGraph()
>>> G.add_nodes_from(['Alice', 'Bob', 'Charles'])
>>> G.add_edge('Alice', 'Bob')
```

add_edges_from(*ebunch*, ***kwargs*)

Add all the edges in ebunch.

If nodes referred in the ebunch are not already present, they will be automatically added. Node names should be strings.

Parameters ebunch : container of edges

Each edge given in the container will be added to the graph. The edges must be given as 2-tuples (u, v).

Examples

```
>>> from pgmpy.base import DirectedGraph
>>> G = DirectedGraph()
>>> G.add_nodes_from(['Alice', 'Bob', 'Charles'])
>>> G.add_edges_from([('Alice', 'Bob'), ('Bob', 'Charles')])
```

add_node(*node*, ***kwargs*)

Add a single node to the Graph.

Parameters **node: node** :

A node can be any hashable Python object.

Examples

```
>>> from pgmpy.base import DirectedGraph
>>> G = DirectedGraph()
>>> G.add_node('A')
```

add_nodes_from(*nodes*, ***kwargs*)

Add multiple nodes to the Graph.

Parameters **nodes: iterable container** :

A container of nodes (list, dict, set, etc.).

Examples

```
>>> from pgmpy.base import DirectedGraph
>>> G = DirectedGraph()
>>> G.add_nodes_from(['A', 'B', 'C'])
```

get_parents(*node*)

Returns a list of parents of node.

Parameters **node: string, int or any hashable python object.** :

The node whose parents would be returned.

Examples

```
>>> from pgmpy.base import DirectedGraph
>>> G = DirectedGraph([('diff', 'grade'), ('intel', 'grade')])
>>> G.parents('grade')
['diff', 'intel']
```

moralize()

Removes all the immoralities in the DirectedGraph and creates a moral graph (UndirectedGraph).

A v-structure X->Z<-Y is an immorality if there is no directed edge between X and Y.

Examples

```
>>> from pgmpy.base import DirectedGraph
>>> G = DirectedGraph([('diff', 'grade'), ('intel', 'grade')])
>>> moral_graph = G.moralize()
>>> moral_graph.edges()
[('intel', 'grade'), ('intel', 'diff'), ('grade', 'diff')]
```

class `pgmpy.base.`**`UndirectedGraph`**(*ebunch=None*)

Base class for all the Undirected Graphical models.

UndirectedGraph assumes that all the nodes in graph are either random variables, factors or cliques of random variables and edges in the graphs are interactions between these random variables, factors or clusters.

Parameters **data: input graph** :

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list or any Networkx graph object.

Examples

Create an empty UndirectedGraph with no nodes and no edges

```
>>> from pgmpy.base import UndirectedGraph
>>> G = UndirectedGraph()
```

G can be grown in several ways

Nodes:

Add one node at a time:

```
>>> G.add_node('a')
```

Add the nodes from any container (a list, set or tuple or the nodes from another graph).

```
>>> G.add_nodes_from(['a', 'b'])
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge('a', 'b')
```

a list of edges,

```
>>> G.add_edges_from([('a', 'b'), ('b', 'c')])
```

If some edges connect nodes not yet in the model, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Shortcuts:

Many common graph features allow python syntax for speed reporting.

```
>>> 'a' in G      # check if node in graph
True
>>> len(G)   # number of nodes in graph
3
```

`add_edge`(*u*, *v*, ***kwargs*)

Add an edge between u and v.

The nodes u and v will be automatically added if they are not already in the graph

Parameters **u,v** : nodes

Nodes can be any hashable Python object.

Examples

```
>>> from pgmpy.base import UndirectedGraph
>>> G = UndirectedGraph()
>>> G.add_nodes_from(['Alice', 'Bob', 'Charles'])
>>> G.add_edge('Alice', 'Bob')
```

add_edges_from(*ebunch*, ***kwargs*)
Add all the edges in ebunch.

If nodes referred in the ebunch are not already present, they will be automatically added.

Parameters **ebunch** : container of edges

Each edge given in the container will be added to the graph. The edges must be given as 2-tuples (u, v).

Examples

```
>>> from pgmpy.base import UndirectedGraph
>>> G = UndirectedGraph()
>>> G.add_nodes_from(['Alice', 'Bob', 'Charles'])
>>> G.add_edges_from([('Alice', 'Bob'), ('Bob', 'Charles')])
```

add_node(*node*, ***kwargs*)
Add a single node to the Graph.

Parameters **node: node** :

A node can be any hashable Python object.

Examples

```
>>> from pgmpy.base import UndirectedGraph
>>> G = UndirectedGraph()
>>> G.add_node('A')
```

add_nodes_from(*nodes*, ***kwargs*)
Add multiple nodes to the Graph.

Parameters **nodes: iterable container** :

A container of nodes (list, dict, set, etc.).

Examples

```
>>> from pgmpy.base import UndirectedGraph
>>> G = UndirectedGraph()
>>> G.add_nodes_from(['A', 'B', 'C'])
```

check_clique(*nodes*)
Check if the given nodes form a clique.

Parameters **nodes: list, array-like** :

List of nodes to check if they are a part of any clique.

is_triangulated()

Checks whether the undirected graph is triangulated or not.

Examples

```
>>> from pgmpy.base import UndirectedGraph
>>> G = UndirectedGraph()
>>> G.add_edges_from([('x1', 'x2'), ('x1', 'x3'), ('x1', 'x4'),
...                   ('x2', 'x4'), ('x3', 'x4')])
>>> G.is_triangulated()
True
```

CHAPTER 3

Example gallery

CHAPTER 4

GSoC 2015 Ideas

4.1 Introduction

A graphical model or probabilistic graphical model (PGM) is a probabilistic model for which a graph expresses the conditional dependence structure between random variables. They are most commonly used in probability theory, statistics (particularly Bayesian statistics) and machine learning.

pgmpy is a Python library to implement Probabilistic Graphical Models and related inference and learning algorithms. Our main focus is on providing a consistent API and flexible approach to its implementation. This is the second year pgmpy is participating in GSoC.

4.2 Want to get involved?

If you're interested in participating in GSoC 2015 as a student, mentor, or community member, you should join the pgmpy's mailing list and post any questions, comments, etc. to pgmpy@googlegroups.com

Additionally, you can find us on IRC at #pgmpy on irc.freenode.org. If no one is available to answer your question, please be patient and post it to the mailing list as well.

4.3 Getting Started

1. Install dependencies:

```
$ sudo pip3 install -r requirements.txt  # use requirements-dev.txt if you want to run tests
```

2. Clone the repo:

```
$ git clone https://github.com/pgmpy/pgmpy
```

3. Install pgmpy:

```
$ cd pgmpy/
$ sudo python3 setup.py install
```

4.3.1 References for PGM:

- Notebooks for basic introduction of PGM and pgmpy: https://github.com/pgmpy/pgmpy_notebook

- Quick intro to Bayesian Networks: http://people.cs.ubc.ca/~murphyk/Bayes/bnintro.html
- Reference book for PGM: Probabilistic Graphical Models - Principles and Techniques

4.4 Ideas

4.4.1 1. Add feature to accept and output state names for models.

At present pgmpy internally assigns a numerical value to each state of a random variable. For example, for a variable `grade` having states `A`, `B` and `C`, the internal representation in pgmpy would be `grade_0`, `grade_1` and `grade_2`. Also if some method needs to output a state name, it gives the state name in this form only. We want improve this and allow the user to work with the state names rather than the internal representations.

Expected Outcome: The user should be able to completely work with the state names (never need to use internal representation) that he has provided.

Difficulty Level: Moderate

PGM knowledge required: Basic

Skills Required: Intermediate Python

Potential Mentor(s): Ankur Ankan, Shashank Garg

4.4.2 2. Approximate Algorithms

At present in `pgmpy`, we have implementation of exact inference algorithms for various graphical models. Although inference algorithms run in polynomial time for simple graphs (such as graphs with low tree-width), they become computationally intractable for larger graphs that arise from real life problem. However there a class of algorithms that can be used to perform approximate inference on the graphical models. This project aims towards implementation of two famous approximate inference algorithms

- Linear Programming Relaxation
- Cutting Plane Algorithms

Expected Outcome: We should be able to run approximate inference algorithms on complex graphical models (used in stereo vision).

Difficulty Level: Difficult

PGM knowledge required: Very good understanding of Graphical Models and Inference Algorithms

Skills Required: Intermediate Python, Cython

Potential Mentor(s): Abinash Panda, Ankur Ankan

4.4.3 3. Adding support for different types of CPDs

Right now pgmpy has the feature for creating `Rule CPDs` and `Tree CPDs` but the current implementation of variable elimination or clique tree don't accept the models if a `Rule CPD` or `Tree CPD` is associated with it. There are algorithms that are able to do inference much efficiently in the case of Rule and Tree CPDs as compared to normal Tabular CPD. Implement those algorithms.

Expected Outcome: We should be able to run inference algorithms over models having associated Rule CPD or Tree CPD.

Difficulty Level: Difficult

PGM knowledge required: Good understanding of Bayesian Models and inference algorithms.

Skills Required: Intermediate Python, Cython

Potential Mentor(s): Jaidev Deshpande, Shashank Garg

4.4.4 4. Adding support for Dynamic Bayesian Networks (DBNs)

Dynamic Bayesian Networks are used to represent models which have repeating pattern. It is mostly used when we are trying to create a model with time as a variable, so for each instant of time we have the same model and hence a repeating model. Currently pgmpy doesn't have support for DBNs.

Expected Outcome: Should be able to create DBNs and do inference over it.

Difficulty Level: Difficult

PGM knowledge required: Very good understanding of PGM.

Skills Required: Intermediate Python, Cython

Potential Mentor(s): Ankur Ankan, Abinash Panda

4.4.5 5. Parsing from and writing to standard PGM file formats

There are various standard file formats for representing the PGM data. PGM data basically consists of a Graph, a table corresponding to each node and a few other attributes of the Graph. Here is a list of some of these formats. pgmpy needs functionality to read networks from and write networks to these standard file formats. Currently only **ProbModelXML** is supported. pgmpy uses lxml for XML formats and we plan to use pyparsing for non XML formats.

Expected Outcome: You are expected to choose at least one file format from the above list and write a sub-module which enables pgmpy to read from and write to the same format.

Difficulty level: Easy

PGM knowledge required: Basic knowledge about representation of PGM models.

Skills Required: Intermediate python

Potential Mentor(s): Pranjal Mittal, Shashank Garg

CHAPTER 5

GSoC 2014 Ideas

5.1 Introduction

Probabilistic Graphical Models (PGM) use graphs to denote the conditional dependence structure between random variables. They are most commonly used in probability theory, statistics (particularly Bayesian statistics) and machine learning.

pgmpy is a Python library to implement Probabilistic Graphical Models and related algorithms. The main focus is on providing a consistent API and flexible approach to its implementation. This is the first time pgmpy is applying for GSoC under the Python Software Foundation's umbrella.

5.2 Want to get involved?

If you're interested in participating in GSoC 2014 as a student, mentor, or interested community member, you should join the pgmpy's mailing list and post any questions, comments, etc. to pgmpy@googlegroups.com

You can also contact the mentors with your ideas.

Anavil Tripathi: anaviltripathi@gmail.com

Shikhar Nigam: snigam3112@gmail.com

Soumya Kundu: samkent.1729@gmail.com

Additionally, you can find us on IRC at #pgmpy on irc.freenode.org. If no one is available to answer your question, please be patient and post it to the mailing list as well.

5.3 Getting Started

Reference book for PGM: Probabilistic Graphical Models - Principles and Techniques

5.3.1 pgmpy

1. Install dependencies:

```
$ sudo pip3 install networkx numpy scipy cython
```

2. Clone the repo:

```
$ git clone https://github.com/pgmpy/pgmpy
```

3. Install pgmpy:

```
$ cd pgmpy/
$ sudo python3 setup.py install
```

5.3.2 pgmpy_viz

1. Install dependencies:

```
$ sudo pip3 install django
```

2. Clone the repo:

```
$ git clone https://github.com/pgmpy/pgmpy_viz
```

3. Run local server:

```
$ cd pgmpy_viz/
$ python3 manage.py runserver
```

Go to `localhost:8000` in your browser to access the pgmpy_viz page.

5.4 Example

```
from pgmpy.models import BayesianModel
from pgmpy.factors import TabularCPD
student = bm.BayesianModel()
# instantiates a new Bayesian Model called 'student'

student.add_nodes_from(['diff', 'intel', 'grade'])
# adds nodes labelled 'diff', 'intel', 'grade' to student

student.add_edges_from([('diff', 'grade'), ('intel', 'grade')])
# adds directed edges from 'diff' to 'grade' and 'intel' to 'grade'

"""
diff cpd:

+--------+--------+
|diff:   |        |
+--------+--------+
|easy    |   0.2  |
+--------+--------+
|hard    |   0.8  |
+--------+--------+
"""
diff_cpd = TabularCPD('diff', 2, [[0.2], [0.8]])

"""
intel cpd:

+--------+--------+
|intel:  |        |
+--------+--------+
|dumb    |   0.5  |
+--------+--------+
|avg     |   0.3  |
+--------+--------+
|smart   |   0.2  |
+--------+--------+
"""
intel_cpd = TabularCPD('intel', 3, [[0.5], [0.3], [0.2]])

"""
grade cpd:

+------+-----------------------+---------------------+
|diff: |          easy         |         hard        |
+------+------+------+---------+------+------+-------+
|intel:| dumb |  avg |  smart  | dumb | avg  | smart |
+------+------+------+---------+------+------+-------+
|gradeA| 0.1  | 0.1  |   0.1   |  0.1 |  0.1 |   0.1 |
+------+------+------+---------+------+------+-------+
|gradeB| 0.1  | 0.1  |   0.1   |  0.1 |  0.1 |   0.1 |
+------+------+------+---------+------+------+-------+
|gradeC| 0.8  | 0.8  |   0.8   |  0.8 |  0.8 |   0.8 |
+------+------+------+---------+------+------+-------+
"""
grade_cpd = TabularCPD('grade', 3,
                    [[0.1,0.1,0.1,0.1,0.1,0.1],
                        [0.1,0.1,0.1,0.1,0.1,0.1],
                        [0.8,0.8,0.8,0.8,0.8,0.8]],
                    evidence=['diff', 'intel'],
                    evidence_card=[2, 3])

student.add_cpds(diff_cpd, intel_cpd, grade_cpd)

# Finding active trail
student.active_trail_nodes('diff')

# Finding active trail with observation
student.active_trail_nodes('diff', observed='grades')
```

5.5 Ideas

5.5.1 1. Parsing from and writing to standard PGM file formats

There are various standard file formats for representing the PGM data. PGM data basically consists of a Graph, a table corresponding to each node and a few other attributes of the Graph. Here is a list of some of these formats. pgmpy needs functionality to read networks from and write networks to these standard file formats. Currently only ProbModelXML is supported. pgmpy uses lxml for XML formats and we plan to use pyparsing for non XML formats.

Expected Outcome: You are expected to choose at least one file format from the above list and write a sub-module which enables pgmpy to read from and write to the same format.

Difficulty level: Medium

PGM knowledge required: Basic knowledge about representation of PGM models.

Skills required: Intermediate python

Potential Mentor(s): Shikhar Nigam

5.5.2 2. Adding features to pgmpy_viz

pgmpy_viz is a web application for creating and visualizing graphical models that runs pgmpy in the back-end. It uses cytoscape.js in the front-end for manipulation of the networks. For reference to a similar application you can look at SamIam.

This project needs you to add:

- Network validation before posting data to the server.
- Options for inference from networks.
- Porting pgmpy_viz from Django to Flask.

Expected Outcome: You are expected to design a Flask based web application which would enable the user to visualize the outcomes of analysis of the network.

Difficulty level: Medium

PGM knowledge required: None

Skills required: HTML5, CSS, JavaScript, Flask

Potential Mentor(s): Soumya Kundu

5.5.3 3. Implementing Markov Networks

There are two common branches of graphical representation of distributions. They are Bayesian networks(Directed Acyclic Graphs) and Markov networks(Undirected graphs which may be cyclic). Currently, pgmpy supports Bayesian Networks. The following features for Markov Networks need to be implemented:

- Create and edit Markov Networks.
- Finding reduced Markov Networks.
- Finding independencies in Markov Networks.

Expected Outcome: You are expected to write a sub-module implementing the above listed features.

Difficulty level: Hard

PGM knowledge required: Good understanding of Markov Networks

Skills required: Intermediate python, Cython

Potential Mentor(s): Anavil Tripathi

5.5.4 4. Implementing Algorithms:

PGM involves many theorems and algorithms such as Belief-Propagation, Variable Elimination etc. The library will eventually implement every PGM algorithm. Here is the proposed set of algorithms to be implemented.

Expected Outcome: You are expected to select at least one algorithm from the list and implement it.

Difficulty level: Hard

PGM knowledge required: Good understanding of PGM

Skills required: Intermediate python, Cython

Potential Mentor(s): Shikhar Nigam

5.5.5 5. Blue Sky Project

If you have any interesting ideas please discuss it over the mailing list.

5.6 Interested Students

If you are interested in participating in GSoC with pgmpy, please introduce yourself on the mailing list.

CHAPTER 6

Community

- **Mailing List**: pgmpy@googlegroups.com
- **IRC**: #pgmpy @ freenode.net

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index