Bayesian Network with Discrete Random Variables and its Python Implementation

Yuwei (Johnny) Meng

Department of Statistical Sciences, University of Toronto

Contents

1 Preface & Acknowledgements

One day before class, I was talking with Jae and Liam, two other students in STA410, about coming up with a cool name for our Python packages, and I was like "Oh man I hate coming up with names!" I do think that cool package names are good, and I admire people coming up with names like `PyTorch` and `Beautiful Soup` (when I first learned to use `Beautiful Soup` I was very confused how this is an HTML parsing library). But yeah, coming up with names doesn't work for me. Just `BayesNetwork`, or `BayesianNetwork`. It's cool. It's straightforward. No need to ponder on why `Beautiful Soup` is for HTML.

Bayesian Network, a variational model for exact inference. The first time I familiarized with this concept was in CSC384, and I thought that this model seems very versatile and powerful. The exposure to the D-Separation, Variable Elimination, and Viterbi algorithms in this course kept my interest in this model. Upon finishing CSC384, I never thought that I would encounter this concept again in CSC412, presented in a more theoretical and mathematical way. What's more, for the third time I was taught Hidden Markov Model in STA410, and I just can't throw this concept out of my mind now. Well, it's something that interests me, so I quickly chose this model to do as my STA410 project.

Here I would like to express my sincere gratitude to Professor Alice Gao in CSC384 for introducing Bayesian Network to me in a very entertaining way, to Professor Denys Linkov/Murat Erdogdu in CSC412/STA414 for incorporating theories of this model into the course, and definitely to Professor Scott Schwartz in STA410 for providing this project opportunity to me to have this model implemented in Python.

2 Bayesian Network

2.1 Motivation

Consider a set of random variables $\mathbf{X} = (X_1, \cdots, X_n)$ where each $X_i \in \{x_i, \neg x_i\}$ is binary. To specify the joint distribution of $\mathbf{X}$, we need to list out all the possible combinations:

$$\begin{aligned}
&\mathbb{P}(x_1, x_2, \cdots, x_n);\\
&\mathbb{P}(\neg x_1, x_2, \cdots, x_n);\\
&\mathbb{P}(x_1, \neg x_2, \cdots, x_n);\\
&\vdots
\end{aligned}$$

For n random variables, the number of joint probabilities is $2^n - 1$, which quickly becomes intractable as n grows large, requiring a more efficient framework for modeling joint probabilities. Bayesian Network, or Directed Acyclic Graphical Model (DAGM), is one such framework exploiting *Theorem 1* below.

Theorem 1. (Chain Rule of Probability) Let $X_1, \cdots, X_n$ be random variables. Then the joint probability can be expressed as

$$\begin{aligned}
\mathbb{P}(X_1, \cdots, X_n) &= \mathbb{P}(X_1)\mathbb{P}(X_2, X_3, \cdots, X_n | X_1)\\
&= \mathbb{P}(X_1)\mathbb{P}(X_2|X_1)\mathbb{P}(X_3, \cdots, X_n|X_1, X_2)\\
&= \cdots\\
&= \mathbb{P}(X_1)\mathbb{P}(X_2|X_1)\mathbb{P}(X_3|X_1, X_2)\cdots\mathbb{P}(X_n|X_1, X_2, \cdots, X_{n-1})\\
&= \prod_{i=1}^{n} \mathbb{P}(X_i|X_1, X_2, \cdots, X_{i-1}).
\end{aligned}$$

■

Thus, if we have prior knowledge about the conditional independence relationships between the random variables we are modeling, then we can express the joint distribution in a more compact form, which motivates the use of Bayesian Networks.

2.2 Introduction to Bayesian Network

An example of Bayesian Network is shown in *Figure 1* below.

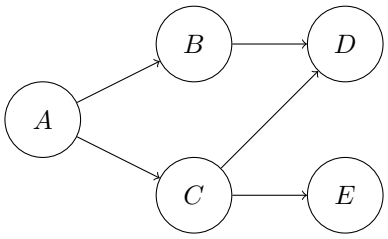

Figure 1: Bayesian Network Example

In this Bayesian Network, each vertex represents a random variable. The arrows denote conditional dependencies between random variables. For example, the edge $A \longrightarrow B$ implies that B is directly conditionally dependent on A. With this structure, we can model the joint distribution of these random variables in the following way:

Definition 1. Let $\mathbf{X} = (X_1, \cdots, X_n)$ be a set of random variables modeled by a Bayesian Network. Then the joint probability can be expressed as

$$\mathbb{P}(X_1, \cdots, X_n) = \prod_{i=1}^{n} \mathbb{P}(X_i | \text{ Parents}(X_i)).$$

■

Example 1. For the Bayesian Network shown in *Figure 1*, the joint probability can be expressed as

$$\mathbb{P}(A, B, C, D, E) = \mathbb{P}(A) \cdot \mathbb{P}(B|A) \cdot \mathbb{P}(C|A) \cdot \mathbb{P}(D|B, C) \cdot \mathbb{P}(E|C).$$

■

Note that *Definition 1* is a special case of *Theorem 1*, given that we have prior knowledge of the conditional independence relationships between the random variables. As a result, we can reduce the number of conditionals in the product and express the joint distribution in a more compact form. To illustrate this concept more clearly, let's define some probabilities for the Bayesian Network shown in *Figure 1*. These probabilities will be used throughout the rest of this document.

Example 2. Let $A, B, C, D, E \in \{0, 1\}$ be binary random variables with the following conditional probabilities:

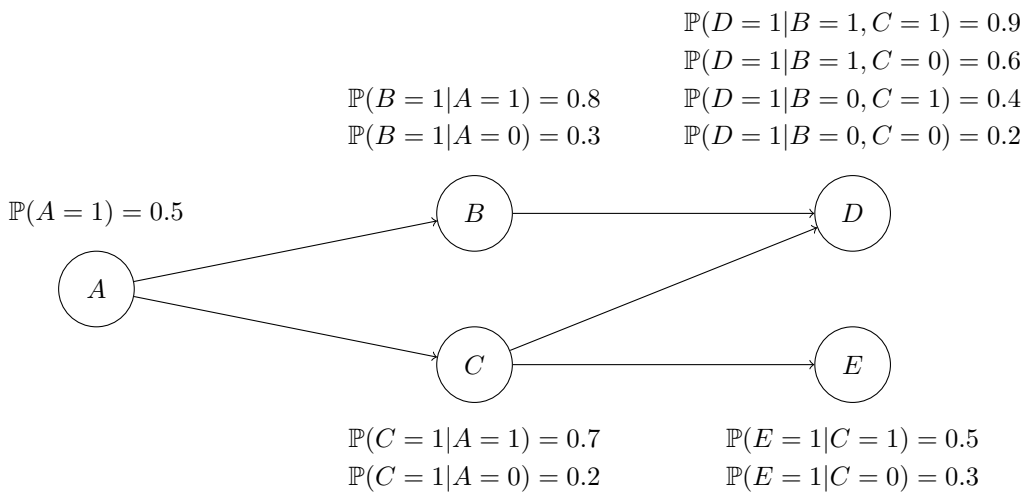

Figure 2: Bayesian Network with Probabilities Example

■

Note that only 11 conditional probabilities are needed to specify the joint distribution of the 5 random variables in our Bayesian Network, which is a significant improvement compared to the $2^5 - 1 = 31$ joint probabilities in the general case. The exact number of required conditional probabilities depends on the structure of the Bayesian Network, though, but as long as there are some conditional independence relationships between the random variables, we can always expect a reduction in the number of required conditional probabilities.

2.3 Implementation

The implementation of Bayesian Network in this project consists of multiple modules which are discussed separately in the following subsections.

2.3.1 `Distribution`

The `distribution.py` file defines an abstract class called `Distribution`, as well as two inherited classes named `UnconditionalDistribution` and `ConditionalDistribution`. The purpose of these classes is to provide a data structure that encodes the probability distribution of a vertex in the Bayesian Network.

The abstract class `Distribution` has the attribute and method definitions:

```
domain: set
```

This attribute defines the set of possible values for the random variables.

```
__call__(self, *args) -> float
```

This method evaluates the probability of the random variable taking on a specific value in the domain.

2.3.2 `UnconditionalDistribution`

The `UnconditionalDistribution` class defines the distribution of a random variable that does not have a parent in the Bayesian Network, as well as the distribution of a random variable conditioned on its parents. Besides the inherited attribute `domain`, it has the following attribute:

```
distribution: dict[Any, float]
```

This attribute serves as a lookup table for the unconditional distribution of the random variable. The keys of the dictionary are the possible values of the random variable, and the values are the corresponding probabilities.

```
__init__(self, domain: set, distribution: dict[Any, float]) -> None
```

This is the constructor of the class. It takes in the domain and the distribution as arguments. The constructor checks if the provided domain and distribution are valid. If `domain` does not match the keys of `distribution`, or the probabilities in `distribution` do not sum to 1, a `ValueError` will be raised.

This class also implements the `__call__` method. The expected argument is a value in the domain of the random variable, and the method outputs the probability of the random variable taking on that value. If the argument is not in the domain, a `ValueError` will be raised.

Example 3. For the Bayesian Network shown in *Figure 2*, the distribution of vertex A would have the following behaviors:

```
>>> dist_A = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.5, 1: 0.5})
>>> dist_A(0)
0.5
>>> dist_A(1)
0.5
```

■

2.3.3 `ConditionalDistribution`

This class defines the distribution of a random variable that has parents in the Bayesian Network. It inherits the `domain` attribute from the `Distribution` class, and it has the following attributes:

```
distributions[frozenset[str], UnconditionalDistribution]
```

The keys of this dictionary are `frozensets` of conditionals of the random variable, and the values define the distribution of the random variable conditioned on the values of its parents.

```
__init__(self,
         domain: set,
```

```
distributions: dict[frozenset[str], UnconditionalDistribution]) -> None
```

This constructor takes in the domain and the distributions as arguments. The constructor checks if the provided domain and distributions are valid. If `domain` does not match the domain of each `UnconditionalDistribution` object in `distribution`, a `ValueError` will be raised.

This class also implements the `__call__` method. The expected arguments are a value in the domain of the random variable and a dictionary of the values of its parents. The method outputs the probability of the random variable taking on that value, given the values of its parents. If the value is not in the domain, or if the parents are not in the dictionary, or if the conditioning values are not in the domain of the parents, a `ValueError` will be raised.

This is slightly confusing, so the following example illustrates this concept.

Example 4. For the Bayesian Network shown in *Figure 2*, the distribution of vertex D would have the following behaviors:

```
>>> dist_D_B1C1 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.1, 1: 0.9})
>>> dist_D_B1C0 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.4, 1: 0.6})
>>> dist_D_B0C1 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.6, 1: 0.4})
>>> dist_D_B0C0 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.8, 1: 0.2})
>>> dist_D = ConditionalDistribution(domain={0, 1}, distributions={
    frozenset({'B: 1', 'C: 1'}): dist_D_B1C1,
    frozenset({'B: 1', 'C: 0'}): dist_D_B1C0,
    frozenset({'B: 0', 'C: 1'}): dist_D_B0C1,
    frozenset({'B: 0', 'C: 0'}): dist_D_B0C0
})
>>> dist_D(1, {'B': 1, 'C': 0})
0.6
>>> dist_D(0, {'C': 0, 'B': 0})
0.8
```

■

The use of a `frozenset` data structure is a compromise to the need for a hashable data structure. The advantage is that we don't need to memorize the order of the parents when we are looking up the distribution because there is no inherent order for a `set` object. When looking up probabilities, the order in which we pass in the parents thus does not matter, as long as the values are correct, which ensures the efficiency of the lookup process.

2.3.4 `Vertex`

The `Vertex` class represents a vertex in the Bayesian Network. It has the following attributes:

```
name: str
```

This attribute defines the name of the vertex.

```
domain: set
```

This attribute defines the set of possible values for the random variable.

```
parents: dict[str, Self]
children: dict[str, Self]
```

These two attributes hold the parents and children of the vertex in the Bayesian Network. The keys of the dictionaries are the names of the vertices, and the values are the corresponding `Vertex` objects.

```
distribution: Optional[Distribution]
```

This attribute holds the distribution of the vertex. Depending on whether the vertex has parents or not, this attribute

can be either an `UnconditionalDistribution` or a `ConditionalDistribution` object.

The `Vertex` class also implements the following methods:

```
__init__(self, name: str, domain: set) -> None
```

This constructor takes in the name and domain of the vertex as arguments. It initializes the `parents`, `children`, and `distribution` attributes to empty dictionaries or `None`.

```
add_parent(self, parent: Self) -> None
add_child(self, child: Self) -> None
```

These two methods add a parent or child to the vertex. If the added vertex is already a parent or child of the vertex, a `ValueError` will be raised.

```
in_domain(self, value: Any) -> bool
```

This method returns `True` if the value is in the domain of the vertex, and `False` otherwise.

```
set_distribution(self, distribution: Distribution) -> None
```

This method sets the distribution of the vertex. This method contains many checks to ensure that `distribution` is valid. If `distribution` is not compatible with the vertex, a `ValueError` will be raised.

```
__call__(self, *args) -> float
```

This method basically calls the `__call__` method of the `distribution` object with the arguments `*args`. The expected arguments are the same as the `__call__` method of the `distribution` object and outputs a probability. If the `distribution` attribute for the vertex is `None`, a `ValueError` will be raised.

2.3.5 BayesNetwork

This class represents the Bayesian Network. It has the following attribute:

```
vertices: dict[str, Vertex]
```

This attribute holds a list of the vertices in the Bayesian Network. The keys of the dictionary are the names of the vertices, and the values are the corresponding `Vertex` objects.

The `BayesNetwork` class implements the following methods:

```
__init__(self) -> None
```

The constructor creates a `BayesNetwork` object and initializes the `vertices` attribute to an empty dictionary.

```
__len__(self) -> int
```

This method returns the number of vertices in the Bayesian Network.

```
add_node(self, vertex: Vertex) -> None
```

This method adds a vertex to the Bayesian Network. If the vertex is already in the network, a `ValueError` will be raised.

```
add_edge(self, parent: Vertex, child: Vertex) -> None
```

This method adds an edge from the parent vertex to the child vertex in the Bayesian Network. If either vertex is not in the network, a `ValueError` will be raised.

```
find_roots(self) -> list[Vertex]
```

This method returns a list of the root vertices in the Bayesian Network. A root vertex is defined as a vertex that does not have any parents in the network.

```
__call__(self, *args) -> float
```

This method evaluates the joint probability of the random variables in the Bayesian Network. The expected argument is a dictionary with the keys being the names of the random variables and the values being the corresponding values. The method outputs the joint probability of the random variables taking on those values. If values for one or more random variables are not provided, a `ValueError` will be raised. To compute the probability of a subset of the random variables, use the *Variable Elimination* algorithm discussed in the next section.

2.3.6 `read_bayes_network_from_txt`

The above description might be very confusing, so it is not recommended to create a Bayesian Network from scratch. Instead, I provide a function to read in a Bayesian Network from a text file. An example of this text file is provided in this `bn_ex.txt` file. The text file requires 3 parts, explained using the example file and *Figure 2*:

```
A: {0, 1}
B: {0, 1}
C: {0, 1}
D: {0, 1}
E: {0, 1}
```

This part defines the vertices in the Bayesian Network. Each line contains the name of the vertex and its domain, enclosed in curly braces. The vertices are separated by new lines.

```
A -> B
A -> C
B -> D
C -> D
C -> E
```

This part defines the edges in the Bayesian Network. Each line contains the name of the parent vertex and the name of the child vertex, separated by an arrow (`->`). The edges are separated by new lines.

```
P(A = 1) = 0.5
P(A = 0) = 0.5

P(B = 1 | A = 1) = 0.8
P(B = 0 | A = 1) = 0.2
P(B = 1 | A = 0) = 0.3
P(B = 0 | A = 0) = 0.7

P(D = 1 | B = 1, C = 1) = 0.9
P(D = 0 | B = 1, C = 1) = 0.1
P(D = 1 | B = 1, C = 0) = 0.6
P(D = 0 | B = 1, C = 0) = 0.4
P(D = 1 | B = 0, C = 1) = 0.4
P(D = 0 | B = 0, C = 1) = 0.6
P(D = 1 | B = 0, C = 0) = 0.2
P(D = 0 | B = 0, C = 0) = 0.8

...
```

This part defines the unconditional and conditional distributions of the vertices in the Bayesian Network. The conditional distributions are separated by new lines. Note that it is necessary to provide the probabilities for both `P(A = 1)` and `P(A = 0)` in the text file, even though the complement can be easily computed.

As long as the format is followed, the order in which the vertices, edges, and distributions are defined does not matter because the function uses regular expression to parse the text file. The function will create a `BayesNetwork` object and add the vertices, edges, and distributions to the object. The function then returns the `BayesNetwork` object. It is strongly recommended to use this function to create a Bayesian Network.

To wrap up this section, below is an example of how to use the function to create a Bayesian Network from a text file.

Example 5. Assume that the text file is named `bn_ex.txt` and is in the same directory as the Python file.

```
>>> bn = read_bayes_network_from_txt('bn_ex.txt')
>>> bn.find_roots()
[A: {0, 1}]
>>> len(bn)
5
```

Regarding joint probability computations:

$$\begin{aligned}\mathbb{P}(A=1,B=0,C=1,D=1,E=0) &= \mathbb{P}(A=1)\mathbb{P}(B=0|A=1)\mathbb{P}(C=1|A=1)\mathbb{P}(D=1|B=0,C=1)\mathbb{P}(E=0|C=1)\\ &= 0.5\cdot 0.2\cdot 0.7\cdot 0.4\cdot 0.5\\ &= 0.014.\end{aligned}$$

The code output confirms this result:

```
>>> bn({'A': 1, 'B': 0, 'C': 1, 'D': 1, 'E': 0})
0.013999999999999999 # Numerical roundoff error
```

■

3 Variable Elimination