Expected Value of a Discrete Distribution

In this lesson, we will learn how to compute the expected value of a discrete distribution.

WE'LL COVER THE FOLLOWING

- Computing the Expected Value
  - What if the die isn't fair?
  - Example
- Implementation

In the previous lesson, we saw that we could compute a continuous posterior distribution when given a continuous prior and a discrete likelihood function; we hope it is clear how that is useful, but we'd like to switch gears for a moment and look at a different (but also extremely useful) computation: the expected value.

Computing the Expected Value

We'll start with a quick refresher on how to compute the expected value of a discrete distribution.

You probably already know what expected value of a discrete distribution is; we've seen it before in this series. But in case you don't recall, the basic idea is: suppose we have a distribution of values of a type where we can meaningfully take an average; **the "expected value" is the average value of a set of samples as the number of samples gets very large**.

A simple example is: what's the expected value of rolling a standard, fair six-sided die? You could compute it empirically by rolling $6000d6$ and dividing by 6000, but that would take a while.

> Again, recall that in Dungeons and Dragons, XdY is "roll a fair

> $Y-sided$ die X times and take the sum".

We could also compute this without doing any rolling; we'd expect that about 1000 of those rolls would be 1, 1000 would be 2, and so on. So we should add up $(1000 + 2000 + ... + 6000) \div 6000$, which is just $(1 + 2 + 3 + 4 + 5 + 6) \div 6$, which is 3.5. On average when you roll a fair six-sided die, you get 3.5.

We can then do variations on this scenario; what if the die is still fair, but the labels are not $1, 2, 3, 4, 5, 6$, but instead $-9, -1, 0, 1, 3, 30$? You can imagine, once again we can compute the expected value by just taking the average: $(-9 - 1 + 0 + 1 + 3 + 30) \div 6 = 4$.

> It's a bit weird that the "expected value" of a distribution is a value that is not even in support of the distribution; We've never once rolled a 3.5 on a $d6$. Beginners sometimes confuse the expected value with the $mode$: that is, the value that you'd expect to get more often than any other value. Remember, the expected value is just an average; it only makes sense in distributions where the sampled values can be averaged.

What if the die isn't fair?

In that case, we can compute a weighted average; the computation is the value of each side, multiplied by the weight of that side, sum that, and divide by the total weight. As we saw in a previous lesson:

```
public static double ExpectedValue(
    this IDiscreteDistribution<int> d) =>
  d.Support()
   .Select(s =>
     (double)s * d.Weight(s)).Sum() / d.TotalWeight();
```

And of course, we could similarly define methods for discrete distributions of double and so on. Hopefully, that is all clear.

The question we want to explore in the next few lessons requires us to make a small extension of the meaning of "expected value":

Suppose we have a distribution of outcomes `d`, in the form of an

`IWeightedDistribution<double>` Suppose we have a function `f` from `double` to `double` which assigns a value to each outcome. We wish to accurately and efficiently estimate the average value of `f(d.Sample())` as the number of samples becomes large.

> If we had implemented `Select` on weighted distributions, that would be the same as the expected value of `d.Select(f)` — but we didn't!

> We could be slightly more general and say that the distribution is on any `T`, and `f` is a function from `T` to `double`, but for simplicity's sake we'll stick to continuous, one-dimensional distributions in this series. At the end of this series, we'll briefly discuss how and why some of these techniques are important in multidimensional distributions.

There's an obvious and extremely concise solution; *if we want the average as the number of samples gets large, compute the average of a large number of samples!* It's like one line of code. Since we make no use of any weights, we can take any distribution:

```
public static double ExpectedValue(
    this IDistribution<double> d,
    Func<double, double> f) =>
  d.Samples().Take(1000).Select(f).Average();
```

We could make this even more general; we only need to get a number out of the function:

```
public static double ExpectedValue<T>(
    this IDistribution<T> d,
    Func<T, double> f) =>
  d.Samples().Take(1000).Select(f).Average();
```

We could also make it more specific, for the common case where the function is an identity:

```
public static double ExpectedValue(
    this IDistribution<double> d) =>
  d.ExpectedValue(x => x);
```

Let's look at an example.

Example

Suppose we have a distribution from 0.0 to 1.0, say the beta distribution from last time, but we'll skew it a bit:

```
var distribution = Beta.Distribution(2, 5);
Console.WriteLine(distribution.Histogram(0, 1));
```

The output is:

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

It looks like we've heavily biased these coins towards flipping `Tails`; suppose we draw a coin from this mint; what is the average fairness of the coins we draw? We can draw a thousand of them and take the average to get an estimate of the expected value:

```
Console.WriteLine(distribution.ExpectedValue());
```

The output is:

```
0.28099740981762
```

That is, we expect that a coin drawn from this mint will come up `Heads` about 28% of the time and `Tails` 72% of the time, which conforms to our intuition that this mint produces coins that are heavily weighted towards `Tails`.

Or, here's an idea; remember the distribution we determined last time: the posterior distribution of fairness of a coin drawn from a `Beta(5, 5)` mint, flipped once, that turned up `Heads`. On average, what is the fairness of such a coin? (Remember, this is the average given that we've discarded all the coins that came up `Tails` on their first flip.)

```
var prior = Beta.Distribution(5, 5);
IWeightedDistribution<Result> likelihood(double d) =>
  Flip<Result>.Distribution(Heads, Tails, d);
var posterior = prior.Posterior(likelihood)(Heads);
Console.WriteLine(posterior.ExpectedValue());
```

The output is:

```
0.55313807698807
```

As we'd expect, if we draw a coin from this mint, flip it once, and it comes up `Heads`, on average if we did this scenario a lot of times, the coins would be biased to about 55% `Heads` to 45% `Tails`.

So, once again, we've implemented a powerful tool in a single line of code! That's awesome.

Right?

Unfortunately, this naive implementation has many problems.

> **Exercise:** What are the potential problems with this implementation?

Implementation

Let's have a look at the code for this lesson:

Program.cs

Beta.cs

Bernoulli.cs

BetterRandom.cs

Distribution.cs

DistributionBuilder.cs

Empty.cs

Episode31.cs

Extensions.cs

Flip.cs

Gamma.cs

IDiscreteDistribution.cs

IDistribution.cs

IWeightedDistribution.cs

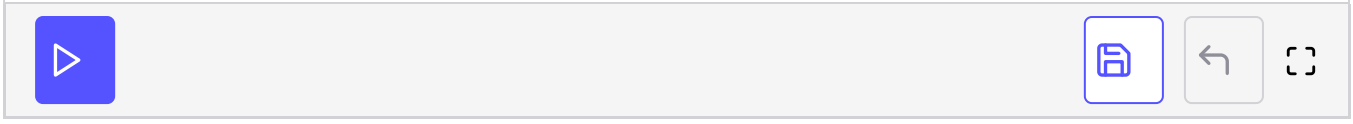
Markov.cs

```
using System;

namespace Probability
{
    using static Result;
    enum Result { Heads, Tails }

    static class Episode31
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 31 -- Expected value refresher");
            Console.WriteLine("Expected fairness of coin drawn from Beta(2, 5)");
            var distribution = Beta.Distribution(2, 5);
            Console.WriteLine(distribution.Histogram(0, 1));
            Console.WriteLine(distribution.ExpectedValue());

            Console.WriteLine("Expected fairness of coin from episode 30");
            var prior = Beta.Distribution(5, 5);
            Func<double, IWeightedDistribution<Result>> likelihood = d =>
                Flip<Result>.Distribution(Heads, Tails, d);
            var posterior = prior.Posterior(likelihood)(Heads);
            Console.WriteLine(posterior.Histogram(0, 1));
            Console.WriteLine(posterior.ExpectedValue());
        }
    }
}
```



In the next lesson, we'll start to address some of the problems with this naive implementation.