Solve Problems in Bayesian Inference

In this lesson, we will review concepts learned so far and learn how to solve problems using Bayesian Inference.

Review of the Concepts Learned So Far

Since we have covered many new concepts this would be a good time to quickly review where we're at:

1. We're representing a particular discrete probability distribution $P(A)$ over a small number of members of a particular type A by `IDiscreteDistribution<A>`.

2. We can condition a distribution — by discarding certain possibilities from it — with `Where`.

3. We can project a distribution from one type to another with `Select`.

4. A conditional probability $P(B|A)$ — the probability of B given that some A is true — is represented as likelihood function of type `Func<A, IDiscreteDistribution<B>>`.

5. We can "bind" a likelihood function onto a prior distribution with `SelectMany` to produce a joint distribution.

These are all good results and we hope you agree that we have already produced a much richer and more powerful abstraction over randomness than `System.Random` provides.

Bayes' Theorem

In this lesson, everything is really going to come together to reveal that we can use these tools to solve interesting problems in *probabilistic inference.*

To show how we'll need to start by reviewing *Bayes' Theorem.*

If we have a prior $P(A)$, and a likelihood $P(B|A)$, we know that we can "bind" them together to form the joint distribution. That is, the probability of A and B both happening is the probability of A multiplied by the probability that B happens given that A has happened:

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

Obviously, that goes the other way. If we have $P(B)$ as our prior, and $P(A|B)$ as our likelihood, then:

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

But $(A\&B)$ is the same as $(B\&A)$, and things equal to the same are equal to each other. Therefore:

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

Let's suppose that $P(A)$ is our prior and $P(B|A)$ is our likelihood. In the equation above the term $P(A|B)$ is called the *posterior* and can be computed like this:

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

Let's move away from abstract mathematics and illustrate an example by using the code we've written so far.

We can step back a few lessons and re-examine our prior and likelihood example for *Frob Syndrome.* Recall that this was a made-up study of a made-up condition which we believe may be linked to height. We'll use the weights from the original episode.

That is to say: we have $P(Height)$, we have likelihood function $P(Severity|Height)$, and we wish to first compute the joint probability distribution $P(Height\&Severity)$:

```csharp
var heights = new List<Height() { Tall, Medium, Short }
var prior = heights.ToWeighted(5, 2, 1);
[...]
IDiscreteDistribution<Severity> likelihood(Height 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);
  }
}
[...]

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

This produces:

```
(Tall, Severe):850
(Tall, Moderate):935
(Medium, Moderate):504
(Medium, Mild):210
(Short, Mild):357
```

> Now the question is: what is the posterior, $P(Height|Severity)$?

💡 **Show Hint**

Computing Marginal Probabilities

We can compute the marginal probabilities "by hand" by looking at the weights above:

- If symptoms are severe, there is a 100% chance that the person is tall.
- If symptoms are moderate, 935 study members are tall for every 504

medium-height members.

- If symptoms are mild, then that's 210 medium people for every 357 short.

We could implement that easily enough; it's just another function like we've seen many times before in this course:

```
IDiscreteDistribution<Height> posterior(Severity s)
{
  switch(s) // ... code here... //
}
```

We have enough information in the `IDiscreteDistribution<(Height, Severity)>` to generate a `Func<Severity<IDiscreteDistribution>>`.

In fact, we can simply add another clause to our query:

```
IDiscreteDistribution<Height> posterior(Severity s) =>
  from pair in joint
  where pair.s == s
  select pair.h;
```

We can compute the posterior with a `Where` clause!

Recall that what we are computing here is $P(A\&B) \div P(B)$; just as `SelectMany` can be thought of as a sort of multiplication, apparently `Where` is a sort of division.

But let's not stop here; we can make a general rule in the form of an extension method:

```
public static Func<B, IDiscreteDistribution<C>> Posterior<A, B, C>(this ID
iscreteDistribution<A> prior, Func<A, IDiscreteDistribution<B>> likelihood
, Func<A, B, C> projection) =>
  b => from a in prior
        from bb in likelihood(a)
        where object.Equals(b, bb)
        select projection(a, b);

public static Func<B, IDiscreteDistribution<A>> Posterior<A, B>(this IDisc
reteDistribution<A> prior, Func<A, IDiscreteDistribution<B>> likelihood) =
>
  Posterior(prior, likelihood, (a, b) => a);
```

> **Question:** Given the prior distribution and the likelihood function, what is the posterior distribution of height amongst the study members with moderate symptoms?

<div align="center">

💡 **Show Hint**

</div>

OK, that's pretty neat, but why is this relevant?

Because Bayesian inference is incredibly important, and incredibly easy to get wrong! Anything we can do to improve developers' ability to use Bayesian analysis correctly is a win.

Another Example

Bayesian inference is counterintuitive. Suppose we have a disease, Tappett's Disease, that has infected 1% of the population, and we have a 99% accurate diagnostic test created by Dr. Jethro. Let's look at an example of this:

```
enum TappetsDisease {Sick, Healthy}
```

and our prior is that 99% of the population is healthy:

```
var prior = new List<TappetsDisease> {Sick, Healthy}.ToWeighted(1, 99);
```

We also have a test:

```
enum JethroTest { Positive, Negative }
```

And the test is 99% accurate. That is, if you are sick, it has a 99% chance of "positive", and if you are healthy, it has a 99% chance of "negative":

```
var tests = new List<JethroTest> { Positive, Negative };

IDiscreteDistribution<JethroTest> likelihood(TappetsDisease d) =>
  d == Sick ? tests.ToWeighted(99, 1) : tests.ToWeighted(1, 99);
```

> You might wonder how we know that the test is 99% accurate, and how we know that 1% of the population has the disease. That's a great question and we are not going to get into the details in this series of how in the real world medical practitioners evaluate the accuracy of a test or the prevalence of a condition. Let's just suppose that we know these facts ahead of time; after all, that's why the prior is called the prior.

> **Question**: You have just tested positive; what is the probability that you have the disease?

Solution

Most people, and even many doctors, will say "the test is 99% accurate, you tested positive, therefore there is a 99% chance that you have the disease". But that is not at all true; we can compute the true result very easily now:

```
var posterior = prior.Posterior(likelihood);
Console.WriteLine(posterior(Positive).ShowWeights());
```

And we get:

```
Sick:1
Healthy:1
```

It's fifty-fifty.

Why?

If a result is confusing, always look at the joint distribution:

```
Console.WriteLine(prior.Joint(likelihood).ShowWeights());
```

The results will be:

```
(Sick, Positive):99
(Sick, Negative):1
(Healthy, Positive):99
```

```
(Healthy, Positive):99
(Healthy, Negative):9801
```

You tested positive. 99 out of every 10000 people are true positives, and 99 out of every 10000 people are false positives. We condition away from the negatives, because you didn't test negative, and what is left? 50% chance that you are positive, not 99%.

> In this example if you test negative then you are not 99% likely to be negative; you are 99.99% likely to be negative! This is also counterintuitive to people.

> **Exercise:** How accurate does the test have to be for you to have a 90% posterior probability of actually being positive given a positive result?

Advantages of Bayesian Inference

Bayesian inference is incredibly powerful and useful. We very frequently have good information on priors and likelihoods. We make observations of the world, and we need to figure out posteriors probabilities given those observations.

A classic example in information technology is:

1. We can ask users to manually classify emails into spam and non-spam. That gives us a prior on $P(Spam)$.
2. From that collection of spam and non-spam emails, we can find out which words are commonly found only in spam. That gives us a likelihood function, $P(Words|Spam)$.
3. We then observe a real email, and the question is: given the words in an email, what is the posterior probability that it is spam? That is, what is the function $P(Spam|Words)$. If the probability passes some threshold, we can put the mail in the spam folder.

Applications in Sensor Technology

There are also real applications in sensor technology:

1. We have a machine in a factory which requires a part on a conveyor to stop moving before it is welded; we manually observe how often the part is stopped correctly, giving us a prior on $P(Stopped)$.

2. We install a sensor that attempts to sense whether the part is stopped and test its accuracy to obtain $P(Sensor\,Reading|Stopped)$

3. Now we have enough information to compute the posterior: given a certain reading from the sensor, what is the probability that the part has stopped moving? That is $P(Stopped|Sensor\,Reading)$.

4. If we do not have a high enough probability that the part is stopped, we can delay the welding step until we have better evidence that the part has stopped.

Applications in Developer Tools

There are even applications in developer tools!

1. We can gather information from open source repositories about how often certain functions are called, giving us a prior on $P(Function\,called)$.

2. We can gather information from IDE keystrokes about how often a letter typed is ultimately the first letter of that function, giving us a likelihood function $P(Keystrokes|Function\,called)$.

3. Now we have enough information to compute the posterior: given a certain set of recent keystrokes, what is the probability distribution of likely functions the user wishes to call? This could give us much better typing hints in an editor.

And so on. The opportunities for taking advantage of Bayesian inference are enormous. We really ought to have Bayesian inference on distributions in the basic toolbox of the language, the same way we have ints, doubles, strings, nullables, functions, tasks, sequences, and so on, in that toolbox.

That's what we mean by "Fixing Random". The fundamental problem is not that `Random` has historically had a confusing interface; that's just a silly historical accident that can be fixed. Rather: we've decided that monads like nullable, sequence, function and task are so important that they are included in the core runtime. Why? Not because they're cool, but because having `Nullable<T>`, `IEnumerable<T>`, `Task<T>`, and so on in the core runtime makes it

problems.

Programming is increasingly about dealing with a world of unknowns; having operators in the language for concisely describing probabilistic workflows seems very valuable to me. This course seeks to make the case for that value.

Implementation

The code so far is as follows:

Program.cs

Bernoulli.cs

BetterRandom.cs

Combined.cs

Distribution.cs

Episode16.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 }
enum TappetsDisease { Sick, Healthy }
enum JethroTest { Positive, Negative }

namespace Probability
{
```

```csharp
using static Height;
using static Severity;
using static Prescription;

using static TappetsDisease;
using static JethroTest;

static class Episode16
{
    public static void DoIt()
    {
        Console.WriteLine("Episode 16");
        ComputeHeightPosterior();
        ComputeTestAccuracy();
    }

    private static void ComputeHeightPosterior()
    {
        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);
            }
        };

        Console.WriteLine("Joint distribution of height and severity:");
        var joint = prior.Joint(likelihood);
        Console.WriteLine(joint.Histogram());
        Console.WriteLine(joint.ShowWeights());

        Console.WriteLine("Posterior distribution of height given moderate symptoms:");
        var posterior = prior.Posterior(likelihood);
        Console.WriteLine(posterior(Moderate).ShowWeights());

    }

    static void ComputeTestAccuracy()
    {
        var prior = new List<TappetsDisease> { Sick, Healthy }.ToWeighted(1, 99);

        var tests = new List<JethroTest> { Positive, Negative };
        Func<TappetsDisease, IDiscreteDistribution<JethroTest>> likelihood = d =>
            d == Sick ?
                tests.ToWeighted(99, 1) :
                tests.ToWeighted(1, 99);

        Console.WriteLine("Joint distribution of sickness and test results");
        Console.WriteLine(prior.Joint(likelihood).ShowWeights());
        Console.WriteLine("Posterior distribution of sickness given positive test:");
        var posterior = prior.Posterior(likelihood);
        Console.WriteLine(posterior(Positive).ShowWeights());
        Console.WriteLine("Posterior distribution of sickness given negative test:");
        Console.WriteLine(posterior(Negative).ShowWeights());
    }
```
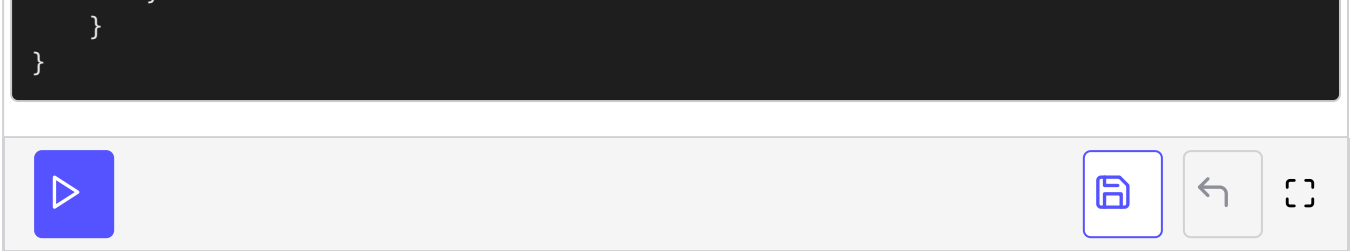
```
    }
}
```

In the next lesson, we'll take a closer look at the discrete probability distribution type as a monad.