Designing a benchmarking suite for probabilistic programming languages

P. Pagni
Department of Engineering, Electronics and Computer Science
Queen Mary University of London
London, United Kingdom
ec23035@qmul.ac.uk

***Abstract*—Probabilistic programming languages (PPLs) find posterior distributions by modelling the parameters of the posterior with specific hyperpriors. They then use advanced sampling algorithms such as Hamiltonian Monte Carlo Markov chains to find the optimal parameters of the posterior distribution. Evaluating the quality of the found parameters, and by extensions the PPLs themselves, is an outstanding question in machine learning. This paper introduces a new benchmarking suite (what's your bench) to evaluate the found distributions against an exact reference distribution. One can find the reference distribution by using conjugate priors to exactly calculate the posterior parameters given a set of samples and a prior distribution. Using the same set of samples, one can also use PPLs to approximate the posterior distribution. The benchmarking suite uses a variant of the d-dimensional Kolmogorov-Smirnov test and the Kullback-Leibler divergence to measure the performance of PPLs. The benchmarking suite has been evaluated against the time taken to calculate the performance measures; and there is comparison of the three implemented PPLs as a proof of concept.**

I. Introduction

Probabilistic programming is a programming paradigm intended to enable programmers to implement probabilistic modelling without requiring deep knowledge of probability theory or machine learning. [6] A programmer can take advantage of advanced techniques, such as Hamiltonian Monte Carlo Markov chains, without requiring the expertise to implement these algorithms from scratch.

The key task for Bayesian practitioners is probabilistic inference: given a prior distribution for a random variable $P(X)$, and a set of observations Y, what is the posterior distribution $P(X|Y)$. Probabilistic programming languages (PPLs) calculate this posterior distribution using sampling techniques such as Hamiltonian Markov chains or derivatives thereof. The idea behind Markov sampling algorithms is to iteratively sample parameter values and accept only the value which maximises the probability of the observations versus the previous value.

For example, imagine that a practitioner hypothesises that $P(X|Y) \sim \mathcal{N}(\mu, \sigma^2)$ where μ and σ are unknown. If the practitioner specifies that $\mu \sim \mathcal{N}(0, 1)$ and $\sigma^2 \sim \mathcal{N}(1, 2)$, a simple Markov sampling algorithm might select the initial values for μ and σ as 0 and 1 respectively and calculate the probability of the observations under the distribution $\mathcal{N}(0, 1)$. The algorithm then samples the candidate values for each parameter μ' and σ' and evaluates the probability of the observations under the candidate values. When the probability of the observations is higher under μ' and σ', *these parameter values are accepted and the prior distributions for each parameter are updated:* $\mu \sim P(\mu', 1)$*;* $\sigma \sim P(\sigma', 2)$. After N draws from the Markov chain, the posterior is returned with the parameters which maximise the probability of the observations. More sophisticated versions of this algorithm exist, including those used by many probabilistic programming languages, but the above example demonstrates the core concept behind Markov sampling algorithms.

For the purpose of this project, the key characteristic of Markov sampling algorithms is that they do not guarantee that the parameters they find for the posterior distribution are the true optimal parameters. It is trivial to see why this undermines the trustworthiness of probabilistic languages. In light of this, developing a benchmarking suite to assess the accuracy of these programming languages is an important step in improving this trustworthiness.

This project introduces a PPL benchmarking suite called what's your bench. This evaluates PPL performance by comparing found distributions against a ground truth distribution. Ground truth distributions can be found where given a set of conditions and a set of samples, one can use conjugate priors to find the true posterior distribution. [5] In principle, one can measure the distance (or divergence) between the found distribution and the true distribution and use this to evaluate a PPL's performance.

II. Related Works

For the purposes of this project, one can consider all probabilistic programming languages as related. While it is out of the scope of this project to build a benchmarking suite compatible with all probabilistic languages, outcomes from this project should be applicable to many probabilistic languages.

There are several probabilistic programming languages which this benchmarking suite applies to. These include Stan, PyMC, Pyro, Numpyro, Edward2, to name just a few. [14][15][8][9][7] Of these five, the most dissimilar to the others is Stan, which is a programming language written in C++. It uses the Stan compiler to compile and execute code – it is essentially a programming language in its own right.

Developers have written a useful python interface to work with Stan: CmdPyStan. [13] To use this, one defines a model in Stan and saves this to a file. This is then loaded using CmdPyStan and variables can be injected using a data variable. This makes it easy to integrate Stan into what's your bench.

The other languages mentioned above are python libraries rather than languages in their own right. In general, the key differences between them is the backend libraries they use for automatic differentiation. For example, PyMC uses a custom built library called pytensor [14]; pyro is built on pytorch [8]; numpyro is built on JAX [9]; and edward2 is built on tensorflow (although it has the option to use JAX or numpy.) [7] All these languages implement the No U-Turn Sampler (NUTS), a Hamiltonian Markov Chain algorithm.

For the first iteration of what's your bench, the PPLs being implemented are PyMC, Pyro and Stan. These languages were chosen as they are popular PPLs with active development communities. PyMC released its latest version at time of writing in July 2024; Pyro June 2024; and Stan is a GitHub organisation whose sub-repositories see regular pushes. As a result, there is value in evaluating the performance of these PPLs against each other since they are being actively used.

Other attempts at benchmarking PPLs include PPL Bench, developed by researchers at Meta. [10] This benchmarking suite evaluates PPLs by training models on a training dataset and evaluating models according to their performance on a held-out test set. It reports the following evaluation metrics: plot of the predicted log likelihood w.r.t test samples; Gelman-Rubin convergence statistic; effective sample size; and inference time. These metrics evaluate the performance with regards to both accuracy of found distributions and the computational performance. Since this benchmarking strategy does not compare PPL distributions with a ground truth distribution, these metrics are also relative performance metrics (i.e. how well did one PPL perform versus another PPL.)

What's your bench differs from this approach since it compares found distributions against a ground truth distribution. While it can be used to compare the performance of PPLs, it is also useful in isolation to evaluate the performance of one PPL. For example, one can evaluate the performance of one PPL on a variety of problems and see which problems it performs well on versus where it can be improved. Again, the comparative aspect is not lost since this can be extended to compare several PPLs, but it differs from PPL Bench insofar as the performance metrics are still meaningful without comparing PPLs.

III. Methodology

This project has the following work-streams: implementing evaluation metrics; developing benchmarking infrastructure; defining a problem set with sufficient coverage over the total set of problems. It is necessary to develop a minimal viable product for each stream in order to have a functional benchmarking suite as a proof of concept. Each stream will be discussed in more detail below as well as the prioritisation of each stream.

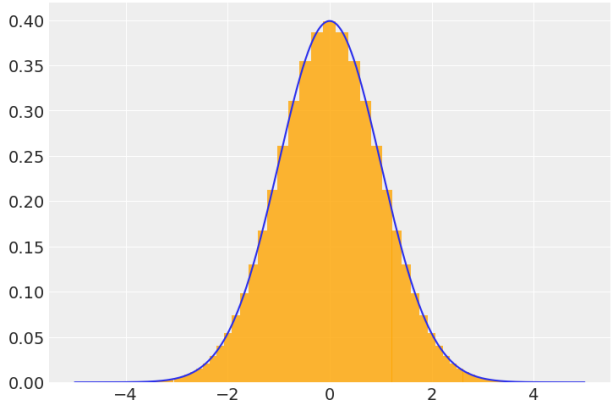


Fig. 1: This shows the Gaussian probability distribution function on top of an empirical density function designed to represent the Gaussian as a series of bins of decreasing size as the density function moves further away from its mean.

Performance Measures

This workstream marries engineering challenges with theoretical challenges since measuring the distance between probability distributions is an active area of research within the machine learning community (see discussion below.) The theoretical and engineering challenges are linked. Pre-existing methods for measuring the distance between probability distributions represent significant engineering challenges, particularly for high-dimensional probability distributions, thus motivating the development of new theoretical measures which can mitigate some of the engineering challenges. As a result, the challenge for this stream is to implement distance measures which scale well to high-dimensional probability distributions.

The accuracy of a PPL's found distribution will be measured in terms its distance from the exact distribution. Again, measuring the distance between probability distributions is non-trivial, especially when comparing high-dimensional distributions. This project has focused on the implementation of two distance measures: the Kullback-Leibler divergence and a d-dimensional Kolmogorov-Smirnov test.

Kullback-Leibler Divergence: The Kullback-Leibler divergence measures how much a probability distribution P diverges from a model distribution Q. [1] For continuous functions it is defined as

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} P(x) log\left(\frac{P(x)}{Q(x)}\right) dx$$

In information-theoretic modelling, this measure is often interpreted as the information loss when model $Q(x)$ is used to approximate $P(x)$. [2] This makes it a natural choice for evaluating the inference performance of PPLs as its use is widespread within Machine Learning for a similar purpose.

Calculating the Kullback-Leibler divergence for high-dimensional continuous distributions presents practical challenges, such as wasting computational resources integrating over low density areas of the support which do not contribute

significantly to the total density; and numerical inaccuracies introduced by floating point arithmetic.

Nevertheless, the Kullback-Leibler divergence calculation has been implemented from scratch in this project and the performance of this implementation has been evaluated in the results section. The implementation partitions the probability space into rectangular sub-regions of decreasing size as the algorithm moves away from the mean (figure 1.) The decreasing bin size should help reduce the error of the computation.

Additionally, the problem set features the normal distribution, student's T distribution, and the multivariate normal distribution. These distributions all have a support of $x \in (-\infty, \infty)$. As mentioned above, however, integrating over this entire support will result in integrating over extremely density areas which do not contribute significantly to the overall calculation. Therefore, it was important to limit the support over which the integral was calculated. This was defined as $\mu \pm 10\sigma^2$ where μ and σ^2 are the parameters of the exact distribution.

In the case of this project, and the problem set associated with it, this static support definition is sufficient because the distributions are well understood and symmetric. Should problems be added to the problem set such that the exact distribution is more complex, it may be necessary to investigate techniques to dynamically define the support. Some interesting areas for further research in this direction include using Hamiltonian Markov Chain sampling to define the typical set, and integrate over this support. Investigating this is beyond the scope of this project.

D-dimensional Kolmogorov-Smirnov Test: The D-dimensional Kolmogorov-Smirnov Test is inspired by the Kolmogorov-Smirnov test developed by Hagen et al. [12] A key difference, however, is that where the one-dimensional KS test compares an empirical cumulative density function with an analytic density function, the d-dimensional test compares two empirical cumulative distribution functions. [3]

To calculate Hagen et al.'s Ddks test statistic for one test point, the sample space is partitioned at this test point. For example, in 3 dimensions the sample space is partitioned at (x_1, y_1, z_1). Partitioning the space at this point yields 8 subspaces. (The number of subspaces after partitioning is given by 2^d.) [12] After partitioning the space, a vector is generated with the number of samples in each orthant.

The test statistic is then calculated as

$$D = max[|C_P(\vec{x}) - C_T(\vec{x})|]$$

where $C_P(\vec{x})$ and $C_T(\vec{x})$ are the respective cumulative density functions of the predicted (P) and true(T) distributions, evaluated at $\vec{x}$.

It should be noted that [12] specifies this slightly differently to enable accelerated computation. Specifically, it details how to implement this using tensor-based computation; but the essential idea is explained above.

This is appropriate for the use case defined in [12], since the intention is to evaluate whether model predictions follow the same distribution as the actual target variable. This is done by sampling the target variable and the predictions and comparing them. For this project, however, the comparison is between two analytic distributions. Therefore, this test statistic has been adapted to compare an empirical CDF generated by sampling from the predicted distribution with the integral of the analytic PDF of the true distribution.

The idea behind this is similar to Hagen et al.'s DDKS test statistic: for N test points, calculate the sample counts per orthant for the empirical CDF and integrate the analytic PDF such that the output is a vector of the densities in each orthant. For example, let $f(x, y)$ be the true analytic probability density function, and let the test point t have coordinates (x_1, y_1). The integrals evaluated will be

$$F(x,y) = \begin{bmatrix} \int_{-\infty}^{y_1}\int_{-\infty}^{x_1} f(x,y)dxdy \\ \int_{-\infty}^{y_1}\int_{x_1}^{\infty} f(x,y)dxdy \\ \int_{y_1}^{\infty}\int_{-\infty}^{x_1} f(x,y)dxdy \\ \int_{y_1}^{\infty}\int_{x_1}^{\infty} f(x,y)dxdy \end{bmatrix} \tag{1}$$

This will give the theoretical densities for each orthant at t. We can then calculate the Ddks statistic in the same way as Hagen et al. (It should be noted that computing this integral from scratch would in $N2^d$ d-dimensional integrals. Fortunately scipy has the CDF pre-computed for distributions currently in the scope of this project. [11])

Finally, it is necessary to measure whether the statistic is significant. Hagen's significance test asks if the predicted distribution and true distribution are the same (the null hypothesis), then what is the probability of seeing a higher maximal difference value in proportional orthant occupation when the orthants are fixed and new samples are generated. [12] Since Hagen et al. conceive landing in an orthant as either a success or a failure, they consider the membership matrix

$$M_X[i,k] \sim Bi(N_x, \lambda_{i,k})$$

where $\lambda_{i,k}$ is the total probability density within an orthant. This is estimated using the Bayes' estimator for the predicted empirical CDF:

$$\lambda_{i,k} = \frac{M_P[i,k] + 1}{N_P + 2}$$

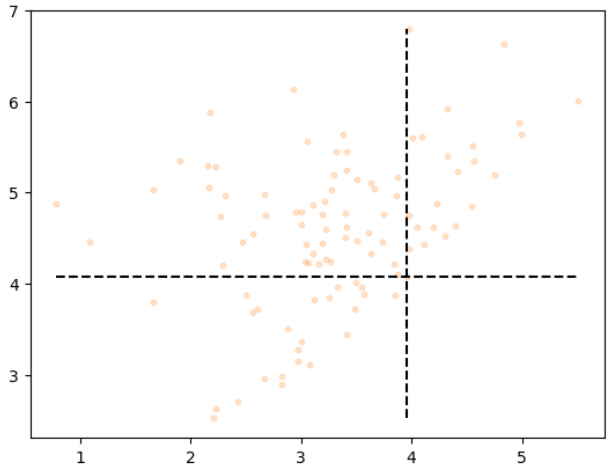
After computing the delta values between the true and predicted membership matrices, one can use the binomial distribution PMF to calculate the probability of observing a specific delta value

$$p(\delta : N_P, N_T, \lambda) = \sum_{n_P, n_T \in \vec{\Delta}} p_{Bi}(n_p, m_p, \lambda) p_{Bi}(n_T, m_T, \lambda)$$

where

$$\vec{\Delta} = \left\{ n_P, n_T : \delta = \left|\frac{n_P}{N_P} - \frac{n_T}{N_T}\right|, 0 \le n_P \le N_P, 0 \le n_T \le N_T \right\}$$

Essentially, $\vec{\Delta}$ is all the different ways of achieving a specific delta value.



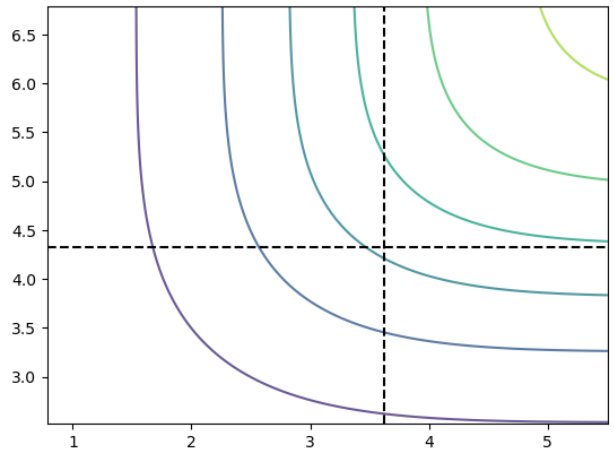


Fig. 2: A visual representation of how the ADKS algorithm partitions a 2-dimensional space into quadrants. Densities are calculated by subtracting densities from the total cumulative density. Fig. 3: A visual representation of how the ADKS algorithm partitions a 2-dimensional space into quadrants. Densities are calculated per quadrant based on how many samples fall into each quadrant.

With the probability of observing a specific δ value defined, Hagen then defines the probability of observing anything as, or less, extreme as the observed δ as

$$p(d \leq \delta : \delta, N_P, N_T, \lambda_{i,k}) = \sum_{d^*=0}^{\delta} p_\delta(d^*, N_P, N_T, \lambda_{i,k})$$

The probability of observing the maximum difference D is the complement to this. Therefore, significance is given by

$$S(D, N_P, N_T, \vec{\lambda}) = 1 - \prod_{i=0}^{2^d} \prod_{k=0}^{N} p_{\leq\delta}(D, N_P, N_T, \lambda_{i,k})$$

[12]

This significance test is appropriate since it reflects the assumption that the PPL distributions are good approximations of the real distribution. Only where the found distribution differs greatly from the true distribution will this test consider it statistically significant.

The implementation for this distance measure exists separately from the what's your bench repository, in a fork of Hagen et al.'s DDKS repository. It has been implemented by adapting the DDKS class into a new class: ADKS. This class generates the empirical CDF density in the same way as the original DDKS class. It generates analytical densities per orthant by taking a test point (x, y) and calculating the boundaries of the space given a notional maximum value (x_{max}, y_{max}). (This maximum value is statically defined in the same way as for the Kullback-Leibler divergence.) As mentioned above, scipy has pre-computed the CDFs for the scoped probability distributions. So, armed with the coordinates $\{(x, y), (x_{max}, y), (x, y_{max}), (x_{max}, y_{max})\}$ one can find the cumulative densities up to these coordinates and use arithmetic to calculate the densities per orthant.

Calculating the density per orthant for an array of cumulative densities was implemented such that it was agnostic to the dimensionality of the data, and therefore to the number of orthants. Let $D(\cdot)$ be equal to the density of an orthant, where $\cdot$ represents the orthant; and let $F(\cdot)$ be the cumulative density of a PDF up to $\cdot$. Finally, let each orthant in figure 3 be labeled as follows: $a = (x, y)$; $b = (x_{max}, y)$; $c = (x, y_{max})$; $d = (x_{max}, y_{max})$ where the coordinates represent the upper right point of the orthant. The orthant densities for each orthant are given as follows:

$$\begin{aligned} D(a) &= F(a) \\ D(b) &= F(b) - D(a) \\ D(c) &= F(c) - D(a) \\ D(d) &= F(d) - (D(a) + D(b) + D(c)) \end{aligned} \tag{2}$$

This motivates the notion that the orthant densities can be calculated iteratively, but scaling it to d-dimensions requires a programmatic approach for identifying which densities should be subtracted from the cumulative density.

An observation which provided an avenue to solve this problem is that for a d-dimensional coordinate set, where there are m coordinates which take the maximum value, the density for this orthant λ is given by

$$D(\lambda) = F(\lambda) - (D(a) + \sum_{u \in U} D(u))$$

where $D(a)$ is the density of the orthant where the coordinates take no maximum values. U is the set of orthants where the count of maximum coordinates is $m-1$ and the index positions of those maximum values are the same as the index position of those maximum values for orthant λ.

Once the theoretical orthant densities are calculated, and the orthants are sorted in the same order as the empirical CDF, the rest of the significance test was implemented in the original DDKS repository.

The loop based implementation of this algorithm has a computational complexity of $O(2^d N^2)$ where d is the dimensionality and N is the total number of sample points. [12] The calculation of the empirical densities is implemented using PyTorch which serves to alleviate this, but the calculation of the analytic densities is still in loop form. As such, this algorithm still suffers from this complexity.

To relieve this, a sub-sampling technique has been implemented where instead of using every point in the sample set as a test point, a random selection of N points are taken from the sample set to use as test points. Using this, one can manage the impact of N^2 on the complexity.

Benchmarking Infrastructure

This stream represents a key engineering challenge of this project. The benchmarking suite should be designed to enable users to add further PPLs or problems, without having to make onerous changes to the rest of the code base. By removing these barriers to entry, it should encourage active development of the repository. (Although the author assumes most changes will be done by himself for the foreseeable future.)

With this in mind the benchmarking suite has the following file structure:

```
whats_your_bench
├── stan_models
│   ├── mvNormalKnownCov.stan
│   ├── normalKnownCov.stan
│   └── normalKnownMean.stan
├── _problem.py
├── conjugate_priors.py
├── distance.py
├── main.py
├── problem_set.py
├── pymc_models.py
├── pyro_models.py
├── stan_models.py
└── utils.py
```

This structure emphasises keeping components separate to enable making additions to the repository simple. For example, imagine one wanted to add another PyMC model which estimated the mean of a normal distribution given a known variance; but instead of modelling the distribution of the mean with a Normal distribution, one wanted to model the mean with a Cauchy distribution. In this case, one could add the code for this model to pymc_models.py; duplicate the relevant problem cases with the PPL prior updated to be appropriate for the Cauchy, and then this suite could test this model against this problem. Making the repository extensible was the goal, so this philosophy permeates throughout the repository.

The workhorse of this repository is the Problem class, defined in _problem.py. This class generates the sample set; invokes the conjugate model which given the exact distribution; calculates the support limits to be supplied to the distance metrics; and evaluates the PPL models. All problems within the problem_set.py are sub-classes of the Problem class; these sub-classes invoke the relevant PPL models and get the estimated parameters given the relevant priors and dataset.

The final notable thing to draw attention to is the implementation of the Stan models. As mentioned in section 2, while PyMC and Pyro are Python libraries, Stan is a separate programming language altogether. Therefore, the PyMC and Pyro models can be implemented directly in a Python file (and this would be true of any other PPLs which are essentially Python libraries.) For Stan, however, the models are implemented in Stan files, which are then run with CmdStanPy, a Stan-Python interface. This interface makes evaluating Stan models in a python environment straightforward.

It is also a harbinger of potential engineering challenges for what's your bench. Two probabilistic languages which have not yet been mentioned and are not python libraries and are languages in their own right are Church and Anglican. Neither of these languages have a Python interface, so to making them compatible with this benchmarking suite would require developing an interface between these languages and Python. Designing this interface is outside of the scope of this project.

Defining the problem set

With a minimum viable product in the benchmarking infrastructure, the engineering challenges of implementing the problem set should be accounted for. (This is to an extent true in the actual implementation of the benchmarking infrastructure.) Therefore, the main challenges for this stream are theoretical.

This stream demands defining a problem set for probabilistic programming languages where one can analytically calculate the exact posterior distribution given a set of data and some priors. In the case where the prior distribution belongs to the same family of distributions as the posterior distribution, conjugate priors can be used to find a closed-form solution for the exact posterior distribution. Using conjugate priors, one can hard code the solution given a set of observations and assumptions. This can be used as the reference solution against which one can compare the found distributions.

This approach is advantageous for two reasons: one, it is an easy to implement methodology for finding exact posterior distributions given specific criteria; two, it provides significant constraints on the infinite set of potential problems.

The problem set presented in this project focused on the following conjugate prior solutions:

- normal distribution with known variance;
- normal distribution with known mean;
- normal distribution with known covariance.

Nevertheless, there is still an infinite set of potential problems since each problem can be adjusted with the following levers:

- sample size;
- distribution of samples;
- data dimensionality;
- prior distribution family of estimated parameters;
- prior distribution hyperparameters for distribution of estimated parameters;

- implementation of probabilistic model.

While the different settings for each of these represent an infinite set, one anticipates that one could find different configurations which provides good coverage over the total set. For example, one could build upon the work completed by [13] on prior choice recommendation for defining a set of priors for a problem which represent the 5 levels of priors: flat prior; vague but proper prior; very weakly informative prior; generic weakly informative prior; and specific informative priors. By defining problems covering each of these categories one can provide coverage over different kinds of priors without defining problems for every specific potential prior (which would be unreasonable.)

For the purposes of this project, however, the priority has been to develop a problem set which allows some comparison of the performance of PPLs, and allows for evaluation of the performance of the distance measures used by the testing suite. As such, the problem set has been designed to test against the following variables:

- *dimensionality*: how does the execution time of the test suite change and the dimensionality of probability distributions increases;
- *sample size*: as sample size is changed, how does the performance of PPLs change;
- *hyperprior parameters*: how does the PPL performance change as hyperprior parameters are changed.

The problem set can be found in Appendix A.

IV. Results

Performance measures

Figure 4 shows the increase in execution time for each performance measure as the dimensionality of the probability distributions is increased. From this chart, it is clear that at low dimensions the time taken to calculate each measure is comparable and low. The time taken begins to diverge at 7 dimensions, where the execution time for calculating the Kullback-Leibler divergence remains low. The execution time for calculating the d-dimensional KS score with all samples being used as test points begins to exponentially increase up to 4606 seconds when the dimensionality is 10. This shows that the current implementation of the d-dimensional KS test cannot reasonably be used for high dimensional probability distributions; at least not where all samples are used as test points.

When the sample points are subsampled to get the test points, we can see the execution time for the KS test is substantially lower than when all test points are used. Indeed, the execution time to calculate the KS score for 10-dimensional probability distributions is 657 seconds: 7 times smaller than when calculating for all test points. While this is an improvement, the time to compute the distance is still approximately 11 minutes, and much slower than the calculation for the Kullback-Leibler divergence. This shows the current loop-based implementation of the d-dimensional KS test is too slow to be considered of practical use.

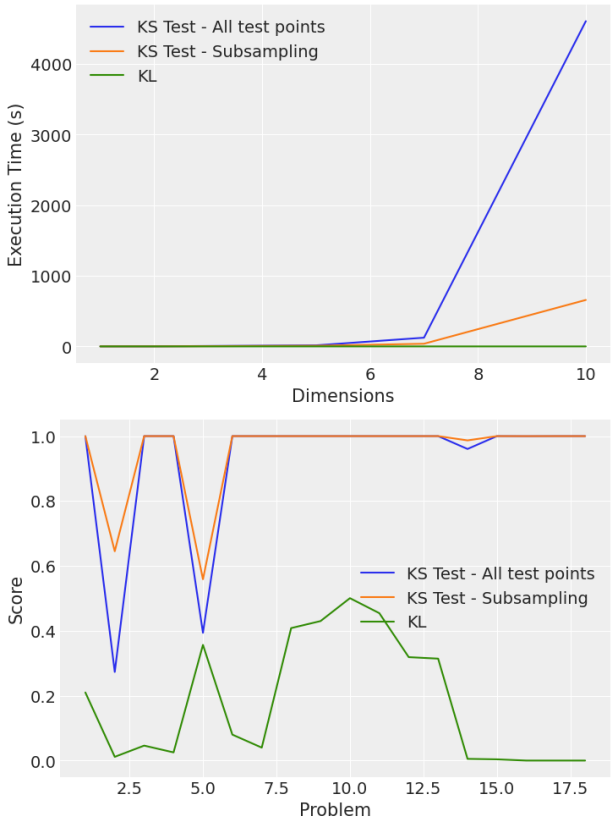


Fig. 4: Shows the increase in execution time of the performance measures tested. Fig 5: Shows the mean value for each performance measure for each problem.

Comparing the overall trends of the measures themselves, there is more fluctuation in the Kullback-Leibler divergence than in the KS Score. Indeed, generally the KS Score is stable around 1.0, even for problems where the reported KL divergence is higher. Nevertheless, the KL divergence is low enough such that it does not indicate any significant divergence between the found probability distributions and the true distribution, so it is not reasonable to say there is a major discrepancy between the scores in this case.

Ultimately, in the case of the performance measures, it is clear the implementation of the Kullback-Leibler divergence is more scalable than the implementation of the d-dimensional Kolmogorov-Smirnov test. It should be noted that the implementation of the d-dimensional KS test is sub-optimal since it does not take advantage of PyTorch's parallelization. Nevertheless, the current implementation is impractical for high dimensions.

Comparison of probabilistic programming languages

When examining the performance of probabilistic programming languages, the two variables against which the performance was tested were the sample size of the observations and the parameters given to the hyperpriors.

Firstly, figures 6a and 6b show no obvious link between PPL performance and sample size (N). In terms of the KS

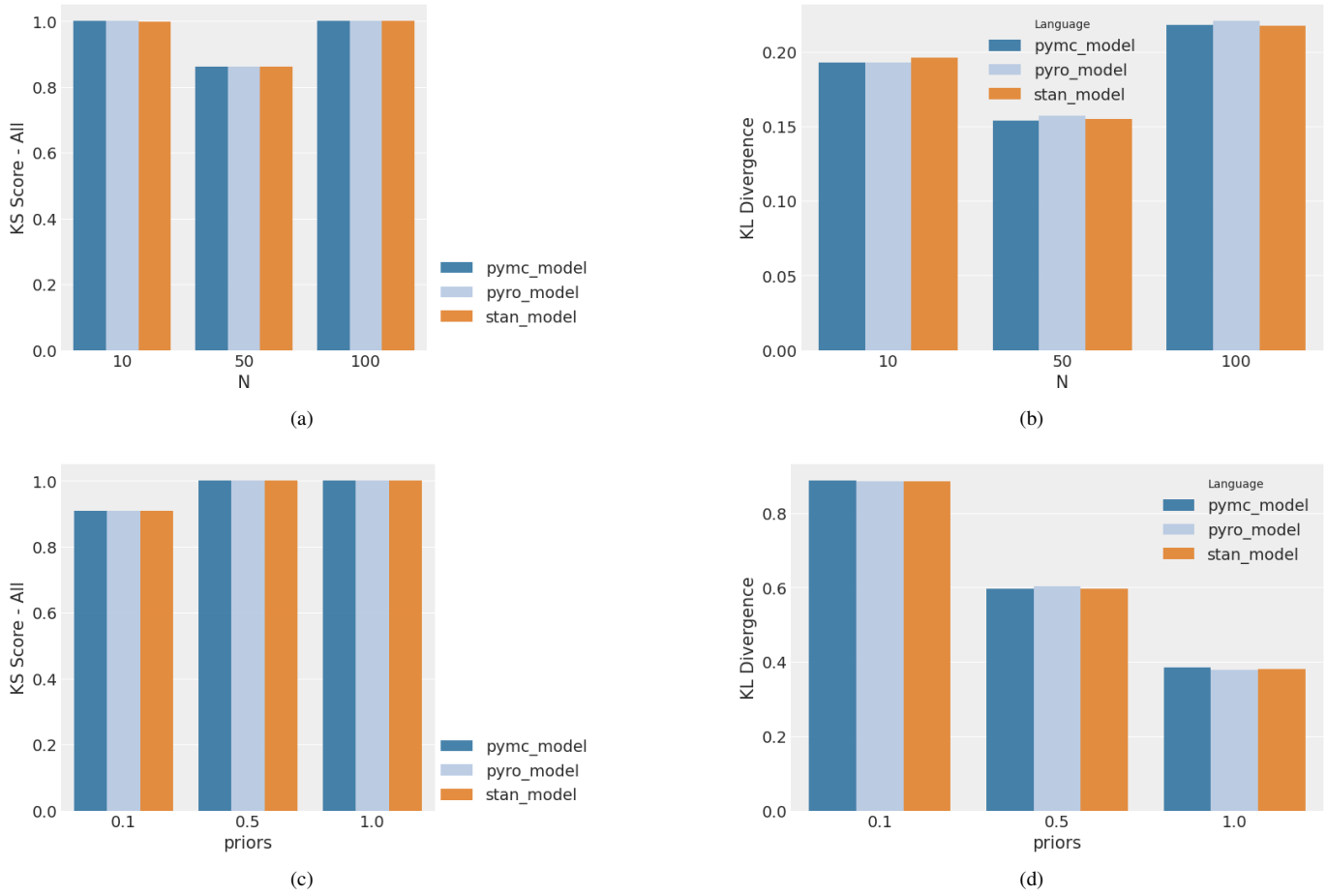


Figure 6 shows the KS score and KL divergence for different values of sample size (10, 50, 100) and priors (0.1, 0.5, 1.0)

Score, the mean performance of the PPL models is worst when N is 50; this is not reflected in the KL divergence however, which shows the mean divergence is lowest when N is 50. For the problems presented in the problem set, the sample size does not appear to have a significant impact on PPL performance. An explanation for this could be that the exact posterior parameters are calculated in relation to the same dataset, regardless of sample size. Since the sample size is the same for both the conjugate models and the PPL models, they are both calculating posterior parameters with the same amount of evidence. Therefore, one can expect the posterior parameters to be similar regardless of sample size.

An interesting avenue for further testing would be to derive the exact posterior parameters with a set of N observations; then subsample these observations to N' and approximate the posterior parameters with a PPL and compare the distributions.

An area where this benchmarking suite shows promise is in evaluating the impact of the hyperpriors on a PPL's approximate distribution. Figures 6c and 6d show the performance of PPL models in predicting the posterior for a Normal distribution with a known mean. The PPL models are estimating ν (degrees of freedom) and ψ (scale) of Student's T distribution. The hyperpriors on both of these parameters are half Normal distributions with standard deviations of 0.1, 0.5 and 1.0 respectively.

It is clear that when the hyperprior parameters are lower (0.1) the KS score is lower and the KL divergence is higher, meaning the approximate distribution is further from the exact distribution. Whereas for wider priors, the approximate distribution is much closer to the exact distribution. This shows that informative priors which are poorly chosen can lead to worse outcomes than weaker priors. This substantiates the advice on prior choice given by the Stan community: "the cost of setting the prior too narrow is more severe than the cost of setting it too wide." [13]

This shows this benchmarking suite to be well-suited to evaluate the appropriateness of the specific implementations of PPL models. By comparing with the true posterior distribution, one can evaluate the efficacy of specific hyperprior parameters; the effect of the choice of distribution on the hyper prior; different composite methods for estimating specific parameters, such as the covariance of a multivariate Gaussian.

Finally, in terms of comparing the performance of each PPL: their performance is very similar and there is no obvious

separation in terms of their approximate distributions. While one must accept that the problem set is limited at this stage and is not testing the PPLs to their limit, there are no conclusions to be drawn other than each PPL performs similarly well across all problems.

It is a different story when it comes to the time taken to execute each model: Stan is considerably faster than both PyMC and Pyro. The execution time for Stan to estimate the parameters of a 10-dimensional probability distribution is 2.26 seconds; compare this to PyMC at 3.12 seconds and Pyro at 5.47 seconds. Additionally, it is able to estimate the parameters for 1 and 2 dimensional distributions in under a second at 0.30 and 0.44 seconds respectively. The times taken for PyMC and Pyro to estimate the parameters of a 1-dimensional distribution are 2.07 seconds for PyMC and 2.65 seconds for Pyro. (Times can be found in Appendix B.)

While there is no clear separation between the PPLs in the quality of the found distributions, Stan is much faster in execution time; followed by PyMC then Pyro.

V. Future Work

There are several opportunities for future work stemming from the creation of this benchmarking suite.

The first is improving the implementation of the d-dimensional KS test. In its current loop-based form the time it takes to be computed makes it impractical for high-dimensional probability distributions. By making use of PyTorch parallelization, the computation should be accelerated. This will make it useful when the dimensionality is high.

Furthermore, additional performance metrics can be investigated to provide more comprehensive analysis of the PPLs. For example, looking at PPL Bench's metrics; it would be interesting to add the Gelman-Rubin convergence metric; and effective sample size. This would enable more thorough evaluation of the different aspects of PPL performance. In it's current version, the what's your bench benchmarking suite evaluates PPL performance through a more narrow lens.

Additionally, it would be wise to extend the problem set to provide more coverage across a range of problems. By doing this one could investigate further into the effect of specific priors and prior hyperparameters on the performance of PPLs. On top of this, a more comprehensive problem set would provide a better window through which to compare PPLs, possibly identifying specific criteria under which some PPLs perform better than others. A proposed approach has been outlined in section 3: defining the problem set.

Another area where the benchmarking suite could be enhanced would be to add more PPLs for evaluation. Only three were added here as a proof of concept. It would be straightforward to add other python libraries such as edward2 and numpyro. These could be implemented in a similar way to the other python-based PPLs. Other PPLs such as Anglican would require the design of an interface between Python and the language in question, an extra obstacle but nevertheless achievable.

VI. Conclusion

The benchmarking suite introduced in this report compares 3 different PPLs: PyMC; Pyro and Stan. While there is no clear separation in their ability to approximate distributions, Stan has a considerably faster execution speed against the problem set included with the benchmarking suite. No specific conclusions will be drawn on the performance of the programming languages themselves, since the problem set is not robust enough at this stage. Several avenues for improving the problem set have been proposed in section 3: defining the problem set.

Additionally, the implemented distance metrics have been tested for probability distributions up to 10 dimensions: these tests show that the implementation of Kullback-Leibler Divergence is sufficiently performant for similarly high-dimensional distributions (although this is not to say it would perform well with continued increases to the dimensionality.) The d-dimensional Kolmogorov-Smirnov test, however, does not perform well for high-dimensional probability distributions. As a result, a high priority improvement would be to implement this distance function using accelerated computing techniques to ensure it is tenable within this benchmarking suite. In its current form, this distance measure is unsuitable for high-dimensional problems and if it cannot be improved it should be replaced.

Ultimately, this benchmarking suite serves as a sufficient proof of concept that a PPL's posterior distribution can be compared with a reference distribution. It has been shown that in spite of a limited problem set, meaningful analysis can be conducted about the efficacy of a specific PPL model implementation; such as the impact of have an overly informative hyperpriors. Furthermore, there are several promising areas for future work outlined in section 5: including adding more PPLs and improving the user interaction with the program.

In its current form this benchmarking suite is somewhat minimal. The hope is that with further development it can become more robust and be a substantial hub where PPLs can be evaluated and researched.

References

[1] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[2] Kenneth P. Burnham. *Model Selection and Multi-Model Inference: A Practical Information-Theoretic Approach*. Rev. ed. of: Model selection and inference. c1998. New York: Springer, 2002. ISBN: 0387953647. URL: https://archive.org/details/modelselectionmu0000burn.

[3] Kevin Ford. "From Kolmogorov's theorem on empirical distribution to number theory". In: *Kolmogorov's Heritage in Mathematics*. Ed. by Éric Charpentier, Annick Lesne, and Nikolaï K. Nikolski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 97–108. ISBN: 978-3-540-36351-4. DOI: 10.1007/978-3-540-36351-

4_5. URL: https://doi.org/10.1007/978-3-540-36351-4_5.

[4] Kevin Murphy. “Conjugate Bayesian analysis of the Gaussian distribution”. In: (Nov. 2007).

[5] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020.

[6] Andrew D. Gordon et al. “Probabilistic Programming”. English. In: *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 167–181. DOI: 10.1145/2593882.2593900.

[7] Dustin Tran et al. “Simple, Distributed, and Accelerated Probabilistic Programming”. In: *Neural Information Processing Systems*. 2018.

[8] Eli Bingham et al. “Pyro: Deep Universal Probabilistic Programming”. In: *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6. URL: http://jmlr.org/papers/v20/18-403.html.

[9] Du Phan, Neeraj Pradhan, and Martin Jankowiak. “Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro”. In: *arXiv preprint arXiv:1912.11554* (2019).

[10] Sourabh Kulkarni et al. *PPL Bench: Evaluation Framework For Probabilistic Programming Languages*. 2020. arXiv: 2010.08886 [cs.PL]. URL: https://arxiv.org/abs/2010.08886.

[11] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[12] Alex Hagen et al. *Accelerated Computation of a High Dimensional Kolmogorov-Smirnov Distance*. 2021. arXiv: 2106.13706 [stat.CO]. URL: https://arxiv.org/abs/2106.13706.

[13] Allen Riddell, Ari Hartikainen, and Matthew Carter. *pystan (3.0.0)*. PyPI. Mar. 2021.

[14] O Abril-Pla et al. “PyMC: A Modern and Comprehensive Probabilistic Programming Framework in Python”. In: (2023).

[15] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual*. Version 2.35. 2023. URL: https://mc-stan.org.

APPENDIX A
PROBLEM SET

Likelihood	Conjugate Prior Model	Sample Size	PPL Hyper Priors	Dimensionality
Normal Known Variance	$\mathcal{N}(3,1)$	10	$\mu=0;\sigma^2=1$	1
Normal Known Variance	$\mathcal{N}(3,1)$	50	$\mu=0;\sigma^2=1$	1
Normal Known Variance	$\mathcal{N}(3,1)$	100	$\mu=0;\sigma^2=1$	1
Normal Known Variance	$\mathcal{N}(5,1)$	10	$\mu=0;\sigma^2=1$	1
Normal Known Variance	$\mathcal{N}(5,1)$	50	$\mu=0;\sigma^2=1$	1
Normal Known Variance	$\mathcal{N}(5,1)$	100	$\mu=0;\sigma^2=1$	1
Normal Known Mean	$\beta(1,1)$	10	$\alpha=1;\beta=1$	1
Normal Known Mean	$\beta(1,1)$	50	$\alpha=1;\beta=1$	1
Normal Known Mean	$\beta(1,1)$	100	$\alpha=1;\beta=1$	1
Normal Known Mean	$\beta(1,1)$	10	$\alpha=0.5;\beta=0.5$	1
Normal Known Mean	$\beta(1,1)$	50	$\alpha=0.5;\beta=0.5$	1
Normal Known Mean	$\beta(1,1)$	100	$\alpha=0.5;\beta=0.5$	1
Normal Known Mean	$\beta(1,1)$	10	$\alpha=0.1;\beta=0.1$	1
Normal Known Mean	$\beta(1,1)$	50	$\alpha=0.1;\beta=0.1$	1
Normal Known Mean	$\beta(1,1)$	100	$\alpha=0.1;\beta=0.1$	1
Multivariate Normal Known Covariance	$\mathcal{N}([3,5],I)$	50	$\mu=[0,0];\Sigma=I$	2
Multivariate Normal Known Covariance	$\mathcal{N}([10,8],I)$	50	$\mu=[0,0];\Sigma=I$	2
Multivariate Normal Known Covariance	$\mathcal{N}([14,12],I)$	50	$\mu=[0,0];\Sigma=I$	2
Multivariate Normal Known Covariance	$\mathcal{N}([3,5,4,6,7],I)$	100	$\mu=[0,0,0,0,0];\Sigma=I$	5
Multivariate Normal Known Covariance	$\mathcal{N}([3,5,4,6,7,8,9],I)$	100	$\mu=[0,0,0,0,0,0,0];\Sigma=I$	7
Multivariate Normal Known Covariance	$\mathcal{N}([3,5,4,6,7,8,9,3,3,2],I)$	100	$\mu=[0,0,0,0,0,0,0,0,0,0];\Sigma=I$	10

TABLE I: Summary of Problems

Conjugate models derived from [4].

APPENDIX B
PERFORMANCE RESULTS

Language	dims	Model Exe Time
PyMC	1	2.068340
	2	2.229888
	5	2.423288
	7	3.054864
	10	3.121639
Pyro	1	2.646281
	2	3.555767
	5	4.113375
	7	4.655225
	10	5.470818
Stan	1	0.303158
	2	0.436851
	5	1.031603
	7	1.289852
	10	2.263650