Probability Distribution Monad

In this lesson, we will discuss probability distribution using monads in C#.

WE'LL COVER THE FOLLOWING

In the previous lesson, we discovered the interesting fact that conditional probabilities can be represented as likelihood functions and that applying a conditional probability to a prior probability looks suspiciously like `SelectMany`, which is usually the bind operation on the sequence monad.

Monads in C#

We created a new implementation of `SelectMany` that creates an object which samples from the prior, calls the likelihood, and then samples from the resulting distribution. Is that the bind operation on the probability distribution monad?

> If you're completely confused by the preceding paragraph, you might want to read an introduction to monads for OO programmers.

We need the following things to have a monad in C#:

1. We need an "embarrassingly generic" type: some `Foo<T>` where it can sensibly take on any `T` whatsoever. `IDiscreteDistribution<T>` meets that condition.
2. The type represents an "amplification of power" of the underlying type. Indeed it does; it allows us to represent a probability distribution of

particular values of that type, which is certainly a new power that we did not have before.

3. We need a way of taking any specific value of any `T` and creating an instance of the monadic type that represents that specific value. `Singleton.Distribution(t)` meets that condition.
4. There is frequently(but not necessarily) an operation that extracts a value of the underlying type from an instance of the monad. `Sample()` is that operation. Note that sampling a singleton always gives you back the original value.
5. There is a way to "bind" a new function onto an existing instance of the monad. That operation has the signature `M<R> SelectMany<A, R>(M<A> m, Func<A, M<R>> f)`. We traditionally call it SelectMany in C# because that's the bind operation on `IEnumerable<T>`, and it produces a projection on all the elements from a sequence of sequences. As we saw last time, we have this function for probability distributions.
6. Binding the "create a new instance" function to an existing monad must produce an equivalent monad. I think it is pretty clear that if we have an `IDiscreteDistribution` in hand, call it d, that `SelectMany(d, t => Singleton.Distribution(t))` produces an object that has the same distribution that d does.

> If that's not clear, play around with the code until it becomes clear to you.

6. Going "the other direction" must also work. That is, if we have a `Func<A, IDiscreteDistribution<B>>` called f, and a value of A, then `SelectMany(Singleton<A>.Distribution(a), f)` and `f(a)` must produce logically the same `IDiscreteDistribution<B>`.

> Again, if that's not clear in your mind, step through the code mentally or write some sample code and convince yourself that it is true.

7. Two bind operations "on top of each other" must produce the same

logical result as a single bind that is the composition of the two bound functions.

All our conditions are met; `IDiscreteDistribution<T>` is a monad. So we should be able to use it in a query comprehension, right?

```
from c in cold
from s in SneezedGivenCold(c)
select s
```

Unfortunately, this gives an error saying that `SelectMany` cannot be found; what's up with that?

The query comprehension syntax requires a slight variation on the traditional "bind" operation; it requires that we also allow a projection on end and that moreover, the projection take both the original value and the transformed value. That is, C# requires us to implement it like this:

```
public sealed class Combined<A, B, C> : IDiscreteDistribution<C>
{
  private readonly List<C> support;
  private readonly IDiscreteDistribution<A> prior;

  private readonly Func<A, IDiscreteDistribution<B>> likelihood;
  private readonly Func<A, B, C> projection;
  public static IDiscreteDistribution<C> Distribution(IDiscreteDistributio
n<A> prior, Func<A, IDiscreteDistribution<B>> likelihood,
      Func<A, B, C> projection) => new Combined<A, B, C>(prior, likelihood
, projection);

  private Combined(IDiscreteDistribution<A> prior, Func<A, IDiscreteDistri
bution<B>> likelihood, Func<A, B, C> projection)
  {
    this.prior = prior;
    this.likelihood = likelihood;
    this.projection = projection;
    var s = from a in prior.Support()
            from b in this.likelihood(a).Support()
            select projection(a, b);
            this.support = s.Distinct().ToList();
  }

  public IEnumerable<C> Support() => this.support.Select(x => x);

  public int Weight(C c) => // NOT YET!
```

```
  public C Sample()
  {
    A a = this.prior.Sample();
    B b = this.likelihood(a).Sample();
    return this.projection(a, b);
  }
}
```

And now we can implement `SelectMany` as

```
public static IDiscreteDistribution<C> SelectMany<A, B, C>(this IDiscreteD
istribution<A> prior, Func<A, IDiscreteDistribution<B>> likelihood,
    Func<A, B, C> projection) => Combined<A, B, C>.Distribution(prior, lik
elihood, projection);
```

and of course, if we want a `SelectMany` with the traditional *monad bind signature*, that's just

```
public static IDiscreteDistribution<B> SelectMany<A, B>(this IDiscreteDist
ribution<A> prior, Func<A, IDiscreteDistribution<B>> likelihood) =>
  SelectMany(prior, likelihood, (a, b) => b);
```

Now that we have a `SelectMany`, we can write conditional probabilities in comprehension syntax, as before:

```
var sneezed = from c in cold
              from s in SneezedGivenCold(c)
              select s;
```

or, if we like, we can extract a tuple giving us both values:

```
public static IDiscreteDistribution<(A, B)> Joint<A, B>(this IDiscreteDist
ribution<A> prior,
    Func<A, IDiscreteDistribution<B>> likelihood) => SelectMany(prior, lik
elihood, (a, b) => (a, b));

var joint = cold.Joint(SneezedGivenCold);
```

and if we graph that, we see that we get the distribution we worked out by hand from last lesson:

```
Console.WriteLine(sneezed.Histogram());
Console.WriteLine(sneezed.ShowWeights());

Console.WriteLine(joint.Histogram());
Console.WriteLine(joint.ShowWeights());
```

The output will be:

```
No|**************************************
Yes|*****

(No, No)|**************************************
(No, Yes)|*
(Yes, No)|
(Yes, Yes)|***
```

It might seem slightly vexing that C# requires us to implement a variation on the standard bind operation, but in this case, this is precisely what we want. Why's that?

Notating Probability Distributions

Let's remind ourselves of how we are notating probability distributions. If we have a collection of possible outcomes of type $Cold$, we notate that distribution as $P(Cold)$; since $Cold$ has two possibilities, this distribution is made up of two probabilities, $P(Cold.Yes)$ and $P(Cold.No)$ which add up to 100%. We represent this in our type system as `IDiscreteDistribution<Cold>`.

A conditional probability distribution $P(Sneezed|Cold)$ is "*given a value from type Cold, what is the associated distribution* `P(Sneezed)`"? In other words, it is `Func<Cold, IDiscreteDistribution<Sneezed>>`.

What then is `P(Cold&Sneezed)`? That is our notation for the joint distribution over all possible pairs. This is made up of four possibilities:

1. $P(Cold.No\ \&\ Sneezed.No)$,
2. $P(Cold.No\ \&\ Sneezed.Yes)$,
3. $P(Cold.Yes\ \&\ Sneezed.No)$, and
4. $P(Cold.Yes\ \&\ Sneezed.Yes)$, which also adds up to 100%.

In our type system, this is `IDiscreteDistribution<(Cold, Sneezed)>`

Now, remember the fundamental law of conditional probability is:

$$P(A) \times P(B|A) = P(A\&B)$$

That is, the probability of A and B both occurring is the probability of A occurring, multiplied by the probability of B occurring given that A has.

That is, we can pick any values from those types, say:

$$P(Cold.Yes)\ P(Sneezed.Yes|Cold.Yes) = \\ P(Cold.Yes\ \&\ Sneezed.Yes)$$

That is, the probability of some value of A and some value of B both occurring is the probability of the value of A occurring multiplied by the probability of the value of B given that the value of A has occurred.

> Here "multiplication" is assuming that the probabilities are between 0.0 and 1.0, but again, you'll see that it's all just weights. In the next lesson, we'll see how to compute the weights as integers by thinking about how to do the multiplication of fractions.

We've implemented $P(A)$ as `IDiscreteDistribution<A>`, we've implemented $P(B|A)$ as `Func<A, IDiscreteDistribution<B>>`, and $P(A\&B)$ as `IDiscreteDistribution<(A, B)>`.

We have a function `Joint<A, B>` that takes the first two and gives you the third, and if you work out the math, you'll see that the probabilities of each member of the joint distribution that results are the products of the probabilities given from the prior and the likelihood.

> Multiplication of a prior probability by a likelihood across all members of a type is implemented by `SelectMany`.

Implementation

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

Program.cs

Program.cs

Bernoulli.cs

BetterRandom.cs

Combined.cs

Distribution.cs

Episode13.cs

Extensions.cs

IDiscreteDistribution.cs

IDistribution.cs

Projected.cs

Pseudorandom.cs

Singleton.cs

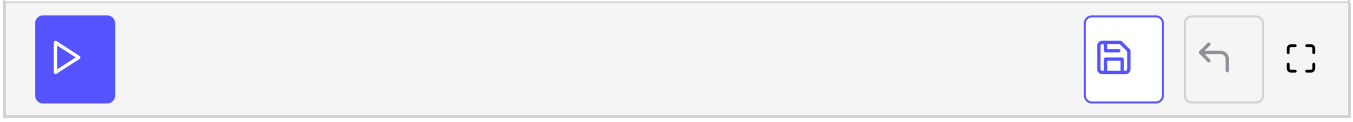
StandardCont.cs

StandardDiscrete.cs

WeightedInteger.cs

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

enum Cold { No, Yes }
enum Sneezed { No, Yes }

namespace Probability
{
    static class Episode13
    {
        static IDiscreteDistribution<Sneezed> SneezedGivenCold(Cold c)
        {
            var list = new List<Sneezed>() { Sneezed.No, Sneezed.Yes };
            return c == Cold.No ?
              list.ToWeighted(97, 3) :
              list.ToWeighted(15, 85);
        }

        public static void DoIt()
        {
            Console.WriteLine("Episode 13");

            var colds = new List<Cold>() { Cold.No, Cold.Yes };
            IDiscreteDistribution<Cold> cold = colds.ToWeighted(90, 10);

            var sneezed = from c in cold
                          from s in SneezedGivenCold(c)
                          select s;
```

```
            var joint = cold.Joint(SneezedGivenCold);

            Console.WriteLine(sneezed.Histogram());
            Console.WriteLine();
            //Console.WriteLine(sneezed.ShowWeights());

            Console.WriteLine(joint.Histogram());
           // Console.WriteLine(joint.ShowWeights());
        }
    }
}
```



In the next lesson, we'll work out the weights correctly, and that will enable us to build an optimized `SelectMany` implementation.