Probability Distribution Monad (Continued)

In this lesson, we will continue our discussion on probability distribution using monads in C# and implement the weight functionality. We will also highlight the advantages of probability distribution monad.

WE'LL COVER THE FOLLOWING

In the previous lesson, we achieved two significant results in our effort to build better probability tools. First, we demonstrated that the `SelectMany` implementation, which applies a likelihood function to a prior probability is the bind operation of the probability monad. Second, we gave an implementation of a wrapper object that implements it. Its action can be summed up as:

1. Sample from the prior distribution
2. Use the likelihood function to get the conditional distribution
3. Sample from the conditional distribution
4. Run the projection on the pair of samples to get the result

Implementing the Weight Function

You will probably recall that we did not implement the `Weight` function.

It’s a little tricky to do so, for two reasons.

1. First, we decided to make weights integers.
    - If the weights are fractions between 0.0 and 1.0, you can multiply the weight of the prior sample by the weight of the conditional

sample. (And if the weights are logarithms, you can add them.) It’s trickier with integers.

2. Second, the projection at the end introduces once again the possibility that there will be “collisions”; the projection could pick non-unique values for unique combinations of the samples, that then have to be weighted as the sum.

That’s all a little abstract, so let’s work an example.

Example

Suppose we have a population of people who have been diagnosed with Frob Syndrome, which seems to be linked with height. We’ll divide the population of Frob Syndrome patients into three categories:

```
enum Height { Tall, Medium, Short}
```

and let’s suppose in our study population there are five tall people, two medium-height people, and one short person in every eight:

```
var prior = new List<Height>() { Tall, Medium, Short}.ToWeighted(5, 2, 1);
```

Now let’s suppose we’ve surveyed each of the tall, medium, and short people to learn the severity of their symptoms:

```
enum Severity { Severe, Moderate, Mild}
```

At this point, we are going to make the numbers a bit odd to illustrate *the mathematics* more clearly. What is the likelihood of a member of each group to report symptoms? Let’s say that 10 out of every 21 tall people report severe symptoms and the remaining 11 report moderate symptoms. For the medium-height people, 12 out of 17 report moderate symptoms and 5 report mild symptoms. And all the short people report mild symptoms:

```
var severity = new List<Severity> { Severe, Moderate, Mild};

Func<Height, IDiscreteDistribution<Severity>> likelihood = h =>
{
  switch (h)
  {

    case Tall:
      return severity.ToWeighted(10, 11, 0);
    case Medium:
      return severity.ToWeighted(0, 12, 5);
    default:
      return severity.ToWeighted(0, 0, 1);
  }
};
```

And now let's suppose we have a recommended prescription level:

```
enum Prescription { DoubleDose, NormalDose, HalfDose}
```

Taller people or people with more severe symptoms get a higher dose; shorter people or people with mild symptoms get a smaller dose:

```
 Func<Height, Severity, Prescription> projection = (h, s) =>
 {
  switch (h)
  {
    case Tall: return s == Severe ? DoubleDose : NormalDose;
    case Medium: return s == Mild ? HalfDose : NormalDose;
    default: return HalfDose;
  }
};
```

The question now is: what is the probability distribution on prescriptions for this study population? That is if we picked a random member of this population, how likely is it that they'd have a double, normal or half dose prescription?

```
IDiscreteDistribution<Prescription> doses = prior.SelectMany(likelihood, p
rojection);
```

The problem is to work out the weightings of the three possible outcomes.

As we mentioned before, it's easiest to do this when the weights are fractions because we can then just multiply them and then add them up:

Height	Severity	Prescription
Tall $(\frac{5}{8})$	Severe $(\frac{10}{21})$	DoubleDose $(\frac{25}{84})$
Tall $(\frac{5}{8})$	Moderate $(\frac{11}{21})$	NormalDose $(\frac{55}{168})$
Medium $(\frac{2}{8})$	Moderate $(\frac{12}{17})$	NormalDose $(\frac{3}{17})$
Medium $(\frac{2}{8})$	Mild $(\frac{5}{17})$	HalfDose $(\frac{5}{68})$
Short $(\frac{1}{8})$	Mild $(\frac{1}{1})$	HalfDose $(\frac{1}{8})$

> To save space we have elided the zero rows.

So the probability of a member of this population getting a double dose is $\frac{25}{84}$, getting a normal dose is $\frac{55}{168} + \frac{3}{17} = \frac{1439}{2856}$, and getting a half dose is $\frac{5}{68} + \frac{1}{8} = \frac{27}{136}$.

> Verifying that those add up to 1.0 is left as an exercise.

But we're going to grit our teeth here and do it all in integers! How might we do that?

Well, we know how to eliminate fractions: multiply all the weights in the first column by 8, and all the weights in the second column by 21×17, and none of the proportions will change:

Height	Severity	Prescription
Tall (5)	Severe (170)	DoubleDose (850)
Tall (5)	Moderate (187)	NormalDose (935)
Medium (2)	Moderate (252)	NormalDose (504)

Medium (2)	Mild (105)	HalfDose (210)
Short (1)	Mild (357)	HalfDose (357)

So the integer weights are: double dose is 850, normal dose is $935 + 504 = 1439$, and half dose is $210 + 357 = 567$.

Let’s implement it!

First off, oddly enough there is a `Sum()` extension method but no `Product()` extension method, so let’s implement that:

```
public static int Product(this IEnumerable<int> items) => items.Aggregate(
1, (a, b) => a * b);
```

And we also need to know the total weight of a distribution:

```
public static int TotalWeight<T>(this IDiscreteDistribution<T> d) => d.Sup
port().Select(t => d.Weight(t)).Sum();
```

And now we can implement the algorithm we just sketched out:

```
int product = prior.Support()
  .Select(a => likelihood(a).TotalWeight())
  .Product();
var w = from h in prior.Support()
        let ps = likelihood(h)
        from s in ps.Support()
        group prior.Weight(h) * ps.Weight(s) *
              product / ps.TotalWeight()
        by projection(h, s);
var dict = w.ToDictionary(g => g.Key, g => g.Sum());
var doses = dict.Keys.ToList();
var weights = dict.Values.ToList();
```

And sure enough, if we print those last two out:

DoubleDose	NormalDose	HalfDose
850	1439	567

850	1459	567

Super, we can now work out the weights in our implementation of `SelectMany`.

But… wait a minute. Why do we have to?

That is, why do we need a *Combined wrapper class* for `SelectMany` at all?

We just worked out the weights of every member of the support, and we did so making no assumptions whatsoever about the prior or the likelihood function. We can delete our Combined wrapper class, and replace our implementation of `SelectMany` with:

```
public static IDiscreteDistribution<C> SelectMany<A, B, C>(this IDiscreteDistribution<A> prior, Func<A, IDiscreteDistribution<B>> likelihood,
  Func<A, B, C> projection)
{
  int product = prior.Support().Select(a => likelihood(a).TotalWeight()).Product();
  var w = from a in prior.Support()
          let pb = likelihood(a)
          from b in pb.Support()
          group prior.Weight(a) * pb.Weight(b) *
            product / pb.TotalWeight()
          by projection(a, b);
  var dict = w.ToDictionary(g => g.Key, g => g.Sum());
  return dict.Keys.ToWeighted(dict.Values);
}
```

> **Exercise:** Do you see any potential pitfalls in this implementation of computing the new weights? Give it some thought; we will provide an answer to this question in the next lesson.

We do a small amount of math upfront, and in exchange, we have computed the exact resulting probability distribution, which we can sample from efficiently just as we did with `Where` and `Select` in previous lessons.

> Once again, if we trace through all the logic we have written so far we will quickly see that it is hugely inefficient in terms of the amount of re-

computation it does and garbage it produces. If we were writing this for production code, we'd be a lot more aggressive about finding code paths that do re-computation and eliminating them. The point of this exercise is that our code produces correct, efficient distribution objects out the other end, even if it is a bit wasteful to do so in this particular pedagogic implementation.

Advantages of Probability Distribution Monad

Think about the power of this: you can write programs that treat discrete probability distributions over arbitrary types as values, the same way you'd treat integers, strings, sequences, or whatever, as values that obey a particular set of algebraic rules. We can project, condition and combine them with the same ease that we do today with sequences, and sample from them to boot!

The idea that we can describe a probabilistic workflow, and have as the output of that workflow a new distribution semantically equivalent to the effect of the workflow, but without any long-running sample-and-reject loops due to the conditions, is called **inference** by the probabilistic programming community.

We've seen that we can make inference on arbitrary discrete distributions provided that the supports are small and the weights are small integers; as we'll see throughout the rest of this series, the problem gets considerably harder as we abandon some of those simplifying assumptions.

Implementation

Let's have a look at the code:

Program.cs

Bernoulli.cs

BetterRandom.cs

Combined.cs

Distribution.cs

Episode14.cs

Extensions.cs

IDiscreteDistribution.cs

IDistribution.cs

Projected.cs

Pseudorandom.cs

Singleton.cs

StandardCont.cs

StandardDiscrete.cs

WeightedInteger.cs

```
using System;
using System.Collections.Generic;

enum Height { Tall, Medium, Short }
enum Severity { Severe, Moderate, Mild }
enum Prescription { DoubleDose, NormalDose, HalfDose }

namespace Probability
{
    using static Height;
    using static Severity;
    using static Prescription;

    static class Episode14
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 14");
            var prior = new List<Height>() { Tall, Medium, Short }.ToWeighted(5, 2, 1);
            var severity = new List<Severity> { Severe, Moderate, Mild };

            Func<Height, IDiscreteDistribution<Severity>> likelihood = h =>
            {
                switch (h)
                {
                    case Tall:
                        return severity.ToWeighted(10, 11, 0);
                    case Medium:
                        return severity.ToWeighted(0, 12, 5);
                    default:
                        return severity.ToWeighted(0, 0, 1);
                }
            };

            Func<Height, Severity, Prescription> projection = (h, s) =>
            {
                switch (h)
                {
                    case Tall: return s == Severe ? DoubleDose : NormalDose;
                    case Medium: return s == Mild ? HalfDose : NormalDose;
                    default: return HalfDose;
                }
            };

            Console.WriteLine("Joint distribution of height and severity:");

            var joint = prior.Joint(likelihood);
            Console.WriteLine(joint.Histogram());
            Console.WriteLine(joint.ShowWeights());

            Console.WriteLine("Distribution of doses:");
            var doses = prior.SelectMany(likelihood, projection);
            Console.WriteLine(doses.Histogram());
            Console.WriteLine(doses.ShowWeights());

        }
    }
}
```

In the next lesson, we are going to implement a minor optimization in our weighted integer distribution. After that, we'll put it all together to show how what we've developed so far can be used for *Bayesian inference*.