Simulation - Sampling and Estimation

Author: Wes Cowan

Abstract

Shamelessly cribbed from what I remember of Sheldon Ross' *Simulations*, an excellent book. Some basic knowledge of probability in discrete and continuous settings is useful.

What does it mean to simulate something? In these notes, we take the view that to simulate something is *generate a population of data or samples subject to a desired statistical distribution.* We can, for instance, simulate a boolean random variable with equal likelihood of being true or false by flipping a fair coin a number of times (and interpreting heads as true, tails as false, etc). If we generate 100 values by this method, it is (relatively) unlikely that this population will have exactly half true, half false - but the population was generated by or simulated from the desired distribution.

Why simulate? In many cases, probabilistic questions can be computed and answered directly - we can compute exactly the probability of getting 7 heads out of 10 flips, for instance. But for much larger systems, especially systems with many interacting variables (large Bayesian networks, for instance), explicitly and exactly computing probabilities can be tremendously inefficient. It can be frequently useful to approximate answers by simulating large populations, and determining various frequencies and properties of the simulated population. If in a large Bayesian Network we are interested in knowing, for instance, the average number of variables that have a True value, we could simply generate a large population of sample assignments from the network, and count, over this population, the average number of True values in each sample. This is a much, much simpler operation than computing the true and explicit expected values.

The problem of generating randomness by a computer is a difficult one, and will largely not be addressed here. In general, we will take the view that a computer has two basic functions that it can perform well: generating a random bit 0 or 1 with equal probabilities, and generating a random value uniformly over the range $[0, 1]$. The focus of these notes is on using these tools to simulate from other distributions of interest. This will culminate in Markov Chain Monte Carlo simulation, particularly important for Bayesian Networks.

1 Sampling

1.1 Discrete Random Variables

A discrete random variable X has as a value or outcome in a discrete set (finite or infinite), with a specified probability for every possible outcome. In a finite case, for instance, we might have

$$X = \begin{cases} \text{Outcome}_1 & \text{with prob. } p_1 \\ \text{Outcome}_2 & \text{with prob. } p_2 \\ \dots & \dots \\ \text{Outcome}_n & \text{with prob. } p_n \end{cases} \tag{1}$$

It is worth noting a few things. One, the value of a discrete random variable need not be numerical. These could represent numbers, as in the outcomes of a die roll, or could represent any discrete objects, such as 'heads'/'tails', True/False, colors, cities, or courses of action. Two, the indexing here is relatively arbitrary (the ordering of the outcomes is not necessarily important); in many contexts it may be beneficial to index the outcomes from 0 to ∞, other times, starting at 1 may be useful. But even though the indexing may be arbitrary (though obvious/natural ones can exist), it can be useful to have a fixed ordering of outcomes. While the outcomes need not be numerical, it is notationally convenient to refer to Outcome_i simply as i.

1.1.1 Direct Sampling

How can we *generate a value of* X? For a fixed ordering of outcomes, define the 'cumulative probabilities' as

$$P_i = p_1 + p_2 + \ldots + p_i. \tag{2}$$

Note that $P_i \leq P_{i+1}$, and if there are n total outcomes, we will have $P_n = 1$ (or in the infinite case $P_n \to 1$ as $n \to \infty$). For any u in $0 \leq u \leq 1$, we can uniquely define the following outcome index

$$I(u) = \min \{i : P_i \geq u\}. \tag{3}$$

With this mapping between u values and indices, we can generate random outcomes in the following way: First, generate a random value $U \sim \text{Unif}[0, 1]$, and then take $X = I(U)$. If we want to generate a sequence of such X, we can simply generate a sequence $U_1, U_2, \ldots$ and generate $X_1, X_2, \ldots$ correspondingly. In this way, we can sample from the distribution of X.

What does this mean? For X constructed in this way, what can we say about $P(X = i)$ for any i? We have the following relations:

$$P(X = i) = P(I(U) = i) = P(P_i \geq U > P_{i-1}). \tag{4}$$

For a uniform random $[0, 1]$ random variable U, the probability that it falls between two values $a < b$ is simply $b - a$, hence

$$P(X = i) = P(I(U) = i) = P(P_i \geq U > P_{i-1}) = P_i - P_{i-1} = p_i. \tag{5}$$

So we see that under this scheme, turning values U into outcomes X, the probability of getting outcome i is exactly p_i, hence X constructed in this way follows the desired distribution.

A note on implementation: in terms of actually implementing this construction scheme, if the number of outcomes is small, it is not unreasonable to have the P_i pre-computed. In that case, simply generate U, and then compare it sequentially to these values to find the right interval $[P_{i-1}, P_i]$ that contains U. In the case of a large number of outcomes (potentially infinite), it is not unreasonable to first generate U, then enter into a loop iterating over the values of $i = 1, 2, 3, \ldots$, and tracking the sum of the p_i until the threshold of U is first crossed, then report this outcome.

Discrete Sampling Algorithm:

```
def generate_sample(p):
    U = rand()
    P = p(1)
    i = 1
    while P < U
        i += 1
        P += p(i)
    return i
```

1.1.2 Conditional Sampling

In many cases, we may not be interested in the distribution of X so much as the distribution of X *subject to some conditions*. The classic example of this is problems like "what is the probability a die roll comes up even?" Or "what is the probability that the sum of two dice is 7?" In these cases, there are some subset of the outcomes that we are interested in, which we can denote Conditions $\subset$ Outcomes. How can we generate samples of X according to its distribution, *given that the result should be in Conditions*?

One immediate approach to this is **rejection sampling**: repeatedly generate samples of X, rejecting them until you generate a sample in the set Conditions.

Rejection Sampling Algorithm:

```
def rejection_sample(p, Conditions):
    i = generate_sample(p)
    while i not in Conditions:
        i = generate_sample(p)
    return i
```

There are a couple of important points to consider about this algorithm. The most important of course is, does it work? If X' is generated via this algorithm, what can we say about $P(X' = i)$? There are a couple of ways to approach this, but one is to marginalize based on the number of times the loop executes in the algorithm. Let $p_c = P(X \in \text{Conditions})$. For any given sample, there is a probability p_c that the sample is in Conditions and that the loop terminates, and probability $1 - p_c$ that the loop doesn't terminate. Let K be the number of times the loop executed, and i_k the outcome generated on round k. Note, the i_k are independent of each other. Hence we have

$$\begin{aligned}
P(X' = i) &= \sum_{k=0}^{\infty} P(i_k = i \text{ and } K = k) \\
&= \sum_{k=0}^{\infty} P(i_0, i_1, \dots, i_{k-1} \notin \text{Conditions and } i_k = i) \\
&= \sum_{k=0}^{\infty} P(i_0, i_1, \dots, i_{k-1} \notin \text{Conditions}) P(i_k = i | i_0, i_1, \dots, i_{k-1} \notin \text{Conditions}) \\
&= \sum_{k=0}^{\infty} (1 - p_c)^k P(i_k = i) \\
&= \sum_{k=0}^{\infty} (1 - p_c)^k p_i \\
&= \frac{p_i}{1 - (1 - p_c)} \\
&= \frac{p_i}{p_c}.
\end{aligned} \tag{6}$$

Additionally, what can we say about $P(X = i | X \in \text{Conditions})$?

$$P(X = i | X \in \text{Conditions}) = \frac{P(X = i \text{ and } X \in \text{Conditions})}{P(X \in \text{Conditions})} = \frac{P(X = i)}{P(X \in \text{Conditions})} = \frac{p_i}{p_c}. \tag{7}$$

Hence we see that $P(X' = i) = P(X = i | X \in \text{Conditions})$, and X' obeys the *conditional* distribution of X given $X \in \text{Conditions}$.

One potential issue with this algorithm however is its inefficiency. If it was incredibly unlikely that $X \in$ Conditions in the first place, it may take many loops to generate a sample that is in the conditions set. This is potentially wasteful. An alternative is the following: if you can compute p_c easily (as the sum of all probabilities of outcomes in Conditions, you can *directly* sample from the conditional distribution by sampling outcome i with probability p_i / p_c

for any $i \in$ Conditions. In this case, we skip having to loop in the hopes of generating something in the desired set, directly sampling from the underlying conditional distribution. This does, however, require knowledge of what p_c is, in order to re-normalize the probabilities of the desired outcomes.

1.1.3 Transformations

In many instances, it is possible to generate random values of certain distributions by building them out of other distributions. For instance, the **Bernoulli** distribution ($X \sim \text{Bernoulli}(p)$) gives 1 with probability p and 0 with probability $1 - p$. These are easy to simulate given the techniques outlined above. The **Binomial** distribution $Y \sim \text{Bin}(n, p)$ gives a random value $Y \in \{0, 1, 2, \ldots, n\}$ where

$$P(Y = k) = \binom{n}{k} p^k (1-p)^{n-k}. \tag{8}$$

These could be sampled directly using the above schemes, but another way of viewing the Binomial distribution is as the sum of some n Bernoulli random values with parameter p:

$$Y \sim X_1 + X_2 + \ldots + X_n, \tag{9}$$

where $X_i \sim \text{Bernoulli}(p)$. Using this relationship, instead of working out the probabilities for the individual outcomes for the Binomial random variable, you could instead simply generate n appropriate Bernoulli random variables, and sum the result, outputting that sample of a Binomial random variable.

Similarly, the **Geometric** distribution ($X \sim \text{Geom}(p)$) has outcomes in the infinite set $\{0, 1, 2, 3, \ldots\}$, with the probability of any outcome being given by

$$P(X = k) = (1-p)^k p. \tag{10}$$

This again could be simulated according to the direct sampling method above, but an alternative view of the geometric distribution is essentially that it is 'the number of flips of a biased coin needed to get heads'. More concretely, the number of Bernoulli(p) random variables needed to be generated in a row before the first 1 appears. This has a natural implementation - simply loop, generating Bernoulli(p) random variables, counting the number of loops until the first 1 is generated. The number of loops may be output as a sample of a geometric distribution.

In many cases, it may simplify implementation if the distributions you are interested in can be composed from other simpler distributions.

1.2 Continuous Random Variables

For continuous random variables X, we generally take them to have values in the real numbers. As there are uncountably infinite possible values, we can no longer specify a probability associated with every outcome. Instead we specify the distribution of X through a *probability density function (p.d.f.)* f so that

$$P(a \leq X \leq b) = \int_a^b f(x)dx. \tag{11}$$

Note the requirements that $f(x) \geq 0$ everywhere, and $\int_{-\infty}^{\infty} f(x)dx = 1$ (*why?*). It is additionally useful to specify the *cumulative distribution function (c.d.f.)* F, given by

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(x')dx'. \tag{12}$$

Here are some common and useful continuous distributions:

- **The Uniform Distribution:** A random variable X has a value uniformly distributed on the interval $[a, b]$ if f is a constant over this interval, and 0 everywhere else. Note, the normalization requirement on f immediately gives that $f(x) = 1/(b - a)$ on this interval (*why?*). This frequently arises whenever a value is *equally likely to be anywhere in a given interval.* The 'typical' uniform distribution is over the interval $[0, 1]$ and has $f(x) = 1$ over this interval.

- **The Exponential Distribution:** A non-negative random variable X has an exponential distribution *with parameter* λ if for $x > 0$,
$$\mathbf{P}(X > x) = e^{-\lambda x}, \tag{13}$$
that is, the probability that X is particularly large drops off exponentially (relative to some constant parameter). Note we get immediately from this that $f(x) = \lambda e^{-\lambda x}$. Exponential distributions frequently arise because of the *memoryless property.* Imagine you are waiting for a bus. Ideally, the longer you waited for the bus to arrive, the more certain you would be that the bus would arrive 'soon' (or perhaps more certain the bus was not coming at all). However, the memoryless property would hold if, having waited some amount of time for the bus to arrive, you know no more about when the bus will come than you did when you first arrived at the bus station. In this sense, the past is 'forgotten' and tells you nothing about the future. Mathematically, this can be expressed as
$$\mathbf{P}(X > s + t | X > t) = \mathbf{P}(X > s), \tag{14}$$
or in terms of the example, given that you have waited t minutes, the probability that you will have to wait at least s *additional* minutes is the same as the probability of having to wait s minutes from the start. Under some weak assumptions, you can show that if X has the memoryless property, it must be an exponential distribution with some parameter λ. Exponential distributions frequently occur in temporal models.

- **The Normal Distribution:** The normal distribution arises frequently in statistics and application (hence the name), commonly referred to as 'bell curve'. A random variable X has a normal distribution if it is most likely to be around or near some value μ, and the probability of it being far away from μ drops off very quickly. In particular, normal distributions have a density function given by
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right). \tag{15}$$
Note that the smaller the value of σ^2, the faster the density drops off around μ - the tighter the resulting concentration, i.e., the more likely X is to be 'close' to μ. The standard normal distribution has a mean of $\mu = 0$, and a variance of $\sigma^2 = 1$.

1.2.1 Direct Sampling

Given a distribution of X specified by some density function f or distribution function F, the most direct way to generate samples from this distribution is via the *inverse function method.* Note that F maps from the real numbers to the interval $[0, 1]$, and is monotonically non-decreasing. Let F^{-1} denote the inverse function of F (if it exists). Consider the following algorithm: generate a uniform random variable on the interval $[0, 1]$, i.e., $U \sim \text{Unif}[0, 1]$, then take
$$X = F^{-1}(U). \tag{16}$$

Inverse Function Algorithm:

```
def inverse_function(F):
    u = rand()
    return F_inv( u )
```

What can we say about the distribution of this X? We have the following relations:

$$P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x) - 0 = F(x) \ \textit{why?} \tag{17}$$

Hence we see that X constructed in this way has the desired distribution. The inverse c.d.f. essentially transforms a uniform $[0, 1]$ random variable to the random variable we desire. As an example of this, note that for the exponential distribution with parameter λ, we have that $F(x) = 1 - e^{-\lambda x}$. In this case, we have that $F^{-1}(u) = -(1/\lambda)\ln(1 - u)$. Hence we can construct an exponential random variable with the indicated parameter simply taking

$$X = -\frac{1}{\lambda}\ln(1 - U). \tag{18}$$

Now it is worth noting two restrictions of this algorithm. One, the inverse function may not exist. This happens in particular if $f(x) = 0$ for any reasonable stretch, and there are essentially 'gaps' in which X will not fall. In this case, $F(x)$ is constant over these stretches, and the inverse is not defined. It is useful then to define the *left-inverse*,

$$F^*_{\text{inv}}(u) = \min\{x : F(x) \geq u\}. \tag{19}$$

The algorithm can proceed with this, to an equivalent result.

The second main restriction of this algorithm is that even if the c.d.f. F exists, it may not be usefully expressible. The c.d.f. of the normal distribution exists, but has no closed form expression, and neither does its inverse. As such, a significant computation may go in to computing the value of $F^{-1}(u)$ for any given value of u.

Suppose that we want to sample from a density function f, but cannot utilize the inverse function method. If we can sample from a density that is 'close' to f, we may be able to modify the sample to bring it more inline with f. Suppose that we can sample from a density g (by whatever means), and that $f(x)/g(x) \leq c$ for some positive constant c, for all the x where $g(x) > 0$. We say that g 'dominates' f in the sense that $f \leq cg$ everywhere. Consider the following algorithm:

Accept-Reject Algorithm:

```
def accept_reject(f,g,c):
    u = rand()
    y = sample_density(g)
    while u > f(y)/(c*g(y))
        u = rand()
        y = sample_density(g)
    return y
```

If X' is the output of this algorithm, what can we say about the distribution of X'? Again, it is useful to marginalize on the number of times the loop executes. Suppose that the algorithm generates a sequence $Y_0, Y_1, Y_2, \ldots$, each with density g, $U_0, U_1, \ldots$ with Unif$[0,1]$ distribution, and terminates after K loops. Note, $X' = Y_K$. We have

$$\begin{aligned}
P(X' \leq x) &= \sum_{k=0}^{\infty} P(X' \leq x, K = k) \\
&= \sum_{k=0}^{\infty} P\left(\frac{f(Y_0)}{cg(Y_0)} < U_0, \ldots, \frac{f(Y_{k-1})}{cg(Y_{k-1})} < U_{k-1} \text{ and } \frac{f(Y_k)}{cg(Y_k)} \geq U_k \text{ and } Y_k \leq x\right) \\
&= \sum_{k=0}^{\infty} P\left(\frac{f(Y)}{cg(Y)} < U\right)^k P\left(\frac{f(Y)}{cg(Y)} \geq U \text{ and } Y \leq x\right) \\
&= \frac{P\left(\frac{f(Y)}{cg(Y)} \geq U \text{ and } Y \leq x\right)}{1 - P\left(\frac{f(Y)}{cg(Y)} < U\right)} \\
&= \frac{P\left(\frac{f(Y)}{cg(Y)} \geq U \text{ and } Y \leq x\right)}{P\left(\frac{f(Y)}{cg(Y)} \geq U\right)}.
\end{aligned} \tag{20}$$

It remains to compute these probabilities. We have

$$P\left(\frac{f(Y)}{cg(Y)} \geq U\right) = \int_{-\infty}^{\infty} \int_0^{f(y)/(cg(y))} g(y) * 1 * du * dy = \int_{-\infty}^{\infty} \left[\frac{f(y)}{cg(y)} - 0\right] g(y) * dy = \int_{-\infty}^{\infty} f(y)/c * dy = 1/c, \tag{21}$$

and

$$P\left(\frac{f(Y)}{cg(Y)} \geq U \text{ and } Y \leq x\right) = \int_{-\infty}^{x} \int_0^{f(y)/(cg(y))} g(y) * 1 * du * dy = \int_{-\infty}^{x} f(y)/c * dy = F(x)/c, \tag{22}$$

which yields

$$P(X' \leq x) = (F(x)/c)/(1/c) = F(x). \tag{23}$$

Hence, we see that X' has the desired distribution.

An interesting note about this algorithm: f need not be an actual p.d.f. so much as 'capture the shape of the density we would like to have'. That is, we do not need to assume that $\int_{-\infty}^{\infty} f(x)dx = 1$. Suppose that $\int_{-\infty}^{\infty} f(x)dx = C$, so that $\tilde{f}(x) = f(x)/C$ is an actual density function. The c.d.f. $\tilde{F}$ would correspond to the distribution we actually want to sample from, f (and F) simply capture the 'shape'. What happens if we use the accept-reject method with f instead of $\tilde{f}$? If c is still chosen so that $f \leq cg$, then $f/(cg)$ will still be a value between 0 and 1. The only thing that changes is the following probabilities:

$$\begin{aligned}
&P\left(\frac{f(Y)}{cg(Y)} \geq U\right) \\
&= \int_{-\infty}^{\infty} \int_{0}^{f(y)/(cg(y))} g(y) * 1 * du * dy \\
&= \int_{-\infty}^{\infty} \left[\frac{f(y)}{cg(y)} - 0\right] g(y) * dy = \int_{-\infty}^{\infty} f(y)/c * dy = \int_{-\infty}^{\infty} C\tilde{f}(y)/c * dy = C/c.
\end{aligned} \tag{24}$$

and

$$\begin{aligned}
&P\left(\frac{f(Y)}{cg(Y)} \geq U \text{ and } Y \leq x\right) \\
&= \int_{-\infty}^{x} \int_{0}^{f(y)/(cg(y))} g(y) * 1 * du * dy = \int_{-\infty}^{x} f(y)/c * dy = \int_{-\infty}^{x} C\tilde{f}(y)/c * dy = (C/c)\tilde{F}(x),
\end{aligned} \tag{25}$$

which yields

$$P(X' \leq x) = ((C/c)\tilde{F}(x))/(C/c) = \tilde{F}(x). \tag{26}$$

Hence we see that even if the 'density' f is un-normalized, we still get the correct distribution ultimately. Note additionally, implementing this algorithm requires no knowledge of C in advance (C may be hard to compute, in many contexts). This is technique is tremendously useful when dealing with conditional densities.

A useful example of this is using exponential distributions to generate normal distributions. Consider a density given by

$$g(x) = \frac{\lambda}{2} e^{-\lambda|x|}. \tag{27}$$

If you consider the plot of this density, it is easy to see that it is simply a 'reflected' exponential distribution, with parameter λ, reflected around $x = 0$. To generate samples of this form, it is sufficient to generate an exponential random variable with parameter λ (as in Eq. (18)), and with probability 1/2 make it negative, or probability 1/2 keep it positive. Suppose we want to generate from the standard normal distribution, i.e.,

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}. \tag{28}$$

Comparing the densities, we have that

$$\frac{f(x)}{g(x)} = \frac{2}{\lambda} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2 + \lambda|x|} \leq \frac{2}{\lambda} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\lambda^2 + \lambda^2} = \frac{2}{\lambda} \frac{1}{\sqrt{2\pi}} e^{\frac{1}{2}\lambda^2} = c(\lambda). \tag{29}$$

At this point, it is worth asking - what is the best value for λ? The previous analysis shows that the probability of the loop successfully exiting at any point is $1/c$. In this case, we see that the probability that the loop successfully exits is given by

$$\frac{1}{c(\lambda)} = \frac{\lambda}{2}\sqrt{2\pi} e^{-\frac{1}{2}\lambda^2}. \tag{30}$$

If we want to maximize efficiency, we want to choose λ to make this quantity as large as possible. This occurs taking $\lambda^* = 1$, yielding a probability of success in any loop of

$$\frac{1}{c(\lambda^*)} = \frac{1}{2}\sqrt{2\pi}e^{-\frac{1}{2}} \leq 0.77. \tag{31}$$

This leads to the following algorithm for generating standard normals:

Standard Normal Generation:

```
def generate_reflected_exp():
    u = rand()
    y = -1 * ln(1-u)
    if rand() <= 0.5:
        return y
    else
        return -1 * y



def generate_standard_normal():
    u = rand()
    y = generate_reflected_exp()
    c = 0.77
    while u > f(y)/(c*g(y))
        u = rand()
        y = generate_reflected_exp()
    return y
```

1.2.2 Conditional Sampling

If we want to generate from a density f subject to some conditions, a natural way of going about it is simply the rejection algorithm as laid out in the discrete section - simply generate repeatedly from f, rejecting samples until the conditions are met.

It is worth noting though that the accept-reject algorithm of this section above can be viewed as a continuous generalization of the rejection algorithm for conditional discrete sampling. We can utilize it to the same effect here, to generate from conditional densities as well. In the extreme case, imagine that we wanted to generate X from density f, *subject to the condition that* $a \leq X \leq b$. In that case, we could consider sampling from a density g strictly over $[a, b]$ (a natural choice is a uniform distribution, as will be discussed in the next subsection) and running accept-reject against f.

As an example of this, suppose that you wanted to generate standard normal values, subject to the condition that they are greater than 10. It is worth noting that the probability that a directly sampled standard normal is greater than 10 is less than $7 * 10^{-24}$. If you were to try to generate from this conditional distribution *purely* by generating standard normals and collecting the ones that fall above 10, you will be waiting a very long time to collect any samples at all. Instead, imagine trying to generate according to the density function

$$f(x) = \begin{cases} 0 & \text{if } x < 10 \\ \frac{1}{\sqrt{2\pi}}e^{-(1/2)x^2} & \text{else.} \end{cases} \tag{32}$$

Now, this is not a 'proper' density function as it will be un-normalized, but as seen above the accept-reject method may be applied to un-normalized densities as well to the same effect. Imagine generating a random value Y according to the distribution

$$g(x) = \begin{cases} 0 & \text{if } x < 10 \\ 10e^{-10(x-10)} & \text{else.} \end{cases} \tag{33}$$

This can be done efficiently by generating an exponential random variable with parameter 10 and just increasing it by 10: $Y = 10 + \text{Exp}(10)$. Note that in this instance, *Every sample generated will be at least* 10. We get that for $x \geq 10$,

$$\frac{f(x)}{g(x)} = \frac{1}{\sqrt{2\pi}} e^{-x^2/2+10(x-10)} \leq \frac{1}{\sqrt{2\pi}} e^{-50} \approx 7.7 * 10^{-23}. \tag{34}$$

The above gives us the necessary c value, which is indeed quite small, but it is worth noting that in the un-normalized case, the probability of the accept-reject loop terminating is actually C/c, where $1/C$ is the normalization constant for f. In this case, we have that

$$\frac{C}{c} = \frac{\int_{10}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx}{e^{-50}/\sqrt{2\pi}} \approx 0.1. \tag{35}$$

This is a remarkable improvement. Essentially, under the naive reject method that simply rejects any sample above 10, the probability of accepting any sample is $7 * 10^{-24}$. Under the improved accept-reject method, the probability of accepting any sample is approximately 0.1. This is a vast improvement in the efficiency of the algorithm.

1.2.3 Transformations

It is frequently useful to avoid the above algorithms (when possible) and construct samples from samples of other distributions. For instance, in order to generate $X \sim \text{Unif}[a, b]$, this can be constructed from a $\text{Unif}[0, 1]$ by taking

$$X = a + (b - a)U. \tag{36}$$

This samples from the desired distribution (*why?*) and avoids other more complex sampling techniques.

Similarly, in order to sample a normal distribution with mean μ and variance σ^2, consider generating a standard normal Z (mean 0, variance 1) and taking

$$X = \mu + \sigma Z. \tag{37}$$

To show that this has the desired distribution, consider taking

$$P(X \leq x) = P(\mu + \sigma Z \leq x) = P(Z \leq (x - \mu)/\sigma) = \int_{-\infty}^{(x-\mu)/\sigma} \frac{1}{\sqrt{2\pi}} e^{-z^2/2} dz, \tag{38}$$

and showing that this can be expressed equivalently as

$$P(X \leq x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(x'-\mu)^2} dx'. \tag{39}$$

Many other distributions of interest can be generated from simpler building blocks - for instance the $X \sim \chi_n^2$ distribution can be sampled directly from its density using the techniques of this section, or it can be efficiently generated by generating n standard normal samples $Z_1, \ldots, Z_n$ and taking

$$X = Z_1^2 + Z_2^2 + \ldots + Z_n^2. \tag{40}$$

Depending on the specific context, it is frequently worthwhile to consider what the fundamental building blocks of your desired distribution is.

1.3 Sets and Sequences of Variables, Sampling in High Dimensions

The techniques of the previous section are incredibly useful when you want to generate independent sequences of one-dimensional random values, for instance repeated (independent) samples from the normal distribution. These are useful in a wide variety of applications. However, it is frequently the case that we want to generate a number of related or dependent values. For instance you might want to simulate a person, as described by (height, weight, age, gender identity, income bracket), all of which might have (potentially complex) relationships between each other. In another instance, you might want to generate a sequence $X_1, X_2, X_3, \dots$ where X_t represents some measurement or quantity that changes over time, and depends on all the previous days.

This section will focus primarily on discrete valued random variables, but much of the discussion extends to continuous distributions as well. Frequently, continuous distributions can be effectively approximated by discrete distributions, chunking or blocking the possible continuous ranges into discrete sections.

1.3.1 Direct Sampling

In some cases, we may know the probabilities of all possible outcomes: we might know the exact probability that someone will be a certain height, weight, age, gender identity, and income bracket, for all possible values of these variables. In this case (while the number of such outcomes could potentially be enormous) we could utilize any of the direct sampling methods, effectively generating values for all of the variables simultaneously.

This requires knowing the full 'joint' probability distribution, describing all possible ways all possible values of all possible variables might interact. In many cases, this information might not be available. However, we might know the base rate in the general population for people identifying as male or female. We could then generate a value for gender based on this probability. We might further know the *conditional* distribution of age among men and women separately. Having already generated a value for gender, we could utilize the corresponding conditional distribution for age, and generate a value for age accordingly. Now armed with a gender and an age, the conditional distribution of height, based on gender and age, might be well understood - we could utilize this conditional distribution to generate a height, and in this way sequentially generate values for all the variables we are interested in.

In general, suppose that we are interested in the variables $X_1, \dots, X_n$, but the full joint distribution $P(X_1, \dots, X_n)$ is unknown. However, the variables are arranged such that the base rate of the first is known, $P(X_1)$, which allows us to generate a sample value for X_1. Then for any $k >$, suppose that at any point the dependence of X_k on all prior variables is known:

$$P(X_k | X_1, X_2, \dots, X_{k-1}). \tag{41}$$

In this case, having generated $X_1, \dots, X_{k-1}$, we can use the above conditional distribution to generate a value of X_k, iterating over all variables until all $X_1, X_2, \dots, X_n$ have been assigned. To see that this works, note that we have

$$\begin{aligned} &P(X_1 = x_1, X_2 = x_2, \dots, X_n = X_n) \\ &= P(X_1 = x_1) P(X_2 = x_2 | X_1 = x_1) \dots P(X_n = x_n | X_1 = x_1, \dots, X_{n-1} = x_{n-1}) \\ &= P(X_1 = x_1) \prod_{t=2}^{n} P(X_t = x_t | X_1 = x_1, \dots, X_{t-1} = x_{t-1}). \end{aligned} \tag{42}$$

Hence, the correct probability for a given outcome is achieved, sequentially generating based on the conditional probability distributions.

This is particularly useful when the dependence of one variable on the previous variables is particularly simple. In the case of a **Markov chain**, for instance, in an infinite sequence of random variables, X_{t+1} depends only on $X_0, X_1, \dots, X_t$ only through X_t.

$$P(X_{t+1} | X_0, X_1, \dots, X_t) = P(X_{t+1} | X_t). \tag{43}$$

If the transition law obeys this simple dependence structure, it is very efficient to simulate one of these sequences - starting with the initial state X_0 and sequentially updating it based on the conditional probability law, using any of the direct sampling techniques described so far.

Sampling a Vector:

```
def generate_x_sequence(int n):
    x[1] = generate_base_sample()
    for i = 2, 3, ..., n
        x[i] = generate_conditional_sample( i, [x[1], x[2], ..., x[i-1] ])
    return x
```

1.3.2 Conditional Sampling

Suppose that we are interested in generating values for $(X_1, \ldots, X_n)$, subject to some constraints, in the form of a partial assignment of some variables. For instance, sample from the conditional distribution of $(X_1, \ldots, X_n)$, given that X_3 is false, $X_1 0$ is true, and $X_1 1 = X_5$. One approach to this is simply the rejection method of previous sections - simply repeatedly generate samples until one is generated that satisfies the conditions, and return that sample. The method carries over directly - it is in fact independent of the dimension of the underlying distribution.

But it runs into the same problem as addressed previously - inefficiency. We may need to generate impossibly large numbers of samples in order to capture any that satisfy the conditions at all. However, we have a mechanism in this case for building samples directly to satisfy the constraints. You could imagine running the following algorithm:

(Proposed) Sampling a Conditioned Vector:

```
def generate_conditioned_sequence(int n, constraints):
    if constraints specify X1:
        set x[1] by constraints
    else:
        x[1] = generate_base_sample()

    for i = 2, 3, ..., n
        if constraints specify Xi given x[1], ..., x[i-1]:
            set x[i] by constraints, x[1], ..., x[i-1]
        else:
            x[i] = generate_conditional_sample( i, [x[1], x[2], ..., x[i-1] ])
    return x
```

Given some set Constraints, and a vector $(X'_1, \ldots, X'_n)$ generated by the above algorithm, we know that X' will satisfy Constraints, but what can be said about $P(X'_1 = x_1, \ldots, X'_n = x_n)$? We can factor it in the following way:

$$P(X'_1 = x_1, \ldots, X'_n = x_n) = \prod_{i \notin \text{Constraints}} P(X_i = x_i | X_{1,\ldots,i-1} = x_{1,\ldots,i-1}). \tag{44}$$

But how does this compare to $P(X_1 = x_1, \ldots, X_n = x_n | \text{Constraints})$?

$$\begin{aligned}
&P(X_1 = x_1, \ldots, X_n = x_n | \text{Constraints}) \\
&= P(X_1 = x_1, \ldots, X_n = x_n \text{ and Constraints}) / P(\text{Constraints}) \\
&= P(X_1 = x_1, \ldots, X_n = x_n) / P(\text{Constraints}) \\
&= \frac{1}{P(\text{Constraints})} \prod_i P(X_i | X_{1,\ldots,i-1} = x_{1,\ldots,i-1}) \\
&= \frac{1}{P(\text{Constraints})} \prod_{i \notin \text{Constraints}} P(X_i = x_i | X_{1,\ldots,i-1} = x_{1,\ldots,i-1}) \prod_{i \in \text{Constraints}} P(X_i = x_i | X_{1,\ldots,i-1} = x_{1,\ldots,i-1}) \\
&= \frac{1}{P(\text{Constraints})} P(X'_1 = x_1, \ldots, X'_n = x_n) \prod_{i \in \text{Constraints}} P(X_i = x_i | X_{1,\ldots,i-1} = x_{1,\ldots,i-1})
\end{aligned} \tag{45}$$

Hence we see that

$$P(X'_1 = x_1, \ldots, X'_n = x_n) \neq P(X_1 = x_1, \ldots, X_n = x_n | \text{Constraints}), \tag{46}$$

But instead the two differ by a normalization constant $1/P(\text{Constraints})$, and a factor that depends on the specific values (potentially of all variables), which we will denote:

$$\text{lik}(x_1, \ldots, x_n) = \prod_{i \in \text{Constraints}} P(X_i = x_i | X_{1,\ldots,i-1} = x_{1,\ldots,i-1}). \tag{47}$$

This is the 'likelihood of the evidence' - essentially, given the values of all the x_i, this is the probability of having generated the values set by the constraints. Hence we get

$$P(X_1 = x_1, \ldots, X_n = x_n | \text{Constraints}) = \alpha P(X'_1 = x_1, \ldots, X'_n = x_n) \text{lik}(x_1, \ldots, x_n) \tag{48}$$

Hence we see that the frequency of a given sample $(x_1, \ldots, x_n)$ generated by this algorithm does not reflect the true conditional probability, but the frequency of a given sample *supplemented by the likelihood of the evidence* will (subject to normalization). It is convenient in this case to augment the earlier algorithm to reflect this:

Likelihood Weighting for Conditional Sampling:

```
def generate_conditioned_sequence(int n, constraints):
    lik = 1

    if constraints specify X1:
        set x[1] by constraints
        lik *= P( x[1] )
    else:
        x[1] = generate_base_sample()

    for i = 2, 3, ..., n
        if constraints specify Xi given x[1], ..., x[i-1]:
            set x[i] by constraints, x[1], ..., x[i-1]
            lik *= P( x[i] given x[1], ..., x[i-1] )
        else:
            x[i] = generate_conditional_sample( i, [x[1], x[2], ..., x[i-1] ])
    return (x, lik)
```

1.4 Markov Chain Monte Carlo

Consider the following problem: generate a binary sequence 100 bits in length, where the probability for a given sequence is proportional to the number of times it switches from 0 to 1 or 1 to 0. In this case, we have a discrete set of possible outcomes (the set of 100-bit binary sequences) and at least notionally a way to assign probabilities to each of them. Thus, in theory, we could use any of the previous methods to try to sample from this distribution.

However, we are hampered in the following way: one, the set of possible outcomes is simply enormous (2^{100}) and difficult to iterate over; two, while we have a notion of comparative frequencies ($0101\ldots0101$ should be highly likely, $000\ldots000$ impossible), computing the exact probability associated with a single outcome is incredibly difficult. If S is the set of all such sequences, and $f(s)$ counts the number of switches in string s, we have that

$$p_s = \frac{f(s)}{\sum_{s' \in S} f(s')}, \tag{49}$$

where computing the denominator (or in general, the normalization constant) is incredibly difficult, barring some clever combinatorial arguments. Even a sequential approach, utilizing conditional distributions for successive bits, would be difficult, as determining the conditional distributions would be very difficult.

But one thing we can do is make local comparisons: (looking at 10 bits instead of 100), we can say that $s_1 = 0000011111$ should be exactly half as likely as $s_2 = 0001110000$, as

$$\frac{p_{s_1}}{p_{s_2}} = \frac{f(s_1)}{f(s_2)} = \frac{1}{2}. \tag{50}$$

In particular, notice that local comparisons in this way meant that we did not have to compute the underlying normalization constant. But can we use local comparisons like this to sample from the underlying distribution?

Enter Markov Chain Monte Carlo.

1.4.1 Markov Chains and Stationary Distributions

A Markov chain (discussed briefly above) can be thought of as a sequence of states evolving over time. The system is in state S_1 at time $t = 1$, and evolves to some state S_2 at time $t = 2$, etc, constructing an infinite sequence $S_1, S_2, S_3, \ldots$ over some (finite) state space S. It is convenient to denote the states as $1, 2, \ldots, N$. The classic example frequently cited is a frog, jumping from lilypad to lilypad in a pond (the state space S in this case being the set of possible lilypads). In particular, what makes it a Markov chain is that the next state at any time depends (probabilistically) only on the current state:

$$P(S_{t+1} | S_1, S_2, \ldots, S_t) = P(S_{t+1} | S_t), \tag{51}$$

and in fact, this transition probability is independent of time, and depends only on the relative states at these two moments:

$$P(S_{t+1} = i | S_t = j) = p_{i,j}. \tag{52}$$

The matrix $P = [p_{i,j}]_{i,j \in \{1,\ldots,N\}}$ is referred to as the transition matrix. The transition matrix can be thought of as taking an initial distribution over the states (probability that the frog is on any of the lilypads) and pushing it forward in time, mixing or spreading those probabilities over states. What can be said about the possible state at time $t = 2$?

$$\begin{aligned} P(S_2 = i) &= \sum_j P(S_1 = j, S_2 = i) \\ &= \sum_j P(S_1 = j) P(S_2 = i | S_1 = j) \\ &= \sum_j P(S_1 = j) p_{i,j}. \end{aligned} \tag{53}$$

So we see that the distribution at time $t = 2$ is the distribution at time $t = 1$ 'mixed' over all possible intermediary or prior states. In practice, to link this to the example at the start of this section, we are particularly interested in the transition probabilities between states and some set of similar 'neighbors' - this is a local behavior that may be well understood, and connects strongly to the global behavior of the system.

A **stationary** distribution is one that doesn't change over time - the distribution in one time step matches the distribution in the previous timestep. If π_i is a 'stationary' probability of being in state i, by the above we must have:

$$\pi_i = \sum_j \pi_j p_{i,j}. \tag{54}$$

With some weak assumptions on P, it can be shown that the stationary distribution for a Markov chain exists, and is unique, and in fact satisfies the following relations: that regardless of the initial distribution on S_1, the 'long term' behavior is that the limiting probability of the system being in state i is π_i, and the fraction of time the system spends in state i is π_i.

This fact suggests the following sampling mechanism: if we wanted to sample outcome i with probability π_i, we could construct a Markov chain with transition matrix P, run it for a long time, and then sample its state. With (approximately, depending on time scales) probability π_i, it will be in state i. If we want to generate multiple samples, this can be done by running the Markov chain even further, and sampling the state again. In this way, we can generate many samples from the underlying distribution π over the state space S.

Typical analysis of Markov chains is of the form: given a transition matrix P, what is the underlying long term stationary distribution π? For the purpose of simulating distributions, we can invert this in the following way: *given* a distribution over the state space S, can we come up with a transition law that yields that distribution as the stationary distribution? Formally, for any $i \in S$, let q_i be a desired probability of outcome i. Can we construct a transition law P such that for all i,

$$q_i = \sum_j q_j p_{i,j}? \tag{55}$$

If so, we can sample from the distribution q simply by constructing the indicated Markov chain and repeatedly sampling along its trajectory. This technique is at the heart of what is known as Markov Chain Monte Carlo techniques.

Note, we do not even have to know the desired distribution q exactly - suppose we knew q up to some normalization constant, for instance $q_s = Cf(s)$ for some unknown constant C but known function f (the situation at the beginning of this section). Looking at the above stationary relations, the normalization constant C cancels out, and we are left with the problem of finding a transition matrix P such that

$$f(i) = \sum_j f(j) p_{i,j}. \tag{56}$$

The space of possible transition matrices is quite large; we can simplify our problem somewhat by restricting the set of transition matrices we are looking over. One way to do this is to restrict to transition matrices that satisfy

Detailed Balance: one way to think about the stationary distribution equation $q_i = \sum_j q_j p_{i,j}$ is to take q_i as an amount of weight assigned to state i, and $p_{i,j}$ to be the fraction of the weight at state j that flows to state i in one time step. This equation then says that the total weight on i after one time step is simply the sum of all the 'inflow' to that state, from every other state. Stationarity is simply that this distribution of weight does not change over time - the amount after one time step is the amount the state started with. The property of detailed balance takes this one step further (as a strong restriction on P) and says that *between any two states i, j, the amount flowing from i to j is exactly the amount flowing from j to i*, i.e., for any pair (i, j):

$$q_i p_{j,i} = q_j p_{i,j}. \tag{57}$$

Given a desired distribution q (or the corresponding f sans normalization constant), if we can construct a transition law that satisfies detailed balance, we get immediately that q must be a stationary distribution for that transition law. **Why?** Constructing from q (or f) such a transition law is the goal of the next section.

1.4.2 The Metropolis-Hastings Algorithm

The general approach of the Metropolis-Hastings algorithm is similar to the acceptance-rejection algorithm of the previous sections. Given a current state j, we will propose a transition to some neighbor i, and then either accept or reject the transition with some probability. Rejection, in this case, can be interpreted as 'the state transitions to itself'. These two mechanisms, when combined, should hopefully produce transition probabilities that satisfy detailed balance.

Given an initial state j, suppose that transition to state i is proposed with probability $Q_{i,j}$. In the example at the beginning of this section, we might pick a bit at random, and consider flipping it from a 0 to a 1 or vice versa. In that example, the proposed transition probabilities are easy to define, based on simple local relationships between bit strings. The 'proposed' transition mechanism can be relatively arbitrary, in the sense that whatever the transition mechanism is, we will choose an acceptance-rejection mechanism for tailoring it appropriately. Hence we may take $Q_{i,j}$ as given, in terms of whatever is natural or reasonable in the context of the problem.

Given q (or f) and a proposal matrix $Q_{i,j}$, suppose that the transition is accepted with probability $A_{i,j}$. In that case, we can argue that, taking K as the number of times the proposal is rejected:

$$\begin{aligned} p_{i,j} &= P(j \to i) \\ &= P(i \text{ is proposed from } j \text{ and } i \text{ is accepted }) \\ &= P(i \text{ is proposed from } j) P(i \text{ is accepted from } j | i \text{ is proposed from } j) \\ &= Q_{i,j} A_{i,j}. \end{aligned} \tag{58}$$

Plugging this into the detailed balance equations:

$$q_i Q_{j,i} A_{j,i} = q_j Q_{i,j} A_{i,j}, \tag{59}$$

or, noting that q and Q are effectively given,

$$\frac{A_{i,j}}{A_{j,i}} = \frac{q_i Q_{j,i}}{q_j Q_{i,j}} \left(= \frac{f(i) Q_{j,i}}{f(j) Q_{i,j}} \right). \tag{60}$$

It remains to choose $A_{i,j}$ and $A_{j,i}$ to make this true for all i, j pairs. A sufficient choice is the following:

$$\begin{aligned} A_{i,j} &= \min\left(1, \frac{q_i Q_{j,i}}{q_j Q_{i,j}}\right) \\ A_{j,i} &= \max\left(1, \frac{q_j Q_{i,j}}{q_i Q_{j,i}}\right) \end{aligned} \tag{61}$$

This leads to the following algorithm:

Metropolis-Hastings:

```
def metropolis_hastings(int n, initial_state, f, Q):
    j = initial_state
    for t = 1, 2, ..., n:
        i = generate neighbor of j with probability Q
        a = min( 1, ( f(i) Q[j,i] ) / ( f(j) Q[i,j] ) )
        if rand () <= a:
            j = i
    return j
```

In the example at the start of this section: for any string s there are 100 possible neighbors (picking any bit at random and flipping it), each equally likely to be chosen. Hence we have that $Q_{i,j} = 1/100$ for any i, j. For any string s and proposed neighbor s' (which differs by one bit from s), the probability of accepting will simply be $a = \min(1, f(s')/f(s))$. Generating neighbors in this way and accepting them as the next 'state' with this probability will, in the long term, converge to a stationary distribution where the probability of string s occurring will be proportional to $f(s)$. Note, in this particular example, transitions to *better* strings will *always* be accepted, but transitions to worse strings will only be accepted with some probability. Hence the states will gravitate towards high value strings, but occasionally move through lower value areas, in a sort of directed random walk. In this way, we transform local comparisons to global effect.

1.4.3 The Metropolis-Hastings Algorithm for Conditional Distributions

Suppose that we are interested in simulating the vector $(X_1, X_2, \ldots, X_N)$ subject to some Conditions. This is the same problem faced in a previous section, but here we consider the problem from the perspective of MCMC techniques. We can consider the 'state' of the system to be an assignment of each variable, and we can consider transitions from one state to another to another by tweaking variable assignments.

If we want to sample according to $P(X_1, X_2, \ldots, X_N)$ (the *unconditional* distribution), we can start at some state $(x_1, x_2, \ldots, x_N)$, pick one of the variables uniformly at random, and change the value of that variable. If we restrict ourselves to boolean random variables, we can consider simply picking one of the variables and swapping it with its negation. In this case, the probability of selecting $(x_1, \ldots, \neg x_j, \ldots, x_N)$ as the next state to $(x_1, \ldots, x_j, \ldots, x_N)$ is simply $1/N$. Further, the acceptance probability will simply be

$$A_{x',x} = \min\left(1, \frac{P(X_1 = x_1, \ldots, X_j = \neg x_j, \ldots, X_N = x_N)}{P(X_1 = x_1, \ldots, X_j = x_j, \ldots, X_N = x_N)}\right) \tag{62}$$

In this way, we don't have to be able to sample globally from the distribution, *we simply have to be able to evaluate the distribution for any proposed set of values.*

We can extend this to conditional probabilities as well. Suppose we want to sample from the distribution of

$$P(X_1, X_2, \ldots, X_N | \text{Conditions}) \tag{63}$$

In this case, the only thing that needs to change is that we **never try to transition to a set of values that violates Conditions** (effectively taking the probability of such a transition to be 0 from all sources). This is due to the following relation for Conditions-satisfying assignments:

$$\frac{P(X_1 = x_1, \ldots, X_j = \neg x_j, \ldots, X_N = x_N | \text{Conditions})}{P(X_1 = x_1, \ldots, X_j = x_j, \ldots, X_N = x_N | \text{Conditions})} = \frac{P(X_1 = x_1, \ldots, X_j = \neg x_j, \ldots, X_N = x_N)}{P(X_1 = x_1, \ldots, X_j = x_j, \ldots, X_N = x_N)}. \tag{64}$$

Why does this relation hold?

The fact that Metropolis-Hastings and related algorithms are invariant/ignorant of the normalization constant makes them tremendously useful for sampling from conditional distributions.

1.5 Bayesian Networks

Much of the previous discussion on sampling sets of variables directly, sequentially, conditionally, or using MCMC techniques, carries over immediately to the notion of sampling from or simulating over Bayesian Networks. The only thing that really changes is that the expression of the joint probabilities, or the conditional distribution factors, can simplify dramatically.

In particular, in applying Metropolis-Hastings (or the related Gibbs Sampler), when examining the ratio of probabilities for two proposed value assignments, one can argue that the dependence of every variable except those in the *Markov blanket* of the altered variable cancel out entirely.

2 Estimation

In the previous section, we were concerned with the ability to generate samples according to some specified distribution. In this section, we want to use those generated samples to estimate things about this distribution. In political science, for instance, we might simulate a large number of voters, and from this simulate the results of an election. By repeating this experiment multiple times, (if our underlying assumptions about voter distributions and dependencies is correct), we may estimate the probability that a given candidate will win or measure will pass.

In general, if X is a random variable known to have some distribution, we may be interested in estimating the following four types of quantities:

- Probabilities of events involving X, $P(\text{Event})$.
- Expected or average outcomes involving X, $E[g(X)]$ for some function g.
- Conditional probabilities of events involving X, $P(\text{ Event } \mid \text{ Conditions })$.
- Conditional expectations involving X, $E[g(X) \mid \text{ Conditions }]$.

Suppose X were, for instance, a large vector describing the vote of every voter in a given population (or abstention). We might be interested, for instance, in the probability that Candidate A wins. We could estimate this by simulating the vote a large number of times, generating a large number of independent samples $X_1, X_2, \ldots, X_n$. We could then utilize the following estimation $P(A \text{ wins under vote } X) \approx \hat{p}_{A \text{ wins under vote } X}$, where

$$\hat{p}_{A \text{ wins under vote } X} = \frac{1}{n} \sum_{t=1}^{n} I\left[A \text{ wins under vote } X_t\right], \tag{65}$$

where I is the indicator function that is 1 if the specified event occurs, 0 if it does not. Note, we have that

$$\begin{aligned}
&E[\hat{p}] \\
&= E\left[\frac{1}{n}\sum_{t=1}^{n} I\left[A \text{ wins under vote } X_t\right]\right] \\
&= \frac{1}{n}\sum_{t=1}^{n} E\left[I\left[A \text{ wins under vote } X_t\right]\right] \\
&= \frac{1}{n}\sum_{t=1}^{n} \left[0 * P(A \text{ does not win under vote } X) + 1 * P(A \text{ wins under vote } X)\right] \\
&= \frac{1}{n}\sum_{t=1}^{n} P(A \text{ wins under vote } X) \\
&= P(A \text{ wins under vote } X)
\end{aligned} \tag{66}$$

So we see that the expected value of our estimator is precisely the thing we want to estimate (a useful property of any estimator). What can we say about the error on this estimator? Let $p = P(A \text{ wins under vote } X)$ for notational convenience. A frequent measure of error on an estimator is the *mean squared error*, in this case,

$$\text{MSE}[\hat{p}] = E\left[(\hat{p} - p)^2\right] \tag{67}$$

Since $E[\hat{p}] = p$, the above is equivalent to saying that $\text{MSE}[\hat{p}] = \text{Var}(\hat{p})$, the variance of the estimator. Using the usual properties of variance and indicator random variables, it can be shown that

$$\text{MSE}[\hat{p}] = \text{Var}(\hat{p}) = \frac{p(1-p)}{n} \leq \frac{1/4}{n}. \tag{68}$$

In this case, the variance of our estimator goes down like $1/n$ where n is the number of samples we take. If we double the number of samples, we halve the variance of our estimator, and halve the mean squared error. (An equivalent statement to this: to halve the *standard deviation* of our estimator, we need to take four times as many samples.)

If we want to estimate some function of the distribution, for instance the spread between the votes gained by Candidate A and those gained by Candidate B, we could define a function $g(X)$ to give the spread of vote X, and then estimate $\hat{g} \approx E[g(X)]$ as

$$\hat{g} = \frac{1}{n}\sum_{t=1}^{n} g(X_t). \tag{69}$$

Similar to the previous case (for good reason), we can show that subject to some weak conditions on g, we will have that $E[\hat{g}] = E[g(X)]$, and indeed that $\text{Var}(\hat{g}) = \text{Var}(g(X))/n$. Note, estimating probabilities is subsumed by the general problem of expectations, taking the function g to be the indicator function for some event.

2.0.1 Conditional Estimation

In many situations, we may be interested in *conditional* expectations, the probability of some event or expected value of some outcome subject to certain conditions. The probability A wins the vote, for instance, subject to the condition that a certain demographic participates at a higher rate (or lower, if you are interested in voter suppression) than normal. In general, we are interested in expectations of the form

$$E[g(X)|\text{Conditions}] \tag{70}$$

Note that if we may sample directly from the conditional distribution of X, that is $X' \sim P(X|\text{Conditions})$, then the previous section applies, as

$$E[g(X')] = E[g(X)|\text{Conditions}]. \tag{71}$$

This follows from the fact that $P(X' = x) = P(X = x|\text{Conditions})$. Since the frequency of samples under the distribution of X' matches the distribution of X conditioned on Conditions, if $X'_1, X'_2, \ldots, X'_n$ are generated independently, we will have that each sample of X' is 'equally important', or contributes equal weight to the total estimate:

$$E[g(X)|\text{Conditions}] \approx \frac{\sum_{t=1}^{n} g(X'_t)}{\sum_{t=1}^{n} 1} = \frac{1}{n}\sum_{t=1}^{n} g(X'_t). \tag{72}$$

In the *likelihood weighting* scheme for conditional sampling on sets of variables (taking X in this case to be a vector, for instance), samples of X that satisfied Conditions were generated by explicitly setting the conditions to be satisfied. To account for possible conflicts between 'free' components in X and the values of set conditioned components in X, we generated not only a sample, but a weight w corresponding to how likely the conditions were to be met given the unconditioned aspects of X. In particular, we had that for any value x, the probability of generating the sample $X' = x$ by this process satisfies

$$P(X = x|\text{Conditions}) \propto P(X' = x)w(x) \tag{73}$$

As the frequencies no longer match, if we generate samples $X'_1, X'_2, \ldots, X'_n$, we need to *weight* them by the corresponding factor of w, essentially replacing the equal 1-weights in the previous scheme. This leads to

$$E[g(X)|\text{Conditions}] \approx \frac{\sum_{t=1}^{n} g(X'_t)w_t}{\sum_{t=1}^{n} w_t}. \tag{74}$$

Note, this leads to the *probability* estimate:

$$P(\text{Event}A|\text{Conditions}) \approx \frac{\sum_{t=1}^{n} I[A \text{ occurred for } X'_t]w_t}{\sum_{t=1}^{n} w_t} \tag{75}$$

2.0.2 MCMC Estimation

It is worth considering MCMC estimation techniques in their own section. One way to utilize MCMC sampling techniques for estimation is to simply use them to generate samples $X_1, X_2, \ldots, X_n$, and apply any of the techniques of the previous section. However, one of the important assumptions for the previous methods was that the samples are *independent* from one another. This is explicitly violated for MCMC techniques, as (due to the underlying Markov chain), the state/sample at any time is explicitly dependent on the state/sample at the previous time. One way to account for this is to essentially run the Markov chain for long periods without recording samples. These waiting periods allow the Markov chain to essentially 're mix', so that for sufficiently long intervening periods the samples collected are relatively independent. One issue with this, however, is that if the state space is particularly large, 'mixing times' may be quite large.

This can be addressed, however. Given a stationary long term distribution q, the expected value of a function g may be defined as

$$E[g(X)] = \sum_{s \in S} g(s)q_s. \tag{76}$$

If a Markov chain is *ergodic* (roughly, every state can be reached (eventually) from every other state) under some assumptions on g it will satisfy

$$E[g(X)] = \lim_{n\to\infty} \frac{1}{n}\sum_{t=1}^{n} g(X_t). \tag{77}$$

That is, the *spatial average* over the state space will be equal to the *long term temporal average* over the chain. What this means is that if we build a Markov chain to simulate a desired distribution q (as in the Hastings-Metropolis algorithm), we can estimate the average value with respect to q by simply taking enough sequential samples of X_t and averaging the result together:

$$E[g(x)] \approx \frac{1}{n}\sum_{t=1}^{n} g(X_t). \tag{78}$$

In terms of the probabilities of events, this can be interpreted as saying the probability of a given event is approximately the fraction of time the chain spends in states that satisfy that event.

Hence, the exact same schemes can be utilized to perform estimations in an MCMC environment. In fact, using algorithms like Hasting-Metropolis, since Markov chains can be constructed for *conditional* state distributions with little additional effort, this immediately carries over to additional sampling as well.