# pgmpy: Support for Continuous Random Variables, GSoC 2016 Application

## Sub Org Information

pgmpy - Python Library for Probabilistic Graphical Models

## Personal Information

**Name** : Yashu Seth

**Github Username** : yashu-seth

**Email** : yashuseth2503@gmail.com

**University** : IIT (BHU) Varanasi

**Major** : Electronics Engineering

**Time Zone** : IST (UTC +5:30)

**Telephone** : +91-8418084769

**GSOC Blog** : Yashu Seth - Blog

## Contributions to pgmpy

I began contributing to pgmpy in September 2015. Since then I have been an active contributor in the community. Here is a list of my Pull Requests -

Merged PRs

- Naive Bayes Model #521
- Added get_cardinality method #491 #500
- Fixed check_model method, added tests for it #498 #487 #621
- Added tests for get_factor methods in models #502
- Fixed bug in BayesianModel.fit() #541

- Fixed bug in Independencies #543
- Added TypeError for string parameters in Factor #479
- Updated README.md and Authors.rst #592 #589

Unmerged PRs

- Variable State Name Feature #648 (I am still working on this.)
- Creates DataFrame object to remove pandas dependency #596

You can have a look at the issues opened by me here.

## Project Information

### Project Title :

**pgmpy: Support For Continuous Random Variables**

### Abstract:

Currently, pgmpy deals with only discrete random variables. In many situations, some variables are best modeled as taking values in some continuous space. Examples include variables such as position, velocity etc.

The first part of the project creates a module to represent nodes having a continuous domain of representation. These nodes would be used in hybrid networks comprising both continuous as well as discrete random variables. The two important features in this part would be -

- Representation of User Defined Continuous Random Variables
- Methods to discretize the continuous distributions.

The second part of the project will deal with Gaussian distributions. Gaussians are a particularly simple subclass of distributions that make very strong assumptions, such as the exponential decay of the distribution away from its mean, and the linearity of interactions between variables.Gaussians are a surprisingly good approximation for many real world distributions.

There will be support for variables comprising the most popular forms of representation in Gaussian distributions.

- Linear Gaussian Distribution
- Joint Gaussian Distribution
- Canonical Forms

# Implementation Details

## Continuous Node Representation

The **scipy.stats.rv_continuous** module provides an efficient way to represent continuous random variables**.** It is a generic class meant for subclassing. I will implement a base class that will inherit the scipy.stats.rv_continuous class.

Now, any type of continuous random variable node in pgmpy shall inherit this base class and will have its own density functions. This will allow user defined random variables that can be defined by providing a function (that defines its pdf) as input. It will also allow representation of different standard continuous distributions that can be added later in pgmpy. The base class will also have have a method that would discretize the continuous random variable into discrete factors.

```python
from scipy.stats import rv_continuous

class ContinuousNode(rv_continuous):
        """
        Base class for continuous node representation.
        It is a subclass of `scipy.stats.rv_continuous`.
        It has an extra method to discretize the continuous node into
        a discrete factor.
        """
        def __init__(self, pdf, lb=None, ub=None, name=None):
                """
                Parameters
                ----------
                lb : float, optional
                        Lower bound of the support of the distribution, default is minus
                        infinity.
                ub : float, optional
                        Upper bound of the support of the distribution, default is plus
                        infinity.
                name : str, optional
                        The name of the instance. This string is used to construct the default
                        example for distributions.
                pdf: function
                        The user defined probability density function.

                Examples
                --------
                >>> from pgmpy.factors import ContinuousNode
                >>> custom_pdf = lambda x: 0.5 if x>-1 and x<1 else 0
                >>> node = ContinuousNode(custom_pdf, -3, 3)
                """

                self.pdf = pdf
                super(ContinuousNode, self.__init__(momtype=0, a=lb, b=ub, name=name))
```

```
    def _pdf(self, *args):
            """
            Defines the probability density function of the given continuous variable.
            """

            return self.pdf(*args)
```

## Discretization

Computes a discrete probability mass function from using the continuous cumulative distribution function (cdf) with various methods. The various methods are -

### Method of Rounding (Mass Dispersal)

Let $f_j$ denote the probability placed at $jh$, $j = 0, 1, 2, \ldots$ Then,

$$\begin{aligned} f_0 &= \Pr\left(X < \frac{h}{2}\right) = F_X\left(\frac{h}{2} - 0\right), \\ f_j &= \Pr\left(jh - \frac{h}{2} \le X < jh + \frac{h}{2}\right) \\ &= F_X\left(jh + \frac{h}{2} - 0\right) - F_X\left(jh - \frac{h}{2} - 0\right), \quad j = 1, 2, \ldots . \end{aligned}$$

This method concentrates all the probability one-half span on either side of $jh$ and places it at $jh$. There is an exception for the probability assigned to zero. This, in effect, rounds all amounts to the nearest convenient monetary unit, $h$, the span of the distribution. When the continuous distribution is unbounded, it is reasonable to halt the discretization process at some point once most all the probability has been accounted for. If the index for this last point is $m$, then $f_m = 1 - F_X[(m - 0.5)h - 0]$. With this method the discrete probabilities are never negative and sum to 1, ensuring that the resulting distribution is legitimate.

### Method of local moment matching

In this method we construct an arithmetic distribution that matches $p$ moments of the arithmetic and the true severity distributions. Consider an arbitrary interval of length $ph$, denoted by $[x_k, x_k + ph)$. We locate point masses $m^k_0, m^k_1, \ldots, m^k_p$ at points $x_k, x_k + h, \ldots, x_k + ph$ so that the first $p$ moments are preserved. The system of $p + 1$ equations reflecting these conditions is -

$$\sum_{j=0}^{p} (x_k + jh)^r m_j^k = \int_{x_k - 0}^{x_k + ph - 0} x^r \, dF_X(x), \quad r = 0, 1, 2, \ldots, p,$$

(Eq. 1.1)

Arrange the intervals so that $x_{k+1} = x_k + ph$ and so the endpoints coincide. Then the point masses at the endpoints are added together. With $x_0 = 0$, the resulting discrete distribution has successive probabilities:

$$f_0 = m_0^0, \quad f_1 = m_1^0, \quad f_2 = m_2^0, \ldots,$$
$$f_p = m_p^0 + m_0^1, \quad f_{p+1} = m_1^1, \quad f_{p+2} = m_2^1, \ldots.$$

(Eq. 1.2)

By summing (1.1) for all possible values of *k*, with $x_0 = 0$, it is clear that the first *p* moments are preserved for the entire distribution and that the probabilities add to 1 exactly. It only remains to solve the system of equations (1.1). The solution of (1.1) is -

$$m_j^k = \int_{x_k-0}^{x_k+ph-0} \prod_{i \neq j} \frac{x - x_k - ih}{(j-i)h} dF_X(x), \quad j = 0, 1, \ldots, p.$$

(For proof refer Klugman, S. A., Panjer, H. H. and Willmot, G. E. (2008), Loss Models, From Data to Decisions, Second Edition, Wiley, Theorem 6.20)

The method of local moment matching with *k* = 1 (matching total probability and the mean) using (1.1) and (1.2) results in -

$$f_0 = 1 - \frac{\mathrm{E}[X \wedge h]}{h}$$
$$f_{i+1} = \frac{2\mathrm{E}[X \wedge ih] - \mathrm{E}[X \wedge (i-1)h] - \mathrm{E}[X \wedge (i+1)h]}{h}, \quad i = 0, 1, 2, \ldots,$$

and that $\{f_i\, ; i = 0, 1, 2, \ldots\}$ forms a valid distribution with the same mean as the original distribution.
(For details refer Loss Models, Second Edition, Exercise 6.36)

## User Defined Methods

The discretize method shall also allow users to define their own discretization algorithm.
For example let us take an example of a user defined method that assigns to point **x = from, from + step, ..., to - step** the probability mass **F(x + step) - F(x)** where from and to are the lower and upper bounds respectively and step is the span size. The way this method will be used has been illustrated in the API documentation below.

```
def discretize(self, from, to, step, method_type=['rounding']):
        """
        Parameters
        ----------
        from: int, float
                lower bound
        to: int, float
                upper bound
        step: int, float
                the discretization step (or span, or lag).
        method_type: List
                A list describing the type of method to use. Default is the rounding
                method type. The first element will be a string that can take the
                following values -
                'rounding' : string for mass dispersal method type
                'unbiased' : string for matching of the first moment method
                custom_function   :  function for user defined method
```

```
            The rest of the elements of this list will depend on the corresponding
            method type. For example, the unbiased type will have a function to compute
            the limited expected value of the distribution corresponding to the cdf.
            For user defined methods the users will pass a function that will compute the
            discretized factor. The function will have the following format -
            custom_fun(cdf, from, to, step, *args)


        Examples
        --------

        # Illustration for a user defined method to discretize.

        >>> def custom_discretize(cdf, from, to, step, *args):
                    fact = []
                    i = from
                    while(i+step<to):
                           fact.append(cdf(i+step)-cdf(i))
                           i+=2*step
                    return fact

        >>> from pgmpy.factors import ContinuousNode
        >>> custom_pdf = lambda x: 0.125 if x>-4 and x<4 else 0
        >>> node = ContinuousNode(custom_pdf, -3, 3)

        >>> node.discretize(from=-4, to=4, step=1, method_type=[custom_discretize])
        [0.125, 0.125, 0.125]
        """
        pass
```

*Note : If I get time I will look into more discretizing methods and implement them during the summer. Else I will continue my work after GSoC.*

## Linear Gaussian Distribution

The Linear Gaussian model is a very useful approximation in many practical applications.
If $X \to Y$ , assume that the mean of $Y$ is a linear function of $X$ and that the variance of $Y$ does not depend on $X$.

For example,

$$p(Y|X) = N(-2x + 0.9;\ 1)$$

Let $Y$ be a continuous variable with continuous parents $X_1$ .......... $X_k$. We say that $Y$ has a linear Gaussian model if there are parameters $\beta_{0,}$..............$\beta_k$ and $\sigma^2$ such that

$$p(Y|x_1..........x_k) = N(\beta_0 + \beta_1 + ............ + \beta_k\ ;\ \sigma^2)$$

In vector notation,

$$p(Y|x) = N(\beta_0 + \beta^T x\ ;\ \sigma^2)$$

```
class LinearGaussianCPD(object):
    def __init__(self, variable, beta_not, variance, parents=None, beta_vector=None):
        """
        Parameters
        ----------

        variable: any hashable python object
               The variable whose CPD is defined.
        beta_not: int, float
               Represents the constant term in the linear equation
        variance: int, float
               The variance of the variable defined.
        parents: iterable of any hashable python object
               An iterable of the parents of the variable. None
               if there are no parents.
        beta_vector: iterable of int or float
               An iterable representing the coefficient vector of the linear equation.

        Examples
        --------

        # For $P(Y|x_1,x_2,x_3) = N(-2x_1 + 3x_2 + 7x_3 + 0.2; 9.6)$

        >>> cpd = LinearGaussianCPD('Y', 0.2, 9.6, ['x1', 'x2', 'x3'], [-2, 3, 7])
        >>> cpd.variable
        'Y'
        >>> cpd.variance
        9.6
        >>> cpd.parents
        ['x1', 'x2', 'x3']
        >>> cpd.beta_vector
        [-2, 3, 7]
        >>> cpd.beta_not
        0.2

        """
        pass
```

The Linear Gaussian CPDs can be converted into Joint Gaussian Distribution (an alternative form of representation of the Gaussian Bayesian Networks).

Let $Y$ be a linear Gaussian of its parents $X_1 \ldots\ldots X_k$.:
$p(Y|x) = N(\beta_0 + \beta^T x\,;\,\sigma^2)$

Assume that $X_1 \ldots\ldots X_k$ are jointly Gaussian with distribution $N(\mu\,;\,\Sigma)$ . Then:

- The distribution of $Y$ is a normal distribution $p(Y) = N(\mu_Y\,;\,\sigma^2{}_Y)$

where:

$$\begin{aligned} \mu_Y &= \beta_0 + \beta^T \mu \\ \sigma_Y^2 &= \sigma^2 + \beta^T \Sigma \beta. \end{aligned}$$

- The joint distribution over $\{X, Y\}$ is a normal distribution where:

$$Cov[X_i; Y] = \sum_{j=1}^{k} \beta_j \Sigma_{i,j}.$$

```
def to_joint_gaussian_distribution(*cpds):
       """
       Example
       -------

       # For network X1 -> X2 -> x3

       >>> cpd1 = LinearGaussianCPD('x1', 1, 4)
       >>> cpd2 = LinearGaussianCPD('x2', -5, 4, ['x1'], [0.5])
       >>> cpd3 = LinearGaussianCPD('x3', 4, 3, ['x2'], [-1])
       >>> dis = to_joint_gaussian_distribution(cpd1, cpd2, cpd3)
       >>> dis.mean_vector
       matrix([[ 1],
               [-3],
               [ 4]])
       >>> dis.covariance_matrix
       matrix([[ 4, 2, -2],
                [ 2, 5, -5],
                [-2, -5, 8]])
       """
       Pass
```

## Positive Definite Matrix

A matrix is said to be a positive definite matrix if for any $x \in IR^n$ such that $x \neq 0,$ we have that $x^T Mx > 0$, where M is the given matrix.

```
def check_positive_definite(matrix):
    """
    Checks if the given matrix is a positive definite matrix or not.
    A matrix is said to be positive definite if for any x belonging to
    IR^n such that x != 0, we have that x.T * M * x > 0.

    Parameters
    ----------
    matrix: numpy matrix
        The matrix that is to be checked for positive definite property.
    """
    pass
```

## Joint Gaussian Distribution

The univariate Gaussian is defined in terms of two parameters: a mean and a variance.
In its most common representation, a multivariate Gaussian distribution over $X_1 .......... X_n$ is characterized by an n-dimensional mean vector $\mu$, and a symmetric $nXn$ covariance matrix $\Sigma$;
the density function is most often defined as:

$$p(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right]$$

where $|\Sigma|$ is the determinant of $\Sigma$.

```
class JointGaussianDistribution(object):
    def __init__(self, variables, mean_vector, covariance_matrix):
       """
       Parameters
       ----------
       variables: iterable of any hashable python object
              The variables for which the distribution is defined.
       mean_vector: nX1 numpy matrix
              n-dimensional vector where n is the number of variables.
       covariance_matrix: nXn numpy matrix
              nXn dimensional matrix where n is the number of variables.

       Examples
       --------
       >>> from pgmpy.factors import JointGaussianDistribution as JGD
       >>> dis = JGD(['x1', 'x2', 'x3'], np.array([[1], [-3], [4]]),
                      np.array([[4, 2, -2], [2, 5, -5], [-2, -5, 8]]))
       >>> dis.variables
       ['x1', 'x2', 'x3']
       >>> dis.mean_vector
       np.matrix([[ 1],
                      [-3],
                      [4]]))
       >>> dis.covariance_matrix
       np.matrix([[4, 2, -2],
                  [2, 5, -5],
                  [-2, -5, 8]])
       """
        pass
```

Marginalization is trivial to perform in the covariance form. Specifically, the marginal Gaussian distribution over any subset of the variables can simply be read from the mean and covariance matrix. Assume that we have a joint normal distribution over {X,Y } where

$X \in IR^n$ and $Y \in IR^m$. Then we can decompose the mean and covariance of this joint distribution as follows.

$$p(\boldsymbol{X}, \boldsymbol{Y}) = \mathcal{N}\left(\begin{pmatrix} \mu_X \\ \mu_Y \end{pmatrix}; \begin{bmatrix} \Sigma_{XX} & \Sigma_{XY} \\ \Sigma_{YX} & \Sigma_{YY} \end{bmatrix}\right)$$

```
    def marginalize(self, variables, inplace=True):
            """
            Parameters
            ----------

            variables: iterator
                    List of variables over which to marginalize.
            inplace: boolean
                    If inplace=True it will modify the distribution itself, else would
                    return a new distribution.

            Examples
            --------
            >>> from pgmpy.factors import JointGaussianDistribution as JGD
            >>> dis = JGD(['x1', 'x2', 'x3'], np.array([[1], [-3], [4]])),
                            np.array([[4, 2, -2], [2, 5, -5], [-2, -5, 8]]))
            >>> dis.marginalize(['x3'])
            dis.variables
            ['x1', 'x2', 'x3']
            >>> dis.mean_vector
            matrix([[ 1],
                     [-3]]))
            >>> dis.covariance_matrix
            np.matrix([[4, 2],
                         [2, 5]])
            """
                    pass
```

The Joint Gaussian distributions can be converted into equivalent canonical forms (discussed later) using the following equations -
$N(\mu; \Sigma) = C(k, h, g)$ where:

$$\begin{aligned} K &= \Sigma^{-1} \\ \boldsymbol{h} &= \Sigma^{-1}\boldsymbol{\mu} \\ g &= -\frac{1}{2}\boldsymbol{\mu}^T\Sigma^{-1}\boldsymbol{\mu} - \log\left((2\pi)^{n/2}|\Sigma|^{1/2}\right). \end{aligned}$$

```
    def to_canonical_factor(self):
            """
            Example
            -------

            >>> from pgmpy.factors import JointGaussianDistribution as JGD
            >>> dis = JGD(['x1', 'x2', 'x3'], np.array([[1], [-3], [4]])),
                            np.array([[4, 2, -2], [2, 5, -5], [-2, -5, 8]]))
            >>> phi = dis.to_canonical_factor()
```

```
        >>> phi.variables
        ['x1', 'x2', 'x3']
        >>> phi.K
        matrix([[0.3125, -0.125, 0.],
                [-0.125, 0.5833, 0.333],
                [     0., 0.333, 0.333]])
        >>> phi.h
        matrix([[  0.6875],
                [-0.54166],
                [ 0.33333]]))
        >>> phi.g
        -6.51533
        """
         pass
```

In order for the joint density function for this multivariate case to induce a well defined density that integrates to 1, The covariance matrix, $\Sigma$ must be positive definite.

```
    def check_distribution(self):
        """
        Checks if the joint density function for the multivariate case induces
        a well defined density that integrates to 1 or not. For this to induce a valid
        density the covariance matrix must be positive definite.

        Returns
        -------
        True if the given distribution has a valid density else False.
        """
        return check_positive_definite(self.covariance_matrix)
```

## Canonical Factors

For inference in continuous networks, we can use any representation for the CPDs or factors that can behave in the same way as the factors behave in the case of discrete variable networks for the sum-product or message passing algorithms.

But in case of Gaussian networks many difficulties in dealing with continuous variables disappear. In particular, the intermediate factors in a Gaussian network can be described compactly using a simple parametric representation called the canonical form. This representation is closed under the basic operations used in inference: factor product, factor division, factor reduction, and marginalization. Thus, we can define a set of simple data structures that allow the inference process to be performed.

As a consequence, a fairly straightforward modification of the discrete sum-product algorithms (whether variable elimination or clique tree) gives rise to an exact inference algorithm for Gaussian networks.

The key difference between inference in the continuous and the discrete case is that the factors can no longer be represented as tables. Thus we use Canonical Factors which are a more general representation for factors. They can accommodate both Gaussian distributions as well as any combination of these models that might arise during the course of inference.

A canonical form C (X; K,h, g) is defined as:

$$\mathcal{C}(\boldsymbol{X}; K, \boldsymbol{h}, g) = \exp\left(-\frac{1}{2}\boldsymbol{X}^T K \boldsymbol{X} + \boldsymbol{h}^T \boldsymbol{X} + g\right).$$

We can represent every Gaussian as a canonical form. Rewriting the above equation, we obtain:

$$\begin{aligned} & \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right) \\ = \; & \exp\left(-\frac{1}{2}\boldsymbol{x}^T \Sigma^{-1}\boldsymbol{x} + \boldsymbol{\mu}^T \Sigma^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}^T \Sigma^{-1}\boldsymbol{\mu} - \log\left((2\pi)^{n/2}|\Sigma|^{1/2}\right)\right). \end{aligned}$$

Thus,

N (μ ; Σ) = C (K,h,g) where:

$$\begin{aligned} K &= \Sigma^{-1} \\ \boldsymbol{h} &= \Sigma^{-1}\boldsymbol{\mu} \\ g &= -\frac{1}{2}\boldsymbol{\mu}^T \Sigma^{-1}\boldsymbol{\mu} - \log\left((2\pi)^{n/2}|\Sigma|^{1/2}\right). \end{aligned}$$

```
class CanonicalFactor(object):
    def __init__(self, variables, K, h, g):
        """
        Parameters
        ----------
        variables : iterable of hashable python object
               variables for which the factor is defined
        K : nXn matrix, where n is the number of variables
        h : nX1 matrix
        g : int, float

        The terms K, h, and g have been defined in the derivation of canonical factors.

        Examples
        --------

        >>> phi = CanonicalFactor(['X', 'Y'], np.array([[1, -1], [-1, 1]]),
                                              np.array([[1], [-1]]), -3)
        >>> phi.variables
        ['X', 'Y']

        >>> phi.K
        matrix([[1, -1],
                [-1, 1]])

        >>> phi.h
        matrix([[1],
                [-1]])
        >>> phi.g
        -3
        """
        pass
```

## Multiply

The product of two canonical product form factors over the same scope X is simply:

$$\mathcal{C}(K_1, \boldsymbol{h}_1, g_1) \cdot \mathcal{C}(K_2, \boldsymbol{h}_2, g_2) = \mathcal{C}(K_1 + K_2, \boldsymbol{h}_1 + \boldsymbol{h}_2, g_1 + g_2).$$

When we have two canonical factors over different scopes X and Y , we simply extend the scope of both to make their scopes match and then perform the operation of the above equation. The extension of the scope is performed by simply adding zero entries to both the K matrices and the h vectors.

```
    def __mul__(self, other):
        """
        Example
        -------

        >>> phi1 = CanonicalFactor(['X', 'Y'], np.array([[1, -1], [-1, 1]]),
                                  np.array([[1], [-1]]), -3)
        >>> phi2 = CanonicalFactor(['X', 'Y'], np.array([[2, -2], [-2, 2]]),
                                  np.array([[2], [-1]]), -1)
        >>> phi3 = phi1 * phi2

        >>> phi3.variables
        ['X', 'Y']

        >>> phi3.K
        matrix([[3, -3],
               [-3, 3]])

        >>> phi3.h
        matrix([[3],
               [-2]])

        >>> phi3.g
        -4
        """
        pass
```

## Divide

The division of canonical forms (which is required for message passing in the belief propagation algorithm) is defined analogously:

$$\frac{\mathcal{C}(K_1, \boldsymbol{h}_1, g_1)}{\mathcal{C}(K_2, \boldsymbol{h}_2, g_2)} = \mathcal{C}(K_1 - K_2, \boldsymbol{h}_1 - \boldsymbol{h}_2, g_1 - g_2).$$

```
    def __truediv__(self, other):
        """
        Example
        -------

        >>> phi1 = CanonicalFactor(['X', 'Y'], np.array([[1, -1], [-1, 1]]),
                                            np.array([[1], [-1]]), -3)
        >>> phi2 = CanonicalFactor(['X', 'Y'], np.array([[2, -2], [-2, 2]]),
                                            np.array([[2], [-1]]), -1)
        >>> phi3 = phi1 / phi2

        >>> phi3.variables
        ['X', 'Y']

        >>> phi3.K
        matrix([[-1, 1],
                [1, -1]])

        >>> phi3.h
        matrix([[-1],
                [0]])

        >>> phi3.g
        -2
        """
       pass
```

**Marginalize**

The marginalization of this function onto the variables X is, as usual, the integral over the variables Y :

$$\int \mathcal{C}(\boldsymbol{X}, \boldsymbol{Y}; K, \boldsymbol{h}, g)\, d\boldsymbol{Y}.$$

We have to guarantee that all of the integrals resulting from marginalization operations are well defined. In the case of canonical forms, the integral is finite if and only if $K_{YY}$ is positive definite (defined before). Let $C(X, Y\ ;\ K, h, g)$ be some canonical form over $X, Y$ where,

$$K = \begin{bmatrix} K_{\boldsymbol{XX}} & K_{\boldsymbol{XY}} \\ K_{\boldsymbol{YX}} & K_{\boldsymbol{YY}} \end{bmatrix} \quad ; \quad h = \begin{pmatrix} h_{\boldsymbol{X}} \\ h_{\boldsymbol{Y}} \end{pmatrix}$$

(Eq. M1)

In this case, the result of the integration operation is a canonical form

$C(X\,;\,K',\,h',\,g)$ is
given by:

$$
\begin{aligned}
K' &= K_{XX} - K_{XY}K_{YY}^{-1}K_{YX} \\
h' &= h_X - K_{XY}K_{YY}^{-1}h_Y \\
g' &= g + \tfrac{1}{2}\left(|Y|\log(2\pi) - \log|K_{YY}| + h_Y^T K_{YY} h_Y\right).
\end{aligned}
$$

```
    def marginalize(self, variables, inplace=True):
        """
        Parameters
        ----------

        variables: iterator
            List of variables over which to marginalize.
        inplace: boolean
            If inplace=True it will modify the distribution itself, else would
            return a new distribution.

        Examples
        --------
        >>> from pgmpy.factors import CanonicalFactor
        >>> phi = CanonicalFactor(['X1', 'X2', 'X3'],
                                  np.array([[1, -1, 0], [-1, 4, -2], [0, -2, 4]]),
                                  np.array([[1], [4], [-1]]), -2)
        >>> phi.K
        matrix([[ 1, -1,  0],
                [-1,  4, -2],
                [ 0, -2,  4]])

        >>>phi.h
        matrix([[ 1],
                [ 4],
                [-1]])

        >>> phi.marginalize(['X3'])

        # X = ('X1', 'X2'), Y = ('X3')

        # K_XX = [[1, -1], [-1, 4]]
        # K_YY = [[4]]
        # K_XY = [[0], [-2]]
        # K_YX = [[0, -2]]
        # K' = K_XX - K_XY * K_YY.I * K_YX

        # h_X = [[1], [4]]
        # h_Y = [[-1]]
        # h' =  h_X - K_XY * K_YY.I * h_Y

        # g' = some confusion with the formula.

        >>> phi.K
        matrix([[ 1., -1.],
                [-1.,  3.]])

        >>> phi.h
```

```
            matrix([[ 1. ],
                    [ 3.5]])
            """
            pass
```

## Reduce

It is possible to reduce a canonical form to a context representing evidence. Assume that the canonical form C (X,Y ; K, h, g) is given by Eq. M1 defined in the marginalize section. Then setting Y = y
results in the canonical form,

C(X; K', h', g') given by,

$$\begin{aligned} K' &= K_{XX} \\ h' &= h_X - K_{XY}y \\ g' &= g + h_Y^T y - \frac{1}{2} y^T K_{YY} y. \end{aligned}$$

```
    def reduce(self, values, inplace=True):
            """
            Reduces the factor to the context of the given variable values.

            Parameters
            ----------
            values: Iterable
            An iterable of tuples of the form (variable_name, variable_value).

            inplace: boolean
            If inplace=True it will modify the factor itself, else would return
            a new factor.

            Returns
            -------
            Factor or None: if inplace=True (default) returns None
                           if inplace=False returns a new CanonicalFactor instance.

            Examples
            --------
            >>> from pgmpy.factors import CanonicalFactor
            >>> phi = CanonicalFactor(['X1', 'X2', 'X3'],
                                np.array([[1, -1, 0], [-1, 4, -2], [0, -2, 4]]),
                                       np.array([[1], [4], [-1]]), -2)
             >>> phi.variables
             ['X1', 'X2', 'X3']

             >>> phi.K
            matrix([[ 1., -1.],
                    [-1.,  3.]])

             >>> phi.h
            matrix([[ 1. ],
                    [ 3.5]])
```

```
        >>> phi.g
        -2

        >>> phi.reduce([('X3', 0.25)])


        >>> phi.variables
        ['X1', 'X2']

        # X = ('X1', 'X2'), Y = ('X3')

        # y = [[0.25]]
        # K_XX = [[1, -1], [-1, 4]]
        # K_YY = [[4]]
        # K_XY = [[0], [-2]]
        # K_YX = [[0, -2]]
        # K' = K_XX

        # h_X = [[1], [4]]
        # h_Y = [[-1]]
        # h' =  h_X - K_XY * y

        # g' = g + h_Y.T * y - 0.5 * y.T * K_YY * y

        >>> phi.K
        matrix([[ 1, -1],
                [-1,  4]])

        >>> phi.h
        matrix([[ 1. ],
                [ 4.5]])

        >>> phi.g
        matrix([[-2.375]])
        """
        pass
```

All of the factor operations can be done in time that is polynomial in the scope of the factor. In particular, the product or division of factors requires quadratic time; factor marginalization, which requires matrix inversion, can be done naively in cubic time.

## Timeline

### Community Bonding Period

I have been an active contributor of the community since September 2015. Hence, most of the time in this period shall be spent on studying the relevant algorithms in detail and strengthening my concepts in PGM. I am also working on a feature in pgmpy which adds state name feature for variable nodes. I will finish this PR as much as possible during this period.

**Week 1-2**

- Implementation of the base class for continuous node representations.

**Week 3**

- Writing tests for the ContinuousNode class.
- Fixing bugs if any.
- Writing more documentations.

**Week 4**

- Implementation of the class LinearGaussianCPD

**Week 5**

- Writing tests for LinearGaussainCPD.
- Fixing bugs.
- Writing documentations.

**Week 6**

- Implementation of the class JointGaussainDistribution.

**Week 7**

- Writing tests and documentation for  the class JointGaussainDistribution.

**Week 8**

- Clear any backlogs.
- Fix bugs.
- Will spend time time to ensure that corner test cases have been covered.

**Week 9**

- Implementation of the class CanonicalFactor
    - Will work on multiply and divide methods

**Week 10**

- Implementation of the class CanonicalFactor.
    - Will work on marginalize and reduce methods.

**Week 11**

- Write tests and more documentation for the class CanonicalFactor.

**Week 12**

- Clear backlogs
- Write more tests
- Fix bugs if any

**If I Get Time,**
I will implement either of the following features -

- More methods for discretizing the continuous distributions.
- Gaussian Bayesian Network for purely continuous random variables.
- Inference (variable elimination) for Gaussian Bayesian Network.
- Canonical Tables and Hybrid Networks

**If I Don't Get Time,**
I plan to remain an active contributor to the community and continue my work not only in the continuous random variable module but in the entire library.

## Other Commitments

- I **do not have** any other commitments during this period and plan to focus completely on my GSoC project if I get selected.
- I do not have exams during this period. My new semester will begin in the last week of July, but it will not be a problem since the beginning of the semester is not demanding and I have also adjusted my timeline accordingly.
- I **do not have** any other jobs or internships during this period.
- As of now, I don't have any other short term commitments.
- I have not applied with any other organisation.

## References

- Probabilistic Graphical Models: Principles and Techniques, Book by Daphne Koller and Nir Friedman
- Klugman, S. A., Panjer, H. H. and Willmot, G. E. (2008), Loss Models, From Data to Decisions, Second Edition, Wiley