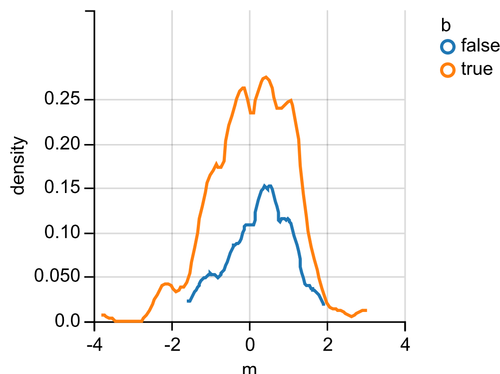
Support and influence analysis for visualizing posteriors of probabilistic programs

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 they wish to inspect and the library tries to construct a useful visual representation of it. For instance, consider this posterior:

```
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, an 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 infers what the variable types in a posterior sample are using heuristics. Another issue is that `webppl-viz` does not scale well with the number of variables 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.

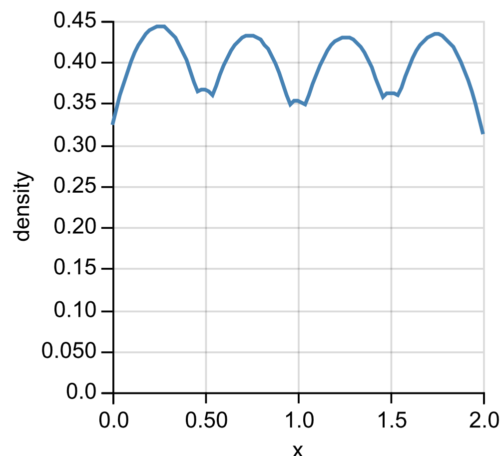
2. Richer types through support analysis

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 visualization choices automatically by mapping components of the posterior 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 ordinal data because it communicates relative magnitude information that doesn't actually exist in the raw data (see Mackinlay 1986 for more).

Currently, `webppl-viz` heuristically infers the types of components in the posterior. 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:



[Copyright notice will appear here once 'preprint' option is removed.]

The presence of non-integer numbers in the posterior sample has lead `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? Looking at the structure of the program, we can tell that x can only take one of five values and that a better plot would be simply 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 lightweight 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:

```
var x = exponential({a: 3});
x;          // => some random value
x.support; // => {lower: 0, upper: Infinity}
```

To make sure that variables derived from random values carry appropriately updated supports, I modified `webppl`'s arithmetic functions to merge supports. Thus, a variable defined as the sum of two random variables has the appropriate tags:

```
var a = uniformDraw([0.1, 0.3, 0.7]),
    b = uniformDraw([10, 20]),
    c = a + b; // support is [10.1, 10.3, 10.7,
               //             20.1, 20.3, 20.7]
```

3. Showing more variables with influence analysis

The number of visual dimensions that we can use to represent information is limited—position in two dimensions, color, size, shape, and perhaps a handful more. So the dimensionality of the posterior can easily outstrip our ability to visualize. For instance, visualizing a 4-dimensional surface is not possible without omitting or severely summarizing certain components.

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. So while a 4-dimensional surface might be in general impossible to visualize, a 4-d posterior distribution with some independence structure in the components might be possible.

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.

[1] This simple approach largely suffices, as `webppl` is a (mostly) static single assignment language. `webppl` does, however, provide a facility for mutation: the `globalStore` variable. This simple influence analysis would fail on models that where dependence between variables is mediated by `globalStore` contents. Such cases would require a more complete approach, such as CFA2.

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 relaxing from perfect to approximate independence.

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).

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.