Bayesian Network with Discrete Random Variables and its Python Implementation

Yuwei (Johnny) Meng

Department of Statistical Sciences, University of Toronto

Contents

1 Preface & Acknowledgements

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.

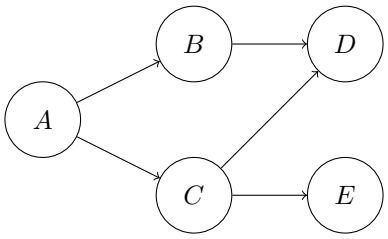

Figure 1: Bayesian Network Example

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:

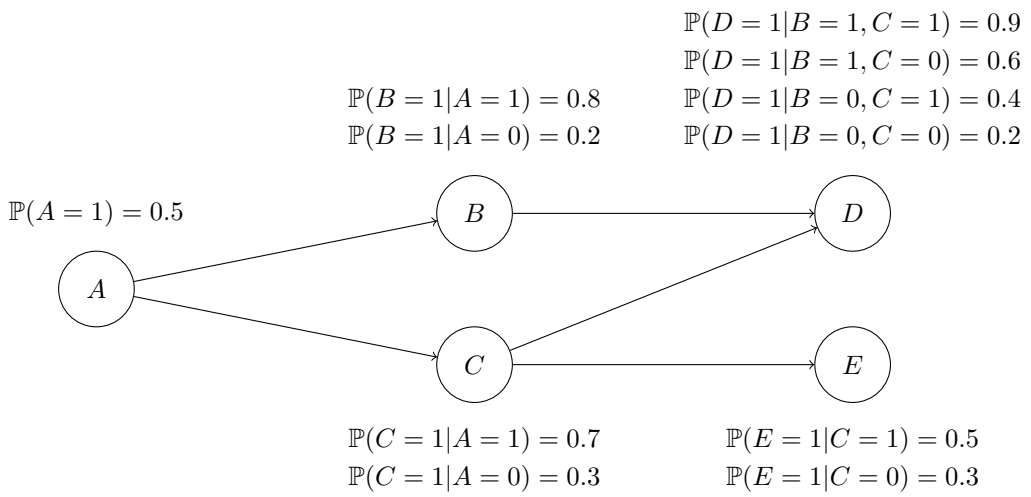

Figure 2: Bayesian Network with Probabilities Example

■

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.

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.

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.

3 Variable Elimination