Support and influence analysis for visualizing posteriors of probabilistic programs

Long Ouyang

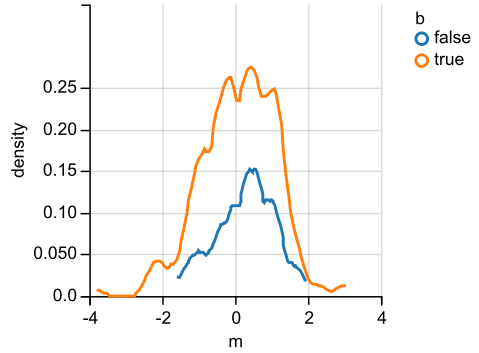
longouyang@post.harvard.edu

1. Introduction

A common way to interpret the results of any computational model is to visualize its output. For probabilistic programming, this often means visualizing a posterior probability distribution. The `webppl` language has a visualization library called `webppl-viz` that facilitates this process. A useful feature of `webppl-viz` is that it does some amount of *automatic* visualization—the user simply passes in the posterior and the library tries to construct a useful visual representation of it. For instance, consider this posterior, with a discrete component b and continuous component m:

```
var dist = Infer(
  {method: 'MCMC', samples: 1000},
  function () {
    var b = flip(0.7)
    var m = gaussian(0, 1)
    var y = gaussian(m, b ? 4 : 2)
    condition(y > 0.3)
    return {b: b, m: m}
  });
```

We visualize it by calling `viz(dist)`, which gives us this picture:



This is a reasonable choice. There are two density curves for m—an orange curve for when b is true, and a blue curve for when b is false. `webppl-viz` often produces helpful graphs but, as we will see, it can also produce graphs with obvious flaws. One reason for this is that `webppl-viz` defines a limited set of variable types for visualization and it uses heuristics to guess the types particular components in a posterior sample. Another issue is that `webppl-viz` does not scale well with the number of components in a posterior. There are only a handful of ways to visually encode data, and `webppl-viz` gives up if the dimensionality of the posterior exceeds the number of available visual channels. In this article, I argue that methods from programming languages research suggest solutions to these two problems.

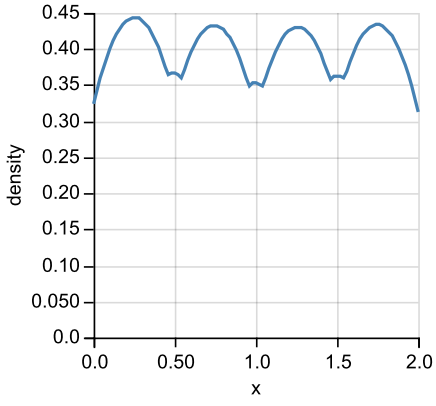
2. Richer types through support analysis

A popular typology in visualization research distinguishes between three kinds of variables: nominal (e.g., Coke vs. Pepsi vs. Sprite), ordinal (terrible vs. neutral vs. excellent), and quantitative (e.g, real numbers). `webppl-viz` makes automatic visualization choices by mapping posterior components to these variable types and then using principles from the psychology of graph perception to make aesthetic choices, e.g., size is a bad way of showing nominal data because it communicates relative magnitude information that doesn't actually exist in the raw data (cf. Mackinlay 1986).

Currently, `webppl-viz` heuristically infers the types of posterior components. If a component does not contain all numbers, it is assumed to be nominal. If a component contains numbers that are all integers, it is assumed to be ordinal. If a component contains numbers that are not all integers, it is assumed to be quantitative. These heuristics are useful but there is much room for improvement. For example, consider this model, which just forward-samples from a regular grid containing non-integer values:

```
viz(Infer(
  {method: 'MCMC', samples: 1000},
  function() {
    var x = uniformDraw([0, 0.5, 1, 1.5, 2])
    return {x: x}
  }))
```

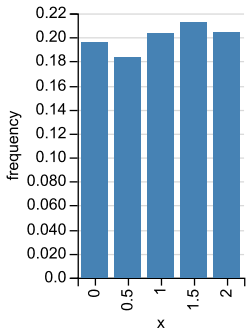
We get a misleading result:



The presence of non-integers in the sample has led `webppl-viz` to assume that x is a real-valued variable and then depict it using a density plot. The combination of type choice and plot choice is surprisingly bad here. Standard non-parametric density estimation performs poorly for multimodal non-smooth distributions, which is exactly the sort of posterior we have. Visually, the most likely values in the plot—0.25, 0.75, 1.25, and 1.75—are not even in the support of the distribution we're drawing from! How could we do better? Inspecting the program, we can tell that x only takes one

of five values, so a better plot would be a histogram over these values. More generally, the nominal-ordinal-quantitative typology is too coarse—we need a more precise notion of type. A natural candidate is the set of values that a variable could take—its support. And mechanically calculating this support information will have to do much more than simple heuristics—we will have to do some abstract interpretation.

To make variable supports available to `webppl-viz`, I modified `webppl`, adding a dynamic abstract interpretation mechanism that operates in tandem with the concrete program evaluation by tagging concrete values with abstract states. I manually declared the supports for base distributions; sampling from one of these distributions returns a value that is tagged with a support (to ensure that variables derived from random values carry appropriately updated supports, I modified `webppl`'s to merge supports using interval arithmetic). Now, rerunning the example above gives something more sensible:



This dynamic tagging approach is both easier to implement and more powerful than static approaches (e.g., k-CFA), which lose precision if random variables are used to direct control flow or to index into arrays. In addition, the support of a variable can depend on other variables in complex ways:

```
var x = randomInteger(10);
var y = x * beta(2,2);
condition(x % 3 == 0);
```

Here, the *a priori* support of y is $[0, x]$. But the later conditioning statement refines our estimate—y's support is one of $[0, 3]$, $[0, 6]$, or $[0, 9]$. Incorporating the effects of belief update into a static approach would be challenging. But we get this for free using a dynamic approach, as this piggybacks on top of inference. However, the dynamic approach does have a cost—propagating supports for every sample incurs some overhead. This might be mitigated by only running the dynamic support tagging for a subset of the samples during inference.

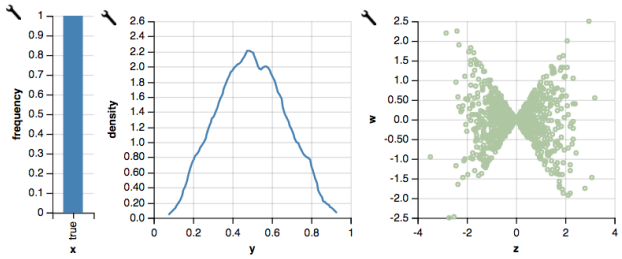
3. Showing more variables with influence analysis

There are only a few visual channels we can use to represent information—position in x and y, color, size, shape, and perhaps a handful more. So the dimensionality of the posterior can easily outstrip our ability to visualize. For instance, visualizing this 4-dimensional distribution is not possible without omitting or severely summarizing certain components:

```
Infer({method: 'forward', samples: 1000},
      function() {
        var x = flip(0.5);
        var y = beta(4,4);
        var z = gaussian(0,1);
        var w = uniform(-z, z);
        condition(x == true)
        return {x: x, y: y, z: z, w: w}})
```

However, the special structure of probability distributions can help. In particular, if two components are *independent*, then there's no need to try to show a single graph of their joint distribution. Instead, we can show the two graphs of their marginals, as these imply the joint distribution in the case of independence.

I take the following approach to computing the independence structure of the posterior. First, I statically analyze the source code of a model to construct an influence graph between the random variables in the posterior. In particular, a variable x is marked as directly depending on another variable y if x's declaration references y.[1] I then use the Bayes ball algorithm (Shachter 1998) to factorize the posterior, i.e., to partition the posterior components into cliques (subsets of components that are independent of all other subsets). It is then possible to call `viz()` on each clique. For the example above, we obtain:



We get univariate distributions for x and for y, as well as the joint distribution on w and z.

4. Future directions

Currently, support analysis is implemented only for numeric values. But a primary selling point of probabilistic programming is that it enables probabilistic modeling over richer representations, like strings, trees, and networks. Accommodating such objects will require designing efficient support representations. For influence analysis, we might be able to further visualize higher dimensional data by using multidimensional scaling techniques or relaxing from perfect to approximate independence (e.g., judging that $A \perp B$ when $D_{\mathrm{KL}}(P(A \mid B = b) \,||\, P(A)) < \epsilon$ for all b).

Another interesting aspect to explore is how the program analysis techniques used for the task of posterior visualization relate to and interact with program analysis for other tasks (e.g., improving inference through dead code elimination, program transformations, or insertion of heuristic factors). For instance, a by-product of the influence analysis is information about possible ways to correctly reorder the program execution.

Acknowledgements

This work was supported as part of the Future of Life Institute (futureoflife.org) FLI-RFP-AI1 program, grant number 2016-162768.

References

J. Mackinlay. Automating the design of graphical presentations of relational information. In *TOG '86*, April 1986.

R. D. Shachter. Bayes-ball: Rational Pastime (for Determining Irrelevance and Requisite Information in Belief Networks and Influence Diagrams) In *UAI '98*, July 1998.

[1] This approach largely suffices, as `webppl` is a static single assignment language that largely forbids mutation. `webppl` does, however, provide a limited facility for mutation: the `globalStore` variable. This simple influence analysis would fail in cases where dependence between variables is mediated by `globalStore` contents. Such cases would require a more complete approach, such as that of Might & Prabhu, 2009.