Correct Implementation of Importance Sampling

In this lesson, we will correct our implementation of Importance Sampling by removing the effect of the constant factor error.

WE'LL COVER THE FOLLOWING

Revisiting Our Previous Example

Let's revisit our example one more time.

Suppose we have our nominal distribution `p` that possibly has "black swans" and our helper distribution `q` which has the same support, but no black swans.

We wish to compute the expected value of `f` when applied to samples from `p`, and we've seen that we can estimate it by computing the expected value of `g`:

```
x => f(x) * p.Weight(x) / q.Weight(x)
```

applied to samples of `q`.

Removing the Constant Factor Error

Unfortunately, in the last two lessons, we saw that **the result will be wrong by a constant factor**; the constant factor is the quotient of the normalization constants of `q` and `p`.

constants of `q` and `p`.

It seems like we're stuck; it can be expensive or difficult to determine the normalization factor for an arbitrary distribution. We've created infrastructure for building weighted distributions and computing posteriors and all sorts of fun stuff, and none of it assumes that weights are normalized so that the area under the PDF is 1.0.

But... **we don't need to know the normalization factors**. We never did.

What do we really need to know? **We need to know the quotient of two normalization constants**. That is less information than knowing two normalization constants, and maybe there is a cheap way to compute that fraction.

Well, let's play around with computing fractions of weights; our intuition is: **maybe the quotient of the normalization constants is the average of the quotients of the weights**. So let's make a function and call it `h`:

```
x => p.Weight(x) / q.Weight(x)
```

What is the Expected Value of `h` When Applied to Samples Drawn From `q`?

Well, we know that it could be computed by:

```
Area(x => h(x) * q.Weight(x)) / Area(q.Weight)
```

But do the algebra: that's equal to

```
Area(p.Weight) / Area(q.Weight)
```

Which is the inverse of the quantity that we need, so we can just divide by it instead of multiplying!

Here's our logic:

1. We can estimate the expected value of `g` on samples of `q` by sampling.
2. We can estimate the expected value of `h` on samples of `q` by sampling.
3. The quotient of these two estimates is an estimate of the expected value

of `f` on samples of `p`, *which is what we've been after this whole time*.

Whew!

> There is one additional restriction that we've got to put on helper distribution `q`: there must be no likely values of x in the support of `q` such that `q.Weight(x)` is tiny but `p.Weight(x)` is extremely large, because their quotient is then going to blow up huge if we happen to sample that value, and that's going to wreck the average.

Modifications in Code

We can now actually implement some code that computes expected values using importance sampling and no quadrature. Let's put the whole thing together, finally: (All the code can be found later in the lesson.)

```
public static double ExpectedValueBySampling<T>(
    this IDistribution<T> d,
    Func<T, double> f,
    int samples = 1000) =>
  d.Samples().Take(samples).Select(f).Average();

public static double ExpectedValueByImportance(
    this IWeightedDistribution<double> p,
    Func<double, double> f,
    double qOverP,
    IWeightedDistribution<double> q,
    int n = 1000) =>
  qOverP * q.ExpectedValueBySampling(
    x => f(x) * p.Weight(x) / q.Weight(x), n);

public static double ExpectedValueByImportance(
  this IWeightedDistribution<double> p,
  Func<double, double> f,
  IWeightedDistribution<double> q,
  int n = 1000)
{
  var pOverQ = q.ExpectedValueBySampling(
    x => p.Weight(x) / q.Weight(x), n);
  return p.ExpectedValueByImportance(f, 1.0 / pOverQ, q, n);
}
```

Look at that; the signatures of the methods are longer than the method bodies! Basically there's only four lines of code here. Obviously we are omitting error handling and parameter checking and all that stuff that would be necessary in a robust implementation, but the point is: even though it took us six math-heavy lessons to justify why this is the correct code to write, actually writing the code to solve this problem is very straightforward.

Once we have that code, we can use importance sampling and never have to do any quadrature, even if we do not give the ratio of the two normalization constants:

```
var p = Normal.Distribution(0.75, 0.09);
Func<double, double> f = x => Atan(1000 * (x - .45)) * 20 - 31.2;
var u = StandardContinuousUniform.Distribution;
var expected = p.ExpectedValueByImportance(f, u);
```

Summing Up

- If we have two distributions `p` and `q` with the same support...
- ... and a function `f` that we would like to evaluate on samples of `p`...
- ... and we want to estimate the average value of `f`...
- ... but `p` has "black swans" and `q` does not.
- We can still efficiently get an estimate by sampling `q`
- Bonus: we can compute an estimate of the ratios of the normalization constants of `p` and `q`.
- Extra Bonus: if we already know one of the normalization constants, we can compute an estimate of the other from the ratio.

Super; are we done?

In the last two lessons we pointed out that there are two problems: we don't know the correction factor, and we don't know how to pick a good `q`. We've only solved the first of those problems.

Implementation

Let's have a look at the code:

Program.cs

Bernoulli.cs

Beta.cs

BetterRandom.cs

Distribution.cs

DistributionBuilder.cs

Empty.cs

Episode36.cs

Extensions.cs

Flip.cs

Gamma.cs

IDiscreteDistribution.cs

IDistribution.cs

IWeightedDistribution.cs

Markov.cs

```
using System;
using static System.Math;
namespace Probability
{
    static class Episode36
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 36 -- Importance sampling for real");

            var p = Normal.Distribution(0.75, 0.09);
            Func<double, double> f = x => Atan(1000 * (x - .45)) * 20 - 31.2;
            var u = StandardContinuousUniform.Distribution;
            Console.WriteLine("Estimate the ratio");
            for (int i = 0; i < 10; ++i)
                Console.WriteLine($"{p.ExpectedValueByImportance(f, u):0.###}");
            Console.WriteLine("Known ratio");
            for (int i = 0; i < 10; ++i)
                Console.WriteLine($"{p.ExpectedValueByImportance(f, 1.0, u):0.###}");
        }
    }
}
```

In the next lesson, we'll dig into the problem of finding a good helper distribution `q`.