Probabilistic Programming in Anglican

David Tolpin(✉), Jan-Willem van de Meent, and Frank Wood

Department of Engineering Science, University of Oxford, Oxford, UK
{dtolpin,jwvdm,fwood}@robots.ox.ac.uk

Abstract. Anglican is a probabilistic programming system designed to interoperate with Clojure and other JVM languages. We describe the implementation of Anglican and illustrate how its design facilitates both explorative and industrial use of probabilistic programming.

Keyword: Probabilistic programming

1 Introduction

For data science practitioners, statistical inference is typically but one step in a more elaborate analysis workflow. The first stage of this work involves data acquisition, pre-processing and cleaning. This is often followed by several iterations of exploratory model design and testing of inference algorithms. Once a sufficiently robust statistical model and corresponding inference algorithm have been identified, analysis results must be post-processed, visualized, and in some cases integrated into a wider production system.

Probabilistic programming systems [1–3,9] represent generative models as programs written in a specialized language that provides syntax for the definition and conditioning of random variables. The code for such models is generally concise, modular, and easy to modify or extend. Typically inference can be performed for any probabilistic program using one or more generic inference techniques provided by the system backend, such as Metropolis-Hastings [3,8, 10], Hamiltonian Monte Carlo [7], expectation propagation [5], and extensions of Sequential Monte Carlo [4,6,9] methods. Although these generic techniques are not always as statistically efficient as techniques that take advantage of model-specific optimizations, probabilistic programming makes it easier to optimize models for a specific application in a manner that is efficient in terms of the dimensionality of its latent variables.

While probabilistic programming systems shorten the iteration cycle in exploratory model design, they typically lack basic functionality needed for data I/O, pre-processing, and analysis and visualization of inference results. In this demonstration, we describe the implementation of Anglican (http://bitbucket.org/dtolpin/anglican/), a probabilistic programming language that tightly integrates with Clojure (http://clojure.org/), a general-purpose programming language that runs on the Java Virtual Machine (JVM). Both languages share a common syntax, and can be invoked from each other. This allows Anglican programs to make use of a rich set of libraries written in both Clojure and Java. Conversely Anglican allows intuitive and compact specification of models for which inference may be performed as part of a larger Clojure project.

A. Bifet et al. (Eds.): ECML PKDD 2015, Part III, LNAI 9286, pp. 308–311, 2015.
DOI: 10.1007/978-3-319-23461-8_36

2 Design Outline

An Anglican program, or *query*, is compiled into a Clojure function. When inference is performed with a provided algorithm, this produces a sequence of return values, or *predicts*. Anglican shares a common syntax with Clojure; Clojure functions can be called from Anglican code and vice versa. A simple program in Anglican can look like the following code:

```
(defquery models
  "chooses a distribution which describes the data"
  (let [;; Model -- randomly choose a distribution and parameters
        dist (sample (categorical [[uniform-discrete 1]
                                   [uniform-continuous 1]
                                   [normal 1]
                                   [gamma 1]]))
        a (sample (gamma 1 1)) b (sample (gamma 1 1))
        d (dist a b)]
    ;; Data  --- samples from the unknown distribution
    (observe d 1) (observe d 2) (observe d 4) (observe d 7)
    ;; Output --- predicted distribution type and parameters
    (predict :d (type d))
    (predict :a a) (predict :b b)))
```

Internally, an Anglican query is represented by a computation in *continuation passing style* (CPS), and inference algorithms exploit the CPS structure of the code to intercept probabilistic operations in an algorithm-specific way. Among the available inference algorithms there are Particle Cascade [6], Lightweight Metropolis-Hastings [8], Iterative Conditional Sequential Monte-Carlo (Particle Gibbs) [9], and others. Inference on Anglican queries generates a lazy sequence of samples, which can be processed asynchronously in Clojure code for analysis, integration, and decision making.

Clojure (and Anglican) run on the JVM and get access to a wide choice of Java libraries for data processing, networking, presentation, and imaging. Conversely, Anglican queries can be called from Java and other JVM languages. Programs involving Anglican queries can be deployed as JVM *jars*, and run without modification on any platform for which JVM is available.

3 Usage Patterns

Anglican is suited to rapid prototyping and exploration, on one hand, and inclusion as a library into larger systems for supporting inference-based decision making, on the other hand.

For exploration and research, Anglican can be run in Gorilla REPL (http://gorilla-repl.org/); a modified version of Gorilla REPL better suited for Anglican

is provided. Gorilla REPL is a notebook-style environment which runs in browser and serves well as a workbench for rapid prototyping and checking of ideas. Figure 1 shows a fragment of an Anglican worksheet in the browser:

Let us visualize the results: red is blind, green is myopic, blue is semioptimal.

```
(let [policy-rewards (fn [f]
                       (map (fn [n r] [n (f r)])
                            +numbers-of-questions+ rewards))]
  (def blind-rewards (policy-rewards #(nth % 0)))
  (def myopic-rewards (policy-rewards #(nth % 1)))
  (def semioptimal-rewards (policy-rewards #(nth % 2))))

(plot/compose
  (plot/list-plot blind-rewards :joined true :color "#ff0000"
                  :plot-range [[0 (reduce max +numbers-of-questions+)] [0. 1.]]
                  :x-title "number of questions"
                  :y-title "success rate")
  (plot/list-plot myopic-rewards :joined true :color "#00ff00")
  (plot/list-plot semioptimal-rewards :joined true :color "#0000ff"))
```

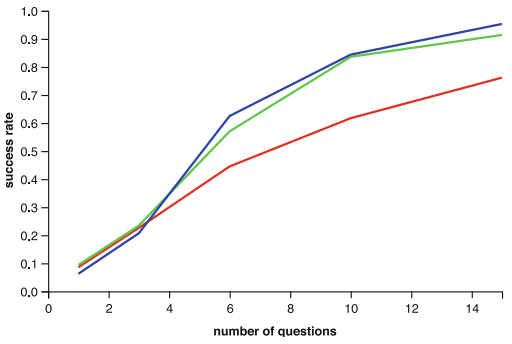

Fig. 1. Anglican worksheet fragment. Post-processed inference results shown in a plot.

Library use is inherent to the Anglican's design for interoperability with Clojure. An Anglican query, along with supporting functions written in either Anglican or Clojure, can be encapsulated in a Clojure module and called from other modules just like Clojure function. Additionally, Anglican functions common for queries of a particular type or structure, such as state-space models or decision-making queries, can be wrapped as libraries and re-used.

4 Anglican Examples

Anglican benefits from a community-maintained collection of problem examples (https://bitbucket.org/fwood/anglican-examples), styled as Gorilla REPL worksheets. Each example is a case study of a problem involving probabilistic inference, includes problem statement, explanations for the solution, and a graphical presentation of inference results. Some of the included examples are:

- Indian GPA,
- Complexity Reduction,
- Bayes Net,
- Kalman Smoother,
- Gaussian Mixture Model,
- DP Mixture Model,
- Hierarchical Dirichlet Process,
- Probabilistic Deterministic Infinite Automata,
- Nested Number Guessing,
- Maximum Likelihood for Logistic Regression.

Anglican users are encouraged to contribute examples, both demonstrating advantages of probabilistic programming and presenting challenges to the current state-of-art of inference algorithms, to the repository.

Acknowledgments. This work is supported under DARPA PPAML through the U.S. AFRL under Cooperative Agreement number FA8750-14-2-0004.

References

1. Goodman, N.D., Stuhlmüller, A.: The Design and implementation of probabilistic programming languages (2015). http://dippl.org/ (electronic; retrieved March 11, 2015)
2. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: Proc. of Uncertainty in Artificial Intelligence (2008)
3. Mansinghka, V.K., Selsam, D., Perov, Y.N.: Venture: a higher-order probabilistic programming platform with programmable inference (2014). CoRR abs/1404.0099
4. van de Meent, J.W., Yang, H., Mansinghka, V., Wood, F.: Particle gibbs with ancestor sampling for probabilistic programs. In: Artificial Intelligence and Statistics (2015). http://arxiv.org/abs/1501.06769
5. Minka, T., Winn, J., Guiver, J., Knowles, D.: Infer.NET 2.4. Microsoft Research Cambridge (2010)
6. Paige, B., Wood, F., Doucet, A., Teh, Y.: Asynchronous anytime sequential Monte Carlo. In: Advances in Neural Information Processing Systems (2014)
7. Stan Development Team: Stan: A C++ Library for Probability and Sampling, Version 2.4 (2014)
8. Wingate, D., Stuhlmüller, A., Goodman, N.D.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Proc. of the 14th Artificial Intelligence and Statistics (2011)
9. Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Artificial Intelligence and Statistics (2014)
10. Yang, L., Hanrahan, P., Goodman, N.D.: Generating efficient MCMC kernels from probabilistic programs. In: Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, pp. 1068–1076 (2014)