pgmpy: Probabilistic Graphical Models using Python

Ankur Ankan*, Abinash Panda

https://www.youtube.com/watch?v=Vcmjqx7lht0

Abstract—Probabilistic Graphical Models (PGM) is a technique of compactly representing a joint distribution by exploiting dependencies between the random variables. It also allows us to do inference on joint distributions in a computationally cheaper way than the traditional methods. PGMs are widely used in the field of speech recognition, information extraction, image segmentation, modelling gene regulatory networks.

pgmpy [pgmpy] is a python library for working with graphical models. It allows the user to create their own graphical models and answer inference or map queries over them. pgmpy has implementation of many inference algorithms like VariableElimination, Belief Propagation etc.

This paper first gives a short introduction to PGMs and various other python packages available for working with PGMs. Then we discuss about creating and doing inference over Bayesian Networks and Markov Networks using pgmpy.

Index Terms—Graphical Models, Bayesian Networks, Markov Networks, Variable Elimination

Introduction

Probabilistic Graphical Model (PGM) is a technique of representing Joint Distributions over random variables in a compact way by exploiting the dependencies between them. PGMs use a network structure to encode the relationships between the random variables and some parameters to represent the joint distribution.

There are two major types of Graphical Models: Bayesian Networks and Markov Networks.

Bayesian Network: A Bayesian Network consists of a directed graph and a conditional probability distribution associated with each of the random variables. A Bayesian network is used mostly when there is a causal relationship between the random variables. An example of a Bayesian Network representing a student [student] taking some course is shown in Fig 1.

Markov Network: A Markov Network consists of an undirected graph and a few Factors are associated with it. Unlike Conditional Probability Distributions, a Factor does not represent the probabilities of variables in the network; instead it represents the compatibility between random variables that is how much a particular state of a random variable likely to agree with the another state of some other random variable. An example of markov [markov] network over four friends A, B, C, D agreeing to some concept is shown in Fig 2.

There are numerous open source packages available in Python for working with graphical models. eBay's bayesian-belief-networks [bbn] mostly focuses on Bayesian Models and has implementation of a limited number of inference algorithms. Another package pymc [pymc] focuses mainly on Markov Chain Monte Carlo (MCMC) method. libpgm [libpgm] also mainly focuses on Bayesian Networks.

pgmpy tries to be a complete package for working with graphical models and gives the user full control on designing the model. The source code is very well documented with proper docstrings and doctests for each method so that users can quickly get upto speed. Furthermore, pgmpy also provides easy extensibility allowing users to write their own inference algorithms or elimination order algorithms without any additional effort to get familiar with the source code.

* *Corresponding author: ankurankan@gmail.com*

Getting Source Code and Installing

pgmpy is released under MIT Licence and is hosted on github. We can simply clone the repository and install it:

```
git clone https://github.com/pgmpy/pgmpy
cd pgmpy
[sudo] python3 setup.py install
```

Dependencies: pgmpy runs only on python3 and is dependent on networkx, numpy, pandas and scipy which can be installed using pip or conda as:

```
pip install -r requirements.txt
```

or:

```
conda install --file requirements.txt
```

Creating Bayesian Models using pgmpy

A Bayesian Network consists of a directed graph where nodes represents random variables and edges represent the the relation between them. It is parameterized using Conditional Probability Distributions(CPD). Each random variable in a Bayesian Network has a CPD associated with it. If a random varible has parents in the network then the CPD represents $P(var|Par_{var})$ i.e. the probability of that variable given its parents. In the case, when the random variable has no parents it simply represents $P(var)$ i.e. the probability of that variable.

For example, we can take the case of student model represented in Fig 1. A possible CPD for the random variable grade is shown in Table 1.

We can represent the CPD shown in Table 1 in pgmpy as follows:

```
from pgmpy.factors import TabularCPD
grade_cpd = TabularCPD(
```

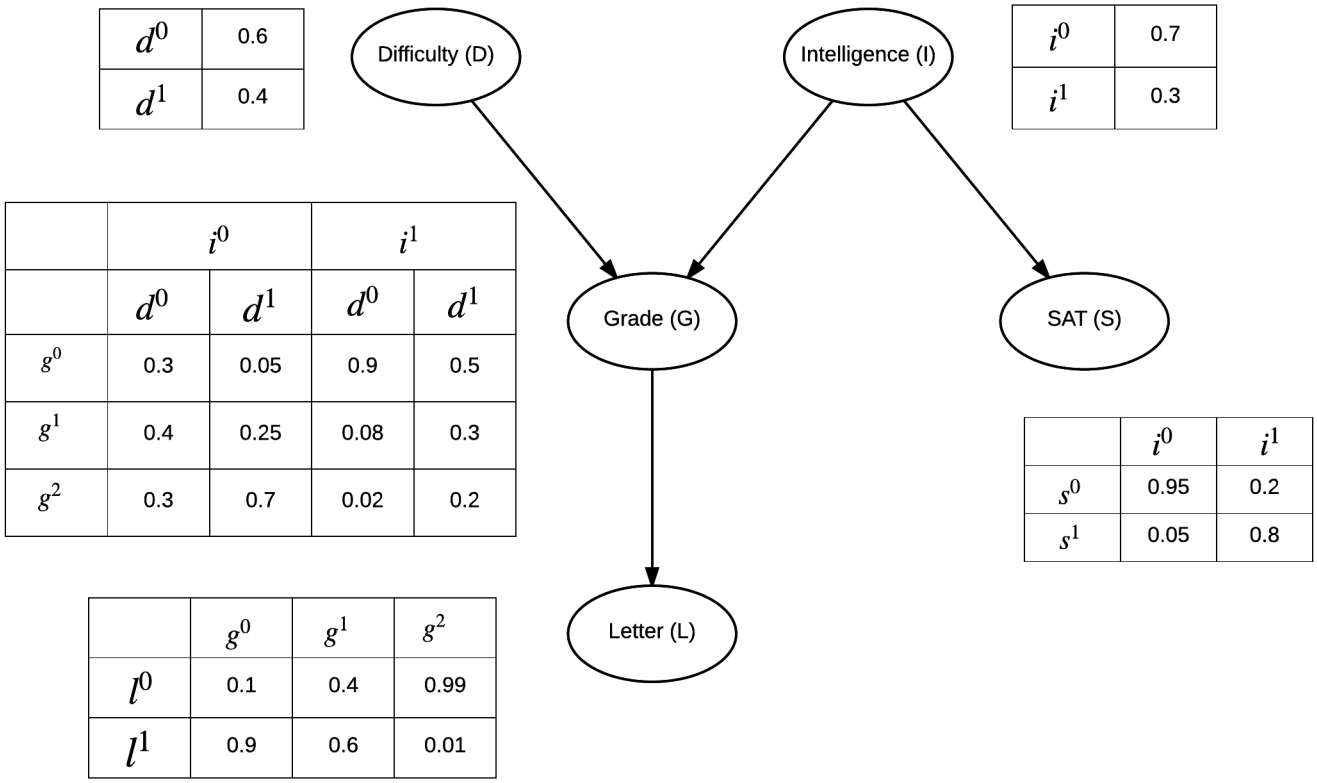

Fig. 1: *Student Model: A simple Bayesian Network.*

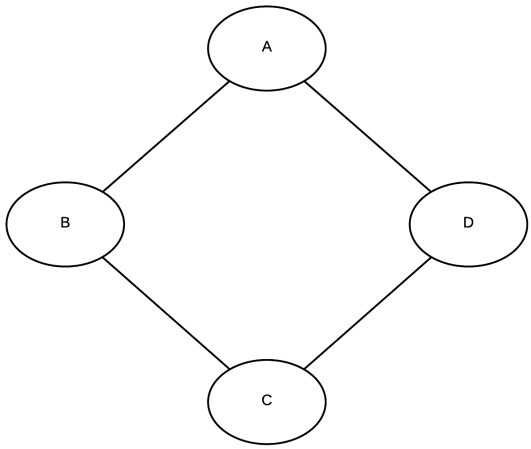

Fig. 2: *A simple Markov Model*

Intelligence (I)	i^0	i^0	i^1	i^1
Difficulty (D)	d^0	d^1	d^0	d^1
g^0	0.3	0.05	0.9	0.5
g^1	0.4	0.25	0.08	0.3
g^2	0.3	0.7	0.02	0.2

TABLE 1: *Conditional Probability Table.*

A	**B**	$\phi(A,B)$
a^0	b^0	30
a^0	b^1	5
a^1	b^0	1
a^1	b^1	10

TABLE 2: *Factor over variables A and B.*

```
                variable='G',
                variable_card=3,
                values=[[0.3, 0.05, 0.9, 0.5],
                        [0.4, 0.25, 0.08, 0.3],
                        [0.3, 0.7, 0.02, 0.2]],
                evidence=['I', 'D'],
                evidence_card=[2, 2])
```

Now, coming back to defining a model using pgmpy. The general workflow for defining a model in pgmpy is to first define the network structure and then add the parameters to it. We can create the student model shown in Fig 1 in pgmpy as follows:

```
from pgmpy.models import BayesianModel
from pgmpy.factors import TabularCPD
student_model = BayesianModel([('D', 'G'),
                               ('I', 'G'),
                               ('G', 'L'),
                               ('I', 'S')])
grade_cpd = TabularCPD(
                variable='G',
                variable_card=3,
                values=[[0.3, 0.05, 0.9, 0.5],
                        [0.4, 0.25, 0.08, 0.3],
                        [0.3, 0.7, 0.02, 0.2]],
                evidence=['I', 'D'],
                evidence_card=[2, 2])
difficulty_cpd = TabularCPD(
                variable='D',
                variable_card=2,
                values=[[0.6, 0.4]])
intel_cpd = TabularCPD(
                variable='I',
                variable_card=2,
                values=[[0.7, 0.3]])
letter_cpd = TabularCPD(
                variable='L',
                variable_card=2,
                values=[[0.1, 0.4, 0.99],
                        [0.9, 0.6, 0.01]],
                evidence=['G'],
                evidence_card=[3])
sat_cpd = TabularCPD(
                variable='S',
                variable_card=2,
                values=[[0.95, 0.2],
                        [0.05, 0.8]],
                evidence=['I'],
                evidence_card=[2])
student_model.add_cpds(grade_cpd, difficulty_cpd,
                       intel_cpd, letter_cpd,
                       sat_cpd)
```

The network structure of a Graphical Model encodes the independence conditions between the random variables. pgmpy also has methods to determine the local independencies, D-Separation, converting to a markov model etc. A few example are shown below:

```
student_model.get_cpds()
[<TabularCPD representing P(G:3 | I:2, D:2)
                              at 0x7f196c0b27b8>,
 <TabularCPD representing P(D:2) at 0x7f196c0b2828>,
 <TabularCPD representing P(I:2) at 0x7f196c0b2908>,
 <TabularCPD representing P(L:2 | G:3)
                              at 0x7f196c0b2978>,
 <TabularCPD representing P(S:2 | I:2)
                              at 0x7f196c0b27f0>]

student_model.active_trail_nodes('D')
{'D', 'G', 'L'}

student_model.local_independencies('G')
(G _|_ S | D, I)

student_model.get_independencies()
(S _|_ I, G, L | D)
(S _|_ D, I | G)
(S _|_ D, I, G | L)
(D _|_ G, L | S)
(D _|_ I, S | G)
(D _|_ G, L | I)
(D _|_ G, I, S | L)
(G _|_ D, I, L | S)
(G _|_ I, L, S | D)
(G _|_ D, L | I)
(G _|_ D, I, S | L)
(I _|_ G, L | S)
(I _|_ G, S, L | D)
(I _|_ D, S | G)
(I _|_ D, G, S | L)
(L _|_ D, G, I | S)
(L _|_ G, I, S | D)
(L _|_ D, G | I)

student_model.to_markov_model()
<pgmpy.models.MarkovModel.MarkovModel
                              at 0x7f196c0b2470>
```

Creating Markov Models in pgmpy

A Markov Network consists of an undirected graph which connects the random variables according to the relation between them. A markov network is parameterized by factors which represent the likelihood of a state of one variable to agree with some state of other variable.

We can take the example of a Factor over variables A and B in the network shown in Fig 2. A possible Factor over variables A and B is shown in Table 2.

We can represent this Factor in pgmpy as follows:

```
from pgmpy.factors import Factor
phi_a_b = Factor(varibales=['A', 'B'],
                 cardinality=[2, 2],
                 value=[100, 5, 5, 100])
```

Assuming some other possible factors as in Table 3, 4 and 5, we can define the complete markov model as:

```
from pgmpy.models import MarkovModel
from pgmpy.factors import Factor
model = MarkovModel([('A', 'B'), ('B', 'C'),
                     ('C', 'D'), ('D', 'A')])
factor_a_b = Factor(variables=['A', 'B'],
```

C	$\phi(B,C)$	
b^0	c^0	100
b^0	c^1	1
b^1	c^0	1
b^1	c^1	100

TABLE 3: *Factor over variables B and C.*

C	D	$\phi(C,D)$
c^0	d^0	1
c^0	d^1	100
c^1	d^0	100
c^1	d^1	1

TABLE 4: *Factor over variables C and D.*

```
                    cardinality=[2, 2],
                    value=[100, 5, 5, 100])
factor_b_c = Factor(variables=['B', 'C'],
                    cardinaity=[2, 2],
                    value=[100, 3, 2, 4])
factor_c_d = Factor(variables=['C', 'D'],
                    cardinality=[2, 2],
                    value=[3, 5, 1, 6])
factor_d_a = Factor(variables=['D', 'A'],
                    cardinality=[2, 2],
                    value=[6, 2, 56, 2])
model.add_factors(factor_a_b, factor_b_c,
                  factor_c_d, factor_d_a)
```

Similar to Bayesian Networks, pgmpy also has the feature for computing independencies, converting to Bayesian Network etc in the case of Markov Networks.

```
model.get_local_independencies()
(D _|_ B | C, A)
(C _|_ A | D, B)
(A _|_ C | D, B)
(B _|_ D | C, A)

model.to_bayesian_model()
<pgmpy.models.BayesianModel.BayesianModel
                            at 0x7f196c084320>

model.get_partition_function()
10000
```

Doing Inference over models

pgmpy support various Exact and Approximate inference algorithms. Generally, to perform inference over models, we need to first create an inference object by passing the model to the inference class. Once an inference object is instantiated, we can call either query method to find the probability of some variable given evidence, or else map_query method to know the state of the variable having maximum probability. Let's perform inference on the student model (Fig 1) using variable elimination :

D	A	$\phi(D,A)$
d^0	a^0	100
d^0	a^1	1
d^1	a^0	1
d^1	a^1	100

TABLE 5: *Factor over variables D and A.*

```
from pgmpy.inference import VariableElimination
student_infer = VariableElimination(student_model)
prob_G = student_infer.query(variables='G')
print(prob_G['G'])
G       phi(G)
G_0     0.4470
G_1     0.2714
G_2     0.2816

prob_G = student_infer.query(
                variables='G',
                evidence=[('I', 1), ('D', 0)])
print(prob_G['G'])
G       phi(G)
G_0     0.0500
G_1     0.2500
G_2     0.7000

student_infer.map_query(variables='G')
{'G': 0}

student_infer.map_query(
                variables='G',
                evidence=[('I', 1), ('D', 0)])
{'G': 2}
```

Fit and Predict Methods

In a general machine learning task we are given some data from which we want to compute the parameters of the model. pgmpy simplifies working on these problems by providing fit and predict methods in the models. fit method accepts the given data as a pandas DataFrame object and learns all the parameters from it. The predict method also accepts a pandas DataFrame object and predicts values of all the missing variables using the model. An example of fit and predict over the student model using some randomly generated data:

```
from pgmpy.models import BayesianModel
import pandas as pd
import numpy as np

# Considering that each variable have only 2 states,
# we can generate some random data.
raw_data = np.random.randint(low=0,
                             high=2,
                             size=(1000, 5))
data = pd.DataFrame(raw_data,
                    columns=['D', 'I', 'G',
                             'L', 'S'])
data_train = data[: int(data.shape[0] * 0.75)]

student_model = BayesianModel([('D', 'G'),
                               ('I', 'G'),
                               ('I', 'S'),
                               ('G', 'L')])
student_model.fit(data_train)
student_model.get_cpds()
[<TabularCPD representing P(C:2) at 0x7f195ee5e400>,
 <TabularCPD representing P(A:2) at 0x7f195ee5e518>,
 <TabularCPD representing P(D:2) at 0x7f195ee5e2b0>,
 <TabularCPD representing P(F:2) at 0x7f195ee5e320>,
 <TabularCPD representing P(P:2 | F:2, A:2, L:2)
                                 at 0x7f195ed620f0>,
 <TabularCPD representing P(L:2 | C:2, D:2)
                                 at 0x7f195ed62048>]

data_test = data[0.75 * data.shape[0] : data.shape[0]]
```

```
data_test.drop('P', axis=1, inplace=True)
student_model.predict(data_test)
     P
750  0
751  0
752  1
753  0
..  ..
996  0
997  0
998  0
999  0

[250 rows x 1 columns]
```

Extending pgmpy

One of the main features of pgmpy is its extensibility. It has been built in a way so that new algorithms can be directly written without needing to get familiar with the code base.

For example, for writing any new inference algorithm we can simply inherit the Inference class. Inheriting this base inference class exposes three variables to the class: self.variables, self.cardinalities and self.factors; using these variables we can write our own inference algorithm. An example is shown:

```
from pgmpy.inference import Inference
class MyNewInferenceAlgo(Inference):
    def print_variables(self):
        print('variables: ', self.variables)
        print('cardinality: ', self.cardinalities)
        print('factors: ', self.factors)

infer = MyNewInferenceAlgo(
            student_model).print_variables()
variables: ['S', 'D', 'G', 'I', 'L']
cardianlity: {'D': 2, 'G': 3, 'I': 2,
              'S': 2, 'L': 2}
factors: defaultdict(<class 'list'>,
{'D': [<Factor representing phi(D:2)
                    at 0x7f195ed61c18>,
       <Factor representing phi(G:3, D:2, I:2)
                    at 0x7f195ed61cf8>],
 'I': [<Factor representing phi(S:2, I:2)
                    at 0x7f195ed61a58>,
       <Factor representing phi(G:3, D:2, I:2)
                    at 0x7f195ed61cf8>,
       <Factor representing phi(I:2)
                    at 0x7f195ed61e10>],
 'G': [<Factor representing phi(G:3, D:2, I:2)
                    at 0x7f195ed61cf8>,
       <Factor representing phi(L:2, G:3)
                    at 0x7f195ed61e48>],
 'S': [<Factor representing phi(S:2, I:2)
                    at 0x7f195ed61a58>],
 'L': [<Factor representing phi(L:2, G:3)
                    at 0x7f195ed61e48>]})
```

Similarly, for adding any new variable elimination order algorithm we can simply inherit from BaseEliminationOrder and define a method named cost(self, variable) which returns the cost of eliminating that variable. Inheriting this class also exposes two variables: self.bayesian_model and self.moralized_graph. We can then call the get_elimination_order method to get the elimination order. Below is an example for returning an elimination order in which the variables are sorted alphabetically.

```
from pgmpy.inference import BaseEliminationOrder
class MyEliminationAlgo(EliminationOrder):
    def cost(self, variable):
        return variable

order = MyEliminationAlgo(
            student_model).get_elimination_order()
['D', 'G', 'I', 'L', 'S']
```

Comparing pgmpy to other libraries

Starting with defining the model, pgmpy provides a very simple to use API. A model can be instantiated simply by using the __init__ method and the structure can be modified using add_node, add_edge etc methods. After the model is created, we can simply add the CPDs using the add_cpds method. In the case of eBay's bayesian belief network, we have to create a separate function for each CPD. And each of these function has a dict of CPD values and logic to return the value when the states are passed as arguments [example_bbn]. Similarly in case of libpgm we have the option to read the data from files defined in a specific format [example_libpgm] but doesn't provide any methods for making changes to the network. For changing the structure we will need to modify the internal variables storing the network information. We have tried to keep pgmpy as modular as possible. We can take the example of creating a model. We define a network structure and separately define different CPDs and then simply associate the CPDs to the structure. At any time we can modify these CPDs, unassociate or associate another CPD to the network.

Other than providing the features to easily create models, pgmpy also supports 4 standard file formats: pomdpX [pomdpX], ProbModelXML [ProbModel], XMLBeliefNetwork [XMLBelief] and XMLBIF [XMLBIF]. Using pgmpy we can read as well as write networks in these formats. Also there's an ongoing GSoC project for adding support for more file formats so hopefully we will be having support for many more formats soon.

There are many more benefits of using networkx to represent the graph structure. For example we can directly run various graph related algorihtms implemented in networkX on our networks. Also we can use networkX's plotting functionality to visualize our networks.

pgmpy also implements methods for getting independencies, D-Separation etc which would help a lot to people who are still new to Graphical Models. These features are not available in most of the other libraries.

We have tried to keep pgmpy as uniform as possible. For example we have fit and predict methods with each of the models which can automatically learn the parameters and structure and you can control the learning by simply passing arguments to these methods. Whereas in the case of libpgm, it has multiple methods for learning like lg_mle_estimateparams, lg_constraint_estimatesstruct, discrete_estimatebn etc. Similarly for each inference algorithm pgmpy prodives query and map_query methods.

Another area in which pgmpy excels is its extensibility. As we have discussed earlier, we can easily add new algorithms to pgmpy without even getting familiar with the code base. We have to tried to build pgmpy in such a way that new components can be easily added which will really help researchers working on new ideas to quickly prototype. Also, since pgmpy is documented very well it is very easy to understand the code base.

Performance wise pgmpy is a bit slower than a few libraries but we are currently actively working on improving the performance so hopefully we will be seeing a major improvement in the coming months.

Conclusion and future work

The pgmpy library provides an easy to use API for working with Graphical Models. It is also modular enough to provide separate classes for most commonly used graphical models like Naive Bayes, Hidden Markov Model etc. so that the user can directly use these special cases instead of contructing them from the base models. For machine learning problems the fit method can be used to learn parameters and predict can be used to predict values for newer data points. pgmpy's easy extensibility allows users to quickly prototype and test their ideas.

pgmpy is in a state of rapid development and some soon to come features are:

- Sampling Algorithms
- Dynamic Bayesian Networks
- Hidden Markov Models
- Support for more file formats
- Structure Learning

REFERENCES

[pgmpy] pgmpy github page https://github.com/pgmpy/pgmpy

[student] Koller, D.; Friedman, N. Probabilistic Graphical Models. Massachusetts: MIT Press, 2009, pp. 103-106.

[markov] Koller, D.; Friedman, N. Probabilistic Graphical Models. Massachusetts: MIT Press, 2009, pp. 53-54.

[bbn] bayesian-belief-networks github page https://github.com/eBay/bayesian-belief-networks

[pymc] pymc home page https://pymc-devs.github.io/pymc/

[libpgm] libpgm github page https://github.com/CyberPoint/libpgm

[pomdpX] http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/index.php?n=Main.PomdpXDocumentation

[ProbModel] http://www.probmodelxml.org/

[XMLBelief] http://xml.coverpages.org/xbn-MSdefault19990414.html

[XMLBIF] http://www.cs.cmu.edu/~fgcozman/Research/InterchangeFormat/

[example_bbn] bayesian belief network examples for creating models https://github.com/eBay/bayesian-belief-networks/tree/master/bayesian/examples/bbns

[example_libpgm] https://github.com/CyberPoint/libpgm/tree/master/examples