Conditional Probabilistic Reasoning

In this lesson, we will learn how to express probabilities with extra conditions.

WE'LL COVER THE FOLLOWING

Expressing Probabilities With an Extra Condition

We're going to spend the next lessons in this course on expressing probabilities that have some sort of extra condition associated with them.

We'll start with a simple example: We want to roll a fair six-sided die, but discard the threes. Why? Maybe we dislike threes. Doesn't matter; we will want to be able to create new distributions by putting conditions on other distributions, so let's figure out how to do so.

Now, of course, we could create the desired distribution easily enough; as we know from the previous lesson, it is

```
WeightedInteger.Distribution(0, 1, 1, 0, 1, 1, 1)
```

But suppose we have a distribution already in hand which we wish to apply a condition to, post hoc. We can see how to do that easily. It's a variation on our "projected" distribution from a while back:

```
public sealed class Conditioned<T> : IDiscreteDistribution<T>
{
```

```
  private readonly List<T> support;
  private readonly IDiscreteDistribution<T> underlying;
  private readonly Func<T, bool> predicate;

  public static IDiscreteDistribution<T> Distribution(IDiscreteDistributio
n<T> underlying, Func<T, bool> predicate)
  {
    var d = new Conditioned<T>(underlying, predicate);
    if (d.support.Count == 0)
      throw new ArgumentException();

    if (d.support.Count == 1)
      return Singleton<T>.Distribution(d.support[0]);

    return d;
  }

  private Conditioned(IDiscreteDistribution<T> underlying, Func<T, bool> p
redicate)
  {
    this.underlying = underlying;
    this.predicate = predicate;
    this.support = underlying.Support()
      .Where(predicate)
      .ToList();
  }

  public T Sample()
  {
    while (true) // Notice this loop
    {
      T t = this.underlying.Sample();
      if (this.predicate(t))
          return t;
    }
  }

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

  public int Weight(T t) => predicate(t) ? underlying.Weight(t) : 0;
}
```

Projected Distributions

Given our discussion a few lessons back about projected distributions and the fact that creating them has the same signature as `Select`, we are going to

fact that creating them has the same signature as `Select`, we are going to make a helper method:

```
public static IDiscreteDistribution<T> Where<T>(
    this IDiscreteDistribution<T> d,
    Func<T, bool> predicate) =>
  Conditioned<T>.Distribution(d, predicate);
```

Now we can use the *comprehension syntax* again:

```
var noThrees = from roll in SDU.Distribution(1, 6)
               where roll != 3
               select roll;
Console.WriteLine(noThrees.Histogram());
```

And we get as expected, no threes:

```
1|**************************************
2|**************************************
4|**************************************
5|**************************************
6|**************************************
```

However, there are some problems here. That possibly-long-running loop is deeply worrisome. Dealing with the existence of that loop will be the major theme of the rest of this course, in one way or another. (This should feel familiar; of course, this is just another version of the “rejection sampling” problem.)

Fixing the problems

We’ll talk about that *loop* problem more in the next lesson. For the remainder of this lesson, we want to examine an assumption we made in the code above; it’s the same assumption that we made when we discussed projections.

Consider the following code, which uses **only sequences, no distributions**:

```
int filterOut = 3;
Func<int, bool> predicate = x => x != filterOut;
var range = Enumerable.Range(1, 6).Where(predicate);
Console.WriteLine(range.CommaSeparated());
filterOut = 4;
Console.WriteLine(range.CommaSeparated());
```

> We have added a handful of extension methods for `string.Join(.....)` to be a little more fluent. See the source code below for details.

If you recall that sequences are computed lazily and lambdas are closed over variables, not values, then this output should be expected:

$1, 2, 4, 5, 6$

$1, 2, 3, 5, 6$

What if we do the same thing to distributions?

```
int filterOut = 3;
Func<int, bool> predicate = x => x != filterOut;
var d = SDU.Distribution(1, 6).Where(predicate);
Console.WriteLine(d.Samples().Take(10).CommaSeparated());
filterOut = 4;
Console.WriteLine(d.Samples().Take(10).CommaSeparated());
```

As we'd expect, we first get no threes, and then no fours:

$1, 1, 4, 6, 6, 5, 4, 1, 2, 6$

$6, 5, 6, 5, 1, 5, 6, 3, 2, 1$ So what's the problem?

```
Console.WriteLine(d.Support().CommaSeparated());
```

$1, 2, 4, 5, 6$

Uh oh. We just produced a 3 in a distribution that does not list 3 in its support!

Why? Because I computed the support in the **constructor** and cached it, but the support changed when the predicate changed its behavior, and it is now out-of-date. The object is now lying to us.

We could fix this problem easily enough: do not compute the support in the constructor; compute it dynamically, on-demand. Is that the right thing to do?

If we do, we lose the ability to reject "*null*" distributions — distributions with no support — and therefore it is possible to get into a situation where

no support — and therefore it is possible to get into a situation where sampling hangs because the predicate is never true. (We could re-check the support before every sample, and throw if the support is empty, but that seems expensive.)

Furthermore, as we'll see in the next few lessons, we can do some pretty good optimizations if we can assume that the predicate does not change.

Therefore we are going to state right now that the predicate passed to `Where` (and the projection passed to `Select`) must be "**pure**" functions when they are intended to operate on probability distributions.

Pure Functions

A *pure* function for our purposes has the following characteristics:

1. It must complete normally; in its regular operation it should not hang and it should not throw.
    - It is permissible for a pure function to throw if its preconditions are violated; for example, a pure function that is documented as not taking negative numbers is permitted to throw an argument exception if passed a negative number.
    - But a pure function should not throw or hang when given a normal, expected input.
2. It must not produce any side effect. For example, it must not write to the console or mutate a global variable or anything like that.
3. Its outputs must depend solely on its inputs; a pure method does not produce one result on its first call and a different result on its second call because something in the world changed; the only reason to produce a different result is if the arguments passed in are different.

Pure functions are nice to work in two particular ways.

1. First, you can reason about their correctness entirely "locally"; you do not have to consider the state of the world at the time the call is made, and you do not have to worry that the state of the world will change depending on how many times the call happens.
2. Second, you can get performance wins by taking advantage of the fact that you know that two calls with the same arguments will produce the same result.

same result.

> Unfortunately, in C# we have no good way of detecting a non-pure method at compile-time and outlawing them as arguments to `Where` and `Select`.

Implementation

Have a look at the source code for this lesson. A new file `Conditioned.cs` has been added. Make sure to have a look at it.

Program.cs
Bernoulli.cs
BetterRandom.cs
Conditioned.cs
Distribution.cs
Episode10.cs
Extensions.cs
IDiscreteDistribution.cs
IDistribution.cs
Projected.cs
Pseudorandom.cs
Singleton.cs
StandardCont.cs
StandardDiscrete.cs
WeightedInteger.cs

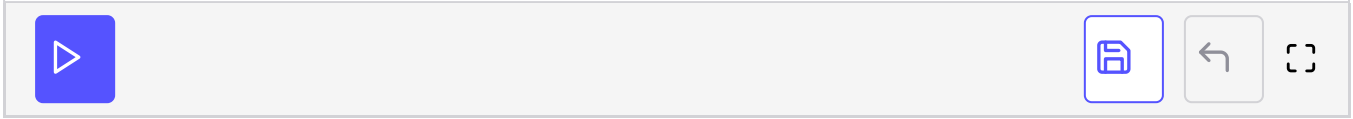
```
using System;
using System.Linq;
namespace Probability
{
    using SDU = StandardDiscreteUniform;
    static class Episode10
    {
        public static void DoIt()
```

```
        {
            Console.WriteLine("Episode 10");
            Console.WriteLine("No threes, weighted integer");
            Console.WriteLine(WeightedInteger.Distribution(0, 1, 1, 0, 1, 1, 1)
                .Samples()
                .Take(20)
                .CommaSeparated());

            Console.WriteLine("No threes, conditioned uniform, rejection sampling");
            var noThrees = from roll in SDU.Distribution(1, 6)
                           where roll != 3
                           select roll;
            Console.WriteLine(noThrees.Histogram());

            Console.WriteLine("Ordinary sequence, filter clause closed over variable");
            int filterOut = 3;
            Func<int, bool> predicate = x => x != filterOut;
            var range = Enumerable.Range(1, 6).Where(predicate);
            Console.WriteLine(range.CommaSeparated());
            Console.WriteLine("Change the variable");
            filterOut = 4;
            Console.WriteLine(range.CommaSeparated());

            Console.WriteLine("Distribution, filter clause closed over variable");
            filterOut = 3;
            var d = SDU.Distribution(1, 6).Where(predicate);
            Console.WriteLine(d.Samples().Take(10).CommaSeparated());
            Console.WriteLine("Change the variable");
            filterOut = 4;
            Console.WriteLine(d.Samples().Take(10).CommaSeparated());
            Console.WriteLine("The support is now wrong!");
            Console.WriteLine(d.Support().CommaSeparated());
            Console.WriteLine("We will require that predicates and projections be pure funct
        }
    }
}
```



Now that we are requiring purity in `Where` and `Select`, what optimizations can we make to the discrete distributions we've created so far? We will be discussing the answer to this question in the next lesson.