Drawing 10e12 samples from a judgmental estimation model

Jorge Sierra, Nuño Sempere

March 28, 2024

Background

Estimators in the tradition of Fermi or Tetlock sometimes create models that seek to capture their subjective credences—as opposed to, for instance, fitting a model to an underlying set of samples.

One way one might go about this is to use the set of idioms that grew from the foretold.io experimental forecasting platform into Squiggle, now a programming language for intuitive estimation. The original iteration of squiggle has been ported to Python (Squiggle.py) and to C (Squiggle.c).

However, those idioms favour relying on Monte Carlo estimation, that is, approximating a model by drawing samples rather than specifying the model exactly. But this approach sometimes has difficulty modelling long-tail behaviour.

To explore those limits, and also just for the sake of the technical challenge, in this paper we describe how to draw 10e12 samples from reasonably complex judgmental estimation model. Then, we show how summary statistics change as we draw more samples. We conclude with [xyz].

1. How to draw 10e9 samples

1.1. Use a fast language

Our time to botec repository compares how fast drawing 10e6 samples from a simple back-of-the-envelope calculation runs in different languages. The comparison depends on our familiarity with the different languages: as we are more familiar with a language, we grow capable of writing better & faster code in it. Nonetheless, speeds are as follows:

Language	Time
C	6.20ms
squiggle.c	7.20ms
go	21.20ms

Language	Time
Nim	41.10ms
Lua (LuaJIT)	68.80ms
Python (numpy)	118ms
OCaml (flambda)	185ms
Squiggle (bun)	384ms
Javascript (node)	395ms
SquigglePy (v0.27)	1,542ms
R (3.6.1)	4,494ms
Python 3.9	11,909ms
Gavin Howard's bc	16,170ms

Readers might have expected this list to include languages such as Rust or zig. FORTRAN, Lisp, Haskell, Java, C#, C++, or Julia are also missing. This is because we are not very familiar with those languages, or because in our attempts to use them we found them too "clunky".

1.2. Use some straightforward tricks

Pass pointers, not values, inline functions. Avoid the overhead of calling a function by inlining.

Define your own primitives, as opposed to calling a library. Reference how to do this here & limits of rng.

Compile to the native architecture. Profile. Compiler flags.

Perhaps concession was defining a mixture One of the few concessions was defining a function to mix several other distributions. This was worse than doing something like:

```
double p = sample_uniform(0,1, seed);
if(p < p1){
  return sample_dist_1(seed);
} else if (p<p1+p2){
  return sample_dist_2(seed);
} else {
  return sample_dist_3(seed);
}
```

but much more convenient

1.3. Introduce parallelism with OpenMP

Introducing parallelism with OpenMP was fairly straightforward.

alignment

1.4. Run the code on a supercomputer with MPI

Talk about challenges here.

Compiling on icc compiler. Requires not using useful gcc extensions, like nested functions. Reader can find recipes in the makefile.

1.5. Run the code for a long time

Finally, we can get around 3x more samples by running the code for 18 hours instead of 6 hours, getting to 10T samples. [don't do this yet]. Increasing time quickly becomes untenable, however.

2. Results as samples increase

2.1. Mean and standard deviation

2.2. Minimum and maximum

2.3. The shape of the distribution represented as a histogram

[to do]

Conclusion

Rethink Priorities' models puny struggled (https://forum.effectivealtruism.org/posts/pniDWyjc9vY5sjGre/reth
priorities-cross-cause-cost-effectiveness-model) to reach more than 150k samples. Eventually, with some optimizations, they moved to the billions. Much more is possible with a focus on speed.