Dive into Bayesian Learning via Stochastic Gradient Langevin Dynamics

Shaoxuan Chen, Instructor: Yves Atchadé

Abstract

The aim of this report is to summarize the exploration of the paper, "Bayesian Learning via Stochastic Gradient Langevin Dynamics"(Welling and Teh, 2011), I did in Bayesian Statistics course under the instruction of Prof. Yves Atchadé, which basically includes two parts: 1.The introduction of Stochastic Gradient Langevin Dynamics (SGLD) by comparing it with Metropolis-Adjusted Langevin Algorithm (MALA) 2.Application of stochastic gradient Langevin algorithm in posterior distribution sampling of Mixture Gaussians with tied means and Bayesian logistic regression model.

1 Introduction

With the advent of the information age, we have witnessed the datasets grow at an exponential rate ranging from all aspects of our life. However, to make full use of these large scale data, the traditional Markov Chain Monte Carlo (MCMC) algorithms may have poor performance or even will not work as those methods requires computations over the whole dataset in each iteration. However, the simple stochastic optimization algorithms like stochastic gradient descent, which updates the model parameters by the gradient of batch-size of data, has already been shown to have intriguing results. And it provides us with the intuition to combine the stochastic optimization with Bayesian learning. The paper of SGLD by Welling and Teh [2011] is the groundbreaking attempt in this direction.

2 Method

Compared with the flow of introduction of SGLD method in the original paper, this section provides a better way, based on my own point of view, to understand the method of SGLD. And it requires some basic understanding of MALA.

2.1 MALA

Recall that in Langevin process, if X is a d-dimensional continuous-time stochastic process satisfying the following stochastic Differential Equation (SDE):

$$dX_t = \nabla log(\pi(X_t)) + \sqrt{2}dW_t$$

where ∇ denotes the gradient, and W is the d-dimensional standard Brownian Motion. Then we can make the conclusion that X_t will have a stationary distribution π if it satisfies the above SDE. However, in order to sample from the stationary distribution by MCMC methods, we need to use Euler-Maruyama discretization to transfer the original continuous distribution into its discrete counterparts. Then we have the following process:

$$X_{t+1} = X_t + \epsilon \nabla log\pi(X_t) + \sqrt{2\epsilon} \cdot \eta_t,\ where\ \eta_t \sim \mathcal{N}(0, I_d)$$

After some rearrangement, we have:

$$X_{t+1} = X_t + \frac{\epsilon}{2} \nabla log\pi(X_t) + \eta_t,\ where\ \eta_t \sim \mathcal{N}(0, \epsilon \cdot I_d)$$

Note that in this discrete stochastic process, the X_t will have stationary distribution π_ϵ such that π_ϵ is "similar" to π. The transition distribution from X_t to X_{t+1} can be denoted as:

$$X_{t+1}|X_t \sim \mathcal{N}(X_t + \frac{\epsilon}{2} \nabla log\pi(X_t),\ \epsilon \cdot I_d)$$

And it is the same rationale for the distribution from X_{t+1} to X_t. Note that the ϵ is the hyperparameter denoting the stepsize. And if we denote the $\pi(X_t)$ above as the posterior distribution (without the normalized constant term) that we want to sample from, combining with the Metropolis-Hasting (MH) accept-reject test, we will have the based framework of MALA.

2.2 SGLD

The SGLD algorithm is very similar to MALA except for three main differences summarized below:

1. Stepsizes decrease towards zero at rates satisfying: $\sum_{t=1}^{\infty} \epsilon_t = \infty$, and $\sum_{t=1}^{\infty} \epsilon_t^2 < \infty$, which allows us to average out of the stochasticity in the gradients.

2. We can ignore the MH acceptance steps as the MH rejection rates go to zero asymptotically.[1]

3. The gradient of log likelihood within the log posterior is calculated by batch-size of data in each iteration and then rescaled to approximate the gradient of likelihood of the whole data set.

Therefore, if we denote $\pi(\theta_t)$ as the posterior distribution (without the normalized constant term). $p(\theta_t)$ as the prior distribution. We then can have the approximately gradient of the log posterior been written as:

$$\nabla log\pi(\theta_t) = \nabla log(p(\theta_t) + \frac{N}{n} logL(\theta_t|x_1, ..., x_n)) = \nabla logp(\theta_t) + \frac{N}{n} \sum_{i=1}^{n} \nabla logf(x_{ti}|\theta_t)$$

Then the stochastic process of SGLD can be written as:

$$\Delta\theta_t = \frac{\epsilon_t}{2}(\nabla logp(\theta_t) + \frac{N}{n} \sum_{i=1}^{N} \nabla logf(x_{ti}|\theta_t)) + \eta_t, \ where \ \eta_t \sim \mathcal{N}(0, \epsilon_t \cdot I_d)$$

Note that the above formula defines a non-stationary Markov Chain. And the stochasticity comes from two parts. The first part is the randomly chosen batch-size data in each iteration. And the second part is from the injected Gaussian noise adjusted by the decreasing stepsize. The author in the paper proved that the SGLD algorithm will transfer from stochastic optimization phase to Langevin dynamics phase, which basically imitates the Langevin dynamics MH algorithm, to sample directly from the posterior distribution.

The stochastic gradient Langevin algorithm is summarized below. Note that for computational convenience, the dimension of parameter θ is set to be 1×(p+1), where p is the number of parameters, and 1 is for the intercept.

Algorithm 1: SGLD

Required: The stepsize ϵ_t, Size of data $\mathcal{N}$, the batch size n. The loss function l. The number iterations. Stepsize adjusting hyperparameters a, b, γ. Gradient calculated by batch size of data, $\mathbf{g}$.

Initialize parameter vector θ_t, where $\theta_t \in \mathbb{R}^{1\times(p+1)}$, and a, b, γ to adjust the range of stepsizes

Initialize Res = θ_t, where Res is used to store the samples

for *t in range(1, number iterations)* **do**

  ϵ_t = a(b+t)$^{\gamma}$

  Randomly draw batch size of data: X_{mini} & y_{mini}

  Calculate the gradient $\nabla logp(\theta_t) + \frac{N}{n} \sum_{i=1}^{n} \nabla logf(x_{mini_{ti}|\theta_t})$, denote as $\mathbf{g}$

  Get Gaussian random noise η_t

  Update the θ_t: $\theta_t = \theta_t + \frac{\epsilon_t}{2} * \mathbf{g} + \eta_t$

  Update the Res by concatenating Res to θ_t vertically

end

Also, in practice, the author mentioned that we can use fixed stepsize SGLD after the algorithm transfer to the Langevin dynamics phase. The application of this method is easy, basically we first run the SGLD in large amount of iterations to make sure it passes the burn-in period, and use the last step updated θ_t to be the initial value for the fixed stepsize SGLD algorithm. Tune and fix the stepsize to run the algorithm and then the algorithm will sample from the posterior distribution. More importantly, different from the original SGLD, the fixed stepsize SGLD is a stationary Markov Chain. And we can use the classical methods to evaluate the MCMC sampler.

3 Experiments

3.1 Mixture Gaussians with tied means

This part is the replication of the first experiment in the paper. Basically it contains only two parameters θ_1 and θ_2. And the same prior as well as same data generation process as in the paper were chosen to apply the SGLD algorithm. Moreover, the fixed stepsize SGLD algorithm was also applied on the data and the samples were evaluated by histogram, ACF plots and box plots. The summary of the model is shown below.
Hyperparameters:

$$\gamma = -0.55, a = 0.03, b = 6, batch\ size = 30$$

Prior:

$$\theta_1 \sim \mathcal{N}(0, \sigma_1^2),\ \theta_2 \sim \mathcal{N}(0, \sigma_2^2), where\ \sigma_1^2 = 10, and\ \sigma_2^2 = 1$$

Data generating process:

$$x_i \sim \frac{1}{2}\mathcal{N}(\theta_1, \sigma_x^2) + \frac{1}{2}\mathcal{N}(\theta_1 + \theta_2, \sigma_x^2), where\ \theta_1 = 0, \theta_2 = 1, \sigma_x^2 = 2$$

Log of prior:

$$logp(\theta) = -log(2\pi\sigma_1\sigma_2) - \frac{1}{2}\theta^\top\Sigma^{-1}\theta, where\ \Sigma = diag(\sigma_1^2, \sigma_2^2)$$

Log likelihood calculated from batch-size of data:

$$logL(\theta|x1, .., xn) = \frac{N}{n}\sum_{i=1}^{n} log\{\frac{1}{4\sqrt{\pi}}exp(-\frac{(x_i - \theta_1)^2}{2\sigma_x^2}) + exp(-\frac{(x_1 - \theta_1 - \theta_2)^2}{2\sigma_x^2})\}$$

Gradient:

$$\nabla log\pi(\theta) = \nabla log(p(\theta) + \frac{N}{n}\nabla logL(\theta_t|x_1, ..., x_n))$$

Note that from the above setting, by the formula of stepsize, $\epsilon_t = a(b + T)^{-\gamma}$, the stepsizes will decrease from about 0.01 to 0.0001 within 30000 iterations. And the posterior sampling results after 30000 iterations are shown in Figure 1.

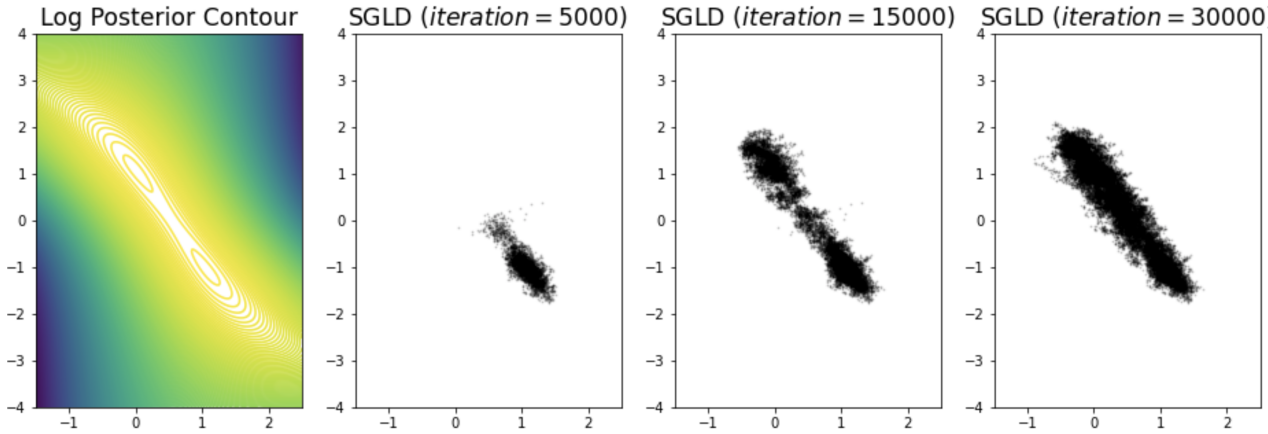

Figure 1: True and estimated posterior distribution after different iterations

From above we can see that the SGLD algorithm works very well to sample from the posterior. And after 15000 iterations, the estimated posterior distribution is already descent enough.

Also, as for posterior samples visualization, from Figure 2 we can see the 3-D plot of posterior samples with respect to the log of posterior values, which is roughly the same as shown in Figure 1 if we project all the points down to the X and Y axis.

Moreover, to verify the statement in the paper that, quote" as $\epsilon_t \to 0$, the discretization error of SGLD will be negligible so that MH rejection probability will approach 0 and we may simply ignore

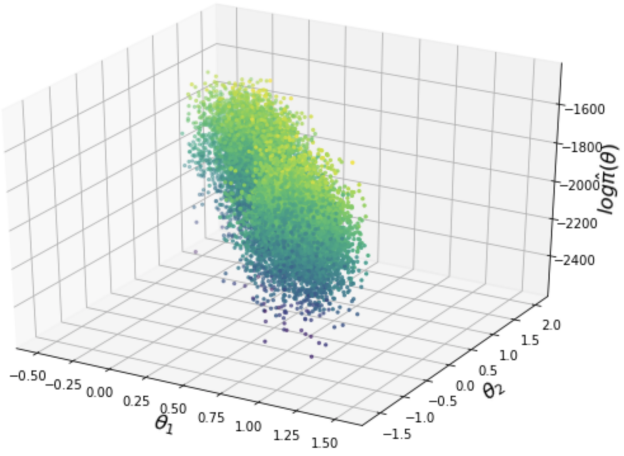

Figure 2: Posterior samples w.r.t log posterior

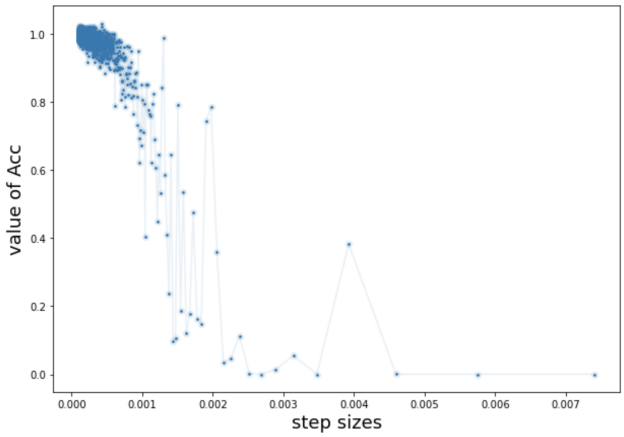

Figure 3: Change of Acc w.r.t iterations

this step". The Acceptance rate (Acc) at each iteration was calculated and shown in Figure 3. To be more specific, from the transition distribution we can have:

$$\begin{aligned}\theta_{t+1}|\theta_t &\sim \mathcal{N}(\theta_t + \frac{\epsilon_t}{2}\nabla[logp(\theta_t) + \frac{N}{n}\sum_{i=1}^{n} logf(x_{ti}|\theta_t)],\ \epsilon_t \cdot I_d) \\ \theta_t|\theta_{t+1} &\sim \mathcal{N}(\theta_{t+1} + \frac{\epsilon_t}{2}\nabla[logp(\theta_{t+1}) + \frac{N}{n}\sum_{i=1}^{n} logf(x_{ti}|\theta_{t+1})],\ \epsilon_t \cdot I_d)\end{aligned} \tag{1}$$

Then we can calculate the Acc by the following formula:

$$Acc = \frac{\pi(\theta_{t+1})q(\theta_t|\theta_{t+1})}{\pi(\theta_t)q(\theta_{t+1}|\theta_t)} \tag{2}$$

From Figure3, we can see that with the decrease of stepsize, the Acc increase to about 1 exponentially fast, which supports the main idea that we can ignore MH accept-reject test in SGLD.

To further evaluate the sampling performance. The fixed stepsize SGLD algorithm was performed to sample from the posterior distribution. The stepsize was fixed at $\epsilon = 0.01$. And 2000 samples were produced by the fixed stepsize SGLD algorithm. The evaluation plots were shown in Figure 4.

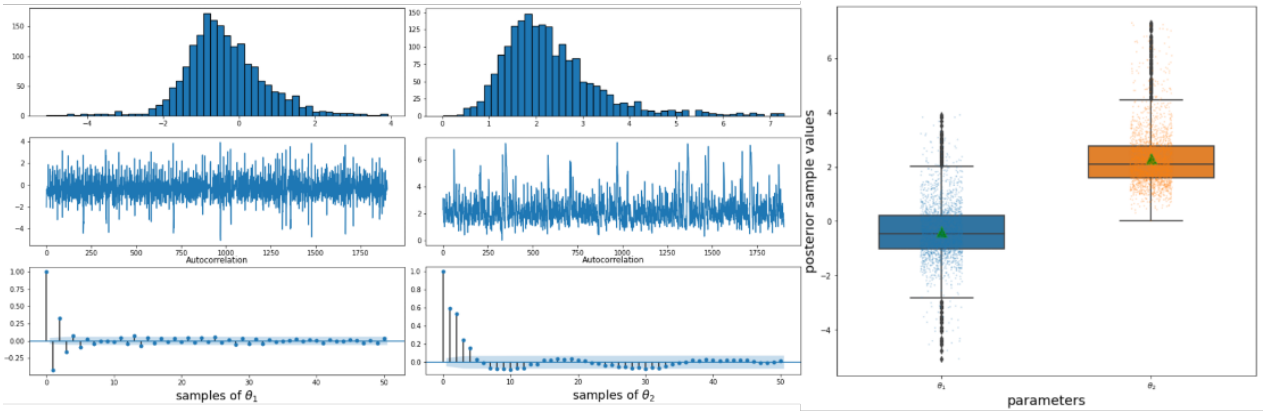

Figure 4: (left two):Histogram and ACF plots of fixed stepsize SGLD samples; (right):Box plot of fixed stepsize SGLD samples

From Figure 4, we can see that, for both of the parameters, the samples are roughly normally distributed, and the samples plotted behave like random noise. The ACF plots also show that the samples have almost no autocorrelation after 5 lags. Moreover, the interquartile range(IQR) of box plots can be seen as the 50% posterior interval in Bayesian setting. We can see that the IQR of θ_1 perfectly

cover the true parameter value 0. Although it is not the case in IQR of θ_2, the true value of θ_2, after taking 95% quantiles of the posterior samples, is roughly covered in the posterior interval. Overall, we can see that the fixed stepsize SGLD has decent performance in posterior sampling.

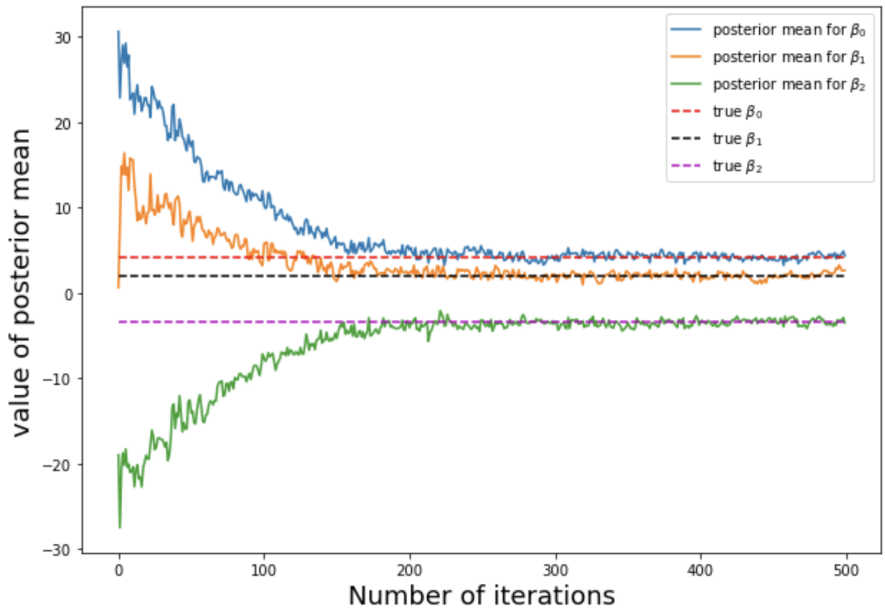

Figure 5: Posterior means w.r.t iterations

3.2 Bayesian Logistic Regression and classification

In this section, the SGLD algorithm was performed in logistic regression to do posterior distribution sampling and classification. Experiments were done in self-generated data and NBA data, separately. And the data were spilt into training & testing data beforehand(note: training v.s testing = 4:1). And the summary of the model is shown below.
Hyperparameters:

$$\gamma = -0.55, a = 0.03, b = 6, batch\ size = 30$$

Prior:

$$\beta_i \sim Laplace(0, 1)$$

Data generating process:

$$f(y_i|X_i) = \sigma(X_i\beta), where\ \sigma(z) = \frac{1}{1+e^{-z}},\ X_i^\top \sim \mathcal{N}(\begin{pmatrix}1 & 0 & 0\end{pmatrix}^\top, diag(0,1,1)),\ \beta = [4, 2, -3.5]^\top$$

Log of prior:

$$logp(\beta) = -log2 - |\beta|$$

Log likelihood calculated from batch-size of data:

$$logL(\theta|X_1, .., X_n) = \frac{N}{n}\sum_{i=1}^{n}(y_i X_i\beta - log(1 + e^{X_i\beta}))$$

Gradient of log posterior:

$$\nabla log\pi(\theta) = -sign(\beta^\top) + \frac{N}{n}\{y\top X - (\frac{e^{X\beta}}{1+e^{X\beta}})^\top X\}$$

Note: X_i is the ith row of matrix X and $X_i \in \mathbb{R}^{1\times(p+1)}$, $\beta \in \mathbb{R}^{(p+1)\times 1}, \nabla log\pi(\theta) \in \mathbb{R}^{1\times(p+1)}, y \in \mathbb{R}^{n\times 1}$

The first experiment is on self-generated data with 12500 entries and 3 parameters (including intercept). The true value of parameters are: $\beta_0 = 4, \beta_1 = 2, \beta_2 = -3.5$. The posterior mean after each iteration was shown in Figure 5. We can see that may because of the simplicity of the true model, the SGLD algorithm changes to posterior sampling phase very fast even after 200 iterations. And the posterior means after 200 iterations are almost the same as the true parameters.

The second experiment is on the real life NBA dataset—the 5-Year Career Longevity for NBA Rookies data, which includes 1340 entries and 19 features like Games Played, Points Per Game, 3 Point Attempts. The response variable is whether the career of the rookies is greater or equal to five years.

The classification experiments were done in both of the self-generated data and NBA data. Basically, after each iteration, the sample from SGLD was plugged back into logistic regression model to do prediction on both the training and testing data. And the model performance were evaluated by cross-entropy loss, training accuracy and testing accuracy. The summary of this two classification experiments were shown in Figure 6 & 7. Note that to show them in the same plot, the cross entropy loss were added fixed value or scaled. And the loss decrease exponentially as the iteration goes. We can also see that for simple self-generated data, the model has very good classification result as the test accuracy are above 0.9 in later iterations. And for real life NBA data, the test accuracy are not bad which are about 0.7 or higher. And it is reasonable to see that test accuracy has more fluctuation than training accuracy.

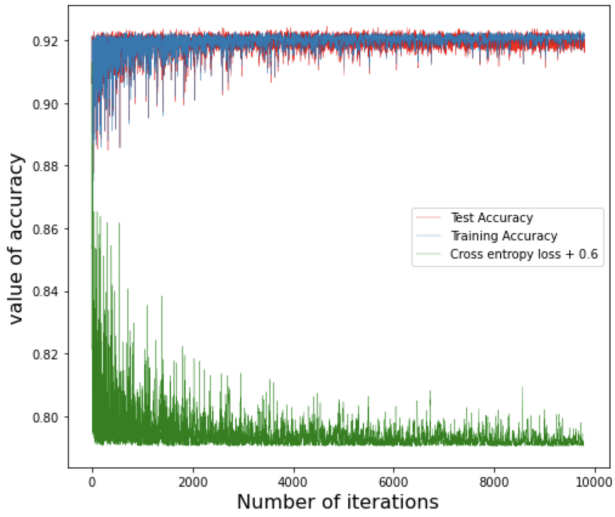

Figure 6: Classification in self-generated data

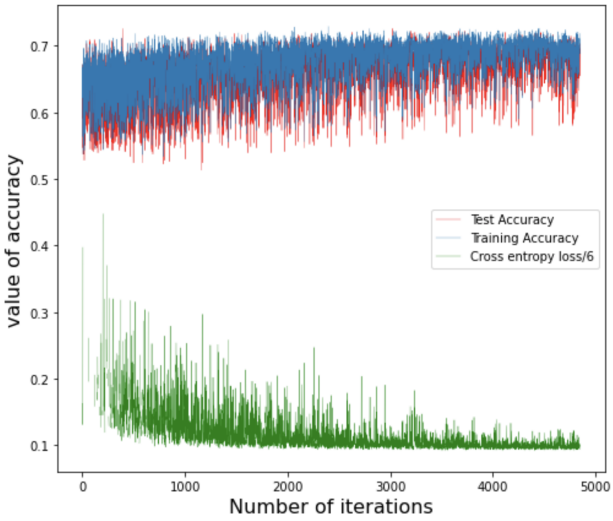

Figure 7: Classification in NBA data

4 Discussion

From what has been discussed above, we can see that SGLD has good performance in posterior sampling. And the fixed stepsize SGLD might be more useful in practice as we will have a stationary Markov Chain and thus we can test the sampling results by traditional ways like ACF plots, posterior intervals and box plots.

For SGLD, since the mixing rate of Markov Chain decrease as the stepsizes decrease. So the simple sample average will overemphasize the tail end of the sequence and the correlation among samples will be high, resulting in higher variance in the estimator.[1] The author also proposed a way to weight the samples by stepsizes: $E[\theta] \approx \frac{\sum_{t=1}^{T} \epsilon_t \theta_t}{\sum_{t=1}^{T} \epsilon_t}$. However, from Figure 5 we may can say that there is no need to do this if the model is simple enough. Since the SGLD will change to the phase sampling from posterior very fast and there are not too much fluctuations in the samples or posterior means.

Moreover, one thing worth mentioning is that we can actually estimate the sampling threshold, α, and relate it to the stepsize. Basically if we choose a stepsize which makes the $\alpha < 1$, the SGLD algorithm will be in the langevin dynamics phase and will be sampling from the posterior distribution.

Note that it requires to generalized the method to allow for preconditioning on the injected noise, but it is only important when we have to faithfully represent the posterior distribution with finite number of samples. [1]

In the end, if time allows the explorations above of the SGLD method can be extended like applying in high dimensional data, e.g., data with thousands of parameters. Note that the models as well as datasets in the above experiments are simple and relatively small. So we have to admit that the merit of SGLD algorithm was not fully shown in this report. Since recall that the main motivation for deriving SGLD method is to deal with huge dataset which is hard for traditional MCMC algorithms to handle with. And one thing which is very important in SGLD is the tradeoff between mixing and computational speed, which also needs further exploration. The SGLD method has been improved in different aspects these years like preconditioned SGLD, variance reduction in SGLD, etc. So exploring those methods and comparing with the original SGLD method is also a meaningful work to do.

References

[1] Welling, M., and Teh, Y. W. 2011. Bayesian learning via stochastic gradient Langevin dynamics. In ICML.

Code in Python:

```
""import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
#%matplotlib notebook
from scipy.stats import t
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits import mplot3d
from numpy.linalg import inv
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
import matplotlib as mpl
import seaborn as sns
import pandas as pd

# Presetting for data generation
N = 1000 # The number of data points
sigma1_sq = 10.0
sigma2_sq = 1.0
sigmax_sq = 2.0
theta1 = 0.0
theta2 = 1.0
cov_matrix = np.array([[sigma1_sq,0],[0,sigma2_sq]])

# Generate the data X.
X = np.zeros(N)
for i in range(N):
    u = np.random.random()
    if (u < 0.5):
        X[i] = np.random.normal(theta1, np.sqrt(sigmax_sq))  #the np.random.normal(...)
    requires STD (not VAR).
    else:
        X[i] = np.random.normal(theta1+theta2, np.sqrt(sigmax_sq))
def log_f(theta, X):
    """
    The function 'f' is the posterior:

        f(\theta) \propto p(\theta) * \prod_{i=1}^N p(x_i | \theta)

    This method returns the *log* of f(\theta), for obvious reasons.
    X is assumed to be of shape (n,) so we reshape it into (n,1) for vectorization.
    We use [0,0] since we get a float in two lists, [[x]], so calling [0,0] gets x.
    """
    scale = N / float(len(X))
    #prior
    cov_inverse = inv(cov_matrix)
    prior = -np.log(2*np.pi*np.sqrt(sigma1_sq)*np.sqrt(sigma2_sq)) - 0.5*(theta.T).dot(
    cov_inverse).dot(theta)
    #log-likelihood
    X_all = X.reshape((len(X),1)) #reshape the X to be vector
    ll_constant = (1.0 / (4*np.sqrt(np.pi)))
    L = ll_constant * (np.exp(-0.25*(X_all-theta[0])**2) + np.exp(-0.25*(X_all-(theta[0]+theta
    [1]))**2))
    log_likelihood = np.sum(np.log(L)) * scale

    assert not np.isnan(prior + log_likelihood)    #np.isnan Test element-wise for NaN and
    return result as a boolean array.
    return (prior + log_likelihood)[0,0]

def grad_f(theta, X):
    """ Computes gradient of log_f by finite differences. X is (usually) a mini-batch. """
    h = 0.00001
    base1 = np.array([[h],[0]])
    base2 = np.array([[0],[h]])
    term1 = log_f(theta+base1, X) - log_f(theta-base1, X)
    term2 = log_f(theta+base2, X) - log_f(theta-base2, X)
    return np.array([[term1],[term2]])/(2*h)

def get_noise(eps):
    """ Returns a 2-D multivariate normal vector with covariance matrix = diag(eps,eps). """
    return (np.random.multivariate_normal(np.array([0,0]), eps*np.eye(2))).reshape((2,1))

def get_ACC(the1, the2, X_mini):
    """
    This function return the ACC after each iteration
```

```
    """
    log_post_1= log_f(the1, X_mini)
    log_post_2= log_f(the2, X_mini)
    ACC_inner1_2 = log_post_2 - log_post_1
    ACC_inner3_up = (the1-the2-eps/2*grad_f(the2, X_mini)).reshape((1, 2))
    ACC_inner3 = -1/(2*eps)*np.inner(ACC_inner3_up, ACC_inner3_up)
    ACC_inner4_up = (the2-the1-eps/2*grad_f(the1, X_mini)).reshape((1, 2))
    ACC_inner4 = 1/(2*eps)*np.inner(ACC_inner4_up, ACC_inner4_up)
    return np.exp(ACC_inner1_2 + ACC_inner3 + ACC_inner4)
##########################################
# Stochastic Gradient Langevin Dynamics
##########################################
num_passes = 30000
theta = np.array([[0.5],[0]])
Res = theta
batch_size = 30

# a and b are based on parameter values similar to those from the SGLD paper.
a = 0.03
b = 6
log_post_Res = np.zeros(num_passes)
ACC = np.zeros(num_passes)
step_size = np.zeros(num_passes)

for T in range(1,num_passes):

    # Step size according to the SGLD paper.
    eps = a*((b + T) ** (-0.55))
    step_size[T] = eps
    # Get a minibatch, then compute the gradient, then the corresponding \theta updates.
    X_mini = X[np.random.choice(N, batch_size, replace=False)]
    gradient = grad_f(theta, X_mini)

    the1 = theta
    the2= the1 + (eps/2.0)*gradient + get_noise(0.001)  #use gradient to update the theta
    log_post_Res[T] = log_f(the2, X_mini)
    ACC[T] = get_ACC(the1, the2, X_mini)
    # Add theta to Res and repeat.
    assert not np.isnan(np.sum(the2))
    Res = np.concatenate((Res,the2), axis=1)
    theta = the2
######Plot of Log Posterior & Samples
mymap2 = plt.get_cmap("Greys")
m_c2 = mymap2(400)
fig, axarr = plt.subplots(1,4, figsize=(16, 5))
(xlim1,xlim2) = (-1.5,2.5)
(ylim1,ylim2) = (-4,4)

# A contour plot of what the posterior really looks like.
axarr[0].set_title("Log Posterior Contour", size="xx-large")
K = 200
xlist = np.linspace(-1.5,2.5,num=K)
ylist = np.linspace(-4,4,num=K)
X_a,Y_a = np.meshgrid(xlist, ylist)
Z_a = np.zeros((K,K))
for i in range(K):
    for j in range(K):
        theta = np.array( [[X_a[i,j]],[Y_a[i,j]]] )
        Z_a[i,j] = log_f(theta, X)
axarr[0].contour(X_a,Y_a,Z_a,300)

# Stochastic Gradient Langevin Dynamics
axarr[1].set_title("SGLD $(iteration=5000)$", size="xx-large")
axarr[1].scatter(Res[0][:5000], Res[1][:5000], color = m_c2, alpha=0.15, s = 1)
axarr[1].set_xlim([xlim1,xlim2])
axarr[1].set_ylim([ylim1,ylim2])
# Stochastic Gradient Langevin Dynamics
axarr[2].set_title("SGLD $(iteration=15000)$", size="xx-large")
axarr[2].scatter(Res[0][:15000], Res[1][:15000], color = m_c2, alpha=0.15, s = 1)
axarr[2].set_xlim([xlim1,xlim2])
axarr[2].set_ylim([ylim1,ylim2])
# Stochastic Gradient Langevin Dynamics
axarr[3].set_title("SGLD $(iteration=30000)$", size="xx-large")
axarr[3].scatter(Res[0][1000:30000], Res[1][1000:30000], color = m_c2, alpha=0.15, s = 1)
axarr[3].set_xlim([xlim1,xlim2])
axarr[3].set_ylim([ylim1,ylim2])
```

```
148
149 start_N = 15000 # for making plot after N iterations
150 fig = plt.figure(figsize = (10, 7))
151 ax = plt.axes(projection='3d')
152 ax.scatter(Res[0][start_N:], Res[1][start_N:], log_post_Res[start_N:], c=log_post_Res[start_N
    :], s =5)
153 ax.set_xlabel('$ _1$', fontsize=16)
154 ax.set_ylabel('$ _2$', fontsize=16)
155 ax.set_zlabel('$log \hat{\pi}({  })$', fontsize=16)
156 plt.title("Log Posterior Samples visulization-3D", fontsize=16)
157 plt.show()
158 ACC_mean_100 = np.mean(np.array(ACC[0:]).reshape(-1, 10), axis=1)
159 step_size_mean_100 = np.mean(np.array(step_size[0:]).reshape(-1, 10), axis=1)
160 plt.figure(figsize = (10, 7))
161 plt.scatter(step_size_mean_100,1/ACC_mean_100, s = 3)
162 plt.plot(step_size_mean_100, 1/ACC_mean_100, marker='o', alpha=0.1)
163 #plt.title("Change of ACC w.r.t step size", size="xx-large")
164 #plt.ylim(0, 2)
165 plt.xlabel("step sizes", size=18)
166 plt.ylabel("value of Acc", size=18)
167 plt.show()
168
169 num_iterations = 2000
170 theta = Res[:,num_passes-1000].reshape(2,1)
171 #theta = np.array([[0.5],[0]])
172 Res_fixed_step = theta
173 batch_size = 30
174 a = 0.03
175 b = 6
176
177 for T in range(1,num_iterations):
178     eps = 0.01
179     # Get a minibatch, then compute the gradient, then the corresponding \theta updates.
180     X_mini = X[np.random.choice(N, batch_size, replace=False)]
181     gradient = grad_f(theta, X_mini)
182     theta = theta + (eps/2.0)*gradient + get_noise(eps)  #use gradient to update the theta
183     # Add theta to Res and repeat.
184     assert not np.isnan(np.sum(theta))
185     Res_fixed_step = np.concatenate((Res_fixed_step,theta), axis=1)
186
187 #Box Plot
188 ##### Get data for plot
189 df = pd.DataFrame({
190     '$ _1$': Res_fixed_step[0],
191     '$ _2$': Res_fixed_step[1],
192 })
193 #print(df.head())
194
195 data_df = df.melt(var_name='parameters',value_name='values')
196 #print(data_df.head())
197
198 ##### Have Box plot
199 plt.figure(figsize = (10, 9))
200 sns.boxplot(x="parameters", y="values", data=data_df, linewidth=1.7, showmeans=True, meanprops
    ={#"marker":"o",
201                       "markerfacecolor":"green",
202                       #"markeredgecolor":"black",
203                      "markersize":"15"}, width=0.6)
204 sns.stripplot(x="parameters", y="values",
205               size=2, alpha=0.3, data=data_df)
206 plt.xlabel("parameters", size=18)
207 plt.ylabel("posterior sample values", size=18)
208 plt.show()
209
210 ########## histogram, ACF plots
211 fig, (ax1, ax2, ax3) = plt.subplots(3, figsize=(10, 10))
212 fig.suptitle('Vertically stacked subplots')
213 ax1.hist(Res_fixed_step[0][100:], edgecolor='black', bins = 50, linewidth=1.2)
214 #ax2.plot(x, -y)
215 ax2.plot(Res_fixed_step[0][100:])
216 plot_acf(Res_fixed_step[0][100:], ax=ax3, lags=50)
217
218 #plt.figure(figsize = (10, 7))
219 #plt.hist(Res_fixed_step[0][:], edgecolor='black', bins = 50, linewidth=1.2)
220 plt.xlabel("samples from $ _1$", size=18)
221 plt.show()
```

```
222
223 fig, (ax1, ax2, ax3) = plt.subplots(3, figsize=(10, 10))
224 fig.suptitle('Vertically stacked subplots')
225 ax1.hist(Res_fixed_step[1][100:], edgecolor='black', bins = 50, linewidth=1.2)
226 #ax2.plot(x, -y)
227 ax2.plot(Res_fixed_step[1][100:])
228 plot_acf(Res_fixed_step[1][100:], ax=ax3, lags=50)
229
230 #plt.figure(figsize = (10, 7))
231 #plt.hist(Res_fixed_step[0][:], edgecolor='black', bins = 50, linewidth=1.2)
232 plt.xlabel("samples from $ _2$", size=18)
233 plt.show()
234
235 #EXPERIMENT 2:
236 import numpy as np
237 import matplotlib.pyplot as plt
238 %matplotlib inline
239 #%matplotlib notebook
240 from scipy.stats import t
241 from scipy.stats import multivariate_normal
242 from mpl_toolkits.mplot3d import Axes3D
243 from mpl_toolkits import mplot3d
244 from numpy.linalg import inv
245 from statsmodels.graphics.tsaplots import plot_acf
246 from statsmodels.graphics.tsaplots import plot_pacf
247 import matplotlib as mpl
248
249 import pandas as pd
250 from sklearn.model_selection import train_test_split
251 from sklearn.preprocessing import LabelEncoder
252
253 # get training & testing data
254 p = 2 #number of parameters except for intercept
255 num_classes = 1
256 num_features = 2
257 num_examples = 12500  # The number of data points
258 N = num_examples
259 #data generating & processing
260 def logit_data(w, b, num_examples):
261     """Generate y = Xw + b."""
262     X = np.zeros((num_examples, w.shape[0]))
263     X += np.random.normal(0,1, size = X.shape)
264     y = np.matmul(X, np.reshape(w, (-1,1))) + b
265     y = np.reshape(y, (-1, 1))
266     return X, y
267
268 true_w = np.array([2, -3.4])
269 true_b = 4.2
270 X_in, y_in = logit_data(true_w, true_b, num_examples)
271 P = 1/(1+np.exp(-y_in))
272 y = np.random.binomial(n=1, p=P)
273 beta_0 = np.repeat(1, num_examples).reshape(num_examples,1)
274 X = np.concatenate((beta_0, X_in), axis = 1)
275 ###separate into training and testing(4:1)
276 X_train, X_test, y_train, y_test = train_test_split(
277     X, y, test_size=0.2, random_state=77)
278
279 def gradient_log_pos(theta, X, y):
280     """
281     The function 'lil' is the likelihood to logistic regression
282     """
283     scale = N / float(X.shape[0])
284     #prior_gradient                    #log_prior= -np.log(2) - np.abs(theta)
285     prior_gradient = -np.sign(theta)  #dim = 1x2
286
287     #log-likelihood_gradient
288     beta = theta.reshape(p+1,1)
289     exp_com = np.matmul(X, beta)
290     #sigmoid = np.exp(exp_com)/(1+np.exp(exp_com))
291     sigmoid = 1/(1 + np.exp(-exp_com))
292     log_li_gradient = (np.matmul(y.T, X) - np.matmul(sigmoid.T, X))*scale
293     log_li_gradient = log_li_gradient.reshape(1,-1)
294     gradient = prior_gradient + log_li_gradient
295     return gradient
296
297 def get_noise(eps):
```

```
    """ Returns a 2-D multivariate normal vector with covariance matrix = diag(eps,eps). """
    noise = np.random.multivariate_normal(np.repeat(0, p+1), eps*np.eye(p+1))
    noise = noise.reshape(1,-1)
    return noise

def adjusted_pos_mean(Res, p, start_N):
    post_mean = np.zeros(p+1)
    """
    The function returns the posterior mean adjuested by the stepsize, as well as the starting iteration point
    """
    for i in range(0,p+1):
        post_multi = Res[:,i]*step_size
        post_multi = post_multi[:start_N]
        post_mean[i] = sum(post_multi)/sum(step_size[:start_N])

    return post_mean.reshape(1,p+1)

def adjusted_pos_mean_step(Res, p, start_N, iter_start):
    post_mean = np.zeros(p+1)
    """
    The function returns the posterior mean adjuested by the stepsize, as well as the starting iteration point
    """
    for i in range(0,p+1):
        post_multi = (Res[iter_start:])[:,i]*step_size[iter_start:]
        post_multi = post_multi[:start_N]
        post_mean[i] = sum(post_multi)/sum(step_size[iter_start:][:start_N])

    return post_mean.reshape(1,p+1)

def sigmoid(X):
    """
    Sigmoid function in numpy
    """
    return 1/(1+np.exp(-X))

def logistic(X, beta, T):
    """
    Use logistic function to get y_hat, dimention of y_hat: nx1
    """
    y_hat = sigmoid(np.matmul(X, beta[T]))
    return y_hat.reshape(-1,1)

def cross_entropy_loss(X, y, beta, T):
    """
    Use y and y_hat to get the loss
    """
    y_hat = logistic(X, beta, T)
    a = np.matmul(y.reshape(1, -1), np.log(y_hat))
    b = np.matmul((1-y).T, np.log(1-y_hat))
    loss = -1/len(y)*(a+b)
    return loss

def prediction_accuracy(X ,y, beta, T):
    """
    Use y and y_hat to get the prediction accuracy
    """
    y_hat = logistic(X, beta, T)
    n = len(y)
    Y_prediction = np.zeros(n)
    for i in range(n):
        if y_hat[i] > 0.5:
            Y_prediction[i] = 1
        else:
            Y_prediction[i] = 0
    Y_pred = Y_prediction.reshape(-1,1)
    wrong = np.abs(y-Y_pred)
    prob_wrong = np.sum(wrong)/n
    accuracy = 1- prob_wrong
    return accuracy

##########################################
# STOCHASTIC GRADIENT LANGEVIN DYNAMICS #
##########################################
num_passes = 10000
```

```
theta = np.array([1,1,1]).reshape(1,3)
Res = theta
batch_size = 30
step_size = np.zeros(num_passes)
# a and b are choose to tune the step size decrease
a = 0.03
b = 6

for T in range(1,num_passes):

    # Step size according to the SGLD paper.
    eps = a*((b + T) ** (-0.55))
    step_size[T] = eps
    # Get a minibatch, then compute the gradient, then the corresponding \theta updates.
    batch_data_num = np.random.choice(len(X_train), batch_size, replace=False)
    X_mini = X_train[batch_data_num]
    y_mini = y_train[batch_data_num]
    gradient = gradient_log_pos(theta, X_mini, y_mini)

    theta = theta + (eps/2.0)*gradient + get_noise(0.0001)  #use gradient to update the theta
    # Add theta to Res and repeat.
    assert not np.isnan(np.sum(theta))
    Res = np.concatenate((Res,theta), axis=0)

############
############
#Get the posterior mean
post_mean_all = np.array([1,1,1]).reshape(1,3)
for S in range(1,num_passes):    #S is the number of iteration
    post_mean_adjust = adjusted_pos_mean_step(Res, p, S, 2)
    post_mean_all = np.concatenate((post_mean_all,post_mean_adjust), axis=0)

plt.figure(figsize = (10, 7))
Res_truncated = Res[2:]
plt.plot(Res_truncated[:500,0], label=("posterior mean for $ _0$" ))
plt.plot(Res_truncated[:500,1], label=("posterior mean for $ _1$" ))
plt.plot(Res_truncated[:500,2], label=("posterior mean for $ _2$" ))
plt.plot(np.repeat(4.2, 500), '--r', label = "true $ _0$")
plt.plot(np.repeat(2, 500), '--k', label = "true $ _1$")
plt.plot(np.repeat(-3.4, 500), '--m', label = "true $ _2$")
plt.xlabel('Number of iterations', fontsize=18)
plt.ylabel('value of posterior mean', fontsize=18)
#plt.title('Posterior of parameters', fontsize=20)
plt.legend()
plt.show()

plt.figure(figsize = (10, 7))
Res_truncated = Res[2:]
plt.plot(post_mean_all[:,0], label=("adjusted pmean for $ _0$" ))
plt.plot(post_mean_all[:,1], label=("adjusted pmean for $ _1$" ))
plt.plot(post_mean_all[:,2], label=("adjusted pmean for $ _2$" ))
plt.plot(np.repeat(4.2, len(Res_truncated)), '--r', label = "true $ _0$")
plt.plot(np.repeat(2, len(Res_truncated)), '--k', label = "true $ _1$")
plt.plot(np.repeat(-3.4, len(Res_truncated)), '--m', label = "true $ _2$")
plt.xlabel('Number of iterations', fontsize=16)
plt.ylabel('value of adjusted posterior mean', fontsize=16)
plt.title('Adjuested posterior mean of parameters', fontsize=20)
plt.legend()
plt.show()


#Prediction
##### get loss & Training Accuracy, after iteration 100
Loss = np.zeros(num_passes)
Train_ACC = np.zeros(num_passes)
Test_ACC = np.zeros(num_passes)
for T in range(100,num_passes):
    Loss[T] = cross_entropy_loss(X_train, y_train, Res, T)
    Train_ACC[T] = prediction_accuracy(X_train, y_train, Res, T)
    Test_ACC[T] = prediction_accuracy(X_test, y_test, Res, T)

#Experiment 3
# NBA data preparation
num_classes = 1
num_features = 19
data_size = 1340
```

```
#train_size = tf.cast(0.8*data_size, tf.int32)
nba = pd.read_csv("nba_logreg.csv") #shape: 1340x21
#fill in NA value with meanx
nba = nba.fillna(nba.mean())
#####training data
X, y = nba.values[:, 1:-1], nba.values[:, -1]
all_features = nba.iloc[:, 1:-1]
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
nba = all_features
X = nba.values
#add intercept to the data
beta_0 = np.repeat(1, len(X)).reshape(-1,1)
X = np.concatenate((beta_0, X), axis = 1)
y = (y.astype('int')).reshape(-1,1) # get rid of the dype in the y dataframe
###separate into training and testing(4:1)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=77)

########################################
# STOCHASTIC GRADIENT LANGEVIN DYNAMICS #
########################################
num_passes = 10000
N = num_passes
p = X_train.shape[1]-1

theta = np.repeat(1,X_train.shape[1]).reshape(1,-1)
Res = theta
batch_size = 30
# a and b are choose to tune the step size decrease
a = 0.03
b = 6

for T in range(1,num_passes):
    # Step size according to the SGLD paper.
    eps = a*((b + T) ** (-0.55))
    # Get a minibatch, then compute the gradient, then the corresponding \theta updates.
    batch_data_num = np.random.choice(len(X_train), batch_size, replace=False)
    X_mini = X_train[batch_data_num]
    y_mini = y_train[batch_data_num]
    gradient = gradient_log_pos(theta, X_mini, y_mini)

    theta = theta + (eps/2.0)*gradient + get_noise(0.0001)  #use gradient to update the theta
    # Add theta to Res and repeat.
    assert not np.isnan(np.sum(theta))
    Res = np.concatenate((Res,theta), axis=0)

#Loss and Prediction accuracy
##### get loss & Training Accuracy, after iteration 100
Loss = np.zeros(num_passes)
Train_ACC = np.zeros(num_passes)
Test_ACC = np.zeros(num_passes)
for T in range(100,num_passes):
    Loss[T] = cross_entropy_loss(X_train, y_train, Res, T)
    Train_ACC[T] = prediction_accuracy(X_train, y_train, Res, T)
    Test_ACC[T] = prediction_accuracy(X_test, y_test, Res, T)

Loss_mean_100 = np.mean(np.array(Loss[100:]).reshape(-1, 2), axis=1)
Train_ACC_mean_100 = np.mean(np.array(Train_ACC[100:]).reshape(-1, 2), axis=1)
Test_ACC_mean_100 = np.mean(np.array(Test_ACC[100:]).reshape(-1, 2), axis=1)
plt.figure(figsize = (8, 7))
plt.plot(Test_ACC_mean_100[100:], label=("Test Accuracy"), color='red', linewidth=0.3)
plt.plot(Train_ACC_mean_100[100:], label=("Training Accuracy"), linewidth=0.3)   # loss can
    change to average of each 100 iterations

plt.plot(Loss_mean_100[100:]/6, label=("Cross entropy loss/6"), color='green', linewidth=0.3)

plt.xlabel('Number of iterations', fontsize=16)
plt.ylabel('value of accuracy', fontsize=16)
#plt.title('Prediction accuracy & Loss after each iteration', fontsize=20)
plt.legend(fontsize=10)
plt.show()
```