Metropolis Algorithm

In this lesson, we will learn about the Metropolis Algorithm, and its implementation.

WE'LL COVER THE FOLLOWING

In the previous lesson, we implemented a technique for sampling from a non-normalized target PDF:

- Find an everywhere-larger helper PDF that we can sample from.
- Sample from it.
- Accept or reject the sample via a coin flip with the ratio of weights in the target distribution and the helper distribution.

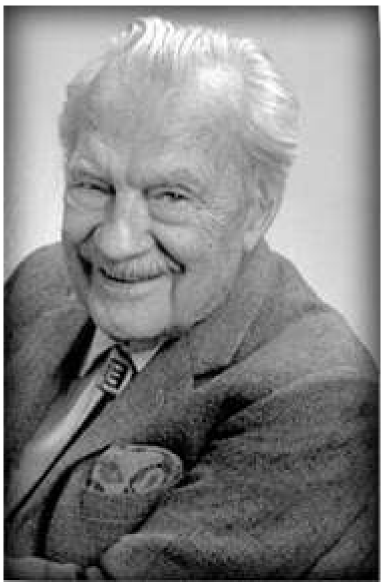
This technique works, but it has a few drawbacks:

- It's not at all clear how to find a suitable helper PDF without humans intervening.
- Even with a pretty tight-fitting helper, you potentially still end up rejecting a lot of samples.

The first problem is the big one. It would be great if there were a technique that didn't require quite so much intervention from experts.

In this lesson, we'll describe just such a technique; it is called the "**Metropolis algorithm**" after one of its inventors, Nicholas Metropolis.

Introduction to the Metropolis Algorithm


Nicholas Metropolis

> The co-authors of the 1953 paper that first presented this algorithm are unfairly not credited in the name of the algorithm, but “the Metropolis algorithm” sounds snappy and cool, whereas “the Metropolis-Rosenbluth-Rosenbluth-Teller-and-Teller algorithm” is a bit of a mouthful.

In modern code, we usually use a variation called the “*Metropolis-Hastings algorithm*”, but we are not going to discuss it in this course.

Pseudocode of the Metropolis Algorithm

Without further ado, here’s a sketch of the *Metropolis algorithm*:

1. We have a non-normalized PDF `f(t)` that takes a `T` and returns a double. We want to produce arbitrarily many samples of `T` that conform to this distribution.
2. The first time that `Sample()` is called, we produce the *initial sample*.
3. Now, if we could produce an initial random sample that matched the distribution described by f, we’d already have solved the problem! All that we require is an initial random sample that is *somewhere* in support of f.
4. We’ll call the distribution of the first sample the *initial distribution*.
5. The initial sample becomes the *current sample*, which is returned.

6. Every subsequent time that `Sample()` is called, we use the current sample to generate a candidate for the new sample; more specifically, we produce a *proposal distribution* of possible next samples.
7. We sample the proposal distribution to get a *proposed sample*.
8. We compute the weights $f(current)$ and $f(proposed)$
9. We divide the proposed weight by the current weight.
10. If the quotient is greater than or equal to 1.0 then the proposed sample automatically becomes the new current sample.
11. Otherwise, we flip a Bernoulli coin with that probability. Heads, we accept the proposed sample, and it becomes the current sample. Tails, we keep the current sample.

> **Exercise:** Do you see why we don't care if the PDFs are non-normalized in this algorithm? All we care about is the quotient of the weights when determining the Bernoulli distribution. Non-normalized PDFs have the same ratios as normalized PDFs!

> There are a few technical restrictions on the proposal distribution that we are not going to discuss; for details, see ergodicity.

That might have been a bit abstract. Let's say in type system terms what we need. First off, we have the target distribution we're attempting to sample from:

```
Func<T, double> target;
```

Next, we have an initial distribution to get the first sample:

```
IDistribution<T> initial;
```

And finally, we have a function that produces proposal distributions:

```
Func<T, IDistribution<T>> proposals;
```

But… wait a minute. This looks familiar.

- We're producing a sequence of samples.

- We randomly choose an initial value.
- We randomly choose each subsequent value based on a probability distribution determined only by the previous value.

This is a Markov process!

We can use a Markov process to simulate sampling from an arbitrary PDF, assuming that the restrictions on the initial and proposal distributions are met.

Well, this should be easy to implement, given that we've already implemented Markov processes, so let's do it. This time, the fun will all be in the factory:

```
sealed class Metropolis<T> : IWeightedDistribution<T>
{
  private readonly IEnumerator<T> enumerator;
  private readonly Func<T, double> target;
  public static Metropolis<T> Distribution(
    Func<T, double> target,
    IDistribution<T> initial,
    Func<T, IDistribution<T>> proposal)
  {
    Func<T, IDistribution<T>> transition = d =>
    {
      T candidate = proposal(d).Sample();
      return Flip<T>.Distribution(candidate, d, target(candidate) / target
(d));
     };
     var markov = Markov<T>.Distribution(initial, transition);
     return new Metropolis<T>(target, markov.Sample().GetEnumerator());
  }

  private Metropolis(
    Func<T, double> target,
    IEnumerator<T> enumerator)
  {
    this.enumerator = enumerator;
    this.target = target;
  }

  public T Sample()
  {
    this.enumerator.MoveNext();
    return this.enumerator.Current;

  }

  public double Weight(T t) => this.target(t);
}
```

Let’s try it out. We’ll need three things. For our target PDF, let’s take the mixed PDF we looked at last time:

```
Func<double, double> Mixture = x =>
  Exp(-x * x) + Exp((1.0 - x) * (x - 1.0) * 10.0);
```

What should our initial distribution be? All doubles are in support of this distribution so that we can choose any, but we’d like it if the double was likely to be high-weight. Let’s suppose that we don’t know the exact shape of this distribution, but we do know that it has a lump near the origin, so we’ll use a standard normal distribution as our initial value.

What should our proposal distribution be? It needs to be based on the current value, not biased to consistently get larger or smaller, and give us the possibility of any value in support of our target distribution being produced. How about a normal distribution centered on the current value? That seems reasonable.

We are going to make a helper method just for this case since it seems likely to be pretty common:

```
public static Metropolis<double> NormalMetropolis(
    this Func<double, double> weight) =>
  Metropolis<double>.Distribution(
    weight,
    Normal.Standard,
    d => Normal.Distribution(d, 1.0));
```

Finally, let’s run it and see if we get a reasonable histogram out:

```
NormalMetropolis(Mixture).Histogram(-2, 2)
```

```
                             ***
                             ***
```

```
                            ***
                            ***

                           *****
                  ****     *****
                 *******  ******
                ******** *******
               *****************
              *******************
             ********************
            *********************
            *********************
           ***********************
          ************************
         *************************
       ****************************
      ******************************
   **********************************
----------------------------------------
```

Nice!

This seems to work well, but unfortunately, there are a few problems to deal with. Some of them are:

1. We need an initial distribution that produces a value in the support. However, this is usually not too hard. And suppose by some accident we do end up getting a zero-weight element as the first sample; what's going to happen? Probably the proposal distribution will pick a non-zero-weight element, and it will then automatically win, so hey, maybe the first sample is bad. No big deal.

2. Let's think a bit more about that issue, though. Even if the first sample has positive weight, we might have a sequence of unusually low-weight samples at the beginning. If we start in a low-weight region, there might be some time spent randomly walking around before we stumble into a high-weight region. Of course, once we are in a high-weight region we are likely to stay there, which is good…

3. … unless the distribution is "multimodal". That is, there are multiple high-weight regions. Our example distribution in this episode is multimodal, but the two "lumps" are close together and are "bridged" by a reasonable high-probability region; imagine a distribution where the "lumps" were

far apart with a super-low-probability region between them. You can end up “stuck” in one of the lumps for a long time, and this can lead to situations where long sequences of samples have way more correlation between them than we would expect if we were sampling from the “real” distribution.

4. The previous point is particularly emphasized by the fact that in a Metropolis simulation of sampling from a distribution, we frequently get repeated values in samples; that sort of correlation is typically not observed at all were we drawing from the “real” distribution rather than a Markovian simulation of it.

All of these problems can be dealt with by a variety of techniques; you’ve probably already thought of some of them:

1. We could “burn” the first few dozen / hundred / whatever of the samples so that we get past any initial low-weight region.
2. If samples are too correlated with each other, we could use a proposal distribution with a higher variance. (For example, we could use a normal distribution with a larger standard deviation.) That would increase the number of rejected samples, which increases repeats, but it would decrease the number of samples that were all close to each other but not rejections.
3. Or, we could sample from a Bernoulli at a probability of our choosing on each sample, and if it’s *heads*, we skip enumerating the value but still keep it as our new state internally; that would decrease the amount of similarity between subsequent samples, while increasing the average computational cost of each sample.
4. Or, we could generate samples in batches of a hundred or a thousand or whatever, randomly shuffle the batch, and then enumerate the shuffled batch, to decrease the amount of similarity between subsequent samples.

We are not going to do any of these, but you can try them on your own.

We’ve made excellent progress here; we have an efficient general mechanism for sampling from a well-behaved PDF, so long as we have some idea of the initial distribution and a good guess at a proposal distribution. We also have some ideas for “tuning parameters” — that is, knobs that experts can turn to make tradeoffs between performance, correlation, and so on, and the knobs are simple enough that you could imagine writing a program to try different settings and figure out what works best.

Implementation

Let’s have a look at the code:

Program.cs

Bernoulli.cs

BetterRandom.cs

Distribution.cs

DistributionBuilder.cs

Empty.cs

Episode28.cs

Extensions.cs

Flip.cs

IDiscreteDistribution.cs

IDistribution.cs

IWeightedDistribution.cs

Markov.cs

MarkovBuilder.cs

Metropolis.cs

```
using System;
using static System.Math;

namespace Probability
{
    static class Episode28
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 28 -- Metropolis");

            Func<double, double> Mixture = x =>
                Exp(-x * x) + Exp((1.0 - x) * (x - 1.0) * 10.0);
            Console.WriteLine(Distribution.NormalMetropolis(Mixture).Histogram(-2, 2));
        }

    }
}
```

We’re going to seemingly take a step backwards and talk about a problem in discrete Bayesian reasoning: given a bag of coins, some unfair, and some observations of the results of flipping a coin pulled from the bag, what can we deduce about the coin we pulled?