# 1 Overview

PPLBench is a benchmark for analyzing and comparing the accuracy and performance of Probabilistic Programming Languages and Libraries (PPLs) over various probabilistic models. It is designed to be open and modular such that contributors can add new models, PPL implementations, and evaluation metrics to the benchmark.

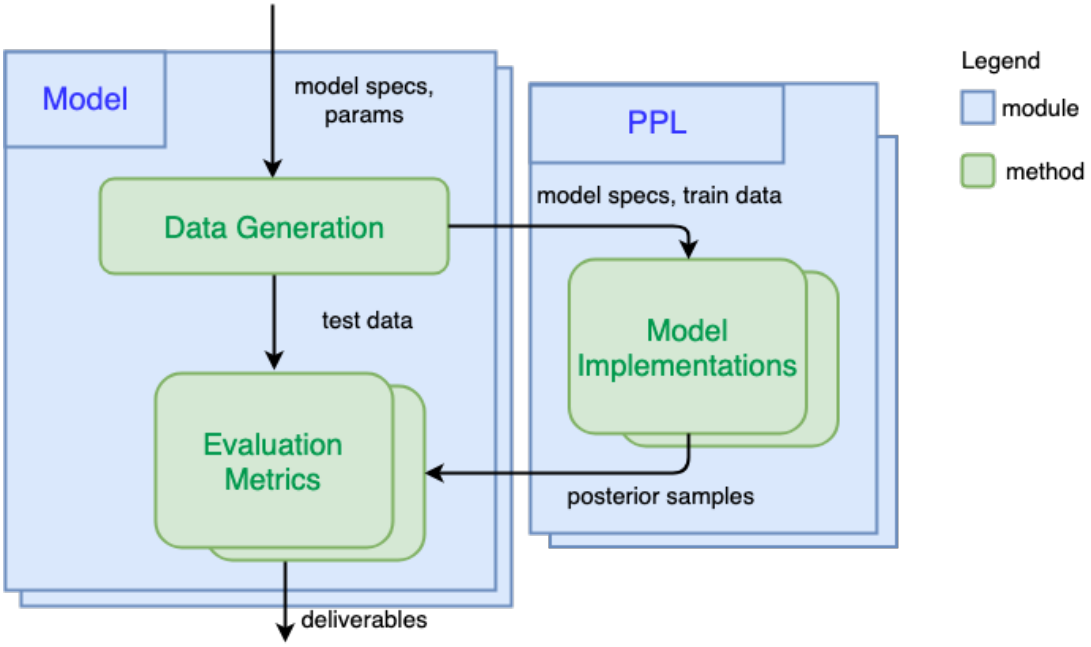


Figure 1: PPLBench system overview

PPLBench is implemented in Python 3[**?**]. Each model in PPLBench as well as a model's PPL implementation is a module as seen in figure 1. The typical PPLBench flow is as follows:

- **Model Instantiation and Data Generation**: We establish a model $P_\theta(X, Z)$ with ground-truth parameter distributions($\theta \sim P_\phi$) where $\phi$ is the hyperprior. In some models theta is sampled while in others it can be specified by users.
  The model factorizes as:
  $$P_\theta(X, Z) = P_\theta(Z)P_\theta(X|Z)$$
  And sample parameters:
  $$Z_1 \sim P_\theta(Z)$$
  We then simulate train and test data from this model:
  $$X_{train} \overset{iid}{\sim} P_\theta(X|Z = Z_1)$$
  $$X_{test} \overset{iid}{\sim} P_\theta(X|Z = Z_1)$$
  Note that this process of data generation is performed independent of any PPL.
- **PPL Implementation and Posterior Sampling**: Using the training data, we train various PPL implementations which learn a model $P_\theta(X = X_1, Z)$; obtain $n$ posterior samples of parameters along with runtime information.
  $$Z^*_{1\ldots n} \sim P_\theta(Z|X = X_{train})$$
  PPLBench is designed to accept an approximate time per PPL per iteration. The PPLBench estimates the number of posterior samples to obtain from the implementations based on this time constraint. The PPL's default warmup/burn-in and initialization parameters were left unchanged. The sampling is restricted to a single chain and any form of multi-threading is disabled.

- **Evaluation**: Using the test data $X_{test}$, posterior samples $Z^*_{1...n}$ and timing information, the PPL implementations are evaluated on various evaluation metrics. The default evaluation metric is average log predictive vs. time for each implemented PPL. A running average of the log posterior predictive is computed w.r.t $n$ and a further mapping $n$ to the elapsed time is computed. The final plot is hence a running average posterior predictive w.r.t time. This metric has been previously used by [**?**] to compare convergence of different implementations of statistical models.

$$AveragePosteriorPredictive(n) = \frac{1}{n}\sum_{i=1}^{n}\log(P(X_{test}|Z = Z^*_i))$$

  After computing the average posterior predictive w.r.t time over each iteration, the min, max and mean values over iterations is obtained. These are plotted on a graph for each PPL implementation so their convergence behavior can be visualized. The time axis in the plot is on a log scale to help accommodate larger timescales and vastly different convergence behaviors. This plot is designed to capture both the relative performance and accuracy of these implementations.

Using the PPLBench framework, we establish three models to evaluate performance of PPLs:

1. Robust regression model with Student-T errors[**?**]
2. Latent Keyphrase Index(LAKI) model, a Noisy-Or Topic Model[**?**]
3. Crowdsourced annotation model, estimating true label of an item given a set of labels assigned by imperfect labelers[**?**]

These models are chosen to cover a wide area of use cases for probabilisic modelling. All the above models have been implemeted in JAGS[**?**] and Stan[**?**] PPLs. The model and PPL implementation portfolio is expected to be expanded with additional open-source contributions. Subsequent sections describe these models and implementations in detail along with analysis of observed results.

## 2 Robust Regression Model

Bayesian regression with gaussian errors, like ordinary least-square regression is quite sensitive to outliers[**?**]. To increase outlier robustness, a Bayesian regression model with Student-T errors is more effective[**?**]. This is the first model which while being relatively straightforward is quite effective in several real-world usecases.

$$y \sim StudentT(\nu, \mu, \sigma) \tag{1}$$

Here, $\nu$ is the degrees of freedom parameter. As $\nu$ approaches $\inf$, the distribution approaches normal. Here we choose prior for $\nu$ as:

$$\nu \sim Gamma(shape = 2, scale = 10)$$

$\mu$ is the mean of the regression and is given by:

$$\mu = \alpha + \beta * X$$

$\sigma$ is the variance of the $StudentT$ distribution and has a prior:

$$\sigma \sim Exponential(scale_\sigma)$$

And $\alpha, \beta, X$ are sampled from:

$$\alpha \sim Normal(0, scale_\alpha), \quad \beta \sim Normal(0, scale_\beta), \quad X \sim Normal(0, 10)$$

### 2.1 Data Generation

PPLBench generates all the data internally. This allows for a more customizable way to evaluate the PPL implementations of the model. In this case, to test the robustness of the model implementations, we generate data by sampling the alphas and betas from the $\sigma = [10, 11]$ tails of the distribution. A total of $2N = 2000$ data points were generated, where each point being generated from a sample drawn for these tails of parameter distributions. We use half of the generated data as evidence for the model implementations to facilitate convergence of sampling process. The other half are used to compute the posterior predictive log-likelihood of the parameter samples.

Table 1: Hyperpriors for Robust Regression Model

| Hyperprior | Value | Notes |
|---|---|---|
| $scale_\alpha$ | 10 | Variance of the Normal prior of $\alpha$ |
| $scale_\beta$ | 2.5 | Variance of the Normal prior of $\beta$ |
| $scale_\sigma$ | 1 | Scale of Exponential prior of $\sigma$ |
| Tails | [10, 11] | Tails of prior distributions |

### 2.2 PPL Implementations

The robust regression model was implemented in Stan and Jags PPLs, with the help of PyStan[**?**] and PyJAGS[**?**] libraries for interface. The compilation and inference times were recorded. The configuration of hyperpriors is listed in table1. One noteworthy difference between the two would be that Jags uses Gibbs sampler while Stan uses NUTS HMC. This difference will be important when we analyze the results.

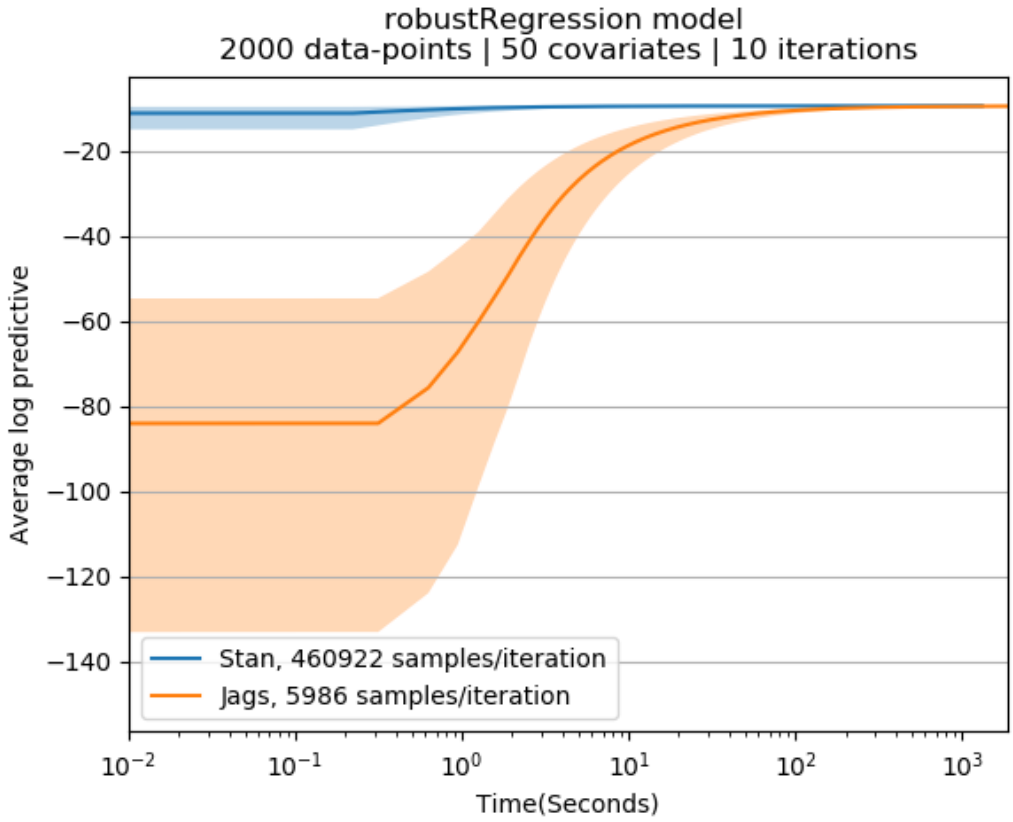


Figure 2: Posterior convergence behaviour of Jags and Stan for Robust Regression model

### 2.3 Results

Figure 1 shows the comparative performance without considering the inference time. From the figure, we can make the following observations:

- Both implementations converge to the same posterior predictive log-likelihood given enough time.
- Without considering compile time, Stan is both faster to converge as well has a much lower time per sample.
- Stan's NUTS inference starts sampling from relatively high log-likelihood space and hence converges much quicker than Jags' Gibbs sampler, which has a much slower convergence.

## 3 Noisy-Or Topic Model

Inferring topics from the words in a document is a crucial task in many natural language processing tasks. The noisy-or topic model[**?**] is one such example, which is quite simple yet effective. It is a Bayesian network consisting of two types of nodes; one type are the domain keyphrases (topics) that are latent while others are the content units (words) which are observed. Words have only topics as their parents. To capture hierachical nature of topics, they can have other topics as parents. See figure 3 which illustrates the model structure.

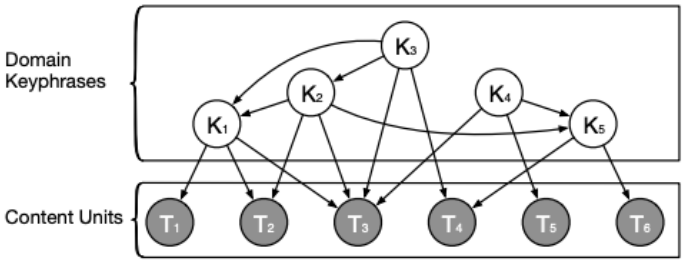


Figure 3: Illustrative Bayesian network for the Noisy-Or Topic model[**?**]

Let $Z$ be the the set of all nodes in the network. Each model has a leak node $O$ which is the parent of all other nodes. The set of keyphrases is denoted by $K$ and the set of content units is represented as $T$. The number of children for each keyphrase $K_i$ is sampled from a poisson prior:

$$|Children(K_i)| \sim Poisson(\lambda_{fanout})$$

The children are chosen at random without replacement from the rest of the nodes in $Z \in [i+1, |Z|]$ . The weight of the leak node $O$ to all the nodes ($W_{oj}$) is sampled from an exponential prior:

$$W_{oj} \sim Exponential(scale_{leakweight})$$

The weight of keyphrase node $K_i$ for child $j$ is sampled from an exponential prior:

$$W_{ij} \sim Exponential(scale_{weight})$$

The likelihood for each node $j$ in $Z$ is given as:

$$P(Z_j|Parents(Z_j)) = 1 - \exp(-W_{oj} - \sum_i (W_{ij} * Z_i))$$

and the state of each node $j$ in $Z$ is sampled from a Bernoulli distribution with the above probability:

$$Z_j \sim Bernoulli(P(Z_j|Parents(Z_j)))$$

### 3.1 Data Generation

For the experiment, we first sample the model structure. This consists of determining the parent key-phrase nodes for each content units, as well as which key-phrase nodes are parent of each key-phrase node. Next, the weights are sampled for all connections from thier respective priors. The hyperprior configuration is listed in table 2. Now the key-phrase nodes are sampled once; and two instances of content units are sampled keeping the key-phrase nodes fixed. One instance is passed to PPL implementation for inference, while the other is used to compute posterior predictive of the obtained samples.

### 3.2 PPL Implementations

The model was implemented in Stan and Jags PPLs, with the help of PyStan[**?**] and PyJAGS[**?**] libraries for interface. The compilation and inference times were recorded. The model requires the support for discrete latent variables, which JAGS has, but Stan does not. This means that the Stan implementation requires a reparameterization for denoting whether a node is activated or not. This was achieved representing boolean variables as a relaxation of 1-hot encoded vectors[**?**]. So, for example, if true is represented by $[1, 0]$ and false by $[0, 1]$ in the 1-hot encoding then a relaxed

Table 2: Hyperpriors for Noisy-Or Topic Model

| Hyperprior | Value | Notes |
|---|---|---|
| $\lambda_{fanout}$ | 3 | Poisson prior for number of children to each key-phrase node |
| $scale_{leakweight}$ | 0.1 | Exponential prior for $W_{oj}$ |
| $scale_{weight}$ | 1 | Exponential prior for $W_{ij}$ |
| word fraction | $\frac{2}{3}$ | ratio of number of content units to all nodes |

representation of a true value might look like [0.9999, 0.0001]. The reparameterization process is as follows: First, you encode the probabilities of a node in paramter vector $\alpha = [\alpha_{true}, \alpha_{false}]$:

$$\alpha_{true} = P(Z_j|Parents(Z_j)), \quad \alpha_{false} = 1 - P(Z_j|Parents(Z_j))$$

We choose a temprature parameter $\tau = 0.1$. Then for each keyphrase, you assign an intermedieate random variable $X_j$:

$$X_j = softmax(\log \alpha_j + G_j/\tau), \quad Where, G_j \overset{iid}{\sim} Gumbel(0, 1)$$

We then obtain $Z_j$ from $X_j$ as follows:

$$Z_j = 1 - argmax(X_j)$$

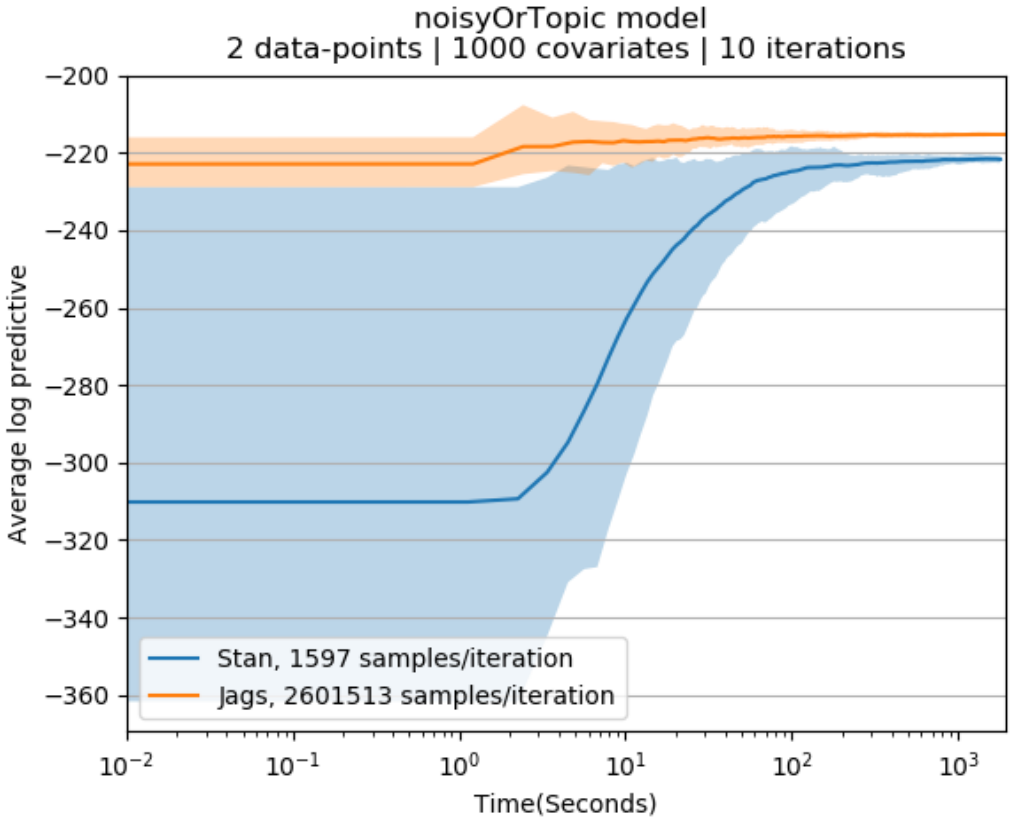


Figure 4: Posterior convergence behaviour of Jags and Stan for Crowdsourced Annotation Model

### 3.3 Results

Figure 6 shows the comparative performance. From the figure, we can make the following observations:

- JAGS converges to higher log-likelihood than Stan, this can be due to the reparameterization of Stan.
- JAGS is both faster to converge as well has a much lower time per sample.
- JAGS' inference starts sampling from relatively high log-likelihood space and hence converges much quicker than STAN's, which can again be attributed to the reparameterization.

## 4 Crowdsourced Annotation Model

There exist several applications where the complexity and volume of the task requires crowdsourcing to a large set of human labelers. Inferring the true label of an item from the labels assigned by several labelers is facilitated by this crowdsourced annotation model[**?**]. The model takes into consideration an unknown prevalence of label categories, an unknown per-item category, and an unknown confusion matrix per-labeler.

The model is shown in plate notation in figure5

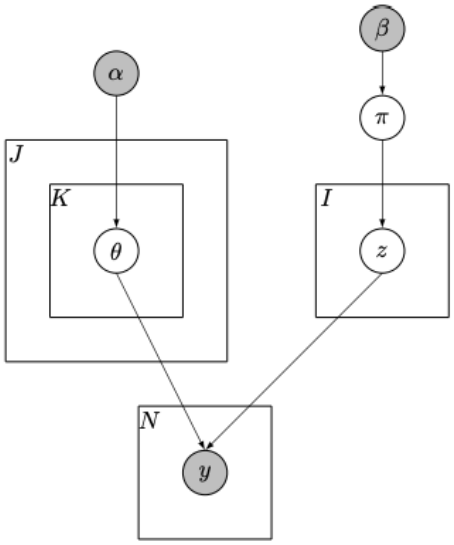


Figure 5: Plate notation of crowdsourced annotation model[**?**]

Let $K$ be the number of possible labels or categories for an item, $I$ the number of items to annotate, $J$ the number of labelers, and $N$ the total number of labels provided by labelers.Each item is labeled by at least one labeler.
The parameters of the model are as follows:

- $z_i \in 1 : K$ for the true category of item $i$,
- $\pi_k \in [0, 1]$ for the probability that an item is of category $k$.
- $\theta_{j,k,k'} \in [0, 1]$ for the probabilty that annotator $j$ assigns the label $k'$ to an item whose true category is $k$; Subject to $\sum_{k'=1}^{K} theta_{j,k,k'} = 1$

The model first selects the true category $z_i$ for item $i$ according to the prevalence $\pi$ of categories where,

$$\pi \sim Dirichlet(\beta);$$

and,

$$z_i \sim Categorical(\pi)$$

Each item $i$ is labelled by $|J_i|$ labelers, where

$$|J_i| \sim Poisson(\lambda_{labeler})$$

These labelers are choosen at random from all the labelers

$$J_i \sim Categorical(J, |J_i|)$$

For each labeler $j$ the confusion matrix $\theta_{j,k}$,(k) is sampled from dirichlet prior $\alpha_k$, where

$$\alpha_{k,k} = E_{correctness}, \quad \alpha_{k,k'} = (1 - E_{correctness})/(K - 1)$$

Here, $E_{correctness}$ denotes the expected proportion of times the labeler correctly identifies the true label. $\theta_{j,k}$ for each labeler is sampled as follows:

$$\theta_{j,k} \sim Dirichlet(\alpha_k)$$

The label $y_n$ for item $i$ is sampled from the labeler $j$'s confusion matrix $\theta_{j,z[i]}$

$$y_n \sim Categorical(\theta_{J_{i[n]},z_i})$$

### 4.1 Data Generation

For the experiment, we simulate a model with $|J| = 100$ labelers, $|I| = 5000$ items, and $|K| = 3$ categories. The hyperprior configuration is listed in table 3. The labels of half of the items are passed to the PPL implementations for training and the other half is used to compute the posterior predictive of the obtained samples.

Table 3: Hyperpriors for Crowdsourced Annotation Model

| Hyperprior | Value | Notes |
|---|---|---|
| $\lambda_{labeler}$ | 2.5 | Poisson prior for number of labelers assigned to each item |
| $\beta$ | $[\frac{1}{K}]$ | Dirichlet prior for $\pi$ |
| $E_{correctness}$ | 0.5 | Expected accuracy of a labeler |

### 4.2 PPL Implementations

The model was implemented in Stan and Jags PPLs, with the help of PyStan[**?**] and PyJAGS[**?**] libraries for interface. The compilation and inference times were recorded. The model requires the support for discrete latent vairables, which JAGS has, but Stan does not. This means that the Stan implementation requires an additional marginaliztion step w.r.t $z[i]$, the true label of each item.

### 4.3 Results

Figure 6 shows the comparative performance. From the figure, we can make the following observations:

- Both implementations converge to the same posterior predictive log-likelihood given enough time.
- Jags is both faster to converge as well has a much lower time per sample.
- Jags' inference starts sampling from relatively high log-likelihood space and hence converges much quicker than STAN's

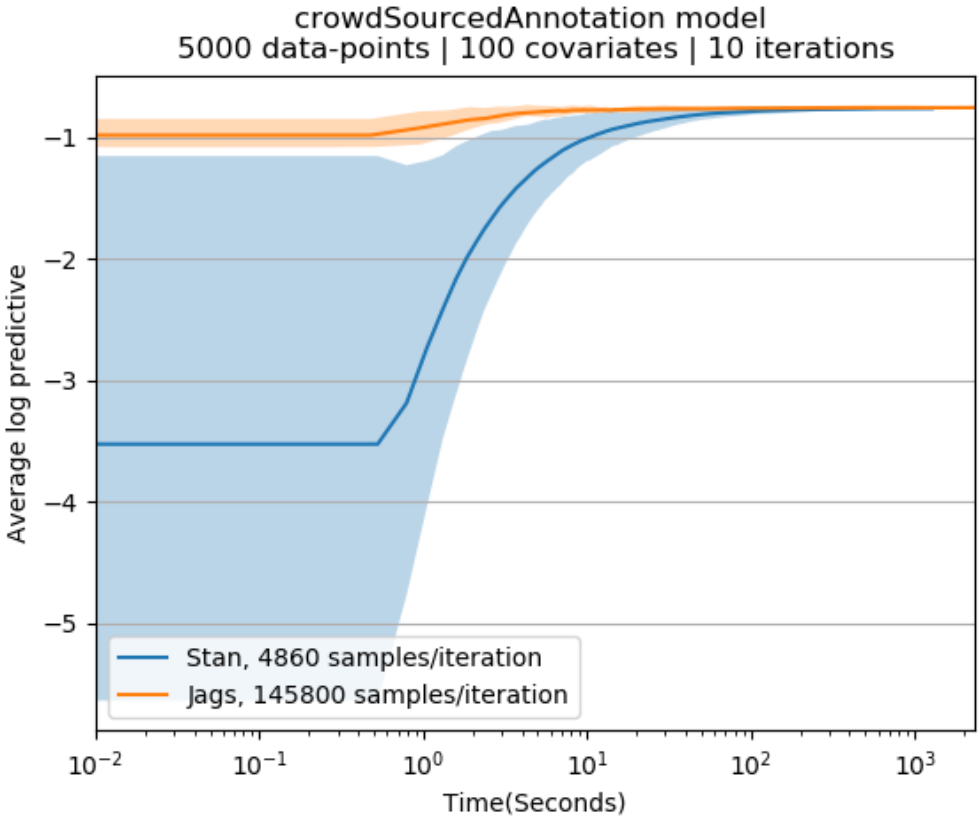


Figure 6: Posterior convergence behaviour of Jags and Stan for Crowdsourced Annotation Model