HackPPL: A Universal Probabilistic Programming Language

Jessica Ai*
Facebook, Inc.
United States
jaix@fb.com

Nimar S. Arora*
Facebook, Inc.
United States
nimarora@fb.com

Ning Dong*
Facebook, Inc.
United States
dnn@fb.com

Beliz Gokkaya*
Facebook, Inc.
United States
belizg@fb.com

Thomas Jiang*
Facebook, Inc.
United States
thomasjiang@fb.com

Anitha Kubendran*
Facebook, Inc.
United States
anithak@fb.com

Arun Kumar*
Facebook, Inc.
United States
arkumar@fb.com

Michael Tingley*
Facebook, Inc.
United States
tingley@fb.com

Narjes Torabi*
Facebook, Inc.
United States
ntorabi@fb.com

Abstract

HackPPL is a probabilistic programming language (PPL) built within the Hack programming language. Its universal inference engine allows developers to perform inference across a diverse set of models expressible in arbitrary Hack code. Through language-level extensions and direct integration with developer tools, HackPPL aims to bridge the gap between domain-specific and embedded PPLs. This paper overviews the design and implementation choices for the HackPPL toolchain and presents findings by applying it to a representative problem faced by social media companies.

CCS Concepts

• **Software and its engineering** → **Language types**; • **Mathematics of computing** → *Statistical software.*

Keywords

Probabilistic Programming, Probabilistic Representations, Hack, Coroutines, Bayesian Inference, Markov-chain Monte Carlo methods, Variational methods, Social Networks, Crowdsourcing

*The authors contribute equally to this paper.

MAPL '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6719-6/19/06.
https://doi.org/10.1145/3315508.3329974

ACM Reference Format:
Jessica Ai, Nimar S. Arora, Ning Dong, Beliz Gokkaya, Thomas Jiang, Anitha Kubendran, Arun Kumar, Michael Tingley, and Narjes Torabi. 2019. HackPPL: A Universal Probabilistic Programming Language. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3315508.3329974

1 Introduction

Probabilistic reasoning has become increasingly important in industrial machine learning applications as practitioners seek to obtain higher-fidelity insights from disparate sources of information. Bayesian modeling provides a compelling way to capture uncertainty in model parameters and incorporate expert domain knowledge, but has historically required significant expertise from the practitioner [16]. Probabilistic programming languages (PPLs) aim to reduce development time of Bayesian modeling by providing a unified syntax to express and compose generative models, and an in-built inference enginer to test probabilistic hypotheses.

PPLs offer many tradeoffs between efficiency and expressivity, with one key consideration being whether the system introduces a new domain-specific language (DSL), or embeds itself within a host language. PPLs such as Stan [7], BUGS [15], and JAGS [27] adopt the former approach, offering specialized syntax and optimization during compilation of a probabilistic model. On the other hand, embedded PPLs such as WebPPL [17], Edward [34], and Pyro [2] offer the compatibility and familiarity of their general-purpose host languages. Balancing performance and ergonomics of a DSL against ease of usability and integration in an embedded language remains an important consideration in PPL design.

We introduce HackPPL, a probabilistic programming language that aims to bridge the gap between these paradigms

with first-class integrations with language and developer productivity tools. HackPPL is built within the Hack programming language [33], a dominant web development language across large technology firms with over 100 million lines of production code [10]. Although Hack originated as an optionally-typed dialect of PHP, the Hack development team is discontinuing PHP support in order to open opportunities for sweeping language advancements [32]. This period of rapid evolution has presented a unique opportunity for us to incorporate language-level benefits typically afforded to DSL-style PPLs by collaborating with the Hack development team. In particular, we have implemented building blocks of a universal PPL within the language, such as multi-shot coroutines [20], as well as high-performance tensor computations backed by PyTorch [8].

We have built HackPPL as a universal probabilistic language [37] in order to target Hack's diverse user base. It is worth noting that, while HackPPL is able to efficiently perform inference over a diverse set of models, ensuring acceptable performance across *all* expressible models remains an active area of research. Nevertheless, our flexible modeling language allows the use of Hack features generally, including recursion and statefulness. It also enables developers to integrate models alongside existing systems, which we believe will expose novel modeling opportunities within the present codebase. To aid with this, we have provided first-class integration with familiar developer productivity tools such as the Nuclide IDE [3] for interactively tuning, debugging, and visualizing models. We have leveraged the rich capabilities of the Hack language to provide user-friendly abstractions for building and analyzing HackPPL models:

1. **Modeling.** HackPPL offers an imperative approach to probabilistic modeling. Models allow for deterministic and stochastic computations, and support the rich set of abstractions and language features offered by Hack.
2. **Inference.** The framework provides a set of generic inference algorithms including Hamiltonian Monte Carlo [24], Black Box Variational Inference [28], and Sequential Monte Carlo [11].
3. **Assessment.** In addition to IDE-integrated visualization, HackPPL provides out-of-the-box tools for model diagnosis and evaluation, such as posterior predictive checks [14] and a statistics module for data analysis.

This paper elaborates on the design, implementation, and selected applications of HackPPL. Section 2 demonstrates modeling and inference in HackPPL with an example. Section 3 details language changes made to accommodate user-friendly modeling. Sections 4, 5, and 6 overview the framework and its integrations with existing systems. Section 7 showcases HackPPL in an industry application.

2 HackPPL Fundamentals

To provide a general overview of the modeling workflow, we begin by discussing an example model in HackPPL.

2.1 Linear Regression Model

HackPPL models are classes that implement the `PPLModel` interface and are demarcated from other Hack code by way of the `<<__PPL>>` attribute. As shown in Listing 1, developers can compose generative models by defining random variables with the `sample` operator. `sample` is used to both draw a new value from a distribution, and to condition a random variable on observations. This dual behavior for `sample` allows users to postpone conditioning of observations until inference. Each `sample` statement must also be associated with a unique identifier for correctness of inference and for examining the posterior after inference. Section 3 further details the implementation of the `sample` operator. In addition, models utilize PyTorch tensors and operations for improved performance, discussed in Section 4.

Listing 1. Linear regression model, $y \sim Normal(\mathbf{x}\mathbf{w}, \tau^{-1/2})$

```
<<__PPL>>
class LinearRegressionModel implements PPLModel<void> {
  public function __construct(private Tensor $x) {}
  public function run(): void {
    $num_coeffs = $this->x->size()[1];
    $w = sample(new Normal(
      Tensor::zeros(vec[$num_coeffs]),
      Tensor::ones(vec[$num_coeffs])), 'w');
    $tau = sample(new Gamma(
      Tensor::scalar(1.),
      Tensor::scalar(1.)), 'tau')
        ->expand($this->x->size()[0]);
    sample(new Normal($x->matmul($w), $tau, true), 'y');
  }
}
```

2.2 Performing Inference

To support universal modeling, HackPPL adopts a trace-based approach to inference [35]. Traces are proposals for parameter values created from model program executions and each trace is evaluated according to the log probability of parameter values conditioned on observations. We do not construct dependency graphs by design, which allows us to support dynamic variables in probabilistic models.

Listing 2. Conditioning and running inference

```
$obs = dict['y' => $y];
$model = new LinearRegressionModel($x);
$hist = PPLInfer::hmc($model, $obs)->history()->run($num_iter);
$hist->getSample('w');
```

Inference in HackPPL is separated entirely from the modeling layer and the user is free to choose an inference method that is most appropriate for their model. Listing 2 presents an example of how to run inference on our linear regression model. We construct a dictionary containing observations `$y`

keyed by the corresponding sample identifier, initiate Hamiltonian Monte Carlo inference using the `PPLInfer` class and retrieve the posterior for the regression coefficients.

3 Language Features

3.1 Coroutines

Monte Carlo inference algorithms draw many samples to approximate the posterior and with a trace-based approach, this is achieved by exploring program execution traces. A key improvement here comes from observing that it may be unnecessary to re-execute the model in its entirety. For instance, to generate many sample traces for the probabilistic program in Listing 3, we may not need to revisit the first sample site 'flip' in each trace. Instead, we can simply generate multiple results from the execution paths following 'flip' given a particular result for 'flip'. This strategy of selectively exploring model subcomputations can save valuable execution time.

Listing 3. Model with stochastic control flow

```
$flip = sample(new Bernoulli(Tensor::scalar(0.7))), 'flip');
if ($flip->first() === 0) {
  sample(new Categorical(Tensor::vector(vec[1, 2])), 'res');
} else {
  sample(new Categorical(Tensor::vector(vec[1, 3])), 'res');
}
```

For this purpose, most universal PPLs utilize a language feature that enables exploring subcomputations. Bingham et al. [2] and Bauer and Pretnar [1] use algebraic effect handlers, whereas Ŝcibior et al. [30] builds models using monads. Wood et al. [37] and Goodman and Stuhlmüller [17] utilize a continuation passing style transform. In HackPPL, models are implemented as coroutines that are reified as multi-shot continuations in inference code. This choice allows us to accommodate more stateful control flow, such as loops and exception handling, that Hack developers are familiar with.

Coroutines have two fundamental characteristics:

1. Values local to a coroutine persist between successive calls.
2. The execution of a coroutine is suspended as control leaves it (e.g., by invoking another coroutine), and the execution continues where it left off when control returns to it [23].

Our implementation of multi-shot coroutines allows models to be resumed multiple times with the same local state restored upon each resumption. This enables the posterior space to be explored efficiently by implementing the `sample` operator, which defines random variables, as a coroutine. When control transfers from the model to the `sample` coroutine, the model is suspended and may be resumed multiple times from the `sample` point.

3.1.1 Using the Coroutine Framework

`sample` utilizes the general coroutine framework to schedule multiple executions of subcomputations of a model. The general coroutine framework allows users to reify coroutines and treat them in their suspended state as continuations.

Listing 4. Example coroutine function `myCoroutine()`

```
class MyClass {
  ?MultishotContinuation<string> $suspended_coroutine;
  public coroutine function myCoroutine(): string {
    print "Started the coroutine";
    $resumed_value = suspend suspendMultiple(
      coroutine ($my_suspended_coroutine) ==> {
        $this->suspended_coroutine = $my_suspended_coroutine;
    });
    print "Continued the coroutine";
    return $resumed_value;
  }
}
```

In Listing 4, we demonstrate how to define a coroutine. In Hack, coroutine functions are marked with the **`coroutine`** modifier. Coroutines may invoke other coroutines via the **`suspend`** operator; however, non-coroutines may not directly invoke a coroutine. In this example, the coroutine suspends to `suspendMultiple`, a special coroutine library function to reify the calling coroutine as a multi-shot continuation.

Listing 5. Coroutine invocation with multiple resumptions

```
class CoroutineCallback implements Continuation<string> {
  public function resume(string $coroutine_return): void {
    print $coroutine_return;
  }
}
StartCoroutine::start(
  coroutine () ==> suspend $my_class->myCoroutine(),
  new CoroutineCallback());
// Later on...
$my_class->suspended_coroutine->resumeAsync("First");
$my_class->suspended_coroutine->resumeAsync("Second");
```

In Listing 5, `myCoroutine()` is invoked via another special coroutine library function, `StartCoroutine::start`. Since the coroutine is called from a noncoroutine context, it requires a callback to be defined, which will be entered any time the coroutine completes. When the coroutine in this example is started, it will immediately print "Started the coroutine", and suspend to `suspendMultiple`. In Listing 4, the developer stores the continuation as an instance variable so that coroutine may be resumed at a later time. Later, this continuation is resumed via `resumeAsync`, at which point the coroutine resumes, completes by returning the string provided to its continuation, and its callback, `CoroutineCallback`, is called with the return value. "First" and then "Second" will then be printed from the callback.

3.1.2 Implementation

As in Kotlin [4] and C# [31], we have implemented coroutines as state machines via source code transformation at

compile time. Coroutines are transformed into continuation passing style where the function continuation is provided as an additional argument. Each coroutine is compiled to an anonymous class, which contains a method implementing the state machine and fields representing its current state. The suspend keywords within a coroutine body are transformed and marked with a goto label. When a coroutine is suspended, local state is stored in the state machine. When a multi-shot coroutine is resumed, local state is copied and restored, and then the coroutine jumps back to its previous suspension point using the saved goto label.

As coroutines invoke their continuations rather than returning, Hack's lack of support of tail recursion can cause executions of long running coroutines to stack overflow. To handle this, we have implemented a coroutine manager in Hack code. This framework is responsible for trampolining the coroutine calls as anonymous function invocations, and is able to avoid such stack overflow issues.

3.2 Syntax Changes

As mentioned previously, inference methods implement the sample operator as a coroutine so that they can control model execution. As a result, sample must be invoked with the suspend keyword and functions invoking it must also be marked with the coroutine modifier. Listing 6 desugars the syntax in Listing 4, and shows the use of the InferenceState object, which helps aggregate information such as the trace's log probability. With the syntax presented thus far, underlying implementation details are actually exposed and results in distracting boilerplate for the user. This unfortunately also provides users direct access to the inference method objects, potentially disrupting the inference state.

Listing 6. Lowered model with stochastic control flow

```
coroutine function model(InferenceState $state): int {
  $flip = suspend $state->sample(new Bernoulli(
    Tensor::scalar(0.7))), 'flip');
  if ($flip->first() === 0) {
    return suspend $state->sample(new Categorical(
      Tensor::vector(vec[1,2])), 'res'));
  } else {
    return suspend $state->sample(new Categorical(
      Tensor::vector(vec[1,3])), 'res'));
  }
}
```

To address this, we introduce a new syntax that separates such implementation details from model writing syntax. The <<__PPL>> attribute, seen previously in Listing 3, treats the sample keyword as if it were a reserved function in the language. This allows it to be specialized for each inference method as a coroutine. Thus, while each inference method implementation may still use the special coroutine library functions to reify multi-shot model continuations, users writing models do not need to understand this in order to write their models. We are then able to completely hide the InferenceState from the model, and users may invoke inference methods like sample as if they were ordinary Hack functions, as shown in Listing 3.

4 Data and Model Representation

In this section, we discuss how random variables and observations are represented. In particular, we emphasize our ubiquitous library abstraction of tensors, which helps to ensure performant inference for large or complex models.

4.1 Continuous Values

We have imported PyTorch's tensor framework [26] as an extension for the Hack virtual machine. Tensors are the data structures underlying distributions, realized samples, and observations in HackPPL, and provide a convenient abstraction for parallelizing computations within a model. Besides providing tensor algebra functionality, they also natively support reverse-mode automatic differentiation [18], which is used in inference algorithms where repeated gradient evaluations of the program trace are needed to guide parameters towards a target distribution. Listings 1 and 3 demonstrates construction of scalar and vector tensors, and higher-rank tensors are also supported. As tensors are used widely throughout model writing, we have ongoing efforts to integrate their syntax more seamlessly into Hack.

4.2 Discrete Values

While PyTorch tensors enable performant computations over continuous values, they do not provide convenient support for discrete values. As sampling and conditioning on discrete random variables is crucial in a universal PPL, we introduce for them an abstraction called DTensor. A DTensor can be thought of as a sequence of categorical labels, which can be represented by either a string or integer. Its primary feature is to enable conversion to and from a one-hot encoded tensor, which is a useful numerical representation for discrete values [19]. Listing 7 shows usage of DTensors where construction requires users to supply category labels and an optional mapping between the encoding indices and the category.

Listing 7. DTensor construction with a vocabulary mapping. The one-hot-encoded tensor will be of the form [[1,0,0], [0,0,1], [0,1,0]].

```
$labels = vec[1, 3, 2];
$vocab = vec['a', 'c', 'b'];
$dtensor = new DTensor($labels, $vocab);
$one_hot_tensor = $dtensor->toOneHotEncodedTensor();
```

4.3 Distributions

HackPPL provides a large repository of performant, tensor-backed distributions, and makes it easy for developers to implement new ones. A Distribution implements sample() and score() methods as follows:

- *sample(n)*: Retrieve *n* i.i.d. samples from the distribution. Note that this method is distinct from the special `sample` operator discussed in Section 3.2.
- *score(x)*: Compute the log probability at *x*.

To take advantage of the performance gains when using tensor operations, we also support what we refer to as batch sampling and batch scoring in our distribution library, shown in Listing 8. Each distribution object is associated with the tensor shape of its parameters, a *batch shape*, and the shape of its support, an *event shape*. In Listing 8, we construct a Dirichlet distribution with a batch shape of *2* where each distribution in the batch will have a sampling event shape of *4*. These two types of shape information are important for allowing users to take advantage of the performance gains that arise from tensorizing their models.

Listing 8. Batch sampling and scoring from a Dirichlet distribution.

```
$alphas = Tensor::matrix(vec[
  vec[0.25, 0.25, 0.25, 0.25],
  vec[0.3, 0.6, 0.2, 0.1]]);
$dirich = new Dirichlet($alphas);
$val = $dirich->sample(5); // $val has shape 5 × 2 × 4
$score = $dirich->score($val);
```

5 Inference Engine

An inference engine's objective is to obtain posterior estimates for model parameters. We have two primary design goals for HackPPL's inference engine. First, we aim to abstract away the details of inference so that it may be reasoned about independently of a model. Second, we aim to make our inference engine generic to support a wide class of models and user-provided constraints. This section demonstrates usage of the inference engine, and then discusses implementation highlights of our inference algorithms.

5.1 Running Inference

Users configure and run inference by interacting with a single helper class, `PPLInfer`. This class provides a centralized way for users to specify inference configurations for their model and describe inference-related constraints. It also offers an interface for constructing inference pipelines, which are procedures tailored to computing specific values of interest from the inference run. In Listing 2, we demonstrate an example inference pipeline where we use Hamiltonian Monte Carlo inference with a probabilistic model and specify configuration options such as the burn-in phase. Inference results are accumulated in a "history" object, which consolidates interpretable information about each inference iteration.

Listing 9. Custom inference pipeline. This returns the expected value for a particular random variable.

```
$results = PPLInfer::hmc($model)
  ->map($samples ==> $samples['w'])->reduce(($d, $val) ==> {
    $weight = $d['weight'] + 1.;
    $mean = ($d['mean'] * $d['weight'] + $val) / $weight;
    return dict['mean' => $mean, 'weight' => $weight];
  })->run($iterations);
```

The builder pattern of `PPLInfer` supports custom pipelines for computing inference results. This allows for the optimization of inference based on the latent variables of interest. Listing 9 shows a pipeline to only track the expected value of random variable 'w'. Here, we obtain the random variable after every iteration in the `map()` step, and combine its value with the results of other iterations in the `reduce()` step.

5.2 Sampling-based Inference

HackPPL provides a library of sampling-based inference methods, including Importance Sampling, Metropolis-Hastings, Sequential Monte Carlo [11], and Hamiltonian Monte Carlo [24] methods. For these, coroutines constitute the building blocks of their trace-based implementations [35]. In subsequent sections, we discuss their implementation highlights.

5.2.1 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) is a Markov Chain Monte Carlo (MCMC) method that is known to avoid inefficient random walk behavior. It instead performs efficient exploration of the parameter space using the gradient estimates of log posterior density. HMC can generate distant proposals by updating sample points through Hamiltonian dynamics simulations, which helps to maintain a high acceptance probability [24]. Hamiltonian functions are defined using the target distribution and an auxiliary momentum variable. The dynamics are simulated in a discretized manner using a leapfrog integrator. The leapfrog integrator introduces step size and number of steps as hyperparameters, and they can be automatically tuned using the No-U-Turn Sampler. [21]

HackPPL uses PyTorch's reverse-mode automatic differentiation to compute gradient estimates of the posterior. We also introduce a *Transform* trait to facilitate transformation of a continuous distribution's support between constrained and unconstrained spaces. This is essential for allowing HMC to operate on distributions with bounded supports. For example, in HackPPL the stick-breaking transform [5] is inherently associated with our Dirichlet distribution and manages implicit conversions between real and simplex spaces.

5.2.2 Auto-marginalization of Discrete Variables

Compared to other sampling-based algorithms, HMC converges faster on large models and, as such, is one of the main inference algorithms in HackPPL. However, HMC can only be used when the target distribution is differentiable with respects to its parameters which excludes discrete variables. Traditionally, marginalizing out discrete samples from the model has been recommended as a workaround [7]. HackPPL automates this process with automatic marginalization in order to support discrete parameter sampling with HMC. Our

implementation of auto-marginalization relies on multi-shot coroutines. Whenever a discrete variable is encountered in a program's execution, the program is suspended and resumed multiple times with all possible values in the support of that distribution. Listing 10, which implements a simple finite mixture model, illustrates the usage.

Listing 10. Finite mixture model in HackPPL

```
$mu = sample(new Normal($mu_p, $sigma_p), 'mu');
$c = sample(new Categorical($p), 'c');
foreach ($data as $i => $y) { // Observe on data bound to y$i
  sample(new Normal($mu->getTensorAt($c[$i]), $sigma), "y$i");
}
```

Here, when we sample the discrete variable c, we run the rest of the program a number of times equal to the number of categories in the support of a Categorical distribution. Throughout the execution, we keep track of the sample points and the likelihood computation. When all execution paths are traversed, we marginalize the discrete variable c out from the log probability function using Equation 1.

$$P(y|p,\mu,\sigma) = \sum_{c=1}^{C} p_c \text{Normal}(y|\mu_c,\sigma) \tag{1}$$

This approach hides the statistical details of marginalization from users, and also allows for the estimation of a discrete variable's posterior distribution.

5.2.3 Resumable Inference

For sampling-based methods, monitoring convergence and diagnosing problems is an essential part of inference. To help with convergence monitoring, we provide standard convergence diagnostics statistics and visualization support in the library, discussed further in Section 6. For cases where convergence is not achieved, we provide functionality for the sampling-based algorithms to be paused and resumed. In essence, our inference runs keep track of their states and this state is restored when the run is resumed. Listing 11 shows how inference can be paused and resumed.

Listing 11. Resuming sampling-based inference in HackPPL.

```
$infer = PPLInfer::hmc($model);
$history = $infer->run($num_iter_first);
$history = $infer->history($history)->run($num_iter_second);
```

This functionality is especially important when performing sampling on production models. We also use resumable inference for tuning of the temperature of inference chains during warm-up phase. This functionality is a key component of ongoing work on evolutionary MCMC methods [12].

5.3 Approximate Inference

For models where sampling-based methods are slow to converge or are too computationally intensive, HackPPL supports scalable inference in the form of Black Box Variational Inference [28, 36]. We estimate the posterior $p(x|y)$ with a mean-field approximation $q(x)$ as shown in Equation 2 where, the variational guide distributions $q(.)$ are parameterized by λ. We then use first-order optimization algorithms to find the guide distribution parameters λ that maximize the Evidence Lower Bound. Each λ_i can be optimized independently according to Equation 2, making this form of variational inference an attractive, parallelizable approach when working with large models or datasets.

$$q(x) = \prod_{i=1}^{m} q_i(x_i|\lambda_i) \tag{2}$$

In HackPPL, users are required to explicitly provide guide distributions for each sample site when constructing their model. In Listing 12, we construct a simple model, which only samples from a Normal distribution, and we demonstrate how variational guide distributions are specified.

Listing 12. `sample` statement with a variational guide.

```
sample(new Normal($mu_p, $sigma_p),
  'x',
  $args ==> new Normal($args['mu'], $args['sigma']->exp()),
  dict['mu' => $mu_guide, 'sigma' => $sigma_guide]);
```

6 Assessment

Building a probabilistic model typically requires multiple iterations where one repeatedly assesses model fit to identify areas of improvement. This section overviews HackPPL's set of model assessment tools, which can be used independently of the modeling and inference modules.

6.1 Posterior Predictive Distributions

Posterior predictive distributions define the distribution of a random variable conditioned on observed values by marginalizing over the posterior distribution is given in Equation 3 where new data points y_{new} are generated and re-weighted using the posterior distribution, $P(\theta|y)$.

$$P(y_{\text{new}}|y) = \int P(y_{\text{new}}|\theta)P(\theta|y)dy \tag{3}$$

HackPPL provides a convenient syntax for obtaining the posterior predictive distribution of random variables. Here, our goal is to reuse the original model but this time run the model in simulation mode (as opposed to inference mode). In essence, our implementation of posterior predictive treats conditioning in the model (i.e. `sample` statements where observations have been bound) as random variable draws, where we generate and determine the likelihood of new data using the posterior distribution. Listing 13 shows how to obtain the posterior predictive distribution using the original model along with the rich inference history object.

Listing 13. Obtaining the posterior predictive distribution

```
PosteriorPredictive::predict($finite_mixture_model, $history);
```

The posterior predictive distribution can then be used in making predictions for future observations.

6.2 Model Criticism

Posterior predictive distributions are also commonly compared against the observed data points through formal statistical checks, known as 'posterior predictive checks' [14, 29]. These checks are fundamental within Bayesian model assessment. HackPPL's model criticism library provides functions for performing posterior predictive checks for user-defined test statistics using the posterior predictive distribution as obtained in Listing 13. One can then obtain the distribution of test statistics and compute Bayesian p-values for model assessment. HackPPL also provides out-of-sample prediction statistics [13] and where there is ground truth data, accuracy scores that take into account the full posterior.

6.3 HackPPL Playground

To enable rapid experimentation and improvement of probabilistic models, it is important to have the ability to quickly visualize the current state of inference (e.g. convergence) and the results of inference. To provide a low-friction execution and visualization environment, we have created the HackPPL Playground, an extension to Nuclide that is built as a React component and hooks into Nuclide's existing debugging capabilities. This makes it easy for developers to run, debug, and visualize HackPPL models from within an editor that is already typically used for Hack development. Users can either run their script in its entirety or set breakpoints to debug specific parts of their model. Any calls to our visualization library will render charts in the playground as the script executes. Figure 1 shows a screenshot of the playground where a breakpoint paused the execution of inference. The user can interact with the debugger as shown in the bottom pane of the figure. The trace plot, displayed on the right pane, is updated in real-time as inference progresses, and can be used to help assess model convergence.

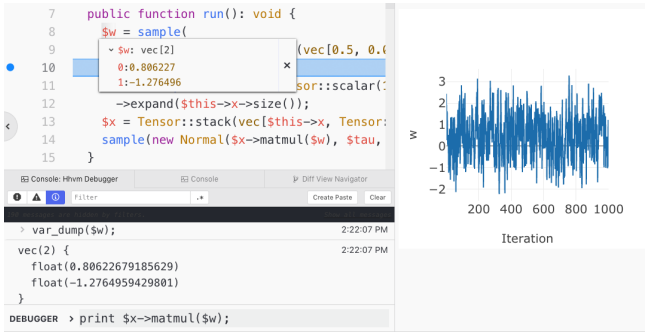

Fig. 1. When users open a compatible Hack script in the Nuclide IDE, the HackPPL playground pane automatically appears and calls to our `Viz` library will be rendered

6.4 Visualization Library

Our visualization library, `Viz`, provides an API for common visualization needs such as rendering distributions and plotting results of inference. It is built on top of Plotly [22], and adds special markers necessary to display charts within the HackPPL Playground. The advantage here is that we are able to use any charts available in the Plotly library for PPL-specific visualizations without needing to change our playground extension. Direct integration with the HackPPL Playground allows us to provide reactive charts, which update in realtime as inference progresses. These reactive charts include running mean trace plots,posterior distribution scatterplots, and marginal distribution plots for latent variables.

7 Case Study

We now present a case study using a crowdsourced annotation model. We introduce the problem followed by a discussion on the probabilistic approach and demonstrate the HackPPL workflow with results from a simulated dataset.

7.1 Motivation and Modeling

Social media companies routinely rely on the annotations provided by reviewers to estimate prevalence, or overall occurrence, of content that violates certain community standards. However, this can often be a challenging task as only a small subset of content can be manually reviewed, and humans reviewers may be prone to error. Fortunately, Bayesian approaches are very effective because they allow for reasoning under such uncertainty and give the ability to infer not just a point estimate of the prevalence, but also its credible intervals.

Previously, Carpenter [6] has proposed various generative models for annotated data. The models help to infer the overall prevalence as well as the confusion matrix of labelers and item level difficulty. However, these models are conditioned on the observed ground truth label of each item. Passonneau and Carpenter [25], on the other hand, describe a model based on the work by Dawid and Skene [9] for cases where the true labels are also unknown. In this case study, we will be focusing on the latter model.

More formally, assume that there are K categories of content, each with prevalence of $\theta_{(.)}$, where $\sum_{k=1}^{K} \theta_k = 1$. The true category y_i of an item i is not directly observed, but we do have the label r_ℓ given by human labeler ℓ. This rating is a noisy observation, corrupted by a confusion matrix ψ^ℓ, such that $\psi^\ell_{a,b}$ is the prior probability that labeler ℓ would assign a label of b to an item with the true category a. The goal of the inference is to estimate: 1) The true prevalence for each category, $\boldsymbol{\theta}$, 2) The latent category that each individual piece of content belongs, $\boldsymbol{y}$, and 3) The accuracy of each labeler on different content types, $\boldsymbol{\psi}$. The joint probability is given in Equation 4:

$$P(\boldsymbol{\theta}, \boldsymbol{\psi}, \boldsymbol{y}|\boldsymbol{r}) \propto P(\boldsymbol{\theta})P(\boldsymbol{y}|\boldsymbol{\theta})P(\boldsymbol{\psi})P(\boldsymbol{r}|\boldsymbol{y}, \boldsymbol{\psi}) \tag{4}$$

7.1.1 Model in HackPPL

For brevity, we present a simplified model in Listing 14 where $K = 2$ and there is a single confusion matrix for all labelers. This matrix is fully specified by the false positive (FPR) and the false negative rates (FNR), where *positive* refers to violating content. To address the bimodal likelihood of the model [6], we constrain false positive and false negative probabilities to follow a Uniform(0.0, 0.5) distribution. We place a uniform prior on the prevalence estimate θ. The true category of each item is defined with the DTensor abstraction and is marginalized out of the model when used with HMC, as discussed in Section 5. We then use vectorization and broadcasting semantics of tensors to efficiently define the likelihood distribution of the labels and obtain the score of the ratings.

Listing 14. Two category annotation model with a single confusion matrix for all labelers using a non-informative prior for prevalence

```
<<__PPL>>
class AnnotationModel implements PPLModel {
  public function __construct(
    private int $n, private DTensor<int> $labels) {}
  public function run(): void {
    $theta = sample(
      new Beta(Tensor::scalar(1.), Tensor::scalar(1.)),
      'theta')->expand(vec[$this->n]);
    $u = new Uniform(Tensor::scalar(0.), Tensor::scalar(.5));
    $fpr = sample($u, 'false_positive_rate');
    $fnr = sample($u, 'false_negative_rate');
    $tpr = $fnr->neg()->add(Tensor::scalar(1.0));
    $cat = sample(new Bernoulli($theta)),
      'item_category')->toRealValueTensor();
    $probs = Tensor::where($cat, $tpr, $fpr);
    sample(new Bernoulli($probs), 'labels', $this->labels);
  }
}
```

7.2 Result

Our primary goal in this case study is to assess how well the model predicts uncertainty in prevalence estimates. For validation, we generate annotation data for 1000 pieces of content, which may be either non-violating or violating, and compare against the known prevalence rates used in this generation. The simulation is repeated for prevalence levels from 0.05 to 0.3 with increments of 0.05. In order to understand the uncertainty around prevalence in the presence ofs labeler error, we also simulate data for different confusion matrices, with FPR and FNR ranging between 0.1 to 0.4. We produce 100 datasets for each prevalence and confusion matrix configuration, and run HMC for 10000 iterations with 5000 burn-in iterations. We also start the sampling from the expected values of the prior distributions.

Since we are interested in assessing our predictions for the uncertainty in prevalence, we show the coverage probabilities against true prevalence values in Figure 3. Overall, the credible intervals cover the ground truth prevalence for most

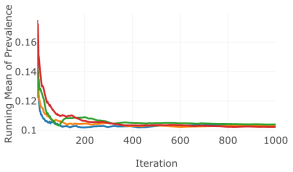

Fig. 2. Running mean of prevalence values for a scenario with a target prevalence of 0.1

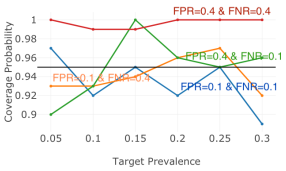

Fig. 3. Coverage probability across 100 simulated datasets for the various levels of prevalence

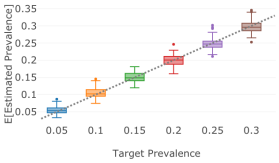

Fig. 4. Expected values of estimated vs. simulated prevalence; $FPR = FNR = 0.1$

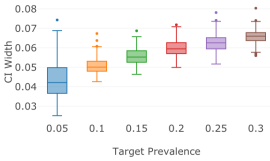

Fig. 5. 95% credible interval width of estimated vs. simulated prevalence; $FPR = FNR = 0.1$

scenarios; the coverage probabilities are obtained above 90% for all scenarios. The boxplots of the expected prevalence in Figure 4 for a particular simulation setting show that the expected values are centered around the true prevalence. The credible intervals in Figure 5 have increasing widths with respect to the true prevalence values. On average, 95% credible intervals have a width of 0.04 and increases to 0.065 when base prevalence is 5% and 30%, respectively. In Figure 2, we also provide the running mean of prevalence for a scenario with a true prevalence of 0.1, where we observe that the mean converges to the true value after 500 iterations.

The annotation model is used to estimate the uncertainty in predicting prevalence by incorporating labeler accuracy. The goal of this model when used in production is to assist with monitoring changes in prevalence over time. In an offline environment, we can validate our model using simulated data. The model can easily be used to observe time patterns of credible intervals, and it can also be used with change point detection techniques to detect changes in prevalence patterns.

8 Conclusion

This paper has overviewed the design motivations and their implementations in the building of HackPPL. By integrating probabilistic programming directly into the development of a general-purpose programming language, HackPPL aims to offer a compelling user experience without compromising on performance. We have integrated directly with familiar tools, including the Hack language, PyTorch tensors, and the Nuclide IDE, in order to provide a modeling experience that is both ergonomic and efficient. HackPPL is now deployed at a large technology firm and is being used to solve business problems through deep integrations with critical services.

References

[1] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

[2] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *arXiv e-prints*, Article arXiv:1810.09538 (Oct 2018), arXiv:1810.09538 pages. arXiv:cs.LG/1810.09538

[3] Michael Bolin. 2015. Building Nuclide, a unified developer experience. https://code.fb.com/developer-tools/building-nuclide-a-unified-developer-experience/ Accessed: 2019-03-16.

[4] Andrey Breslav and Roman Elizarov. 2018. Kotlin Coroutines. https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md

[5] Tamara Broderick, Michael I Jordan, Jim Pitman, et al. 2012. Beta processes, stick-breaking and power laws. *Bayesian analysis* 7, 2 (2012), 439–476.

[6] Bob Carpenter. 2008. Multilevel bayesian models of categorical data annotation. *Unpublished manuscript* 17, 122 (2008), 45–50.

[7] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles* 76, 1 (2017), 1–32. https://doi.org/10.18637/jss.v076.i01

[8] Torch Contributors. 2018. PyTorch C++ API. https://pytorch.org/cppdocs/ Accessed: 2019-03-16.

[9] Alexander Philip Dawid and Allan M Skene. 1979. Maximum likelihood estimation of observer error-rates using the EM algorithm. *Applied statistics* (1979), 20–28.

[10] Dino Distefano, Manuel Fahndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019 (to appear). Scaling Static Analyses at Facebook. *Commun. ACM* (2019 (to appear)).

[11] Arnaud Doucet, Simon Godsill, and Christophe Andrieu. 2000. On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing* 10, 3 (01 Jul 2000), 197–208. https://doi.org/10.1023/A:1008935410038

[12] Madalina M. Drugan and Dirk Thierens. 2003. Evolutionary Markov Chain Monte Carlo. In *Artificial Evolution*.

[13] Andrew Gelman, Jessica Hwang, and Aki Vehtari. [n.d.]. Understanding predictive information criteria for Bayesian models. *Statistics and Computing* 24 ([n. d.]), 997–1016.

[14] Andrew Gelman, Xiao-Li Meng, and Hal Stern. 1996. Posterior predictive assessment of model fitness via realized discrepancies. *Statistica sinica* (1996), 733–760.

[15] Walter R. Gilks, Antonia Thomas, and David J. Spiegelhalter. 1994. A Language and Program for Complex Bayesian Modelling. *Journal of the Royal Statistical Society. Series D (The Statistician)* 43, 1 (1994), 169–177. http://www.jstor.org/stable/2348941

[16] Noah D. Goodman. 2013. The principles and practice of probabilistic programming. In *POPL*, Vol. 13. 399–402.

[17] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2019-03-16.

[18] A. Griewank and A. Walther. 2008. *Evaluating Derivatives* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898717761 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898717761

[19] Cheng Guo and Felix Berkhahn. 2016. Entity Embeddings of Categorical Variables. *arXiv e-prints*, Article arXiv:1604.06737 (Apr 2016), arXiv:1604.06737 pages. arXiv:cs.LG/1604.06737

[20] Christopher Haynes, Daniel Friedman, and Mitchell Wand. 1986. Obtaining coroutines with continuations. *Computer Languages* 11 (12 1986), 143–153. https://doi.org/10.1016/0096-0551(86)90007-X

[21] Matthew D Hoffman and Andrew Gelman. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 1 (2014), 1593–1623.

[22] Plotly Technologies Inc. 2015. Collaborative data science. https://plot.ly

[23] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (Feb. 2009), 31 pages. https://doi.org/10.1145/1462166.1462167

[24] Radford M. Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo* 2, 11 (2011), 2.

[25] Rebecca J. Passonneau and Bob Carpenter. 2014. The benefits of a model of annotation. *Transactions of the Association for Computational Linguistics* 2 (2014), 311–326.

[26] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[27] Martyn Plummer. 2003. JAGS: A Program for Analysis of Bayesian Graphical Models using Gibbs Sampling. *3rd International Workshop on Distributed Statistical Computing (DSC 2003); Vienna, Austria* 124 (04 2003).

[28] Rajesh Ranganath, Sean Gerrish, and David Blei. 2014. Black box variational inference. In *Artificial Intelligence and Statistics*. 814–822.

[29] Donald B. Rubin et al. 1984. Bayesianly justifiable and relevant frequency calculations for the applied statistician. *The Annals of Statistics* 12, 4 (1984), 1151–1172.

[30] Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. 2015. Practical probabilistic programming with monads. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 165–176.

[31] Jon Skeet. 2013. *C# in Depth* (3rd ed.). Manning Publications Co., Greenwich, CT, USA.

[32] Facebook Open Source. 2018. Ending PHP Support, and The Future Of Hack. https://hhvm.com/blog/2018/09/12/end-of-php-support-future-of-hack.html Accessed: 2019-03-16.

[33] Facebook Open Source. 2019. Hack - Programming Productivity Without Breaking Things. https://hacklang.org/ Accessed: 2019-03-16.

[34] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv e-prints*, Article arXiv:1610.09787 (Oct 2016), arXiv:1610.09787 pages. arXiv:stat.CO/1610.09787

[35] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 770–778.

[36] David Wingate and Theophane Weber. 2013. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299* (2013).

[37] Frank Wood, Jan Willem van de Meet, and Vikash Masinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS)*.