3 Inference

The previous chapter explained how to represent probability distributions. This chapter will show how to use these probabilistic representations for *inference*. Inference involves determining the distribution over one or more unobserved variables given the values associated with a set of observed variables. This chapter begins by introducing exact inference methods. Because exact inference can be computationally intractable depending on the structure of the network, we will also discuss several algorithms for approximate inference.

3.1 Inference in Bayesian Networks

In inference problems, we want to infer a distribution over *query variables* given some observed *evidence variables*. The other nodes are referred to as *hidden variables*. We often refer to the distribution over the query variables given the evidence as a *posterior distribution*.

To illustrate the computations involved in inference, recall the Bayesian network from example 2.5, the structure of which is reproduced in figure 3.1. Suppose we have B as a query variable and evidence $D = 1$ and $C = 1$. The inference task is to compute $P(b^1 \mid d^1, c^1)$, which corresponds to computing the probability that we have a battery failure given an observed trajectory deviation and communication loss.

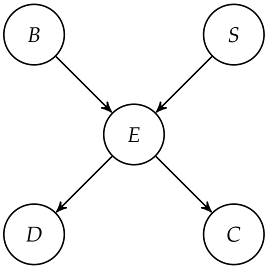


Figure 3.1. Bayesian network structure from example 2.5.

From the definition of conditional probability introduced in equation (2.21), we know

$$P(b^1 \mid d^1, c^1) = \frac{P(b^1, d^1, c^1)}{P(d^1, c^1)} \tag{3.1}$$

To compute the numerator, we must use a process known as *marginalization*, where we sum out variables that are not involved, in this case S and E:

$$P(b^1, d^1, c^1) = \sum_s \sum_e P(b^1, s, e, d^1, c^1) \tag{3.2}$$

We know from the chain rule for Bayesian networks introduced in equation (2.30) that

$$P(b^1, s, e, d^1, c^1) = P(b^1)P(s)P(e \mid b^1, s)P(d^1 \mid e)P(c^1 \mid e) \tag{3.3}$$

All of the components on the right-hand side are specified in the conditional probability distributions associated with the nodes in the Bayesian network. We can compute the denominator in equation (3.1) using the same approach but with an additional summation over the values for B.

This process of using the definition of conditional probability, marginalization, and applying the chain rule can be used to perform exact inference in any Bayesian network. We can implement exact inference using factors. Recall that factors represent discrete multivariate distributions. We use the following three operations on factors to achieve this:

- We use the *factor product* (algorithm 3.1) to combine two factors to produce a larger factor whose scope is the combined scope of the input factors. If we have $\phi(X, Y)$ and $\psi(Y, Z)$, then $\phi \cdot \psi$ will be over X, Y, and Z with $(\phi \cdot \psi)(x, y, z) = \phi(x, y)\psi(y, z)$. The factor product is demonstrated in example 3.1.

- We use *factor marginalization* (algorithm 3.2) to sum out a particular variable from the entire factor table, removing it from the resulting scope. Example 3.2 illustrates this process.

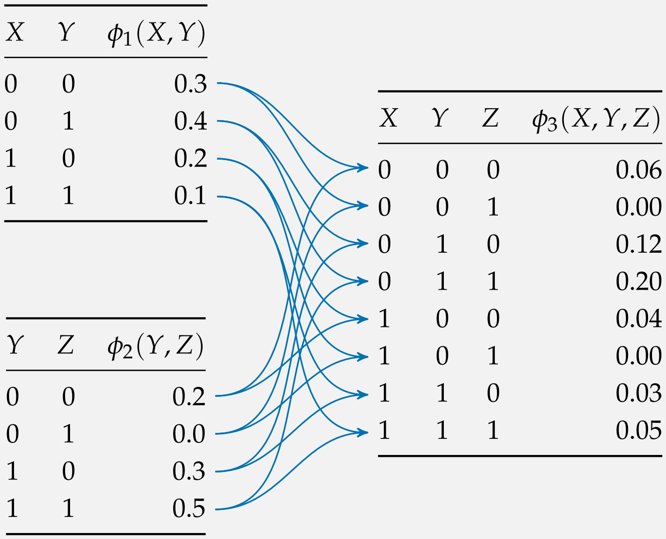
- We use *factor conditioning* (algorithm 3.3) with respect to some evidence to remove any rows in the table inconsistent with that evidence. Example 3.3 demonstrates factor conditioning.

These three factor operations are used together in algorithm 3.4 to perform exact inference. It starts by computing the product of all of the factors, conditioning on the evidence, marginalizing out the hidden variables, and normalizing. One potential issue with this approach is the size of the product of all of the factors. The size of the factor product is equal to the product of the number of values each variable can assume. For the satellite example problem, there are only $2^5 = 32$ possible assignments, but many interesting problems would have a factor product that is too large to practically enumerate.

```julia
function Base.:*(ϕ::Factor, ψ::Factor)
    ϕnames = variablenames(ϕ)
    ψnames = variablenames(ψ)
    shared = intersect(ϕnames, ψnames)
    ψonly = setdiff(ψ.vars, ϕ.vars)
    table = FactorTable()
    for (ϕa,ϕp) in ϕ.table
        for a in assignments(ψonly)
            a = merge(ϕa, a)
            ψa = isempty(ψ.vars) ? NamedTuple() : select(a, ψnames)
            table[a] = ϕp * get(ψ.table, ψa, 0.0)
        end
    end
    vars = vcat(ϕ.vars, ψonly)
    return Factor(vars, table)
end
```

Algorithm 3.1. An implementation of the factor product, which constructs the factor representing the joint distribution of two smaller factors ϕ and ψ. If we want to compute the factor product of ϕ and ψ, we simply write ϕ*ψ.

The factor product of two factors produces a new factor over the union of their variables. Below we produce a new factor from two factors that share a variable.



X	Y	$\phi_1(X,Y)$
0	0	0.3
0	1	0.4
1	0	0.2
1	1	0.1

Y	Z	$\phi_2(Y,Z)$
0	0	0.2
0	1	0.0
1	0	0.3
1	1	0.5

X	Y	Z	$\phi_3(X,Y,Z)$
0	0	0	0.06
0	0	1	0.00
0	1	0	0.12
0	1	1	0.20
1	0	0	0.04
1	0	1	0.00
1	1	0	0.03
1	1	1	0.05

Example 3.1. An illustration of a factor product combining two factors representing $\phi_1(X,Y)$ and $\phi_2(Y,Z)$ to produce a factor representing $\phi_3(X,Y,Z)$.

```
function marginalize(ϕ::Factor, name)
    table = FactorTable()
    for (a, p) in ϕ.table
        a′ = delete(a, name)
        table[a′] = get(table, a′, 0.0) + p
    end
    vars = filter(v -> v.name != name, ϕ.vars)
    return Factor(vars, table)
end
```

Algorithm 3.2. A method for marginalizing a variable named `name` from a factor `ϕ`.

Recall the joint probability distribution $P(X, Y, Z)$ from table 2.1. We can marginalize out Y by summing the probabilities in each row that have matching assignments for X and Z.

X	Y	Z	$\phi(X, Y, Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

X	Z	$\phi(X, Z)$
0	0	0.17
0	1	0.68
1	0	0.03
1	1	0.12

Example 3.2. A demonstration of factor marginalization.

```
in_scope(name, ϕ) = any(name == v.name for v in ϕ.vars)

function condition(ϕ::Factor, name, value)
    if !in_scope(name, ϕ)
        return ϕ
    end
    table = FactorTable()
    for (a, p) in ϕ.table
        if a[name] == value
            table[delete(a, name)] = p
        end
    end
    vars = filter(v -> v.name != name, ϕ.vars)
    return Factor(vars, table)
end

function condition(ϕ::Factor, evidence)
    for (name, value) in pairs(evidence)
        ϕ = condition(ϕ, name, value)
    end
    return ϕ
end
```

Algorithm 3.3. Two methods for factor conditioning given some evidence. The first takes a factor `ϕ` and returns a new factor whose table entries are consistent with the variable named `name` having value `value`. The second takes a factor `ϕ` and applies evidence in the form of a named tuple. The `in_scope` method returns true if a variable named `name` is within the scope of the factor `ϕ`.

Factor conditioning involves dropping any rows inconsistent with the evidence. Below is the factor from table 2.1, and we condition on $Y = 1$. All rows for which $Y \neq 1$ are removed.

X	Y	Z	$\phi(X,Y,Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

$Y \leftarrow 1$

X	Z	$\phi(X,Z)$
0	0	0.09
0	1	0.37
1	0	0.02
1	1	0.07

Example 3.3. An illustration of setting evidence, in this case for Y, in a factor. The resulting values must be renormalized.

```
struct ExactInference end

function infer(M::ExactInference, bn, query, evidence)
    ϕ = prod(bn.factors)
    ϕ = condition(ϕ, evidence)
    for name in setdiff(variablenames(ϕ), query)
        ϕ = marginalize(ϕ, name)
    end
    return normalize!(ϕ)
end
```

Algorithm 3.4. A naive exact inference algorithm for a discrete Bayesian network `bn`, which takes as input a set of query variable names `query`, and `evidence` associating values with observed variables. The algorithm computes a joint distribution over the query variables in the form of a factor. We introduce the `ExactInference` type to allow for `infer` to be called with different inference methods, as shall be seen in the rest of this chapter.

3.2 *Inference in Naive Bayes Models*

The previous section presented a general method for performing exact inference in any Bayesian network. This section discusses how this same method can be used to solve *classification* problems for a special kind of Bayesian network structure known as a *naive Bayes* model. This structure is shown in figure 3.2. An equivalent but more compact representation is shown in figure 3.3 using a *plate*, shown as a rounded box. The $i = 1 : n$ in the bottom of the box specifies that the i in the subscript of the variable name is repeated from 1 to n.

In the naive Bayes model, the class C is the query variable, and the observed features $O_{1:n}$ are the evidence variables. The naive Bayes model is called naive because it assumes conditional independence between the evidence variables given the class. Using the notation introduced in section 2.6, we can say $(O_i \perp O_j \mid C)$ for all $i \neq j$. Of course, if these conditional independence assumptions do not hold, then we can add the necessary directed edges between the observed features.

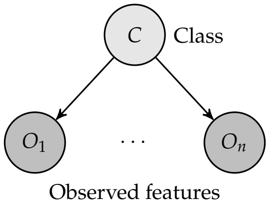


Figure 3.2. A naive Bayes model.

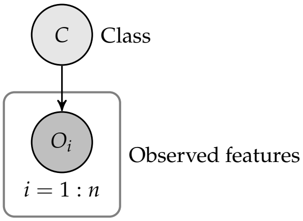


Figure 3.3. Plate representation of a naive Bayes model.

We have to specify the *prior* $P(C)$ and the *class-conditional distributions* $P(O_i \mid C)$. As done in the previous section, we can apply the chain rule to compute the joint distribution:

$$P(c, o_{1:n}) = P(c) \prod_{i=1}^{n} P(o_i \mid c) \tag{3.4}$$

Our classification task involves computing the conditional probability $P(c \mid o_{1:n})$. From the definition of conditional probability, we have

$$P(c \mid o_{1:n}) = \frac{P(c, o_{1:n})}{P(o_{1:n})} \tag{3.5}$$

We can compute the denominator by marginalizing the joint distribution:

$$P(o_{1:n}) = \sum_c P(c, o_{1:n}) \tag{3.6}$$

The denominator in equation (3.5) is not a function of C and can therefore be treated as a constant. Hence, we can write

$$P(c \mid o_{1:n}) = \kappa P(c, o_{1:n}) \tag{3.7}$$

where κ is a *normalization constant* such that $\sum_c P(c \mid o_{1:n}) = 1$. We often drop κ and write

$$P(c \mid o_{1:n}) \propto P(c, o_{1:n}) \tag{3.8}$$

where the *proportional to* symbol $\propto$ is used to represent that the left-hand side is proportional to the right-hand side. Example 3.4 illustrates how inference can be applied to classifying radar tracks.

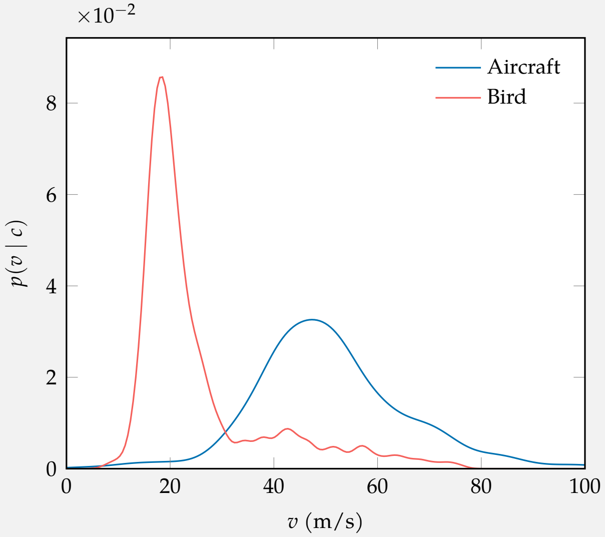
We can use this method to infer a distribution over classes, but for many applications, we have to commit to a particular class. It is common to classify according to the class with the highest posterior probability, $\arg\max_c P(c \mid o_{1:n})$. However, choosing a class is really a decision problem that often should take into account the consequences of misclassification. For example, if we are interested in using our classifier to filter out targets that are not aircraft for the purpose of air traffic control, then we can afford to occasionally let a few birds and other clutter tracks through our filter. However, we would want to avoid filtering out any real aircraft because that could lead to a collision. In this case, we would probably only want to classify a track as a bird if the posterior probability were close to 1. Decision problems will be discussed in chapter 6.

3.3 *Sum-Product Variable Elimination*

A variety of methods can be used to perform efficient inference in more complicated Bayesian networks. One method is known as *sum-product variable elimination*, which interleaves eliminating hidden variables (summations) with applications of the chain rule (products). It is more efficient to marginalize variables out as early as possible to avoid generating large factors.

Example 3.4. Radar target classification example where we want to determine whether a radar track corresponds to a bird or an aircraft.

Suppose we have a radar track and we want to determine whether it was generated by a bird or an aircraft. We base our inference on airspeed and the amount of heading fluctuation. The prior represents our belief about whether a target is a bird or an aircraft in the absence of any information about the track. Below are example class-conditional distributions for airspeed v as estimated from radar data.



Suppose from the chain rule we determine:

$$\begin{aligned} P(\text{bird}, \text{slow}, \text{little heading fluctuation}) &= 0.03 \\ P(\text{aircraft}, \text{slow}, \text{little heading fluctuation}) &= 0.01 \end{aligned}$$

Of course, these probabilities do not sum to 1. If we want to determine the probability that a target is a bird given the evidence, then we would make the following calculation:

$$P(\text{bird} \mid \text{slow}, \text{little heading fluctuation}) = \frac{0.03}{0.03 + 0.01} = 0.75$$

We will illustrate the variable elimination algorithm by computing the distribution $P(B \mid d^1, c^1)$ for the Bayesian network in figure 3.1. The conditional probability distributions associated with the nodes in the network can be represented by the following factors:

$$\phi_1(B), \phi_2(S), \phi_3(E, B, S), \phi_4(D, E), \phi_5(C, E) \tag{3.9}$$

Because D and C are observed variables, the last two factors can be replaced with $\phi_6(E)$ and $\phi_7(E)$ by setting the evidence $D = 1$ and $C = 1$.

We then proceed by eliminating the hidden variables in sequence. Different strategies can be used for choosing an ordering, but for this example, we arbitrarily choose the ordering E and then S. To eliminate E, we take the product of all the factors involving E and then marginalize out E to get a new factor:

$$\phi_8(B, S) = \sum_e \phi_3(e, B, S)\phi_6(e)\phi_7(e) \tag{3.10}$$

We can now discard ϕ_3, ϕ_6, and ϕ_7 because all the information we need from them is contained in ϕ_8.

Next, we eliminate S. Again, we gather all remaining factors that involve S and marginalize out S from the product of these factors:

$$\phi_9(B) = \sum_s \phi_2(s)\phi_8(B, s) \tag{3.11}$$

We discard ϕ_2 and ϕ_8, and are left with $\phi_1(B)$ and $\phi_9(B)$. Finally, we take the product of these two factors and normalize the result to obtain a factor representing $P(B \mid d^1, c^1)$.

The above procedure is equivalent to computing the following:

$$P(B \mid d^1, c^1) \propto \phi_1(B) \sum_s \left(\phi_2(s) \sum_e \Big(\phi_3(e \mid B, s)\phi_4(d^1 \mid e)\phi_5(c^1 \mid e) \Big) \right) \tag{3.12}$$

This produces the same result as, but is more efficient than, the naive procedure of taking the product of all of the factors and then marginalizing:

$$P(B \mid d^1, c^1) \propto \sum_s \sum_e \phi_1(B)\phi_2(s)\phi_3(e \mid B, s)\phi_4(d^1 \mid e)\phi_5(c^1 \mid e) \tag{3.13}$$

The sum-product variable elimination algorithm is implemented in algorithm 3.5. It takes as input a Bayesian network, a set of query variables, a list of observed values, and an ordering of the variables. We first set all observed values. Then, for each variable, we multiply together all factors containing it and then marginalize that variable out. This new factor replaces the consumed factors and we repeat the process for the next variable.

For many networks, variable elimination allows inference to be done in an amount of time that scales linearly with the size of the network, but it has exponential time complexity in the worst case. What influences the amount of computation is the variable elimination order. Choosing the optimal elimination order is *NP-hard*,[1] meaning that it cannot be done in polynomial time in the worst case (section 3.5). Even if we found the optimal elimination order, variable elimination can still require an exponential number of computations. Variable elimination heuristics generally try to minimize the number of variables involved in the intermediate factors generated by the algorithm.

```
struct VariableElimination
    ordering # array of variable indices
end

function infer(M::VariableElimination, bn, query, evidence)
    Φ = [condition(ϕ, evidence) for ϕ in bn.factors]
    for i in M.ordering
        name = bn.vars[i].name
        if name ∉ query
            inds = findall(ϕ->in_scope(name, ϕ), Φ)
            if !isempty(inds)
                ϕ = prod(Φ[inds])
                deleteat!(Φ, inds)
                ϕ = marginalize(ϕ, name)
                push!(Φ, ϕ)
            end
        end
    end
    return normalize!(prod(Φ))
end
```

Algorithm 3.5. An implementation of the sum-product variable elimination algorithm, which takes in a Bayesian Network `bn`, a list of query variables `query`, and evidence `evidence`. The variables are processed in the order given by the ordering `ordering`.

[1] S. Arnborg, D. G. Corneil, and A. Proskurowski, "Complexity of Finding Embeddings in a *k*-Tree," *SIAM Journal on Algebraic Discrete Methods*, vol. 8, no. 2, pp. 277–284, 1987.

3.4 *Belief Propagation*

An approach to inference known as *belief propagation* works by propagating "messages" through the network using the *sum-product algorithm* in order to compute the marginal distributions of the query variables.[2] Belief propagation requires linear time but only provides an exact answer if the network does not have undirected cycles. If the network has undirected cycles, then it can be converted into a tree by combining multiple variables into single nodes by using what is known as the *junction tree algorithm*. If the number of variables that have to be combined into any one node in the resulting network is small, then inference can be done efficiently. A variation of belief propagation known as *loopy belief propagation* can provide approximate solutions in networks with undirected cycles. Although this approach does not provide any guarantees and may not converge, it can work well in practice.[3]

3.5 *Computational Complexity*

We can show that inference in Bayesian networks is NP-hard by using an NP-complete problem called 3SAT.[4] It is easy to construct a Bayesian network from an arbitrary 3SAT problem. For example, consider the following 3SAT formula:[5]

$$F(x_1, x_2, x_3, x_4) = \begin{array}{ccccccccc} (& x_1 & \lor & x_2 & \lor & x_3 &) & \land \\ (& \neg x_1 & \lor & \neg x_2 & \lor & x_3 &) & \land \\ (& x_2 & \lor & \neg x_3 & \lor & x_4 &) & \end{array} \tag{3.14}$$

where $\neg$ represents *logical negation* ("not"), $\land$ represents *logical conjunction* ("and"), and $\lor$ represents *logical disjunction* ("or"). The formula consists of a conjunction of *clauses*, which are disjunctions of what are called *literals*. A literal is simply a variable or its negation.

Figure 3.4 shows the corresponding Bayesian network representation. The variables are represented by $X_{1:4}$, and the clauses are represented by $C_{1:3}$. The distributions over the variables are uniform. The nodes representing clauses have as parents the participating variables. Because this is a 3SAT problem, each clause node has exactly three parents. Each clause node assigns probability 0 to assignments that do not satisfy the clause and probability 1 to all satisfying assignments. The remaining nodes assign probability 1 to true if all their parents

[2] A tutorial on the sum-product algorithm with a discussion of its connections to many other algorithms developed in separate communities is provided by F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor Graphs and the Sum-Product Algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[3] Belief propagation and related algorithms are covered in detail by D. Barber, *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[4] G. F. Cooper, "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks," *Artificial Intelligence*, vol. 42, no. 2–3, pp. 393–405, 1990. The Bayesian network construction in this section follows that text. See appendix C for a brief review of complexity classes.

[5] This formula also appears in example C.3 in appendix C.

are true. The original problem is satisfiable if and only if $P(y^1) > 0$. Hence, inference in Bayesian networks is at least as hard as 3SAT.

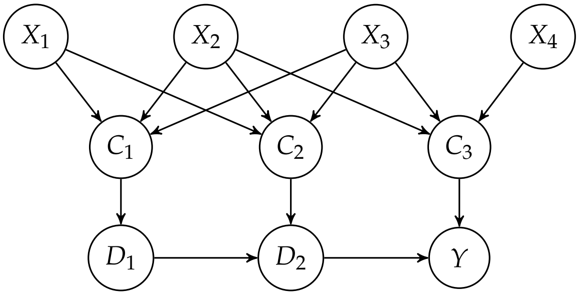


Figure 3.4. Bayesian network representing a 3SAT problem.

The reason we go to the effort of showing that inference in Bayesian networks is NP-hard is so that we know to avoid wasting time looking for an efficient, exact inference algorithm that works on all Bayesian networks. Therefore, research over the past couple decades has focused on approximate inference methods, which are discussed next.

3.6 *Direct Sampling*

Motivated by the fact that exact inference is computationally intractable, many approximation methods have been developed. One of the simplest methods for inference is based on *direct sampling*, where random samples from the joint distribution are used to arrive at a probability estimate.[6] To illustrate, suppose we want to infer $P(b^1 \mid d^1, c^1)$ from a set of n samples from the joint distribution $P(b, s, e, d, c)$. We use parenthetical superscripts to indicate the index of a sample, where we write $(b^{(i)}, s^{(i)}, e^{(i)}, d^{(i)}, c^{(i)})$ for the ith sample. The direct sample estimate is

$$P(b^1 \mid d^1, c^1) \approx \frac{\sum_i (b^{(i)} = 1 \land d^{(i)} = 1 \land c^{(i)} = 1)}{\sum_i (d^{(i)} = 1 \land c^{(i)} = 1)} \tag{3.15}$$

We use the convention where a logical statement in parentheses is treated numerically as 1 when true and 0 when false. The numerator is the number of samples consistent with b, d, and c all set to 1, and the denominator is the number of samples consistent with d and c all set to 1.

[6] Sometimes approaches involving random sampling are referred to as *Monte Carlo methods*. The name comes from the Monte Carlo Casino in Monaco. An introduction to randomized algorithms and their application to a variety of problem domains is provided by R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.

Sampling from the joint distribution represented by a Bayesian network is straightforward. The first step involves finding a *topological sort* of the nodes in the Bayesian network. A topological sort of nodes in a directed acyclic graph is an ordered list such that if there is an edge $A \to B$, then A comes before B in the list.[7] For example, a topological sort for the network in figure 3.1 is B, S, E, D, C. A topological sort always exists, but it may not be unique. Another topological sort for the network is S, B, E, C, D.

Once we have a topological sort, we can begin sampling from the conditional probability distributions. Algorithm 3.6 shows how to sample from a Bayesian network given an ordering $X_{1:n}$ that represents a topological sort. We draw a sample from the conditional distribution associated with X_i given the values of the parents that have already been assigned. Because $X_{1:n}$ is a topological sort, we know that all the parents of X_i have already been instantiated, allowing this sampling to be done. Direct sampling is implemented in algorithm 3.7 and is demonstrated in example 3.5.

```julia
function Base.rand(ϕ::Factor)
    tot, p, w = 0.0, rand(), sum(values(ϕ.table))
    for (a,v) in ϕ.table
        tot += v/w
        if tot >= p
            return a
        end
    end
    return NamedTuple()
end

function Base.rand(bn::BayesianNetwork)
    a = NamedTuple()
    for i in topological_sort(bn.graph)
        name, ϕ = bn.vars[i].name, bn.factors[i]
        value = rand(condition(ϕ, a))[name]
        a = merge(a, namedtuple(name)(value))
    end
    return a
end
```

Algorithm 3.6. A method for sampling an assignment from a Bayesian network `bn`. We also provide a method for sampling an assignment from a factor `ϕ`.

[7] A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962. An implementation of topological sorting is provided by the `LightGraphs.jl` package.

Suppose we draw ten random samples from the network in figure 3.1. We are interested in inferring $P(b^1 \mid d^1, c^1)$. Only two of the ten samples (pointed to in the table) are consistent with the observations d^1 and c^1. One sample has $b = 1$ and the other sample has $b = 0$. From these samples, we infer that $P(b^1 \mid d^1, c^1) = 0.5$. Of course, we would want to use more than just two samples to accurately estimate $P(b^1 \mid d^1, c^1)$.

B	S	E	D	C	
0	0	1	1	0	
0	0	0	0	0	
1	0	1	0	0	
1	0	1	1	1	←
0	0	0	0	0	
0	0	0	1	0	
0	0	0	0	1	
0	1	1	1	1	←
0	0	0	0	0	
0	0	0	1	0	

Example 3.5. An example of how direct samples from a Bayesian network can be used for inference.

```julia
struct DirectSampling
    m # number of samples
end

function infer(M::DirectSampling, bn, query, evidence)
    table = FactorTable()
    for i in 1:(M.m)
        a = rand(bn)
        if all(a[k] == v for (k,v) in pairs(evidence))
            b = select(a, query)
            table[b] = get(table, b, 0) + 1
        end
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

Algorithm 3.7. The direct sampling inference method, which takes a Bayesian network `bn`, a list of query variables `query`, and evidence `evidence`. The method draws `m` samples from the Bayesian network and retains those samples that are consistent with the evidence. A factor over the query variables is returned. This method can fail if no samples that satisfy the evidence are found.

3.7 *Likelihood Weighted Sampling*

The problem with direct sampling is that we may waste time generating samples that are inconsistent with the observations, especially if the observations are unlikely. An alternative approach is called *likelihood weighted sampling*, which involves generating weighted samples that are consistent with the observations.

To illustrate, we will again attempt to infer $P(b^1 \mid d^1, c^1)$. We have a set of n samples, where the ith sample is again denoted $(b^{(i)}, s^{(i)}, e^{(i)}, d^{(i)}, c^{(i)})$. The weight of the ith sample is w_i. The weighted estimate is

$$\begin{aligned} P(b^1 \mid d^1, c^1) &\approx \frac{\sum_i w_i (b^{(i)} = 1 \land d^{(i)} = 1 \land c^{(i)} = 1)}{\sum_i w_i (d^{(i)} = 1 \land c^{(i)} = 1)} && (3.16) \\ &= \frac{\sum_i w_i (b^{(i)} = 1 \land d^{(i)} = 1 \land c^{(i)} = 1)}{\sum_i w_i} && (3.17) \end{aligned}$$

To generate these weighted samples, we begin with a topological sort and sample from the conditional distributions in sequence. The only difference in likelihood weighting is how we handle observed variables. Instead of sampling their values from a conditional distribution, we assign variables to their observed values and adjust the weight of the sample appropriately. The weight of a sample is simply the product of the conditional probabilities at the observed nodes. Likelihood weighted sampling is implemented in algorithm 3.8. Example 3.6 demonstrates inference with likelihood weighted sampling.

```
struct LikelihoodWeightedSampling
    m # number of samples
end

function infer(M::LikelihoodWeightedSampling, bn, query, evidence)
    table = FactorTable()
    ordering = topological_sort(bn.graph)
    for i in 1:(M.m)
        a, w = NamedTuple(), 1.0
        for j in ordering
            name, ϕ = bn.vars[j].name, bn.factors[j]
            if haskey(evidence, name)
                a = merge(a, namedtuple(name)(evidence[name]))
                w *= ϕ.table[select(a, variablenames(ϕ))]
            else
                val = rand(condition(ϕ, a))[name]
                a = merge(a, namedtuple(name)(val))
            end
        end
        b = select(a, query)
        table[b] = get(table, b, 0) + w
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

Algorithm 3.8. The likelihood sampling inference method, which takes a Bayesian network `bn`, a list of query variables `query`, and evidence `evidence`. The method draws `m` samples from the Bayesian network but sets values from evidence when possible, keeping track of the conditional probability when doing so. These probabilities are used to weight the samples such that the final inference estimate is accurate. A factor over the query variables is returned.

The table below shows five likelihood-weighted samples from the network in figure 3.1. We sample from $P(B)$, $P(S)$, and $P(E \mid B, S)$, as we would with direct sampling. When we come to D and C, we assign $D = 1$ and $C = 1$. If the sample has $E = 1$, then the weight is $P(d^1 \mid e^1)P(c^1 \mid e^1)$; otherwise, the weight is $P(d^1 \mid e^0)P(c^1 \mid e^0)$. If we assume

$$\begin{aligned} P(d^1 \mid e^1)P(c^1 \mid e^1) &= 0.95 \\ P(d^1 \mid e^0)P(c^1 \mid e^0) &= 0.01 \end{aligned}$$

then we may approximate from the samples in the table below

$$P(b^1 \mid d^1, c^1) = \frac{0.95}{0.95 + 0.95 + 0.01 + 0.01 + 0.95} \approx 0.331$$

B	S	E	D	C	Weight
1	0	1	1	1	$P(d^1 \mid e^1)P(c^1 \mid e^1)$
0	1	1	1	1	$P(d^1 \mid e^1)P(c^1 \mid e^1)$
0	0	0	1	1	$P(d^1 \mid e^0)P(c^1 \mid e^0)$
0	0	0	1	1	$P(d^1 \mid e^0)P(c^1 \mid e^0)$
0	0	1	1	1	$P(d^1 \mid e^1)P(c^1 \mid e^1)$

Example 3.6. Likelihood weighted samples from a Bayesian network.

Although likelihood weighting makes it so that all samples are consistent with the observations, it can still be wasteful. Consider the simple chemical detection Bayesian network shown in figure 3.5, and assume that we detected a chemical of interest. We want to infer $P(c^1 \mid d^1)$. Because this network is small, we can easily compute this probability exactly by using Bayes' rule:

$$P(c^1 \mid d^1) = \frac{P(d^1 \mid c^1)P(c^1)}{P(d^1 \mid c^1)P(c^1) + P(d^1 \mid c^0)P(c^0)} \tag{3.18}$$

$$= \frac{0.999 \times 0.001}{0.999 \times 0.001 + 0.001 \times 0.999} \tag{3.19}$$

$$= 0.5 \tag{3.20}$$

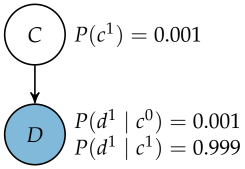


Figure 3.5. Chemical detection Bayesian network with C indicating whether the chemical is present and D indicating whether the chemical is detected.

If we use likelihood weighting, then 99.9% of the samples will have $C = 0$ with a weight of 0.001. Until we get a sample of $C = 1$, which has an associated weight of 0.999, our estimate of $P(c^1 \mid d^1)$ will be 0.

3.8 Gibbs Sampling

An alternative approach to inference is to use *Gibbs sampling*,[8] which is a kind of *Markov chain Monte Carlo* technique. Gibbs sampling involves drawing samples consistent with the evidence in a way that does not involve weighting. From these samples, we can infer the distribution over the query variables.

Gibbs sampling involves generating a sequence of samples, starting with an initial sample $x_{1:n}^{(1)}$ generated randomly with the evidence variables set to their observed values. The kth sample $x_{1:n}^{(k)}$ depends probabilistically on the previous sample $x_{1:n}^{(k-1)}$. We modify $x_{1:n}^{(k-1)}$ in-place to obtain $x_{1:n}^{(k)}$ as follows. Using any ordering of the unobserved variables, which need not be a topological sort, we assign $X_i^{(k)}$ a value from the distribution represented by $P(X_i \mid x_{-i}^{(k)})$. Here, $x_{-i}^{(k)}$ represents the values of all other variables except X_i in sample k. Sampling from $P(X_i \mid x_{-i}^{(k)})$ can be done efficiently because we only need to consider the Markov blanket of variable X_i (section 2.6).

Unlike the other sampling methods discussed so far, the samples produced by this method are not independent. However, it can be proven that, in the limit, samples are drawn exactly from the joint distribution over the unobserved variables given the observations. Algorithm 3.9 shows how to compute a factor for $P(X_i \mid x_{-i})$. Gibbs sampling is implemented in algorithm 3.10.

[8] Named for the American scientist Josiah Willard Gibbs (1839–1903). Together with James Clerk Maxwell and Ludwig Boltzman, he created the field of statistical mechanics.

```
function blanket(bn, a, i)
    name = bn.vars[i].name
    val = a[name]
    a = delete(a, name)
    Φ = filter(φ -> in_scope(name, φ), bn.factors)
    φ = prod(condition(φ, a) for φ in Φ)
    return normalize!(φ)
end
```

Algorithm 3.9. A method for obtaining $P(X_i \mid x_{-i})$ for a Bayesian network bn given a current assignment a.

```
function update_gibbs_sample(a, bn, evidence, ordering)
    for i in ordering
        name = bn.vars[i].name
        if !haskey(evidence, name)
            b = blanket(bn, a, i)
            a = merge(a, namedtuple(name)(rand(b)[name]))
        end
    end
    return a
end

function gibbs_sample(a, bn, evidence, ordering, m)
    for j in 1:m
        a = update_gibbs_sample(a, bn, evidence, ordering)
    end
    return a
end

struct GibbsSampling
    m_samples # number of samples to use
    m_burnin  # number of samples to discard during burn-in
    m_skip    # number of samples to skip for thinning
    ordering  # array of variable indices
end

function infer(M::GibbsSampling, bn, query, evidence)
    table = FactorTable()
    a = merge(rand(bn), evidence)
    a = gibbs_sample(a, bn, evidence, M.ordering, M.m_burnin)
    for i in 1:(M.m_samples)
        a = gibbs_sample(a, bn, evidence, M.ordering, M.m_skip)
        b = select(a, query)
        table[b] = get(table, b, 0) + 1
    end
    vars = filter(v->v.name ∈ query, bn.vars)
    return normalize!(Factor(vars, table))
end
```

Algorithm 3.10. Gibbs sampling implemented for a Bayesian network bn with evidence evidence and an ordering ordering. The method iteratively updates the assignment a for m iterations.

Gibbs sampling can be applied to our running example. We can use our m samples to estimate

$$P(b^1 \mid d^1, c^1) \approx \frac{1}{m} \sum_i (b^{(i)} = 1) \tag{3.21}$$

Figure 3.6 compares the convergence of the estimate of $P(c^1 \mid d^1)$ in the chemical detection network using direct, likelihood weighted, and Gibbs sampling. Direct sampling takes the longest to converge. The direct sampling curve has long periods during which the estimate does not change because samples are inconsistent with the observations. Likelihood-weighted sampling converges faster in this example. Spikes occur when a sample is generated with $C = 1$ and then gradually decrease. Gibbs sampling, in this example, quickly converges to the true value of 0.5.

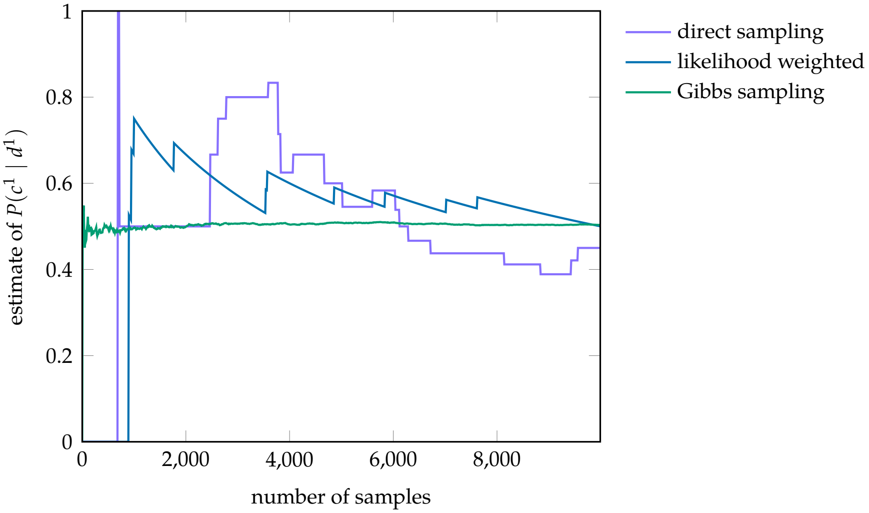


Figure 3.6. A comparison of sampling-based inference methods on the chemical detection network. Both likelihood-weighted and direct sampling have poor convergence due to the rarity of events, whereas Gibbs sampling is able to efficiently converge to the true value even with no burn-in period or thinning.

As mentioned earlier, Gibbs sampling, like other Markov chain Monte Carlo methods, produces samples from the desired distribution in the limit. In practice, we have to run Gibbs for some amount of time, called the *burn-in period*, before converging to a steady state distribution. The samples produced during burn-in are normally discarded. If many samples are to be used from a single Gibbs sampling series, it is common to *thin* the samples by only keeping every hth sample because of potential correlation between samples.

3.9 *Inference in Gaussian Models*

If the joint distribution is Gaussian, we can perform exact inference analytically. Two jointly Gaussian random variables $\mathbf{a}$ and $\mathbf{b}$ can be written

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_\mathbf{a} \\ \boldsymbol{\mu}_\mathbf{b} \end{bmatrix}, \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^\top & \mathbf{B} \end{bmatrix} \right) \tag{3.22}$$

The marginal distribution of a multivariate Gaussian is also Gaussian:

$$\mathbf{a} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{a}, \mathbf{A}) \qquad \mathbf{b} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{b}, \mathbf{B}) \tag{3.23}$$

The conditional distribution of a multivariate Gaussian is also Gaussian with a convenient closed-form solution:

$$p(\mathbf{a} \mid \mathbf{b}) = \mathcal{N}\left(\mathbf{a} \mid \boldsymbol{\mu}_{\mathbf{a}|\mathbf{b}}, \boldsymbol{\Sigma}_{\mathbf{a}|\mathbf{b}}\right) \tag{3.24}$$

$$\boldsymbol{\mu}_{\mathbf{a}|\mathbf{b}} = \boldsymbol{\mu}_\mathbf{a} + \mathbf{C}\mathbf{B}^{-1}(\mathbf{b} - \boldsymbol{\mu}_\mathbf{b}) \tag{3.25}$$

$$\boldsymbol{\Sigma}_{\mathbf{a}|\mathbf{b}} = \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top \tag{3.26}$$

Algorithm 3.11 shows how to use these equations to infer a distribution over a set of query variables given evidence. Example 3.7 illustrates how to extract the marginal and conditional distributions from a multivariate Gaussian.

```julia
function infer(D::MvNormal, query, evidencevars, evidence)
    μ, Σ = D.μ, D.Σ.mat
    b, μa, μb = evidence, μ[query], μ[evidencevars]
    A = Σ[query,query]
    B = Σ[evidencevars,evidencevars]
    C = Σ[query,evidencevars]
    μ = μ[query] + C * (B\(b - μb))
    Σ = A - C * (B \ C')
    return MvNormal(μ, Σ)
end
```

Algorithm 3.11. Inference in a multivariate Gaussian distribution `D`. A vector of integers specifies the query variables in the `query` argument, and a vector of integers specifies the evidence variables in the `evidencevars` argument. The values of the evidence variables are contained in the vector `evidence`. The `Distributions.jl` package defines the `MvNormal` distribution.

For example, consider

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}\right)$$

The marginal distribution for x_1 is $\mathcal{N}(0, 3)$, and the marginal distribution for x_2 is $\mathcal{N}(1, 2)$.

The conditional distribution for x_1 given $x_2 = 2$ is

$$\begin{aligned} \boldsymbol{\mu}_{x_1 \mid x_2 = 2} &= 0 + 1 \cdot 2^{-1} \cdot (2 - 1) = 0.5 \\ \boldsymbol{\Sigma}_{x_1 \mid x_2 = 2} &= 3 - 1 \cdot 2^{-1} \cdot 1 = 2.5 \\ x_1 \mid (x_2 = 2) &\sim \mathcal{N}(0.5, 2.5) \end{aligned}$$

We can perform this inference calculation using algorithm 3.11 by constructing the joint distribution

```
D = MvNormal([0.0,1.0],[3.0 1.0; 1.0 2.0])
```

and then calling `infer(D, [1], [2], [2.0])`.

Example 3.7. Marginal and conditional distributions for a multivariate Gaussian.

3.10 Summary

- Inference involves determining the probability of query variables given some evidence.

- Exact inference can be done by computing the joint distribution over the variables, setting evidence, and marginalizing out any hidden variables.

- Inference can be done efficiently in naive Bayes models, in which a single parent variable affects many conditionally independent children.

- The variable elimination algorithm can make exact inference more efficient by marginalizing out variables in sequence.

- Belief propagation represents another method for inference, in which information is iteratively passed between factors to arrive at a result.

- Inference in a Bayesian network can be shown to be NP-hard through a reduction to the 3SAT problem, motivating the development of approximate inference methods.

- Approximate inference can be done by directly sampling from the joint distribution represented by a Bayesian network, but it may involve discarding many samples that are inconsistent with the evidence.

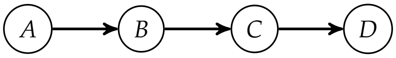
- Likelihood weighted sampling can reduce computation required for approximate inference by only generating samples that are consistent with the evidence and weighting each sample accordingly.

- Gibbs sampling generates a series of unweighted samples that are consistent with the evidence and can greatly speed approximate inference.

- Exact inference can be done efficiently through matrix operations when the joint distribution is Gaussian.

3.11 Exercises

Exercise 3.1. Given the following Bayesian network and its associated conditional probability distributions, write the equation required to perform exact inference for the query $P(a^1 \mid d^1)$.



Solution: We first expand the inference expression using the definition of conditional probability.

$$P(a^1 \mid d^1) = \frac{P(a^1, d^1)}{P(d^1)}$$

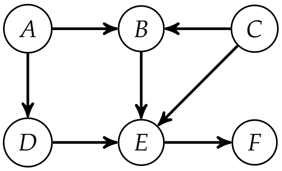
We can rewrite the numerator as a marginalization over the hidden variables and we can rewrite the denominator as a marginalization over both the hidden and query variables.

$$P(a^1 \mid d^1) = \frac{\sum_b \sum_c P(a^1, b, c, d^1)}{\sum_a \sum_b \sum_c P(a, b, c, d^1)}$$

The definition of the joint probability in both the numerator and the denominator can be rewritten using the chain rule for Bayesian networks and the resulting equation can be simplified by removing constants from inside the summations.

$$\begin{aligned} P(a^1 \mid d^1) &= \frac{\sum_b \sum_c P(a^1)P(b \mid a^1)P(c \mid b)P(d^1 \mid c)}{\sum_a \sum_b \sum_c P(a)P(b \mid a)P(c \mid b)P(d^1 \mid c)} \\ &= \frac{P(a^1) \sum_b \sum_c P(b \mid a^1)P(c \mid b)P(d^1 \mid c)}{\sum_a \sum_b \sum_c P(a)P(b \mid a)P(c \mid b)P(d^1 \mid c)} \\ &= \frac{P(a^1) \sum_b P(b \mid a^1) \sum_c P(c \mid b)P(d^1 \mid c)}{\sum_a P(a) \sum_b P(b \mid a) \sum_c P(c \mid b)P(d^1 \mid c)} \end{aligned}$$

Exercise 3.2. Given the following Bayesian network and its associated conditional probability distributions, write the equation required to perform exact inference for the query $P(c^1, d^1 \mid a^0, f^1)$.



Solution: We first expand the inference expression using the definition of conditional probability.

$$P(c^1, d^1 \mid a^0, f^1) = \frac{P(a^0, c^1, d^1, f^1)}{P(a^0, f^1)}$$

We can rewrite the numerator as a marginalization over the hidden variables and we can rewrite the denominator as a marginalization over both the hidden and query variables.

$$P(c^1, d^1 \mid a^0, f^1) = \frac{\sum_b \sum_e P(a^0, b, c^1, d^1, e, f^1)}{\sum_b \sum_c \sum_d \sum_e P(a^0, b, c, d, e, f^1)}$$

The definition of the joint probability in both the numerator and the denominator can be rewritten using the chain rule for Bayesian networks and the resulting equation can be simplified by removing constants from inside the summations. Note, there are multiple possible orderings of the summations in the final equation.

$$\begin{aligned}
P(c^1, d^1 \mid a^0, f^1) &= \frac{\sum_b \sum_e P(a^0)P(b \mid a^0, c^1)P(c^1)P(d^1 \mid a^0)P(e \mid b, c^1, d^1)P(f^1 \mid e)}{\sum_b \sum_c \sum_d \sum_e P(a^0)P(b \mid a^0, c)P(c)P(d \mid a^0)P(e \mid b, c, d)P(f^1 \mid e)} \\
&= \frac{P(a^0)P(c^1)P(d^1 \mid a^0) \sum_b \sum_e P(b \mid a^0, c^1)P(e \mid b, c^1, d^1)P(f^1 \mid e)}{P(a^0) \sum_b \sum_c \sum_d \sum_e P(b \mid a^0, c)P(c)P(d \mid a^0)P(e \mid b, c, d)P(f^1 \mid e)} \\
&= \frac{P(c^1)P(d^1 \mid a^0) \sum_b P(b \mid a^0, c^1) \sum_e P(e \mid b, c^1, d^1)P(f^1 \mid e)}{\sum_c P(c) \sum_b P(b \mid a^0, c) \sum_d P(d \mid a^0) \sum_e P(e \mid b, c, d)P(f^1 \mid e)}
\end{aligned}$$

Exercise 3.3. Suppose we are developing an object detection system for an autonomous vehicle driving in a city. Our autonomous vehicle's perception system reports an object's size S (either small, medium, or large) and an object's speed V (either slow, moderate, or fast). We want to design a model that will determine the class C of an object—either a vehicle, pedestrian, or a ball—given observations of the object's size and speed. Assuming a naive Bayes model with the following class prior and class-conditional distributions, what is the detected class given observations S = medium and V = slow?

C	$P(C)$
vehicle	0.80
pedestrian	0.19
ball	0.01

C	S	$P(S \mid C)$
vehicle	small	0.001
vehicle	medium	0.009
vehicle	large	0.990
pedestrian	small	0.200
pedestrian	medium	0.750
pedestrian	large	0.050
ball	small	0.800
ball	medium	0.199
ball	large	0.001

C	V	$P(V \mid C)$
vehicle	slow	0.2
vehicle	moderate	0.2
vehicle	fast	0.6
pedestrian	slow	0.5
pedestrian	moderate	0.4
pedestrian	fast	0.1
ball	slow	0.4
ball	moderate	0.4
ball	fast	0.2

Solution: To compute the posterior distribution $P(c \mid o_{1:n})$, we use the definition of the joint distribution for a naive Bayes model in equation (3.4):

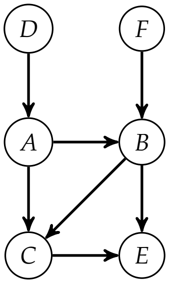
$$\begin{aligned}
P(c \mid o_{1:n}) &\propto P(c) \prod_{i=1}^{n} P(o_i \mid c) \\
P(\text{vehicle} \mid \text{medium}, \text{slow}) &\propto P(\text{vehicle}) P(S = \text{medium} \mid \text{vehicle}) P(V = \text{slow} \mid \text{vehicle}) \\
P(\text{vehicle} \mid \text{medium}, \text{slow}) &\propto (0.80)(0.009)(0.2) = 0.00144 \\
P(\text{pedestrian} \mid \text{medium}, \text{slow}) &\propto P(\text{pedestrian}) P(S = \text{medium} \mid \text{pedestrian}) P(V = \text{slow} \mid \text{pedestrian}) \\
P(\text{pedestrian} \mid \text{medium}, \text{slow}) &\propto (0.19)(0.75)(0.5) = 0.07125 \\
P(\text{ball} \mid \text{medium}, \text{slow}) &\propto P(\text{ball}) P(S = \text{medium} \mid \text{ball}) P(V = \text{slow} \mid \text{ball}) \\
P(\text{ball} \mid \text{medium}, \text{slow}) &\propto (0.01)(0.199)(0.4) = 0.000796
\end{aligned}$$

Since $P(\text{pedestrian} \mid \text{medium}, \text{slow})$ has the largest probability, the object is classified as a pedestrian.

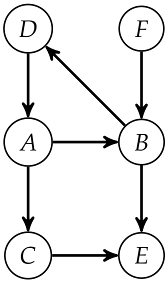
Exercise 3.4. Given the 3SAT formula in equation (3.14) and the Bayesian network structure in figure 3.4, what are the values of $P(c_3^1 \mid x_2^1, x_3^0, x_4^1)$ and $P(y^1 \mid d_2^1, c_3^0)$?

Solution: We have $P(c_3^1 \mid x_2^1, x_3^0, x_4^1) = 1$ because x_2^1, x_3^0, x_4^1 makes the third clause true, and $P(y^1 \mid d_2^1, c_3^0) = 0$ because $Y = 1$ requires that both D_2 and C_3 be true.

Exercise 3.5. Give a topological sort for each of the following directed graphs:



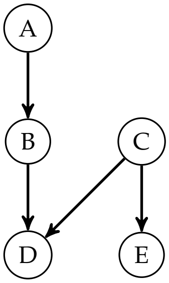
(1)



(2)

Solution: There are three valid topological sorts for the first directed graph (Bayesian network): (F, D, A, B, C, E), (D, A, F, B, C, E), and (D, F, A, B, C, E). There are no valid topological sorts for the second directed graph since it is cyclic.

Exercise 3.6. Suppose we have the following Bayesian network and we are interested in generating an approximation of the inference query $P(e^1 \mid b^0, d^1)$ using likelihood-weighted sampling. Given the following samples, write the expressions for each of the sample weights. Additionally, write the equation for estimating $P(e^1 \mid b^0, d^1)$ in terms of the sample weights w_i.



A	B	C	D	E
0	0	0	1	0
1	0	0	1	0
0	0	0	1	1
1	0	1	1	1
0	0	1	1	0
1	0	1	1	1

Solution: For likelihood-weighted sampling, the sample weights are the product of the distributions over evidence variables conditioned on the values of their parents. Thus, the general form for our weights is $P(b^0 \mid a)P(d^1 \mid b^0, c)$. We then match each of the values for each sample from the joint distribution.

A	B	C	D	E	Weight
0	0	0	1	0	$P(b^0 \mid a^0)P(d^1 \mid b^0, c^0)$
1	0	0	1	0	$P(b^0 \mid a^1)P(d^1 \mid b^0, c^0)$
0	0	0	1	1	$P(b^0 \mid a^0)P(d^1 \mid b^0, c^0)$
1	0	1	1	1	$P(b^0 \mid a^1)P(d^1 \mid b^0, c^1)$
0	0	1	1	0	$P(b^0 \mid a^0)P(d^1 \mid b^0, c^1)$
1	0	1	1	1	$P(b^0 \mid a^1)P(d^1 \mid b^0, c^1)$

To estimate $P(e^1 \mid b^0, d^1)$, we simply need to sum the weights of samples consistent with the query variable and divide this by the sum all of the weights.

$$P(e^1 \mid b^0, d^1) = \frac{\sum_i w_i(e^{(i)} = 1)}{\sum_i w_i} = \frac{w_3 + w_4 + w_6}{w_1 + w_2 + w_3 + w_4 + w_5 + w_6}$$

Exercise 3.7. Each year, we receive student scores on standardized mathematics M, reading R, and writing W exams. Using data from prior years, we create the following distribution:

$$\begin{bmatrix} M \\ R \\ W \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 81 \\ 82 \\ 80 \end{bmatrix}, \begin{bmatrix} 25 & -9 & -16 \\ -9 & 36 & 16 \\ -16 & 16 & 36 \end{bmatrix} \right)$$

Compute the parameters of the conditional distribution over a student's math and reading test scores, given a writing score of 90.

Solution: If we let $\mathbf{a}$ represent the vector of math and reading scores and $\mathbf{b}$ represent the writing score, the joint and conditional distributions are as follows:

$$\begin{aligned}
\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} &\sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{bmatrix}, \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^\top & \mathbf{B} \end{bmatrix} \right) \\
p(\mathbf{a} \mid \mathbf{b}) &= \mathcal{N}\left(\mathbf{a} \mid \boldsymbol{\mu}_{a|b}, \boldsymbol{\Sigma}_{a|b} \right) \\
\boldsymbol{\mu}_{a|b} &= \boldsymbol{\mu}_a + \mathbf{C}\mathbf{B}^{-1}(\mathbf{b} - \boldsymbol{\mu}_b) \\
\boldsymbol{\Sigma}_{a|b} &= \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top
\end{aligned}$$

In the example, we have the following definitions:

$$\boldsymbol{\mu}_a = \begin{bmatrix} 81 \\ 82 \end{bmatrix} \qquad \boldsymbol{\mu}_b = \begin{bmatrix} 80 \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} 25 & -9 \\ -9 & 36 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 36 \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} -16 \\ 16 \end{bmatrix}$$

Thus, the parameters of our conditional distribution given $\mathbf{b} = W = 90$ are:

$$\begin{aligned}
\boldsymbol{\mu}_{M,R|W=80} &= \begin{bmatrix} 81 \\ 82 \end{bmatrix} + \begin{bmatrix} -16 \\ 16 \end{bmatrix} \frac{1}{36}(90 - 80) \approx \begin{bmatrix} 76.5 \\ 86.4 \end{bmatrix} \\
\boldsymbol{\Sigma}_{M,R|W=80} &= \begin{bmatrix} 25 & -9 \\ -9 & 36 \end{bmatrix} - \begin{bmatrix} -16 \\ 16 \end{bmatrix} \frac{1}{36} \begin{bmatrix} -16 & 16 \end{bmatrix} \approx \begin{bmatrix} 25 & -9 \\ -9 & 36 \end{bmatrix} - \begin{bmatrix} 7.1 & -7.1 \\ -7.1 & 7.1 \end{bmatrix} = \begin{bmatrix} 17.9 & -1.9 \\ -1.9 & 28.9 \end{bmatrix}
\end{aligned}$$

Given that the student scores a 90 on the writing test, based on our conditional distribution, we expect the student to earn a 76.5 on the math test with a standard deviation of $\sqrt{17.9}$ and an 86.4 on the writing test with a standard deviation of $\sqrt{28.9}$.