Automatic Sampler Discovery via Probabilistic Programming and Approximate Bayesian Computation

Yura Perov and Frank Wood

Department of Engineering Science, University of Oxford, United Kingdom,
{perov,fwood}@robots.ox.ac.uk

Abstract. We describe an approach to automatic discovery of samplers in the form of human interpretable probabilistic programs. Specifically, we learn the procedure code of samplers for one-dimensional distributions. We formulate a Bayesian approach to this problem by specifying an adaptor grammar prior over probabilistic program code, and use approximate Bayesian computation to learn a program whose execution generates samples that match observed data or analytical characteristics of a distribution of interest. In our experiments we leverage the probabilistic programming system Anglican to perform Markov chain Monte Carlo sampling over the space of programs. Our results are competive relative to state-of-the-art genetic programming methods and demonstrate that we can learn approximate and even exact samplers.

Keywords: Probabilistic Programming, Automatic Programming, Program Synthesis, Bayesian Inference, Automatic Modelling.

1 Introduction

In this paper we present an approach to automatic sampler discovery in the framework of probabilistic programming. Our aim is to induce program code that, when executed repeatedly, returns values whose distribution matches that of observed data. Ultimately, the artificial general intelligence machinery can use such approach to synthesise and update the model of the world in the form of probabilistic programs. As a starting point, we consider the induction of programs that sample from parametrised one-dimensional distributions.

Probabilistic programming is relevant to this problem for several reasons. Programs in Turing-complete languages can represent a wide range of generative probabilistic models. Samples from these models can be generated efficiently by simply executing the program code. Finally, in higher-order languages like Anglican, that is languages where procedures may act on other procedures, it is possible to write a generative model for program code that is itself a probabilistic program. This enables us to perform inference by specifying an adaptor grammar prior over program code and to use general-purpose Markov chain Monte Carlo algorithms implemented by the inference engine to sample over the space of programs.

To assess whether the distribution of samples generated by a program candidate matches the distribution of interest, we use approximate Bayesian computation methods that specify an approximate likelihood in terms of the similarity between a summary statistic of the generated samples and that of the observed values. While this approach is inherently approximate, it still can be used to find exact sampler code. This argument is supported by the fact that we were able to successfully learn an exact sampler for the Bernoulli distribution family given only the adaptor grammar prior learnt from a corpus of sampler code not including Bernoulli. We also successfully found approximate samplers for other common one-dimensional distributions and for real-world data. Finally, our approach holds its own in comparison to state-of-the-art genetic programming methods.

2 Related Work

Our approach to learning probabilistic programs relates to both program induction [11,20,15,22,4,17] and statistical generalisation from sampled observations [23,9]. The former is usually treated as search in the space of program text where the objective is to find a deterministic function that exactly matches outputs given parameters. The latter, generalising from data, is usually referred to as either density estimation or learning, and also includes automatic modelling [10,7].

3 Approach

Our approach can be described in terms of a Markov Chain Monte Carlo (MCMC) approximate Bayesian computation (ABC) [19] targeting

$$\pi(\mathcal{X}|\hat{\mathcal{X}})p(\hat{\mathcal{X}}|\mathcal{T})p(\mathcal{T}), \tag{1}$$

where at a high level $\pi(\mathcal{X}|\hat{\mathcal{X}})$ is a compatibility function between summary statistics computed between observed data $\mathcal{X}$ and data, $\hat{\mathcal{X}}$, generated by interpreting latent sampler program text $\mathcal{T}$. The higher $\pi(\mathcal{X}|\hat{\mathcal{X}})$, the more similar distributions $\mathcal{X}$ and $\hat{\mathcal{X}}$.

Consider a parametric distribution F with parameter vector θ. Let $\mathcal{X} = \{x_i\}_{i=1}^{I}, x_i \sim F(\cdot|\theta)$ be a set of samples from F. Consider the task of learning program text $\mathcal{T}$ that when repeatedly interpreted returns samples whose distribution is close to F. Let $\hat{\mathcal{X}} = \{\hat{x}_j\}_{j=1}^{J}$, $\hat{x}_j \sim \mathcal{T}(\cdot)$ be a set of samples generated by repeatedly interpreting $\mathcal{T}$ J times.

Let s be a summary function of a set of samples and let $d(s(\mathcal{X}), s(\hat{\mathcal{X}})) = \pi(\mathcal{X}|\hat{\mathcal{X}})$ be an unnormalised distribution function that returns high probability when $s(\mathcal{X}) \approx s(\hat{\mathcal{X}})$. We refer to d as a compatibility function, or penalty interchangeably.

We use probabilistic programming to write and perform inference in such a model, i.e. to generate samples of $\mathcal{T}$ from the marginal of (1) and generalisations to come of the same. Refer to the probabilistic program code in Figure 1 where the first line establishes a correspondence between $\mathcal{T}$ and the variable `program-text` then samples it from $p(\mathcal{T})$ where `grammar` is an adaptor-grammar-like [14] prior on program text that is described in Section 3.1. In this particular example θ is implicitly specified since the learning goal here is to find a sampler for the standard normal distribution. Also $\hat{\mathcal{X}}$ corresponds to the program variable `samples` and here $J = 100$. Here s and d are computed on the last four lines of the program with s being implicitly defined as returning a four dimensional vector consisting of the estimated mean, variance, skewness, and kurtosis of the set of samples drawn from $\mathcal{T}$. The penalty function d is also defined to be a

```
(defquery lpp-normal
  (let
    [n 0.001
     prog (grammar `() `real)
     prog-c (extract-compound prog)
     samples (apply-n-times prog-c 100 '())]
    (observe (normal (mean samples) n) 0.0)
    (observe (normal (std  samples) n) 1.0)
    (observe (normal (skew samples) n) 0.0)
    (observe (normal (kurt samples) n) 0.0)
    (predict (extract-text prog))))
```

Fig. 1. Probabilistic program to infer program text for a $\mathrm{Normal}(0, 1)$ sampler. Variable `n` is a noise level. Procedure `grammar` samples a probabilistic program from the adaptor grammar. This procedure returns a tuple: a generated program candidate in the form of nested compound procedures and the same program candidate in the form of program text.

```
(defquery lpp-bernoulli
  (let
    [prog (grammar `(real) `int)
     proc-c (extract-compound prog)
     J 100]
    (let
      [samples-1 (apply-n-times prog-c J '(0.5))]
      (observe (flip (G-test-p-value samples-1
                       `Bernoulli (list 0.5))) true))
    ... ;; So on for samples-2, etc; until N.
    (predict (extract-text prog))
    (predict (apply-n-times prog-c J '(0.3)))))
```

Fig. 2. Probabilistic program to infer program text for a $\mathrm{Bernoulli}(\theta)$ sampler and generate `J` samples from the resulting procedure at a novel input argument value, `0.3`.

multivariate normal with mean $[0.0, 1.0, 0.0, 0.0]^T$ and diagonal covariance $\sigma^2\mathbf{I}$. Note that this means that we are seeking the sampler source code whose output distribution has mean 0, variance 1, skew 0, and kurtosis 0 and that we penalise deviations from that by a squared exponential loss function with bandwidth σ^2, named `noise` in the code.

The more moments we match, the better the approximation will be achieved, but inevitably this will require more samples J.

This example highlights an important generalisation of the original description of our approach. For the standard normal example we chose a form of s such that we can compute the summary statistic of $s(\mathcal{X})$ analytically. There are at least three kinds of scenarios in which d can be computed in different ways. The first occurs when we search for efficient code for sampling from known distributions. In many such cases, as in the standard normal case just described, the summary statistics of F can be computed analytically. The second is the fixed dataset cardinality setting and corresponds to the setting of learning program text generative model for arbitrary observed data. The third, similar to the previous one, happens when we can only sample from F. This corresponds to a situations when, for instance, there is a running, computationally expensive MCMC sampler that can be asked to produce additional samples.

```
(def box-muller-normal
  (fn [mean std]
    (+ mean (* std
      (* (cos (* 2 (* 3.14159
        (uniform-cont 0.0 1.0))))
      (sqrt (* -2
        (log (uniform-cont
               0.0 1.0)))))))))
```

```
(def poisson (fn [rate]
  (let
   [L (exp (* -1 rate))
    inner-loop (fn inner-loop [k p]
        (if (< p L) (dec k)
            (inner-loop (inc k)
              (* p (uniform-cont 0 1)))))]
   (inner-loop 1 (uniform-cont0 1)))))
```

Fig. 3. Human-written sampling procedure program text for, left, Normal(μ, σ) [3] and, right, Poisson(λ) [16]. Counts of the constants, procedures, and expression expansions in these programs (and that of several other univariate samplers) are fed into our hierarchical generative prior over sampler program text.

Figure 2 illustrates the another important generalisation of the formulation in (1). When learning a standard normal sampler we did not have to take into account parameter values. Interesting sampler program text is endowed with arguments, allowing it to generate samples from an entire family of parameterised distributions. Consider the well known Box-Muller algorithm shown in Figure 3. It is parameterised by mean and standard deviation parameters. For this reason we will refer to it and others like it as a conditional distribution samplers. Learning conditional distribution sampler program text requires recasting our MCMC-ABC target slightly to include the parameter θ of the distribution F:

$$\pi(\mathcal{X}|\hat{\mathcal{X}}, \theta)p(\hat{\mathcal{X}}|\mathcal{T}, \theta)p(\mathcal{T}|\theta)p(\theta). \tag{2}$$

Here in order to proceed we must begin to make approximating assumptions. This is because in our case we need $p(\theta)$ to be truly improper as our learned sampler program text should work for all possible input arguments and not simply a just a high prior probability subset of values. Assuming that program text that works for a few settings of input parameters is fairly likely to generalise well to other parameter settings we approximately marginalise our MCMC-ABC target (2) by choosing a small finite N of θ_n parameters yielding our approximate marginalised MCMC-ABC target:

$$\frac{1}{N}\sum_{n=1}^{N} \pi(\mathcal{X}_n|\hat{\mathcal{X}}_n, \theta_n)p(\hat{\mathcal{X}}_n|\mathcal{T}, \theta_n)p(\mathcal{T}|\theta_n) \approx \int \pi(\mathcal{X}|\hat{\mathcal{X}}, \theta)p(\hat{\mathcal{X}}|\mathcal{T}, \theta)p(\mathcal{T}|\theta)p(\theta)d\theta. \tag{3}$$

The probabilistic program for learning conditional sampler program text for $\text{Bernoulli}(\theta)$ in Figure 2 shows an example of this kind of approximation. It samples from $\mathcal{T}$ N times, accumulating summary statistic penalties for each invocation. In this case each individual summary penalty computation involves computing both a G-test statistic

$$G_n = 2\sum_{i\in 0,1} \#[\hat{\mathcal{X}}_n = i]\ln\left(\frac{\#[\hat{\mathcal{X}}_n = i]}{\theta_n^i(1-\theta_n)^{(1-i)} \cdot |\hat{\mathcal{X}}_n|}\right),$$

where $\#[\hat{\mathcal{X}}_n = i]$ is the number of samples in $\hat{\mathcal{X}}_n$ that take value i and its corresponding p-value under the null hypothesis that $\hat{\mathcal{X}}_n$ are samples from $\text{Bernoulli}(\theta_n)$.

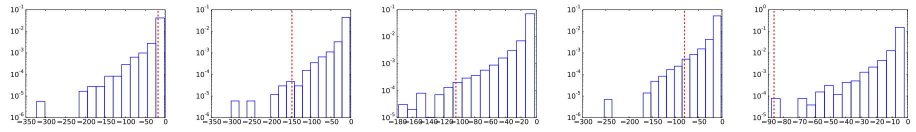

Fig. 4. Blue log-scaled histograms illustrate distributions over program codes' log likelihoods given production rules $P(\mathcal{T})$, which facilitate search of random variate generators for Bernoulli, Gamma, Normal, Poisson and Standard Normal distributions. Red dashed lines show the program codes' log likelihoods of true samplers written by humans as in Figure 3.

Since the G-test statistic is approximately χ^2 distributed, i.e. $G \sim \chi^2(1)$, we can construct d in this case by computing the probability of falsely rejecting the null hypothesis $H_0 : \hat{\mathcal{X}}_n \sim \text{Bernoulli}(\theta_n)$. Falsely rejecting a null hypothesis is equivalent to flipping a coin with probability given by the p-value of the test and having it turn up heads. These are the summary statistic penalties accumulated in the `observe` lines in Figure 2.

As an aside, in the probabilistic programming compilation context θ could be all of the `observe`'d data in the original program. By this parameterising compilation links our approach to that of [13].

3.1 Grammar and Production Rules

As we have the expressive power of a higher-order probabilistic programming language at our disposal, our prior over conditional distribution sampler program text is quite expressive. At a high level it is similar to the adaptor grammar [14] prior used in [17] but diverges in details, particularly those having to do with creation of local environments and the conditioning of subexpression choices on type signatures. To generate a probabilistic program we apply these production rules starting with $expr_{type}$, where $type$ is the output signature of the inducing program.

The set of types we used for our experiments was {`real, bool, int`}. To avoid numerical errors while interpreting generated programs we replace functions like `log(a)` with `safe-log(a)`, which returns 0 if $a < 0$, and `uniform-continuous` with `safe-uc(a, b)` which swaps arguments if $a > b$ and returns a if $a = b$. The general set of procedures in the global environment included `+, −, *, safe-div, safe-uc, cos, safe-sqrt, safe-log, exp, inc, dec`. Schematically our prior is provided below. Prior probabilities $\{p_i\}$ are learnt from a small corpus of probabilistic programs in the way described in Section 3.2.

1. $expr_{type} \mid env \xrightarrow{p_1} v$, a random variable v from the environment env specified by $type$. An environment is a mapping of typed symbols to values, but these values are not produced until the actual probabilistic program runs.
2. $expr_{type} \mid env \xrightarrow{p_2} c$, a random constant c with the type $type$. Constants were drawn from the predefined constants set (including 0.0, π, etc.)[1].

[1] For experiments described in Sections 4.4 constants were also sampled from Normal and Uniform continuous distributions.

3. $expr_{type} \mid env \xrightarrow{p_3} (procedure_{type}\ expr_{arg\_1\_type}\ ...\ expr_{arg\_N\_type})$, where $procedure$ is a primitive or compound, and deterministic or stochastic procedure chosen randomly from the global environment with output type signature $type$.
4. $expr_{type} \mid env \xrightarrow{p_4} (let\ [new\text{-}symbol\ expr_{real}]\ expr_{type} \mid env\ \cup\ new\text{-}symbol))$, where $env\ \cup\ new\text{-}symbol$ is an extended environment with a new variable named $new\text{-}symbol$. Its value is defined by an expression, generated according to the same production rules.
5. $expr_{type} \mid env \xrightarrow{p_5} (cp_{type}\ expr_{arg\_1\_type}\ ...\ expr_{arg\_N\_type})$, where cp_{type} is a compound procedure. Its body is generated using the same production rules given an environment that incorporates argument input variable names and values.
6. $expr_{type} \mid env \xrightarrow{p_6} (if\ (expr_{bool})\ expr_{type}\ expr_{type})$.
7. $expr_{type} \mid env \xrightarrow{p_7} (recur\ expr_{arg\_1\_type}\ ...\ expr_{arg\_M\_type})$, i.e. recursive call to the current compound procedure if we are inside, or to the main inducing procedure otherwise.

3.2 Probabilities for Production Rules

While it is possible to manually specify production rule probabilities for the grammar in Section 3.1 we took a hierarchical Bayesian approach instead, learning from human-written sampler source code. To do this we translated existing implementations of common one-dimensional samplers [6] into Anglican source (see examples in Figure 3). Conveniently all of them require only one stochastic procedure `uniform-continuous` so we also only include that stochastic procedure in our grammar.

We compute held-out production rules prior probabilities from this corpus. When we are inferring a probabilistic program to sample from F we update our priors using counts from all other sampling code in the corpus, specifically excluding the sampler we are attempting to learn. Our production rule probability estimates are smoothed by Dirichlet priors. Note that in the following experiments (Sections 4.4) the production rule priors were updated. True hierarchical coupling and joint inferences approaches are straightforward from a probabilistic programming perspective [18], but result in inference runs that take longer to compute.

4 Experiments

The experiments we perform illustrate uses cases outlined for automatically learning probabilistic programs. We begin by illustrating the expressiveness of our prior over sampler program text in Section 4.1. We then report results from experiments in which we test our approach in all three scenarios for how we can compute the ABC penalty d.

The first set of experiments in Section 4.2 tests our ability to learn probabilistic programs that produce samples from known one-dimensional probability distributions. In these experiments d either probabilistically conditions on p-values of one-sample statistical hypothesis tests or on approximate moment matching. In Section 4.3 the evaluation against genetic programming is presented. The second set of experiments in Section 4.4 addresses the cases where only a finite number of samples from an unknown real-world source are provided.

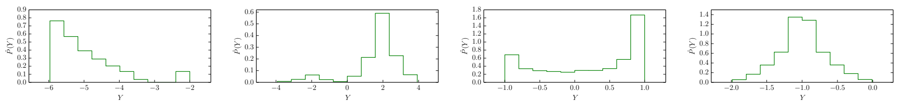

Fig. 5. Histograms of samples generated by repeatedly evaluating probabilistic procedures sampled from our prior over probabilistic sampling procedure text. The prior is constrained to generate samplers with univariate output but is clearly otherwise flexible enough to represent a nontrivial spectrum of distributions.

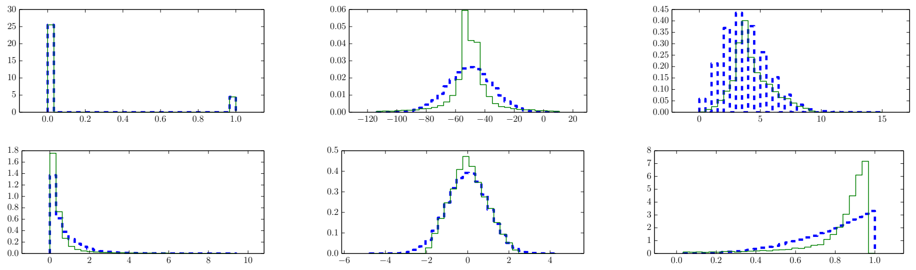

Fig. 6. Representative histograms of samples *(green solid lines)* drawn by repeatedly interpreting inferred sampler program text versus *(blue dashed lines)* histograms of exact samples drawn from the corresponding true distribution. Top row left to right: $\mathrm{Bernoulli}(p)$, $\mathrm{Normal}(\mu, \sigma)$, $\mathrm{Poisson}(\lambda)$. Bottom row same: $\mathrm{Gamma}(a, 1.0)$, $\mathrm{Normal}(0, 1)$, $\mathrm{Beta}(a, 1.0)$. The parameters used to produce these plots do not appear in the training data. In the case of $\mathrm{Bernoulli}(p)$ we inferred programs that sample exactly from the true distribution (see Figure 7). Not all finite-time inference converges to good approximate sampler code as illustrated by the $\mathrm{Beta}(a, 1.0)$ and $\mathrm{Normal}(\mu, \sigma)$ examples. With limited experimental time, a better sampler was found for $\mathrm{Normal}(\mu, \sigma)$ in comparison to $\mathrm{Normal}(0, 1)$. A possible explanation is that it is harder to find a sampler with two parameters rather than a sampler without any parameters. Future experiments should benefit from learning more complex probabilistic programs given already learnt simpler ones, similarly to [12,5].

4.1 Samples from Sampled Probabilistic Programs

To illustrate the flexibility of our prior over probabilistic programs source code, we show probabilistic programs sampled from it. In Figure 5 we show six histograms of samples from six sampled probabilistic programs from our prior over probabilistic programs. Such randomly generated samplers constructively define considerably different distributions. Note in particular the variability of the domain, variance, and even number of modes.

4.2 Learning Sampler Code for Common One-Dimensional Distributions

Source code exists for efficiently sampling from many if not all common one-dimensional distributions. We conducted experiments to test our ability to automatically discover such sampling procedures and found encouraging results.

In particular we performed a set of leave-one-out experiments to infer sampler program text for six common one-dimensional distributions: $\mathrm{Bernoulli}(p)$, $\mathrm{Poisson}(\lambda)$,

```
(fn [p stack-id]
  (if (< (sample
          (uniform-cont
            0.0 1.0)) p)
    1.0 0.0)
  )
```

```
(fn [p stack-id]
  (if (< 1.0 (safe-sqrt (safe-div p
    (safe-uc p (dec p))))) 1.0 0.0))
(fn [p stack-id]
  (if (< 1.0 (safe-uc (safe-sqrt p)
    (+ par (cos p)))) 1.0 0.0))
```

Fig. 7. *(left)* Human-written exact $\mathrm{Bernoulli}(p)$ sampler. *(right, two instances)* Inferred sampler program text. The first is also an exact sampler for $\mathrm{Bernoulli}(p)$. The last is another sampler also assigned non-zero posterior probability but it is not exact.

$\mathrm{Gamma}(a, 1.0)$, $\mathrm{Beta}(a, 1)$, $\mathrm{Normal}(0, 1)$, $\mathrm{Normal}(\mu, \sigma)$. For each distribution we performed MCMC-ABC inference with approximately marginalising over the parameter space using a small random set $\{\theta_1, \ldots, \theta_N\}$ of parameters and conditioning on statistical hypothesis tests or on moment matching as appropriate. Note that the pre-training of the hierarchical program text prior was never given the text of the sampler for the distribution being learned.

Representative histograms of samples from the best posterior program text sample discovered in terms of summary statistics match are shown in Figure 6. A pleasing result is the discovery of the exact $\mathrm{Bernoulli}(p)$ distribution sampler program, the text of which is shown in Figure 7.

4.3 Evaluating our Approach versus Evolutionary Algorithms

The approach was evaluated against genetic programming, one of state-of-the-art methods to search in the space of programs. Genetic programming is an evolutionary based metaheuristic optimisation algorithm to generate a program from a specification. The same grammar we used before was reproduced in the evolutionary computation framework DEAP [8]. The fitness function was selected to be the log probability presented in the eq. 3 with omitted $p\left(\hat{X} \mid \mathcal{T}\right)$ part in accordance with the assumption that desired probabilistic programs will repeatedly appear in results of search over programs. Note that another possible way is to marginalise over $\hat{X}$ during its fitness function evaluation, however this requires more programs runs.

Figure 8 shows that PMCMC inference performance is similar to genetic programming. In contrast to genetic programming, PMCMC is statistically valid estimator of the target distribution. In addition, probabilistic programming system allows to reason about the model over models and inferring models within the same framework, while genetic programming is an external machinery which requires to consider optimising probabilistic programs as a black box[2].

4.4 Generalising Arbitrary Data Distributions

We also explored using our approach to learn generative models in the form of sampler program text for real world data of unknown distribution. We arbitrarily chose three

[2] An interesting work for future is to run experiments in the framework of probabilistic programming with the inference engine that is itself based on evolutionary algorithms, in a similar way to [2].

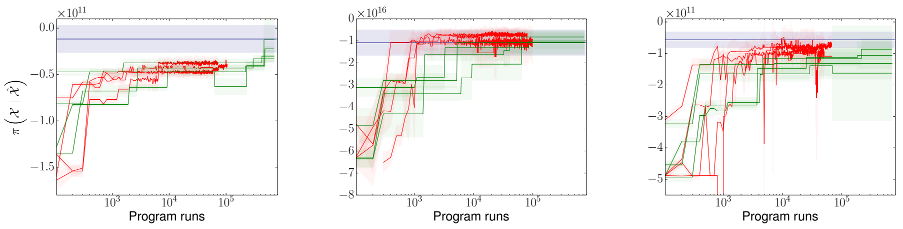

Fig. 8. Convergence of unnormalised penalty function $\pi\left(\mathcal{X} \mid \hat{\mathcal{X}}\right)$ for $\text{Bernoulli}(p)$, $\text{Normal}(\mu, \sigma)$, and $\text{Geometric}(p)$ correspondingly. $\hat{\mathcal{X}}$ is a samples set from a probabilistic program $\mathcal{T}$ as described in Section 3. Navy lines show the true sampler's penalty function value (averaged by 30 trials), red lines correspondent to genetic programming, and green lines – to PMCMC. Transparent filled intervals represent standard deviations within trials.

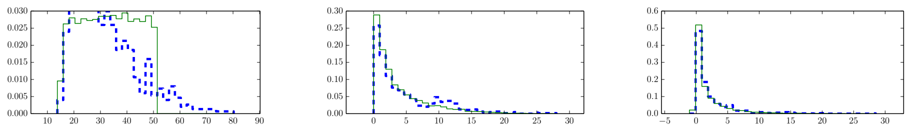

Fig. 9. Histograms of samples *(green solid)* generated by repeatedly interpreting inferred sampler program text and the empirical distributions *(blue dashed)* that they were trained to match.

continuous indicator features from a credit approval dataset [21,1] and inferred sampler program text using two-sample Kolmogorov-Smirnov distribution equality tests (vs. the empirical data distribution) analogously to the G-test described before. Histograms of samples from the best inferred sampler program text versus the training empirical distributions are shown in Figure 9. The data distribution representation, despite being expressed in the form of sampler program text, matches salient characteristics of the empirical distribution well.

5 Discussion

Our approach to program synthesis via probabilistic programming raises at least as many questions as it answers. One key high level question this work invokes is, really, what is the goal of program synthesis? By framing program synthesis as a probabilistic inference problem we are implicitly naming our goal to be that of estimating a distribution over programs that obey some constraints rather than as a search for a single best program that does the same. On one hand, the notion of regularising via a generative model is natural as doing so predisposes inference towards discovery of programs that preferentially possess characteristics of interest (length, readability, etc.). On the other hand, exhaustive computational inversion of a generative model that includes evaluation of program text will clearly remain intractable for the foreseeable future. For this reason greedy and stochastic search inference strategies are basically the only options available. We employ the latter, to explore the posterior distribution of programs whose outputs match constraints knowing full-well that its actual effect in this problem domain, and, in particular finite time, is more-or-less that of stochastic search.

It is pleasantly surprising, however, that the Monte Carlo techniques we use were able to find exemplar programs in the posterior distribution that actually do a good job of generalising observed data in the experiments we report. It remains an open question whether or not sole sampling procedures are the best stochastic search technique to use for this problem in general however. Perhaps by using them in combination with one of search we might do better, particularly if our goal is a single best program.

References

1. Bache, K., Lichman, M.: UCI Machine Learning Repository (2013)
2. Batishcheva, V., Potapov, A.: Genetic programming on program traces as an inference engine for probabilistic languages. In: Artificial General Intelligence. Springer (2015)
3. Box, G.E., Muller, M.E.: A note on the generation of random normal deviates. The Annals of Mathematical Statistics 29(2), 610–611 (1958)
4. Briggs, F., Oneill, M.: Functional genetic programming with combinators. In: Proceedings of the Third Asian-Pacific workshop on Genetic Programming, ASPGP (2006)
5. Dechter, E., Malmaud, J., Adams, R.P., Tenenbaum, J.B.: Bootstrap learning via modular concept discovery. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013) (2013)
6. Devroye, L.: Non-uniform random variate generation. Springer-Verlag (1986)
7. Duvenaud, D., Lloyd, J.R., Grosse, R., Tenenbaum, J.B., Ghahramani, Z.: Structure discovery in nonparametric regression through compositional kernel search. In: Proceedings of the 30th International Conference on Machine Learning (ICML 2013) (2013)
8. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research (2012)
9. Friedman, N., Getoor, L., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 2013). vol. 99, pp. 1300–1309 (1999)
10. Grosse, R., Salakhutdinov, R.R., Freeman, W.T., Tenenbaum, J.B.: Exploiting compositionality to explore a large space of model structures. In: Proceedings of the 28th International Conference on Machine Learning (ICML 2012) (2012)
11. Gulwani, S., Kitzelmann, E., Schmid, U.: Approaches and applications of inductive programming (Dagstuhl seminar 13502). Dagstuhl Reports (2014)
12. Henderson, R.: Incremental learning in inductive programming. In: Approaches and Applications of Inductive Programming. Springer (2010)
13. Hwang, I., Stuhlmüller, A., Goodman, N.D.: Inducing probabilistic programs by Bayesian program merging. arXiv e-print arXiv:1110.5667 (2011)
14. Johnson, M., Griffiths, T.L., Goldwater, S.: Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. Advances in Neural Information Processing Systems (NIPS 2007) (2007)
15. Kersting, K.: An inductive logic programming approach to statistical relational learning. In: Proceedings of the Conference on An Inductive Logic Programming Approach to Statistical Relational Learning 2005 (2005)
16. Knuth, D.E.: The art of computer programming, volume 2: Seminumerical algorithms (3rd edition) (1998)
17. Liang, P., Jordan, M.I., Klein, D.: Learning programs: a hierarchical Bayesian approach. In: Proceedings of the 27th International Conference on Machine Learning (ICML 2010) (2010)
18. Maddison, C., Tarlow, D.: Structured generative models of natural source code. In: Proceedings of the 31st International Conference on Machine Learning (ICML 2014) (2014)
19. Marjoram, P., Molitor, J., Plagnol, V., Tavaré, S.: Markov chain Monte Carlo without likelihoods. Proceedings of the National Academy of Sciences (2003)
20. Muggleton, S.: Stochastic logic programs. Advances in Inductive Logic Programming

21. Quinlan, J.R.: Simplifying decision trees. International journal of Man-Machine Studies (1987)
22. Raedt, L.D., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic inductive logic programming – theory and applications. Lecture Notes in Computer Science, Springer (2008)
23. Silverman, B.W.: Density estimation for statistics and data analysis. CRC Press (1986)