PROBABILISTIC PROGRAMMING LANGUAGES?

Maxwell Clarke – maxeonyx@gmail.com – 300559026

COMP440 Essay – Thu 17 Jun 2021 – Victoria University of Wellington

ABSTRACT

Probabilistic programming languages (PPLs) lie at the intersection of machine learning and programming languages. They allow easy and rapid development of statistical models, using approximate Bayesian inference algorithms.

There are currently many PPLs being actively developed, however only a few are widely used for modeling. I compare three of these on some example programs to give an overview of some of the differences between the languages.

Contents

1. MOTIVATION

I came into this course with no
Questions:

1. What are probabilistic programming languages?
2. Why are there so many and what are the differences between different languages?
3. What are the limitations?
4. Should I be using probabilistic programming languages?

2. WHAT IS A PROBABILISTIC PROGRAMMING LANGUAGE?

2.1. Programming Languages

Programming languages exist to simplify interactions with a computer system. They are essentially collections of useful functions, syntax etc. which allow a reduction the amount of information required for a user to express a certain program. Most general purpose programming languages allow defining libraries of functions to express procedures or concepts without repetition. Creating a domain-specific language typically allows even further reduction by removing redundant information required by a general purpose language but not for a specific domain.

Probabilistic programming languages are domain specific languages for defining probabilistic models. They allow concise definition of a model which can be used to sample from some distribution, and importantly to then condition this distribution on some data or observations and then sample from the posterior.

The purpose of probabilistic programming languages is:

- To make defining and performing inference on probabilistic models more ergonomic, when compared to libraries - improving development time, readability and correctness.
- To enable the creation of more complex models, by allowing user defined functions, and correctly supporting inference on programs with mixed and complex data types, unbounded loops, recursive functions, and different variables in different branches.

2.2. How do they work?

A probabilistic program represents the distribution over all possible program traces. A *sample* statement introduces randomness, and branches the program trace.

This by itself would simply be performing simulations, and we would not need a dedicated programming language. However, the key difference between a probabilistic program and a simulation is that a probabilistic program can also condition on observations of the program variables in order to perform inference. In *Practical Probabilistic Programming* [1], Pfeffer says "A probabilistic program is like a simulation that you can analyze, not just run.". This extra step is done using approximate Bayesian inference algorithms.

2.3. Bayesian Inference

Bayesian inference is a technique in statistics which updates a prior probability distribution $p(x)$ over some hypothesis or model parameters, and finds the posterior $p(x \mid y)$ as more information or data y becomes available. One common kind of inference is to update a predictive distribution given more data, such as in sequence prediction. Another common kind of inference is model training - updating a distribution over model parameters given a dataset.

For simple models, Bayesian inference can sometimes be done analytically or exactly, however for many kinds of models Bayesian inference is intractable. For these models, we can find a model-specific approximation, or we can use general approximate inference methods.

Probabilistic programming languages allow us to define models and perform inference on them using some of these general approximate inference methods, such as importance sampling and Markov chain Monte Carlo methods. They perform sampling and return an empirical distribution over the posterior.

3. WHY ARE THERE SO MANY?

3.1. Overview

There are a lot of active probabilistic programming languages. Each has a history and motivation behind its existence. Some notable ones are listed below.

BUGS (*Bayesian Inference using Gibbs Sampling*) [2], was one of the first PPLs (sometimes called instead a PPS (Probabilistic Programming System)). BUGS only supports continuous variables and does not support user defined functions [1].

Stan [3] is one of the most popular PPLs, used widely in statistics. Stan introduced the Markov chain Monte Carlo method called NUTS (No U-Turn Sampling) [4]. Stan is very performant, because it translates models into C++, which are then compiled into native code.

Pyro [5] is a fluent library for Python open sourced by Uber, which uses PyTorch or JAX to provide automatic differentiation. Pyro focuses strongly on the stochastic variational inference (SVI) inference algorithm.

WebPPL [6] is a JavaScript-like PPL which can be run and embedded in a web browser.

BLOG [7] is a hybrid between probabilistic and logic programming languages, allowing "open-world" modeling.

Anglican [8] aims to distill recent PPL research into a practical tool for the Clojure ecosystem.

Venture [9] is a higher-order PPL which aims to allow correctly combining different exact and approximate inference methods.

Turing.jl [10] is a fluent library for the Julia programming language. Turing.jl is to Julia as Pyro is to Python.

In my opinion there is a split between languages that exist primarily as programming language research projects (BLOG, Venture), vs. those that exist primarily as tools (BUGS, Stan, Pyro).

In the following sections I compare the ergonomics, performance and flexibility of three languages: Stan, Pyro, and the language I implemented in the Assignments: "ThisPPL".

3.2. Comparison: Sprinkler model

As a first comparison between Stan, Pyro and ThisPPL, I attempted to implement the classic "Sprinkler" model in each. The model is as follows:

- The weather may be cloudy.
- The weather may be raining, which is more likely if it is cloudy.
- The sprinkler may be on, which is less likely if it is cloudy.
- The grass is likely to be wet if the sprinkler is on, or the weather is raining.

The Stan implementation of this model is shown below in Listing 1.

Listing 1. Sprinkler in Stan

```
data {
   int<lower=0, upper=1> sprinkler_on;
   int<lower=0, upper=1> grass_wet;
}
parameters {
   int<lower=0, upper=1> cloudy;
   int<lower=0, upper=1> raining;
}
model {
    // It's cloudy half the time.
    cloudy ~ bernoulli(0.5);

    if (cloudy) {
        // If it's cloudy it's usually raining.
        raining ~ bernoulli(0.8);
    } else {
        // Sometimes it rains when it's not cloudy.
        raining ~ bernoulli(0.2);
    }

    if (cloudy) {
        // We don't usually turn the sprinkler on when
             it's cloudy
        sprinkler_on ~ bernoulli(0.1);
    } else {
        // We turn it on half of the time when it's fine.
        sprinkler_on ~ bernoulli(0.5);
    }

    if (sprinkler_on && raining) {
        // The grass is almost certainly wet if it's
             raining and the sprinkler is on.
        grass_wet ~ bernoulli(0.99);
    } else if (sprinkler_on || raining) {
        // the sprinkler and the rain each have a 90%
             chance of making the grass wet
        grass_wet ~ bernoulli(0.9);
    } else {
        // if neither the sprinkler nor the rain are
             going then the grass is almost certainly dry.
        grass_wet ~ bernoulli(0.01);
    }
}
```

Stan does not support this model! In particular, Stan does not support latent discrete parameters, here the boolean variables *cloudy*, *sprinkler_on* and *raining*. When attempting to run the code pictured, we get the error "Parameters cannot be integers". There is no neat way around this - the Stan docs

The Pyro implementation of this model is shown below in Listing 2.

Listing 2. Sprinkler in Pyro

```
import torch
import pyro
from matplotlib import pyplot as plt

torch_true = torch.ones(())
torch_false = torch.zeros(())

def weather():
    # It's cloudy half the time.
    cloudy = pyro.sample('cloudy',
         pyro.distributions.Bernoulli(0.5))

    if cloudy:
        # If it's cloudy it's usually raining.
        raining = pyro.sample('raining',
             pyro.distributions.Bernoulli(0.8))
    else:
        # Sometimes it rains when it's not cloudy.
        raining = pyro.sample('raining',
             pyro.distributions.Bernoulli(0.2))

    if cloudy:
        # We don't usually turn the sprinkler on when
             it's cloudy
        sprinkler_on = pyro.sample('sprinkler_on',
             pyro.distributions.Bernoulli(0.1))
    else:
        # We turn it on half of the time when it's fine.
        sprinkler_on = pyro.sample('sprinkler_on',
             pyro.distributions.Bernoulli(0.5))

    if sprinkler_on and raining:
        # The grass is almost certainly wet if it's
             raining and the sprinkler is on.
        grass_wet_theta = 0.99
    elif sprinkler_on or raining:
        # the sprinkler and the rain each have a 90%
             chance of making the grass wet
        grass_wet_theta = 0.9
    else:
        # if neither the sprinkler nor the rain are going
             then the grass is almost certainly dry.
        grass_wet_theta = 0.01
    grass_wet = pyro.sample('grass_wet',
         pyro.distributions.Bernoulli(grass_wet_theta))

    return torch.tensor([cloudy, raining, sprinkler_on,
         grass_wet])

def conditioned_weather():
    pyro.condition(weather, data={"grass_wet":
         torch_true})
```

```
# nuts_kernel = pyro.infer.NUTS(conditioned_weather)

# mcmc = pyro.infer.MCMC(nuts_kernel,
#     num_samples=10000,
# )
# mcmc.run(weather)
# samples = mcmc.get_samples()
# print(samples)

importance = pyro.infer.Importance(conditioned_weather,
     guide=None, num_samples=1000)
thing = importance.run()
print(thing.get_normalized_weights())

def plot_bool_hist(ax, data, weights):
    data = np.array(data)
    ax.set_xlabel("Value")
    ax.set_ylabel("Mass")

    ax.set_xticks([0.5, 1.5])
    ax.set_xticklabels(["false", "true"])

    ax.hist(data, density=True, weights=weights, bins=2)
```

Pyro has a very confusing API for performing inference on this model. Because Pyro is a library in Python, note that it is also more verbose than Stan as well. Also note that Pyro only accepts tensors as random variables, which provides limitations to the flexibility of the programs. Pyro does support discrete variables, but booleans must be represented as integers. It does not support compound objects or heterogeneous arrays as random variables.

Despite being verbose, Pyro has the benefit of being able to inter-operate with Python code, including importing any Python library, such as matplotlib for visualizations.

Listing 3. Sprinkler in ThisPPL

```
(let [
    cloudy (sample (flip 0.5))
    raining (sample (if cloudy
                        (flip 0.8)
                        (flip 0.2)))
    sprinkler_on (sample (if cloudy
                        (flip 0.1)
                        (flip 0.5)))
    grass_wet (sample (if (and sprinkler_on raining)
                        (flip 0.99)
                        (if (or sprinkler_on raining)
                            (flip 0.9)
                            (flip 0.01))))
    ]
    (observe grass_wet true)
    [cloudy raining sprinkler_on grass_wet])
```

This is the most concise of the three. Stan is more verbose because it has static typing, and Pyro is more verbose because it is a DSL within Python. Similarly to Stan, configuration of the inference is done through commandline.

3.3. Comparison: Linear Regression

As a second comparison, I implemented the basic linear regression model in all three languages, shown below in Listings 4 to 6. This time all three languages do support this.

Listing 4. Linear Regression in Stan

```
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
}
parameters {
    real slope;
    real bias;
}
model {
    slope ~ normal(0, 5);
    bias ~ normal(0, 5);
    y ~ normal(bias + slope * x, 1);
}
```

Listing 5. Linear Regression in Pyro

```
from pyro.primitives import sample
import torch
import pyro
from matplotlib import pyplot as plt

torch_true = torch.ones(())
torch_false = torch.zeros(())

def linear():
    x_data, y_data = [1, 2, 3, 4, 5, 6],
         torch.tensor([2.2, 4.2, 5.5, 8.3, 9.9, 12.1])
    slope = pyro.sample('slope',
         pyro.distributions.Normal(0, 5))
    bias = pyro.sample('bias',
         pyro.distributions.Normal(0, 5))

    for i in range(len(x_data)):
        x = x_data[i]
        mu = x * slope + bias
        y = pyro.sample(f"y_{i}",
             pyro.distributions.Normal(mu, 1),
             obs=y_data[i])

nuts_kernel = pyro.infer.NUTS(linear)

mcmc = pyro.infer.MCMC(nuts_kernel,
    num_samples=100,
)
mcmc.run()
mcmc.summary()
samples = mcmc.get_samples()

print(samples)
```

Listing 6. Linear Regression in ThisPPL

```
(defn observe-data [_ data slope bias]
        (let [xn (first data)
              yn (second data)
              zn (+ (* slope xn) bias)]
          (observe (normal zn 1.0) yn)
          (rest (rest data))))

(let [slope (sample (normal 0.0 10.0))
             bias  (sample (normal 0.0 10.0))
             data (vector 1.0 2.1 2.0 3.9 3.0 5.3
                          4.0 7.7 5.0 10.2 6.0 12.9)]
   (loop 6 data observe-data slope bias)
   (vector slope bias))
```

On this program Stan is now the most concise of the three due to implicit vector maths and implicit observation statements (note that the observation values come from a separate file not here pictured.

Pyro and Stan are dramatically different in sampling speed on this program, as shown in Table 1. This is a factor of nearly 1000x, which even more significant than we would expect from the difference between Python and compiled binary.

Language	# Samples	Time
Stan	10000 samples	0.215 s
Pyro	10000 samples	158 s

Table 1. Performance comparison between Pyro and Stan, both using NUTS MCMC with 10000 samples, 1 sample thinning and 1000 sample warmup.

4. LIMITATIONS OF PPLS

Stand-alone PPLs such as Stan and ThisPPL suffer from poor interoperability with existing tools compared to fluent DSLs like Pyro (Python), Anglican (Clojure) and Turing.jl [10] (Julia).

Most of the remaining limitations to PPLs are equivalent to the limitations of Bayesian inference.

5. IDEAS FOR IMPROVEMENTS TO PPLS

I see three areas where improvements to PPLs can be made.

1. Inference algorithm improvements
2. Language ergonomic improvements
3. Type system and correctness improvements

5.1. Inference algorithm improvements

The inference algorithms used by PPLs have changed over time. BUGS [2], one of the first systems, supported only Gibbs sampling. Systems with MCMC methods were then introduced and significant improvements such as Stan's NUTS sampler have been introduced as recently as 2014, which has since adopted by other projects such as Pyro. Current languages whose authors are working on improved inference algorithms include Venture [9] and Daphne [11].

5.2. Language ergonomic improvements

As we saw earlier, in some languages we are not free to use any type of variable. Stan restricts discrete variables to be observed variables only, and Pyro restricts variables to those which can be represented via a torch tensor object. Both libraries have these restrictions in order to support more, and more performant inference algorithms. However, restrictions are ultimately undesirable. Improving the diversity of data types usable within the language would result in a better experience. Venture [9] appears to be researching this.

Additionally, it appears that few of these languages have built-in tools for visualizing the distributions that they generate. WebPPL has some limited tools, and the library-PPLs Pyro and Turing.jl have easy interoperability with existing plotting libraries. However, Stan and Daphne for example do not appear to have integrated visualization tools. Especially in standalone PPLs like these, the ability to graph and interactively investigate the relationship between variables in the posterior distribution would be very useful. This could be an easy area for improvements. I am not sure which languages are currently working on this.

5.3. Type systems for PPLs

As far as I can tell, most of the mentioned PPLs are dynamically typed languages, with the exception of Stan and possibly Turing.jl. I expect there is some room for type systems in probabilistic languages, to improve ergonomics and prevent common mistakes. Here are some ideas, which I did not have time to research:

1. Tensor shape types
2. Types to restrict the usage of certain language features with certain inference algorithms, such as preventing the use of discrete variables with gradient-based methods such as HMC and NUTS.

6. SUMMARY: SHOULD I BE USING PPLS?

My answer: Yes (sometimes). The current languages are advanced enough that if you ever find yourself writing a simulation in your language of choice to express an inference problem, you should likely instead reach for a PPL. By learning a general purpose PPL and having it in your toolbox, you can create and test probabilistic models quckly. This may be a niche task, but current PPLs handle it well.

References

[1] A. Pfeffer, *Practical probabilistic programming*. Manning Publ., 2016.

[2] D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks, "Bugs 0.5: Bayesian inference using gibbs sampling manual (version ii)," *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, pp. 1–59, 1996.

[3] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: a probabilistic programming language." *Grantee Submission*, vol. 76, no. 1, pp. 1–32, 2017.

[4] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo." *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1593–1623, 2014.

[5] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep Universal Probabilistic Programming," *Journal of Machine Learning Research*, 2018.

[6] N. D. Goodman and A. Stuhlmüller, "The Design and Implementation of Probabilistic Programming Languages," http://dippl.org, 2014, accessed: 2021-06-16.

[7] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov, "1 blog: Probabilistic models with unknown objects," *Statistical relational learning*, p. 373, 2007.

[8] F. Wood, J. W. Meent, and V. Mansinghka, "A new approach to probabilistic programming inference," in *Artificial Intelligence and Statistics*. PMLR, 2014, pp. 1024–1032.

[9] V. Mansinghka, D. Selsam, and Y. Perov, "Venture: a higher-order probabilistic programming platform with programmable inference," *arXiv preprint arXiv:1404.0099*, 2014.

[10] K. Xu, H. Ge, and contributors., "Turing.jl," https://github.com/TuringLang/Turing.jl, 2021, accessed: 2021-06-16.

[11] C. Weilbach, B. Beronov, W. Harvey, and F. Wood, "The Daphne probabilistic programming compiler," https://github.com/plai-group/daphne, 2020, accessed: 2021-06-16.