STA663 Final Project: Stochastic Gradient Hamiltonian Monte Carlo

Boyao Li
Master of Biostatistics
Duke University
boyao.li@duke.edu

Shota Takeshima
Master in Interdisciplinary Data Science
Duke University
shota.takeshima@duke.edu

GitHub repository

https://github.com/shttksm/SGHMC

Abstract

Hamiltonian Monte Carlo (HMC) is a sampling technique which has a mechanism for defining target distributions with higher acceptance probabilities. Thus, it allows a more effective search of the state space and a faster convergence than standard sampling methods. However, one of its limitation is that the required computation of gradient terms is infeasible in real problems. To address the computation problem, Chen et al. proposed a sampling method named stochastic gradient Hamiltonian Monte Carlo (SGHMC), which is a HMC sampling with *friction* added to the momentum update, and this noisy gradient term is calculated per mini-batch of the data provided. In this project, we implemented the SGHMC algorithm in Python, applied optimization to speed the function up and then explored it with both simulated and real datasets. The experimental results are also shown in following sections.
key words: *Hamiltonian, Monte Carlo, stochastic gradient, friction, momentum, minibatch*

1 Background

Hamiltonian Monte Carlo (HMC) [1] sampling method is an advanced and powerful Markov chain Monte Carlo (MCMC) sampling algorithm. The methods define a Hamiltonian function including the *potential energy* and the *kinetic energy* terms from the physics definition to find the target distribution. It defines distant proposals with high acceptance probabilities in a Metropolis-Hastings (MH) framework, enabling more efficient exploration of the state space and faster convergence, which also make HMC methods grow in popularity recently. However, HMC has a limitation that the requirement of computing gradient terms of potential energy for the whole dataset is time consuming and infeasible, especially for a large dataset. Instead, we need to estimate the noisy gradient from a subset of the data. The paper Stochastic Gradient Hamiltonian Monte Carlo by Chen et al. [2] developed a variant HMC to sidestep the computational limitation of the HMC method. Instead of calculating the gradient of potential energy function for the whole dataset, Chen applied stochastic gradient on mini-batches of data and introduced a friction term to the momentum update. This friction term could counteract the inner noise and maintain the desired target distribution to be invariant.

SGHMC algorithm is an efficiency improvement to traditional HMC algorithm and can be applied in many situations where MCMC methods are necessary. It could decrease the total computation cost particularly when applied to a large dataset, while also produce samples that need low correction during the sampling process. A number of simulated experiments in this project validate this improvement. We also apply the SGHMC algorithm to Iris Flower Data set, which is a binary classification task. Our experimental results demonstrate the effectiveness of the SGHMC proposed by Chen et al..

2 Descriptions of Algorithms

2.1 Hamiltonian Monte Carlo

Suppose we want to sample from the posterior distribution of θ given a set of independent observations $x \in \mathcal{D}$:

$$p(x \mid D) \propto \exp(-U(\theta)) \tag{1}$$

where the potential energy $U(\theta)$ is given by:

$$U(\theta) = -\sum_{x \in D} \log p(x \mid \theta) - \log p(\theta) \tag{2}$$

A Hamiltonian system generates samples of θ based on introducing a set of auxiliary momentum variables r, which means sampling from a joint distribution of (θ, r) defined by

$$\pi(\theta, r) \propto \exp\left(-U(\theta) - \frac{1}{2} r^T M^{-1} r\right) \tag{3}$$

Here M is a mass matrix, and $K(r) = \frac{1}{2} r^T M^{-1} r$ defines a kinetic energy term. The Hamiltonian function is defined by

$$H(\theta, r) = U(\theta) + K(r) = U(\theta) + \frac{1}{2} r^T M^{-1} r \tag{4}$$

Here H means the total energy of the physical system. To propose samples, HMC simulates the Hamiltonian dynamics.

$$\begin{cases} d\theta &= M^{-1} r dt \\ dr &= -\nabla U(\theta) dt \end{cases} \tag{5}$$

In practice, we cannot usually simulate from the continuous system of Eq.(5), and instead use a discretized system. One common approach is the "leapfrog" method which is outlined in Alg.1. Due to the inaccuracies introduced through the discretization, an MH step is necessary.

Algorithm 1 Hamiltonian Monte Carlo

Input: Starting position $\theta^{(1)}$ and step size ϵ
  for $t \leftarrow 1, 2, ...$ **do**
    Resample momentum r
    $r^{(t)} \sim \mathcal{N}(0, M)$
    $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)})$
    Simulate discretization of Hamiltonian dynamics in Eq.(5)
    $r_0 \leftarrow r_0 - \frac{\epsilon}{2} \nabla U(\theta_0)$
    for $i \leftarrow 1$ to m **do**
      $\theta_i \leftarrow \theta_{i-1} + \epsilon M^{-1} r_{i-1}$
      $r_i \leftarrow r_{i-1} - \epsilon \nabla U(\theta_i)$
    end for
    $r_m \leftarrow r_m - \frac{\epsilon}{2} \nabla U(\theta_m)$
    $(\hat{\theta}, \hat{r}) = (\theta_m, r_m)$
    Metropolis-Hasting correction:
    $u \sim$Uniform$[0, 1]$
    $\rho = \exp(H(\hat{\theta}, \hat{r}) - H(\theta^{(t)}, r^{(t)}))$
    if $u < \min(1, \rho)$ **then** $\theta^{(t+1)} = \hat{\theta}$
    end if
  end for

The HMC algorithm needs to compute the gradient of the potential energy function for the whole dataset, which could be infeasible. The paper shows that SGHMC could have a better performance with a large-scale dataset.

2.2 Stochastic Gradient Hamiltonian Monte Carlo

SGHMC introduces a stochastic gradient term with noise into the system based on mini-batch $\tilde{D}$. However, with the noisy gradient term, HMC will no longer be invariant. To minimize the injected noise, the paper also adds a friction term to the momentum update. The gradient term is defined as

$$\nabla \tilde{U}(\theta) = -\frac{|D|}{|\tilde{D}|} \sum_{x \in \tilde{D}} \nabla \log p(x \mid \theta) - \nabla \log p(\theta) \tag{6}$$

By the central limit theorem (CLT), the gradient could be approximated as $\nabla \tilde{U}(\theta) \approx \nabla U(\theta) + N(0, V(\theta))$. Based on this setting above, the simulation would be

$$\begin{cases} d\theta &= M^{-1} r dt \\ dr &= -\nabla U(\theta) dt - B M^{-1} r dt + N(0, 2B dt) \end{cases} \tag{7}$$

where $N(0, 2Bdt)$ is the normally approximated noise, $B = \epsilon V(\theta)$ is the diffusion matrix, ϵ is the step size that has a similar behavior with learning rate η in gradient descent. $-BM^{-1}rdt$ is the friction term. In practice, B will be unknown, instead we simply have an estimate $\hat{B}$. The paper introduced a user specified friction term $C \succeq \hat{B}$ and consider the following dynamics:

$$\begin{cases} d\theta &= M^{-1} r dt \\ dr &= -\nabla U(\theta) dt - C M^{-1} r dt + N(0, 2(C - \hat{B}) dt) + N(0, 2B dt) \end{cases} \tag{8}$$

Adding a momentum term to stochastic gradient descent (SGD) is common practice. Conceptually, SGHMC is associated with SGD with momentum. Let $v = \epsilon M^{-1} r$, the dynamic function Eq.8 could be written as

$$\begin{cases} \Delta\theta &= v \\ \Delta v &= -\epsilon^2 M^{-1} \nabla \tilde{U}(\theta) - \epsilon M^{-1} C v + \mathcal{N}(0, 2\epsilon^3 M^{-1} (C - \hat{B}) M^{-1}) \end{cases} \tag{9}$$

Define $\eta = \epsilon^2 M^{-1}, \alpha = \epsilon M^{-1} C, \hat{\beta} = \epsilon M^{-1} \hat{B}$. The updated rule becomes

$$\begin{cases} \Delta\theta &= v \\ \Delta v &= -\eta \nabla \tilde{U}(\theta) - \alpha v + \mathcal{N}(0, 2(\alpha - \hat{\beta})\eta) \end{cases} \tag{10}$$

When comparing to an SGD with momentum method, we can see that η correspond to the learning rate and $1 - \alpha$ the momentum term. The relevant algorithm is shown in Alg.2.

Algorithm 2 Stochastic Gradient Hamiltonian Monte Carlo

Input: Starting position $\theta^{(1)}$, learning rate η, momentum decay α, and noise estimate $\hat{\beta}$
  for $t \leftarrow 1, 2, \ldots$ **do**
    optionally, resample momentum v *as*
    $v^{(t)} \sim \mathcal{N}(0, M)$
    $(\theta_0, v_0) = (\theta^{(t)}, v^{(t)})$
    simulate dynamics in Eq.10
    for $i \leftarrow 1$ to m **do**
      $\theta_i \leftarrow \theta_{i-1} + v_{i-1}$
      $v_i \leftarrow (1 - \alpha) v_{i-1} - \eta \nabla \tilde{U}(\theta)(\theta_i) + \mathcal{N}(0, 2(\alpha - \hat{\beta})\eta)$
    end for
    $(\theta^{(t+1)}, v^{(t+1)}) = (\theta_m, v_m)$
  end for

In experiments, we set $\hat{\beta} = \eta \hat{V}/2$, where $\hat{V}$ is estimated based on the empirical Fisher Information. Additionally, $\alpha = 0.1$ or 0.01, and $\eta = \gamma / |\mathcal{D}|$, where $\gamma = 0.1$ or 0.01.

3 Optimization for Performance

We apply different methods to optimize SGHMC on the simulated data shown in Fig. 1. The algorithm for the SGHMC sampler is shown in Alg. 2. Regarding the optimization, we use two methods including Numba JIT and Cython to speed up the SGHMC sampling and compare their time cost with the plain Python implementation. The different results shown in Table 1 are generated by running 500 samples and 1 leapfrog step onto the noisy gradient computation.

Based on the profiling analysis (shown in optimization notebook), we can see that the bottleneck of the whole function is from the call of random sampling from the standard normal distribution and the lambda function that defines the gradient of potential energy function. So, our optimization is applied to speed up the gradient computing function. From the result shown in Table 1, we can see that the Numba JIT is about 20 times faster than plain Python and shows the best performance on speeding up the gradient function. Cython part could also speed up the function twice, however it is not so good as the Numba JIT. One of the possible reasons is that our implementation still has some Python part and could not be faster via running as C code.

In practice, we may not implement our own gradient function. Instead, we may use the gradient calculate by some built-in functions in packages such as TensorFlow, which could have a better optimization through Cython or optimized C++. In addition, we also find a build-in HMC function in TensorFlow Probability that implement HMC based on TensorFlow system.

Table 1: Optimization using Numba and Cython on simulation data in Section 4.

Method	Time per loop
Plain Python	$13.7ms \pm 166\mu s$
Numba	$633\mu s \pm 12.7\mu s$
Cython	$6.26ms \pm 414\mu s$

4 Applications to Simulated Data Sets

We conduct experiments on a simulated data set. As a baseline sampler, we use the standard HMC implementation. In the implementation, we can designate whether MH correction is applied or not. Regarding HMC, we also test how it performs with $\nabla\tilde{U}$ instead of ∇U. Lastly, we investigate how SGHMC works, which does not use an MH correction. Fig. 1 illustrates that the empirical distributions the methods mentioned above generate. We can observe that even without an MH correction, both HMC and SGHMC methods can provide the distributions that are similar to the true distribution. However, the result of naive stochastic gradient HMC diverges from the true distribution in case the algorithm does not use an MH correction. These findings match with the theoretical insights that both HMC and SGHMC maintain the invariance of Hamiltonian as $\epsilon \to 0$ while naive stochastic gradient HMC does not, though this can be corrected by an MH feature. On the other hand, the MH generally becomes costly as the size of the data increases.

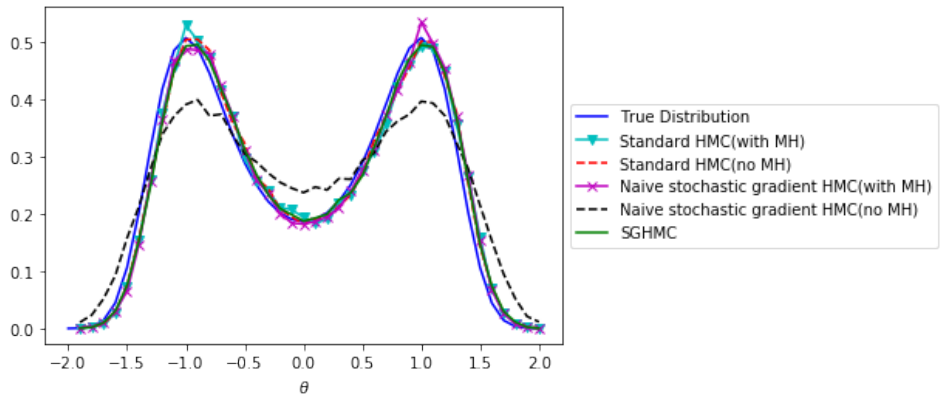

Figure 1: Empirical distributions associated with various sampling algorithms relative to the true target distribution with $U(\theta) = -2\theta^2 + \theta^4$. We compare the HMC method with and without the MH step to: (i) a naive variant that replaces the gradient with a stochastic gradient, again with and without an MH correction: (ii) SGHMC method, which does not use an MH correction. We use $\nabla\tilde{U}(\theta) = \nabla U(\theta) + \mathcal{N}(0, 4)$ in the stochastic gradient based samplers and $\epsilon = 0.1$ in all cases. Momentum is resampled every 50 steps in all variants of HMC.

Compared to other MCMC algorithms, the benefit of HMC algorithm is that it can sample efficiently from correlated distributions. SGHMC also has this property because it is in a subset of HMC. Fig. 2 shows that the first 50 samples generated by our SGHMC sampler. The momentum variable associated with SGHMC helps the sampler to move along the distribution contours. In the beginning of the sampling, SGHMC can obtain non-autocorrerated samples and explore widely over the distribution.

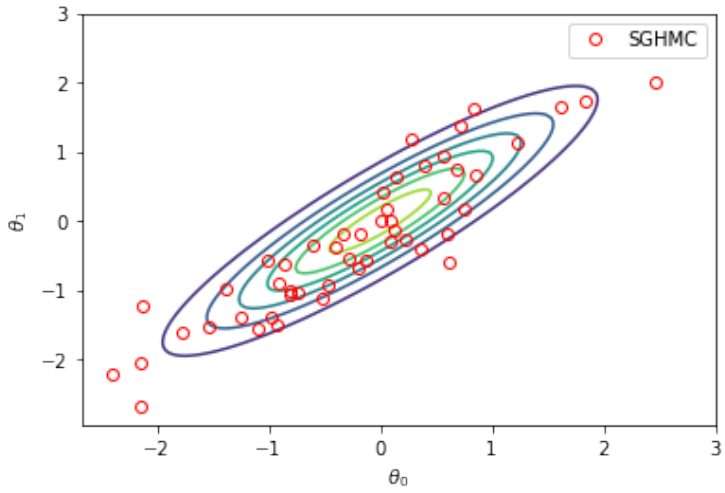

Figure 2: Sampling of a bivariate Gaussian with correlation using SGHMC. Here, $U(\theta) = \frac{1}{2}\theta^T\Sigma^{-1}\theta$, $\nabla\tilde{U}(\theta) = \Sigma^{-1}\theta + \mathcal{N}(0, I)$ with $\Sigma_{11} = \Sigma_{22} = 1$ and correlation $\rho = \Sigma_{12} = 0.9$. Circle points indicate the first 50 samples of SGHMC.

5 Applications to Real Data Sets

We also conduct our SGHMC implementation on a binary plant type classification using Iris Flower Data Set. [3] The data set consists of 70 training instances and 30 test instances. In this training and test data sets, there are two categories of iris flowers. (Originally, Iris Data Sets has three types of flower data. To conduct an experiment of binary classification, we drop all data of one

class.) The number of features for each observation is four. Fig. 3 shows the distribution of the observations in the first two features' plane. For classification, we consider a logistic regression, $y_i|x_i \sim \text{Bernoulli}(\pi_i), \log(\frac{\pi_i}{1-\pi_i}) = \sum_{p=0}^{3} \beta_p x_{ip}$, where y_i is a label and x_i is a feature vector of ith observation. The sampling procedure for β is carried out by running SGHMC using mini batch of $\lfloor\sqrt{70}\rfloor = 8$ training instances, then resampling hyperparameters after an entire pass over the training set. We run the samplers for 10^3 iterations and discard the initial 50 samples as burn-in.

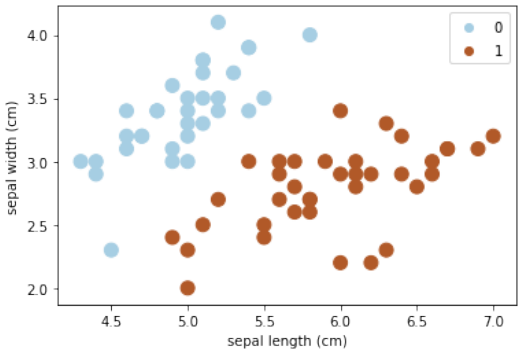

Figure 3: Iris class distribution on the first two features' plane

Fig. 4 shows the first 100 samples of $\beta_0, \cdots, \beta_3$. After about 10 iterations, each parameter is converged into a particular value. After about 10 iterations, each parameter converged into a particular value. To evaluate the trained logistic regression model, we take $10^3 - 50 = 950$ possible samples of β. In Fig. 5, we show the frequencies of $\beta_0, \cdots, \beta_3$. Using all (950) possible β, we calculate 950 probabilities, $Pr[y_i = 1|x_i]$ and average them as a final classification probability. After checking the model's performance on the test data set, we obtained 1.0 accuracy.

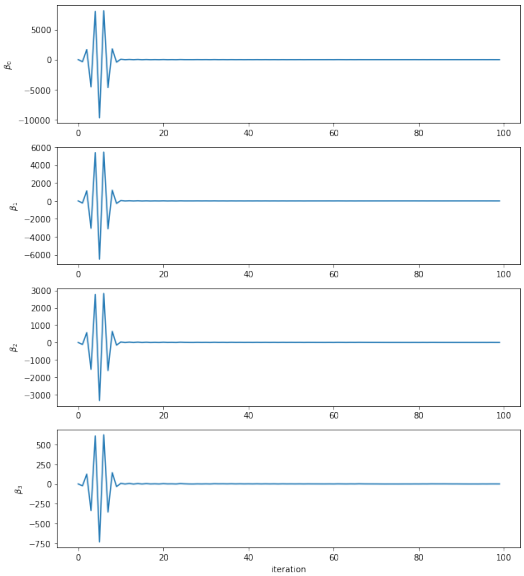

Figure 4: Initial 100 iterations of sampling process for hyperparameter, β

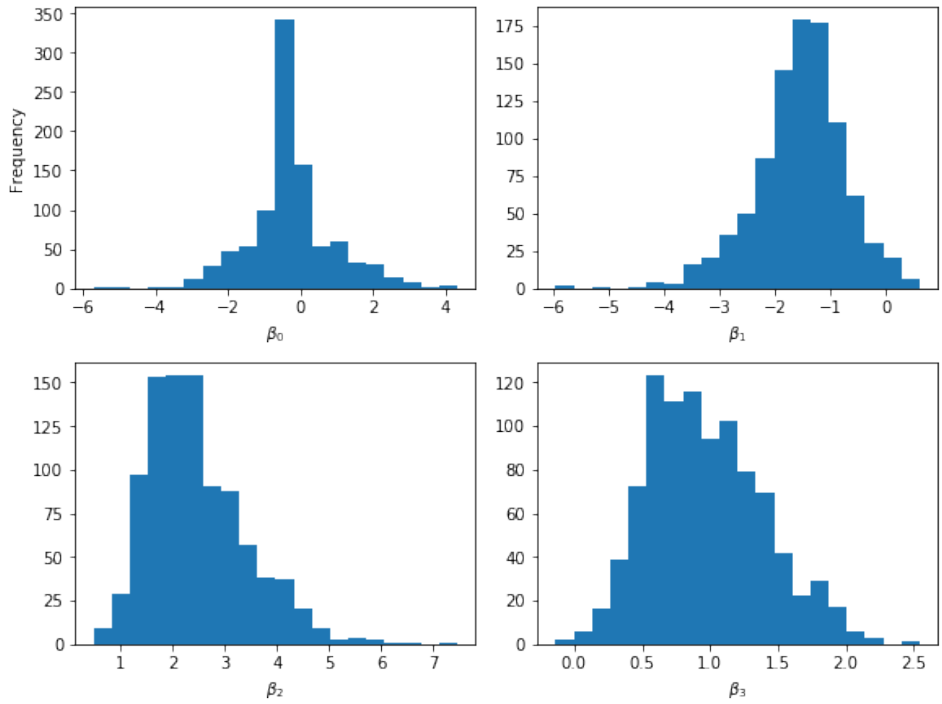

Figure 5: Distributions of sampled $\beta_0, \cdots, \beta_3$

6 Comparative Analysis with Competing Algorithms

As a comparative analysis, we compare the performance of our SGHMC implementation with NUT samplers in PyMC3 and PyStan packages. Fig. 6 and 7 demonstrate samples' distributions of their Gaussian Mixture models. PyMC3 and PyStan samplers can capture two posterior modes of parameters. On the other hand, our SGHMC sampler only captures either one mode because we do not implement multiple chains. As our future work, we should implement multiple chains to detect more complicated posterior distribution.

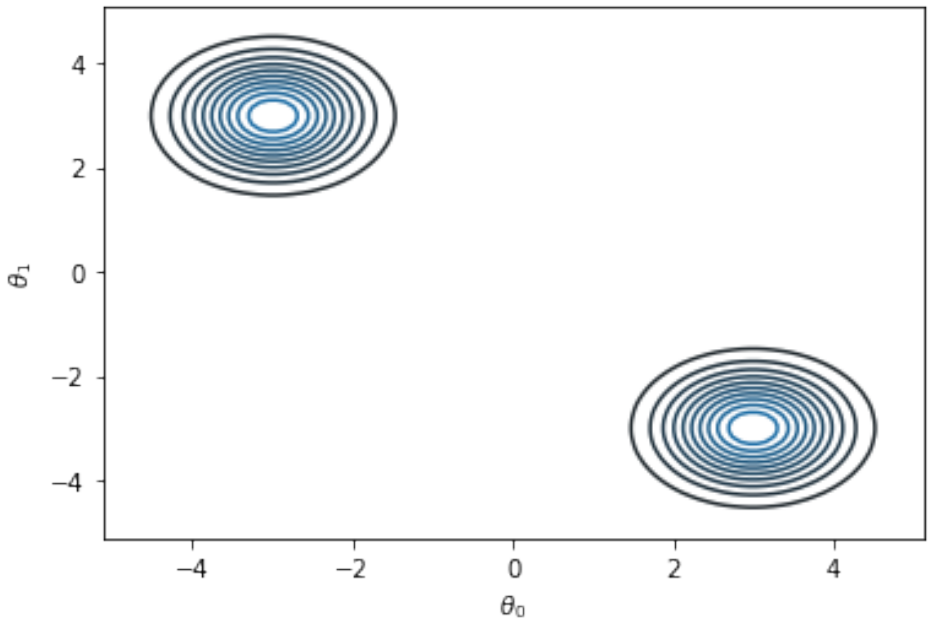

Figure 6: Using PyMC3 NUT sampler to construct sampling from the posterior distribution of mean parameters in Gaussian Mixture model. $X_i|\theta \sim 0.5\mathcal{N}(\theta_0, 1)+0.5\mathcal{N}(\theta_1, 1)$ where $(\theta_0, \theta_1) = (-3, 3)$. Also, we assume a weakly uniformative prior with $(\theta_0, \theta_1) \sim \mathcal{N}(0, 10I_2)$

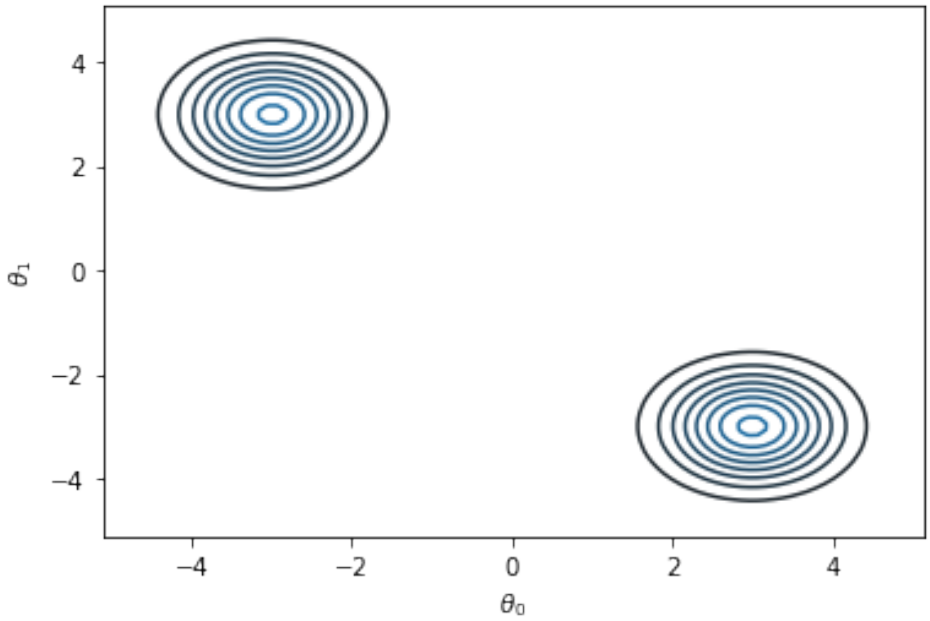

Figure 7: Using PyStan NUT sampler to construct sampling from the posterior distribution of mean parameters in Gaussian Mixture model.

7 Discussion and Conclusion

In this report, we implemented SGHMC in Python and optimize its performance. SGHMC inherits the framework of HMC and takes advantage of stochastic estimates of the gradient to reduce the

costly rigid calculation of the gradient. From the experiments on the simulated and actual dataset, we can confirm SGHMC method is efficient to sample over the entire distribution preventing samples' autocorrelation. Regarding the result of optimization using different methods, we can use numba JIT and Cython to speed up the sampling procedure, and the JIT shows a better performance. One natural next attempt is to write an optimized C++ code for SGHMC, and another possible way is to use the built-in function such as HMC function in TensorFlow Probability. In addition, from the result of comparative analysis with competing algorithms, to improve our SGHMC performance, we can add multiple chains feature into our implementation. In this case, we also need to take advantage of multiprocessing approach to reduce the increase of computational time. We believe efficient sampling methods like this SGHMC will contribute to scaling up Bayesian methods.

A Instruction to Install Our Implementation

We uploaded our implementation as a package in TestPyPI. You can install our package via 'pip' command and import it.

```
pip install -i https://test.pypi.org/simple/ SGHMC-bs
import sghmc
```

B Team Members' Contributions

- Boyao: I implemented SGHMC function, its test code and the optimization code in the jupyter notebook. Also, I wrote the sections in the report including the abstract, background, algorithm and optimization.
- Shota: I implemented applications on simulated and real data sets. Also, I wrote the sections in the report that are corresponding to my implementation. Finally, I prepared and maintained our git repository.

References

[1] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

[2] Tianqi Chen, Emily Fox, and Carlos Guestrin. Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pages 1683–1691. PMLR, 2014.

[3] R. A. FISHER. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.