

Computing Continuous Posterior Distribution

In this lesson we will try answering the question: what is the continuous posterior when we are given an observation of the discrete result?

WE'LL COVER THE FOLLOWING



- Continuous Posterior Given an Observation of Discrete Result
 - Beta Distribution
 - Using **Metropolis**
- Implementation

In the [previous lesson](#), we posed and solved a problem in Bayesian reasoning involving only discrete distributions, and then proposed a variation on the problem whereby we change the prior distribution to a continuous distribution while preserving that the likelihood function produced a discrete distribution.

Continuous Posterior Given an Observation of Discrete Result

The question is: what is the continuous posterior when we are given an observation of the discrete result?

More specifically, the problem we gave was: suppose we have a prior in the form of a process which produces random values between 0 and 1. We sample from that process and produce a coin that is *heads* with the given probability. We flip the coin; it comes up *heads*. What is the posterior distribution of coin probabilities?

Here's one way to think about it: Suppose we stamp the probability of the coin coming up *heads* onto the coin. We mint and then flip a million of those coins once each. We discard all the coins that came up *tails*. The question is: *what is*

the distribution of probabilities stamped on the coins that came up heads?

Let's remind ourselves of **Bayes' Theorem**. For prior $P(A)$ and likelihood function $P(B|A)$, that the posterior is:

$$P(A|B) = P(A) \times \frac{P(B|A)}{P(B)}$$

Remembering of course that $P(A|B)$ is logically a function that takes a B and returns a distribution of A and similarly for $P(B|A)$.

But so far we've only seen examples of Bayes' Theorem for discrete distributions. Fortunately, it turns out that we can do almost the same arithmetic in our weighted non-normalized distributions and get the correct result.

A formal presentation of the continuous version of Bayes' Theorem and proving that it is correct would require some calculus and distract from where we want to go in this lesson, so we are just going to wave my hands here. Rest assured that we could put this on a solid theoretical foundation if we chose to.

Let's think about this in terms of our *type system*. If we have a prior:

```
IWeightedDistribution<double> prior = // whatever;
```

and a likelihood:

```
Func<double, IWeightedDistribution<Result>> likelihood = // whatever;
```

then what we want is a function:

```
Func<Result, IWeightedDistribution<double>> posterior = // ???
```

Let's suppose our result is `Heads`. The question is: what is `posterior(Heads).Weight(d)` equal to, for any `d` we care to choose? We just apply Bayes' Theorem on the weights. That is, this expression should be equal to:

```
prior.Weight(d) * likelihood(d).Weight(Heads) / ????.Weight(Heads)
```

We have a problem; we do not have an `IWeightedDistribution<Result>` to get `Weight(Heads)` from to divide through.

That is: we need to know what the probability is of getting `Heads` if we sample a coin from the mint, flip it, and do not discard `Heads`.

We could estimate it by repeated computation. We could call:

```
likelihood(prior.Sample()).Sample()
```

a billion times; the fraction of them that are `Heads` is the weight of `Heads` overall.

That sounds expensive though. Let's give this some more thought.

Whatever the `Weight(Heads)` is, it is a positive constant, right? And we have already abandoned the requirement that weights have to be normalized so that the area under the curve is exactly 1.0.

Positive constants do not affect proportionality.

We do not need to compute the denominator at all to solve a continuous Bayesian inference problem; we just assume that the denominator is a positive constant, and so we can ignore it.

So `posterior(Heads)` must produce a distribution such that `posterior(Heads).Weight(d)` is proportional to:

```
prior.Weight(d) * likelihood(d).Weight(Heads)
```

But that is just a non-normalized weight function, and we already have the gear to produce a distribution when we are given a non-normalized weight function; we can use our `Metropolis` class from [two lessons ago](#). It can take a non-normalized weight function, an initial distribution, and a proposal distribution, and produce a weighted distribution from it.

Notice that we don't even need a distribution that we can sample from:

Notice that we don't even need a distribution that we can sample from, all we need is its weight function.

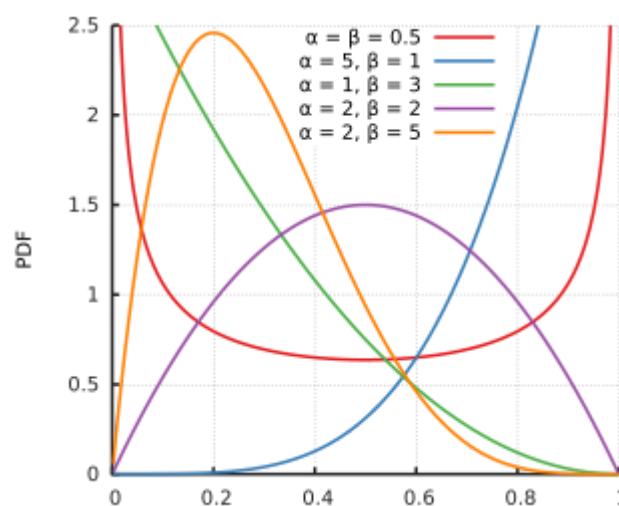
That was all very abstract, so let's look at the example we proposed last time: a mint with poor quality control produces coins with a particular probability of coming up *heads*; that's our prior.

Therefore we'll need a PDF that has zero weight for all values less than 0 and greater than 1. We don't care if it is normalized or not.

Remember, this distribution represents the quality of coins that come from the mint, and the value produced by sampling this distribution is the bias of the coin towards *heads*, where 0 is “double-tailed” and 1 is “double-headed”.

Beta Distribution

Let's choose a beta distribution.



Probability density function for the Beta distribution

Let's take `Beta.Distribution(5, 5)` as our prior distribution. That is, the “fairness” of the coins produced by our terrible mint is distributed like this:

```
var prior = Beta.Distribution(5, 5);  
Console.WriteLine(prior.Histogram(0, 1));
```

The output is:

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

.....

We need a likelihood function, but that is easy enough. Let's make an enumerated type for coins called `Result` with two values, `Heads` and `Tails`:

Using Metropolis

```
Func<Result, IWeightedDistribution<double>> posterior = r =>
    Metropolis<double>.Distribution(
        d => prior.Weight(d) * likelihood(d).Weight(r), // weight
        prior, // initial
        d => Normal.Distribution(d, 1)); // proposal
```

But you know what, let's once again make a generic helper function:

```
public static Func<B, IWeightedDistribution<double>> Posterior<B>(
    this IWeightedDistribution<double> prior,
    Func<double, IWeightedDistribution<B>> likelihood) =>
    b =>
        Metropolis<double>.Distribution(
            d => prior.Weight(d) * likelihood(d).Weight(b),
            prior,
            d => Normal.Distribution(d, 1));
```

And now we're all set:

```
var pos = prior.Posterior(likelihood);
Console.WriteLine(pos(Heads).Histogram(0, 1));
```

The output is:

[illegible]

It is subtle, but if you compare the prior to the posterior you can see that the posterior is slightly shifted to the right, which is the “more *heads*” end of the distribution.

In this specific case we could have computed the posterior exactly. It turns out that the posterior of a beta distribution in this scenario is

another beta distribution, just with different shape parameters. But the point is that we don't need to know any special facts about the *prior* to create a distribution that estimates the posterior. (In cases where we can know mathematically what kind of posterior distribution we're going to get is, we say that the prior and posterior distributions are conjugates. The mathematics of conjugate distributions is quite interesting but we're not going to cover that further in this course.)

Does this result make sense?

The prior is that a coin randomly pulled from this mint is biased, but equally likely to be biased towards *heads* or *tails*. But if we get a *heads*, that is evidence, however slight, that the coin is not biased towards *tails*. We don't know if this coin is close to fair, or very biased towards *heads*, or very biased towards *tails*, but we should expect based on this single observation that the first and second possibilities are more probable than the prior, and the third possibility is less probable than the prior. We see that reflected in the histogram.

Or, another way to think about this is: suppose we sampled from the prior many times, we flipped each coin exactly once, and we threw away the ones that came up *tails*, and then we did a histogram of the fairness of those that came up *heads*. Unsurprisingly, *the coins more likely to turn up tails got thrown away more often than the coins that came up heads, so the posterior histogram is biased towards heads*.

Exercise: Suppose we changed the scenario up so that we were computing the posterior probability of two flips. If we get two *heads*, obviously the posterior will be even more on the "*heads*" side. How do you think the posterior looks if we observe one *head*, one *tail*? Is it different than the prior, and if so, how? Try implementing it and see if your conjecture was correct.

What have we learned? That given a continuous prior, a discrete likelihood, and an observation, we can simulate sampling from the continuous posterior.

Why is this useful? The coin-flipping example is not particularly interesting

why is this useful? The coin flipping example is not particularly interesting, but of course, coin-flipping is a stand-in for more interesting examples.

Perhaps we have a prior on how strongly a typical member of a population believes in the dangers of smoking; if we observe one of them smoking, what can we deduce is the posterior strength of their beliefs? Or perhaps we have a prior on whether a web site user will take action on the site, like buying a product or changing their password, or whatever; if we observe them taking that action, what posterior deductions can we make about their beliefs?

As we've seen a number of times in this course, it's hard for humans to make correct deductions about posterior probabilities even when given good priors; if we can make tools available to software developers that help us make accurate deductions, that has a potentially powerful effect on everything from the design of developer tools to the design of public policy.

Implementation

The code for the Beta and Gamma distributions and the rest of the code in this lesson can be found below:

Program.cs

Beta.cs

Bernoulli.cs

BetterRandom.cs

Distribution.cs

DistributionBuilder.cs

Empty.cs

Episode30.cs


Extensions.cs

Flip.cs

Gamma.cs

IDiscreteDistribution.cs

IDistribution.cs




```

using System;

namespace Probability
{
    using static Result;
    enum Result { Heads, Tails }
    static class Episode30
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 30 -- Continuous prior");

            Console.WriteLine("Beta(5, 5) as the prior");
            var prior = Beta.Distribution(5, 5);
            Console.WriteLine(prior.Histogram(0, 1));

            Func<double, IWeightedDistribution<Result>> likelihood = d =>
                Flip<Result>.Distribution(Heads, Tails, d);

            /*
            Func<Result, IWeightedDistribution<double>> posterior = r =>
                Metropolis<double>.Distribution(
                    d => prior.Weight(d) * likelihood(d).Weight(r),
                    prior,
                    d => Normal.Distribution(d, 1));
            Console.WriteLine(posterior(Heads).Histogram(0, 1));
            */

            var pos = prior.Posterior(likelihood);
            Console.WriteLine(pos(Heads).Histogram(0, 1));
        }
    }
}

```



Thus far in this course, we've been very interested in the question "given a description of a distribution, how can we sample from it efficiently?" But there are other things we want to do with distributions; for example, suppose we have a random process that produces outcomes, and each outcome is associated with a gain or loss of some amount of money. What is the expected amount we will gain or lose in the long run? We'll look at some of these "expected value" problems, and see how the gear we've developed so far can help compute the answers.

