Efficiently Sampling a Conditioned Distribution

In this lesson we will learn how to implement the conditioned distribution from the previous lesson without using rejection sampling; we want to avoid the possibly-long-running loop.

WE'LL COVER THE FOLLOWING

In the previous lesson, we concluded that the predicates and projections used by `Where` and `Select` on discrete distributions be *pure* functions:

1. They must complete normally, produce no side effects, consume no external state, and never change their behavior.
2. They must produce the same result when called with the same argument, every time.

If we make these restrictions then we can get some big performance wins out of Where and Select. Let’s see how.

Dealing With the “Long-Running Loop”

The biggest problem we face is that possibly-long-running loop in the `Where`. We are “*rejection sampling*” the distribution, and we know that can take a long time. Is there a way to directly produce a new distribution that can be efficiently sampled?

Of course, there is. Let’s make a helper method:

```
public static IDiscreteDistribution<T> ToWeighted<T>(this IEnumerable<T> i
tems, IEnumerable<int> weights)
{
  var list = items.ToList();
  return WeightedInteger.Distribution(weights).Select(i => list[i]);
```

```
}
```

There's an additional helper method that we are going to need in a couple of lessons, so let's just make it now:

```
public static IDiscreteDistribution<T> ToWeighted<T>(this IEnumerable<T> items,
  params int[] weights) => items.ToWeighted((IEnumerable<int>)weights);
```

And now we can delete our `Conditioned` class altogether, and replace it with:

```
public static IDiscreteDistribution<T> Where<T>(this IDiscreteDistribution<T> d, Func<T, bool> predicate)
{
  var s = d.Support().Where(predicate).ToList();
  return s.ToWeighted(s.Select(t => d.Weight(t)));
}
```

Recall that the `WeightedInteger` factory will throw an exception if the support is empty, and return a `Singleton` or `Bernoulli` if its size one or two.

> **Exercise:** We're doing probabilistic workflows here; it seems strange that we are either 100% rejecting or 100% accepting in `Where`. Can you write an optimized implementation of this method?

```
public static IDiscreteDistribution<T> Where<T>
(this IDiscreteDistribution<T> d, Func<T, IDiscreteDistribution<bool>> predicate)
```

> That is, we accept each `T` with a probability distribution given by a probabilistic predicate.

Show Hint

What about optimizing `Select`?

Of course, we can do the same trick. We just have to take into account the fact

that the projection might cause some members of the underlying support to

“merge” their weights:

```
public static IDiscreteDistribution<R> Select<A, R>(this IDiscreteDistribu
tion<A> d, Func<A, R> projection)
{
  var dict = d.Support().GroupBy(projection, a => d.Weight(a)).ToDictionar
y(g => g.Key, g => g.Sum());

  var rs = dict.Keys.ToList();

  return Projected<int, R>.Distribution(WeightedInteger.Distribution(
      rs.Select(r => dict[r])), i => rs[i]);
}
```

That is, we compute the new support and the weights for each element of it. Now we can construct a weighted integer distribution that chooses an offset into the support.

> **Exercise:** Why did we not write the far more concise: `return rs.ToWeighted(rs.Select(r => dict[r]));`

Show Hint

Let’s think for a minute about whether this really does what we want.

Suppose for example we do a projection on a `Singleton`:

1. We’ll end up with a single-item dictionary with some non-zero weight.
2. The call to the weighted integer factory will return a `Singleton<int>` that always returns zero.
3. We’ll build a projection on top of that, and the projection factory will detect that the support is of size one, and return a `Singleton<T>` with the projected value.

Though we’ve gone through a bunch of unnecessary object constructions, we end up with what we want.

Furthermore, suppose we have a projection, and we do a `Select` on that: we avoid the problem we have in `LINQ-to-objects`, where we build a projection on top of a projection and end up with multiple objects representing the workflow.

> That last sentence is an oversimplification; in `LINQ-to-objects` there is an optimization for this scenario; we detect `Select-on-Select` (and `Select-on-Where` and so on) and build up a single object that represents the combined projection, but that single object still calls all the delegates. So in that sense there are still many objects in the workflow; there's just a single object coordinating all the delegates.

In this scenario we spend time up front calling the projection on every member of the support once, so we don't need to ever do it later.

We are not trying to make the series of factories here the most efficient it could be; we're creating a lot of garbage. Were we building this sort of system for industrial use, we'd be more aggressive about taking early outs that prevent this sort of extra allocation. What we are trying to illustrate here is that we can use the rules of probability (and the fact that we have pure predicates and projections) to produce distribution objects that give the correct results.

> Of course, none of this fixes the original problems with weighted integer: that in the original constructor, we do not optimize away "all weights zero except one" or "trailing zeros". Those improvements are still left as exercises.

Implementation

The code below is similar to the one in the previous lesson, but notice changes in the file `Distribution.cs`.

> `Select`, `Where` and `ToWeighted` methods have been modified. We have

also removed `Conditioned.cs`

Program.cs

Bernoulli.cs

BetterRandom.cs

Distribution.cs

Episode10.cs

Extensions.cs

IDiscreteDistribution.cs

IDistribution.cs

Projected.cs

Pseudorandom.cs

Singleton.cs

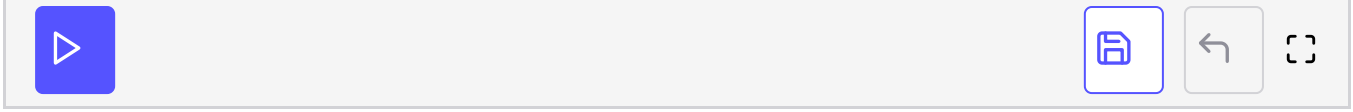
StandardCont.cs

StandardDiscrete.cs

WeightedInteger.cs

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

namespace Probability
{
    using SDU = StandardDiscreteUniform;
    // Weighted integer distribution using alias method.
    public sealed class WeightedInteger : IDiscreteDistribution<int>
    {
        private readonly List<int> weights;
        private readonly IDistribution<int>[] distributions;

        public static IDiscreteDistribution<int> Distribution(params int[] weights) =>
            Distribution((IEnumerable<int>)weights);

        public static IDiscreteDistribution<int> Distribution(IEnumerable<int> weights)
        {
            List<int> w = weights.ToList();
            if (w.Any(x => x < 0) || !w.Any(x => x > 0))
                throw new ArgumentException();
            if (w.Count == 1)
                return Singleton<int>.Distribution(0);
            if (w.Count == 2)
                return Bernoulli.Distribution(w[0], w[1]);
```

```
            return new WeightedInteger(w);
        }

        private WeightedInteger(IEnumerable<int> weights)
        {
            this.weights = weights.ToList();
            int s = this.weights.Sum();
            int n = this.weights.Count;
            this.distributions = new IDistribution<int>[n];
            var lows = new Dictionary<int, int>();
            var highs = new Dictionary<int, int>();

            for (int i = 0; i < n; i += 1)
            {
                int w = this.weights[i] * n;
                if (w == s)
                    this.distributions[i] = Singleton<int>.Distribution(i);

                else if (w < s)
                    lows.Add(i, w);

                else
                    highs.Add(i, w);
            }

            while (lows.Any())
            {
                var low = lows.First();
                lows.Remove(low.Key);
                var high = highs.First();
                highs.Remove(high.Key);
                int lowNeeds = s - low.Value;
                this.distributions[low.Key] = Bernoulli.Distribution(low.Value, lowNeeds).Se

                int newHigh = high.Value - lowNeeds;

                if (newHigh == s)
                    this.distributions[high.Key] = Singleton<int>.Distribution(high.Key);

                else if (newHigh < s)
                    lows[high.Key] = newHigh;

                else
                    highs[high.Key] = newHigh;
            }
        }

        public IEnumerable<int> Support() => Enumerable.Range(0, weights.Count).Where(x => w

        public int Weight(int i) => 0 <= i && i < weights.Count ? weights[i] : 0;

        public int Sample()
        {
            int i = SDU.Distribution(0, weights.Count - 1).Sample();
            return distributions[i].Sample();
        }

        public override string ToString() => $"WeightedInteger[{weights.CommaSeparated()}]"
    }
}
```



We've seen that we can use `Where` to filter a distribution to make a conditioned distribution; we'll look at a more rich and complex way to represent conditional probabilities, and discover a not-so-surprising fact about our distribution interface, in the next lesson.