Likelihood Functions

In this lesson, we will have a look at Conditional Probabilities as well as Joint Probability.

WE'LL COVER THE FOLLOWING

In the previous lesson, we implemented an efficient *conditioned* probability using the `Where` operator on distributions; that is, we have some “underlying” distribution, and we ask the question “if a particular condition has to be met, what is the derived distribution that meets that condition?” For discrete distributions, we can compute that distribution directly and just return it.

Conditional Probabilities as Likelihood Functions

There is another kind of conditional probability though, which is much more rich, complex and counter-intuitive, and that is exploring the relationship between “what is the probability of X?" and "what is the probability of Y given that we know X?

For example: pick a random person in the world who has a cold. What is the probability that they sneezed in the last 24 hours? Probably something like 85%.

Now pick a random person who does not have a cold. For them, the probability is maybe more like 3%. In months when we do not have a cold, we sneeze maybe one or two days.

So what we've got here is a rather more complex probability distribution; in fact, we have two entirely different distributions, and which one we use

depends on a condition.

Notice how this is related to our recent discussion of conditioned probabilities but different. With a `Where` clause we are saying make the support of this distribution smaller because some outcomes are impossible based on a condition. What we're talking about here is choosing between two (or more) distributions depending on a condition.

The standard notation for this kind of probability in mathematics is a bit unfortunate. We would say something like

$$P(sneezed|no\ cold) = 0.03$$

to represent 3% chance that we sneezed if we didn't have a cold and

$$P(sneezed|cold) = 0.85$$

to represent 85% chance that we sneezed if we had a cold. That is, the syntax is $P(A|B)$ means *what is the probability of A given that B happened?*

How might we represent this in our system? It seems like `IDiscreteDistribution<T>` is not rich enough. Let's just start making some types and see what we can come up with.

"Has sneezed recently" and *" has a cold"* are Booleans, but we want the types of everything to be very clear in the analysis which follows, so we are going to make our custom types:

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

We want to be slightly abusive of notation here and say that `P(Cold.Yes)` and `P(Cold.No)` are the weights of a probability distribution that we are going to call by the shorthand `P(Cold)`. Similarly for `P(Sneezed)`; that's the probability distribution that gives weights to `P(Sneezed.Yes)` and `P(Sneezed.No)`. Normally we think of `P(something)` as being a value between 0.0 and 1.0, but if you squint at it, really those values are just weights.

It doesn't matter what convention we use for weights; a bunch of integers that give ratios of probabilities and a bunch of doubles that give fractions has pretty much the same information content.

Plainly what we would very much like is to have `IDiscreteDistribution<Cold>` be the `C#` type that represents `P(Cold)`.

But how can we represent our concept of "There's a 3% chance we sneezed if we do not have a cold, but an 85% chance if we do have a cold?"

That sure sounds like precisely this:

```
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);
}
```

That is, if we do not have a cold then the odds are 97 to 3 that we did not sneeze, and if we do have a cold, then the odds are 15 to 85 that we did not sneeze.

We want to represent `P(Cold.Yes)` and `P(Cold.No)` by the shorthand `P(Cold)`, and that this in our type system is `IDiscreteDistribution<Cold>`. Now I want to represent the notion of `P(Sneezed)` given a value of *Cold* as `P(Sneezed|Cold)`, which is implemented by the function above. So, what type in our type system is that? Well, suppose we wanted to assign `SneezedGivenCold` to a variable; what would its type be? Clearly, the type of `P(Sneezed|Cold)` is `Func<Cold, IDiscreteDistribution<Sneezed>>`!

How interesting! Conditional probabilities are actually functions.

This sort of function has a name; it is called a **likelihood function**. That is, given some condition, how likely is some outcome?

So that's interesting, but how is this useful?

Example of a Likelihood Function

Let's randomly choose a person in the world, where we do not know whether they have a cold or not. What is the probability that they sneezed recently? It depends entirely on the prevalence of colds! If 100% of the world has a cold, then there's an 85% chance that a randomly chosen person sneezed recently, but if 0% of the world has a cold, then there's only a 3% chance. And if it is but if 0% of the world has a cold, then there's only a 3% chance. And if it is

somewhere in between, the probability will be different from either 85% or 3%.

To solve this problem we need to know the probability that the person we've chosen has a cold. The probability that a randomly chosen person has a cold is called the **prior** probability.

What if 10% of the world has a cold? Let's work it out by multiplying the probabilities:

Cold (prior)	**Sneezed** (likelihood)	**Result** (conditional)
10% Yes	85% Yes	8.5% have a cold, and sneezed
	15% No	1.5% have a cold, did not sneeze
90% No	3% Yes	2.7% do not have a cold and sneezed
	97% No	87.3%$ do not have a cold, did not sneeze

Sure enough, those probabilities in the right column add up to 100%. The probability that a randomly chosen person in the world sneezed recently (given that these numbers that we made up are accurate) is $8.5\% + 2.7\% = 11.2\%$.

The rightmost column of the table that we have sketched out here is called the **joint probability**, which we will notate as `P(Cold&Sneezed)`.

Joint Probability

We can write this table more compactly like this:

	Cold Yes	Cold No	Total
Sneezed Yes	8.5%	2.7%	11.2%
Sneezed No	1.5%	87.3%	88.8%
Total	10%	90%	100%

The rightmost column of this table is called the **marginal probability**, so-called because of the way the sums end up at the margins of the table.

What if we expressed the marginal probability as integers? The odds that a random person has sneezed is 11.2% to 88.8%, which if you work out the math, is exactly odds of 14 to 111.

> 111 represents the 88.8%, not the 11.2%.

How can we do this math given the set of types we've created so far? Let's start with the prior:

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

We've got the prior, and we've got the likelihood function `SneezedGivenCold`. We would like to get the marginal probability `IDiscreteDistribution<Sneezed>`
.

We could implement such a distribution by first sampling from the prior, then calling `SneezedFromCold`, and then sampling from the returned distribution. Let's implement it.

> We are of course assuming that the likelihood function is pure.

```
public sealed class Combined<A, R> : IDiscreteDistribution<R>
{
  private readonly List<R> support;
  private readonly List<R> support;
  private readonly IDiscreteDistribution<A> prior;
  private readonly Func<A, IDiscreteDistribution<R>> likelihood;
  public static IDiscreteDistribution<R> Distribution(IDiscreteDistribution<A> prior, Func<A, IDiscreteDistribution<R>> likelihood) =>
    new Combined<A, R>(prior, likelihood);
  private Combined(IDiscreteDistribution<A> prior, Func<A, IDiscreteDistribution<R>> likelihood)
  {
    this.prior = prior;
    this.likelihood = likelihood;
    var q = from a in prior.Support()
            from b in this.likelihood(a).Support()
            select b;
    this.support = q.Distinct().ToList();
  }

  public IEnumerable<R> Support() => this.support.Select(x => x);
  public R Sample() => this.likelihood(this.prior.Sample()).Sample();

  public int Weight(R r) => // WE'LL COME BACK TO THIS ONE
}
```

We haven't implemented `Weight`, but we don't need it to run a histogram. Let's try it out:

```
Combined<Cold, Sneezed>.Distribution(cold, SneezedGivenCold).Histogram()
```

The output will be:

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

Sure enough, it looks like there is about an 11% chance that a randomly chosen person sneezed, given these distributions.

Now, of course as we have done throughout this series, let's make a little helper function to make the call sites look a little nicer:

```
public static IDiscreteDistribution<R> MakeCombined<A, R>(this IDiscreteDistribution<A> prior, Func<A, IDiscreteDistribution<R>> likelihood) =>
  Combined<A, R>.Distribution(prior, likelihood);
```

Once again, that should look very familiar! We should change the name of this helper.

If you are still surprised at this point, you have not been paying attention. I've already made `Select` and `Where`, so the obvious next step is…

```
public static IDiscreteDistribution<R> SelectMany<A, R>(this IDiscreteDist
ribution<A> prior, Func<A, IDiscreteDistribution<R>> likelihood) =>
  Combined<A, R>.Distribution(prior, likelihood);
```

… the **bind operation** on the probability monad.

And the inelegant call site above is now the much more clear:

```
cold.SelectMany(SneezedGivenCold).Histogram()
```

Implementation

The code for this lesson is as follows:

Program.cs

Bernoulli.cs

BetterRandom.cs

Combined.cs

Distribution.cs

Episode12.cs

Extensions.cs

IDiscreteDistribution.cs

IDistribution.cs

Pseudorandom.cs

Projected.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 Episode12
    {
        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 12");

            var colds = new List<Cold>() { Cold.No, Cold.Yes };
            IDiscreteDistribution<Cold> cold = colds.ToWeighted(90, 10);
            Console.WriteLine("Combined distribution: prior probability of having a cold");
            Console.WriteLine("conditional probability of having sneezed");
            Console.WriteLine(cold.SelectMany(SneezedGivenCold).Histogram());
        }
    }
}
```

In the next lesson, we will verify that the distribution type really is a *monad*, and make a few tweaks to get it working with query comprehension syntax. Then we'll figure out how to implement the `Weight` function.