Problems with Discrete Probability Distributions

In this lesson, we will discuss some of the issues with discrete probability distributions.

WE'LL COVER THE FOLLOWING

Some Issues with Discrete Probability Distributions

So… we have good news and bad news.

The good news is:

- We've described an interface for discrete probability distributions and implemented several distributions.
- We've shown how projecting a distribution is logically equivalent to the LINQ `Select` operator.
- We've shown how conditional probabilities of the form $P(B|A)$ "probability of B given A" are likelihood functions.
- We've shown that combining a prior with a likelihood to form a joint distribution is the application of the monadic bind operator on the probability monad, and implemented this as `SelectMany`.
- We've shown that computing the posterior distribution from the joint distribution can be done by applying a `Where` clause.

- We’ve sketched out a statement-based workflow DSL that can be compiled down to our LINQ operators, and that lowering and then executing programs written in this DSL can compute a “sampling-loop-free” distribution automatically.
- And that distribution object can then be efficiently sampled.

This is all great. But the bad news is that there are some problems; let’s list them in order roughly from smallest to largest:

First Problem

It’s often inconvenient to express weights as integers, particularly when you end up with largish co-prime weights; we’re risking integer overflow when we multiply or add them, and we lack facilities for common operations like “make me a Bernoulli distribution based on this arbitrary double”.

That criticism is very valid, but the problem is fixable. I chose integer weights because it is easy to get integer arithmetic right in simple cases and we do not require any additional library support. A realistic implementation of a weighted discrete distribution would probably choose arbitrary-precision integers, arbitrary-precision rationals or doubles depending on its needs.

Using doubles slightly complicates the math because you either have to make sure that probabilities add up to 1.0 even when double representation error bites you, or you have to deal with situations where the total weight is not 1.0. The latter problem is not so bad; we can always divide through by the total weight if we need to. We’ll return to the more general problem of dealing with double-weighted distributions that do not add up to 1.0 later in this series.

> Since probabilities are always between 0 and 1, and often very small, a common choice is to take the log of the probability as the weight. In that scheme, we can implement multiplying two probabilities together as adding, which is nice, but lots of other operations get harder. This is also a nice representation if you’re doing the math that involves computations of entropy, as entropy is defined as the negative log of the mass function. We won’t go there in this series.

Second problem

The basic idea of both our LINQ-based and statement-based implementations is to take a workflow that might contain conditions — `Where` clauses, in the query form, `condition` statements in the statement form — and infer a distribution object that can be sampled from efficiently.

But our technique for that is for `SelectMany` to produce every possible outcome, and then group them to determine the weights! For example, suppose we had a workflow like

```
var d10 = SDU.Distribution(1, 10);
var sum = from a in d10
          from b in d10
          from c in d10
          from d in d10
          let s = a + b + c + d
          where s >= 20
          select s;
```

If these were sequences instead of distributions, we'd know what happens here: we enumerate all ten thousand possible combinations, run a filter over their sum, and then yield a whole sequence of sums that match the filter.

But that's almost exactly what we're doing with the distributions too! Our query here is syntactic sugar for

```
var d10 = SDU.Distribution(1, 10);
var sum = from a in d10.Support()
          from b in d10.Support()
          from c in d10.Support()
          from d in d10.Support()
          let s = a + b + c + d
          where s >= 20
          group s ...
```

And you know how this goes from our implementation of `SelectMany` on distributions: we weight each member, group them by the outcome, sum up the weights, and so on. The math at the end isn't the expensive part; the number of nested loops we go through to get to that math is!

Now imagine if this were a much more complex workflow, whether written in our query form or our statement form.

> Inferring the result of the workflow means evaluating every possible value for every local/range variable and recording the result.

If there are lots of possible values for lots of variables, we could end up exploring millions or billions of possibilities, some of which are very low probability.

Now, the benefit here is of course that we only do that exploration once, and the probability distribution that comes out the other end is then cheap to use forever; we can memoize it because by assumption everything is pure. But still, it seems odd that we're implementing inference by *running every possible path through the workflow* and recording the results; it seems like we could trade off a small amount of inaccuracy for a significant improvement in speed.

Put another way: in our original naive implementations of `Where` and `SelectMany` we didn't make any inference; we just sampled the underlying distributions and lived with the fact that `Where` could loop a lot. Compare constructing the naive implementation to constructing our current implementation:

- Constructing the naive version of the distribution is $O(n)$ in the size of the number of query operators. In this case, we've got a tiny handful of `SelectMany`s and a `Where`. The number of query operators is usually minimal.
- Constructing the exact distribution is $O(n)$ in the *number of possible combinations of all variables*, in this case, ten thousand. Those ten thousand computations then produce a correct weighted integer distribution.

And compare sampling:

- Sampling the naive version would ultimately do four samples from $d10$,

add them up, and reject the resulting sum a little bit less than half of the time. On average, we would do around seven or eight samples of underlying distributions per one sample of the sum distribution.
- Sampling the exact version is just sampling from a weighted integer, which we know does exactly two samples from its implementation distributions.

And compare the knowledge of the weights:

- The naive version can compute weights cheaply for `Where` conditioned distributions, but not distributions that have `SelectMany` clauses; it's computing the expensive weights.
- The exact version computes all weights exactly for all workflows.

The Right Way?

Which way is the win? Consider that we're never going to call `Sample()`, but we want an accurate inference of the true weights of outcomes given a workflow. When we're computing the posterior distribution of "is this email spam?" we don't care to ever sample from that distribution; we want to know is it 99% or 1%?

We're going to sample millions or billions of times, so keeping the per-sample cost as low as possible is important.

We are sampling and a `Where` clause rejects the vast majority of possible outcomes, causing long waits for a sample in the naive case.

In those three scenarios, our current "exact but expensive inference" step is better. But if those conditions are not met, then the current version is doing a lot of up-front work to save possibly very little work later.

Therefore, we can conclude that:

- It is not immediately obvious where to draw the line that indicates whether the inference optimization's high up-front cost is worth the benefit of cheap sampling later.
- It seems plausible that we could trade off a small amount of inaccuracy in the inference for significant speed improvements in inference.

Third problem

The biggest problem with this system is: *in practice, it only works for discrete distributions with small supports!*

We started this course by making a general-purpose distribution type for discrete or continuous distributions, waved my hands a bit about continuous distributions, and then abandoned them entirely. But they are important!

Moreover, we can’t easily work with discrete distributions where any integer is a possible output, like the geometric distribution or the Poisson distribution. Sure, we can approximate them by restricting their ranges to something reasonable, but it would be nice to be able to have an object that represents a Poisson distribution and not have to worry about the repercussions of the fact that technically, the support is any positive integer.

Conclusion

- We have spent the last 22 lessons of this course solving the easiest possible problems in this space. All the operations we care about on “categorical distributions” admit easy exact solutions to the inference problem.

- If you take away one thing from this course, well, it should be that `System.Random` is way too simple for the needs of modern programmers dealing with stochastic systems. By making a couple simple interfaces and implementations, we can represent very rich probabilistic workflows, compose them, and do math automatically.

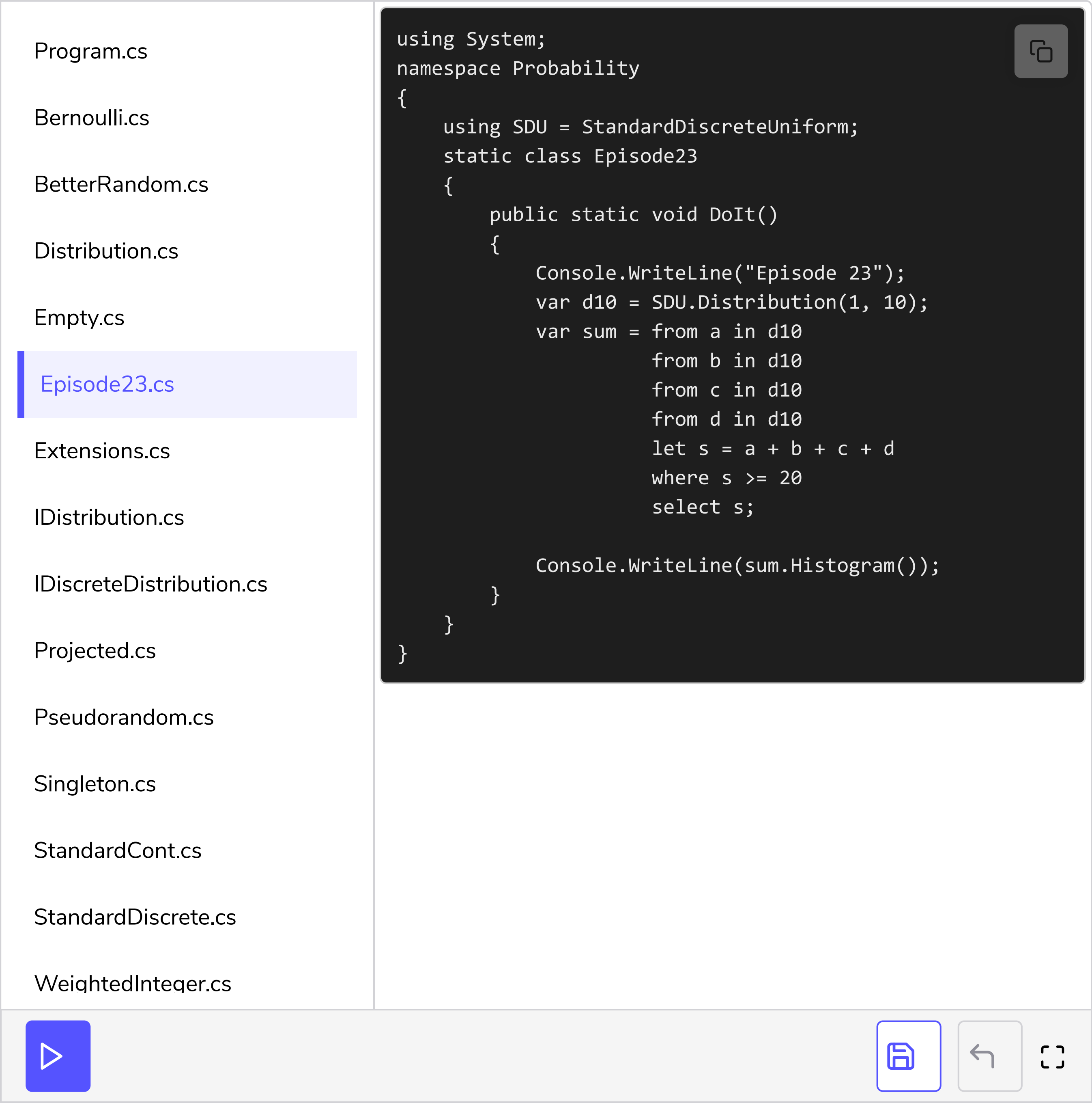
- If you take away two things, the second thing should be that **sampling from a given distribution efficiently is a hard problem**. By restricting the problem enormously, we can make good progress on it, but once we leave the realm of tiny discrete distributions, we will be in some pretty deep water.

We’ve taken small discrete distributions up till now. In the remainder of this course, we are going to sketch out some ways we can use the ideas we’ve developed so far to tackle harder problems in stochastic programming.

Implementation

Implementation

Let’s have a look at the code:



```
using System;
namespace Probability
{
    using SDU = StandardDiscreteUniform;
    static class Episode23
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 23");
            var d10 = SDU.Distribution(1, 10);
            var sum = from a in d10
                      from b in d10
                      from c in d10
                      from d in d10
                      let s = a + b + c + d
                      where s >= 20
                      select s;

            Console.WriteLine(sum.Histogram());
        }
    }
}
```

In the next lesson, we will learn how to use these techniques to model Markov processes.