Markov Chains and Algorithmic Applications

COM-516 - EPFL

Course Mini-Project

MCMC Method for Generalized Linear Estimation and Simulated Annealing

Sadra Boreiri, Alexandre Carlier and Alireza Modirshanechi

December, 2018

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

1 Introduction

The goal of this project is to apply the Markov Chain Monte-Carlo (MCMC) method to a generalized linear estimation problem. Given a known matrix W of dimension $m \times n$, we would like to recover the vector $X = (X_1, \dots, X_n)^T \in S = \{-1, 1\}^n$ from the vector of observations $Y = \frac{1}{n}\text{ReLU}(WX)$.

Since every entry in X can take only two values, this is a discrete optimization problem and solving it by a brute force approach becomes computationally infeasible as soon as n becomes large. We follow the proposed method and use the MCMC method to construct a Markov chain that samples from the Gibbs-Boltzmann distribution $p_Y(x) = \frac{e^{-\beta H_Y(x)}}{Z_\beta}, x \in S$. When β is sufficiently large, the vectors x that minimize the energy function correspond to peaks in this distribution. We hope that such vectors x will be close to the ground-truth vector X.

In section 2, we will explain our implementation choices regarding its design and optimization, and propose three cooling strategies for simulated annealing. Then, we will compare and discuss our obtained results using different approaches in section 3.

2 Methods

2.1 Implementation Details

Our implementation is written in Python using the library PyTorch so that the same program can be executed both on CPU and GPU by adding just a few lines of code.

In order to take advantage of the great parallelism capabilities of a GPU and benefit from a large speed-up in execution time, we decide to *parallelize* our algorithm (hence the name of our team *Game of Threads*!). In the context of the MCMC method, parallelizing the algorithm basically means that we perform several independent runs in parallel using different initial conditions. To do so, we replace the vector x by a matrix of shape $n \times$ `batch_size` and scalars such as the energy $H_Y(x)$ or the acceptance probability $a_\beta(x^{(t)}, x^{(t+1)})$ by vectors of size `batch_size`.

We implement both the Metropolis-Hastings and Glauber algorithms and compare them in section 3.1 in order to choose the best one for the competition.

Finally, we notice that we can optimize the computation of the energy $H_Y(x^{(t)})$. Instead of computing a matrix-vector multiplication at each step of the algorithm, we can deduce the value of $Wx^{(t)}$ using the result of $Wx^{(t-1)}$. E.g. in the Metropolis-Hastings algorithm, if i is the coordinate of $x^{(t)}$ that has been flipped, then:

$$Wx^{(t)} = Wx^{(t-1)} - 2W_{*i}x_i^{(t-1)}$$

where W_{*i} represents the i^{th} column of W.

We compare the execution times using the standard one run at a time implementation and batchwise computations (on both CPU and GPU) in table 1 (see the notebook **ExecutionTime.ipynb**). We observe that the batch implementation adds a significant computational overhead to the one

	1 run	100 runs
One run at a time	**1.08 s ± 2.93 ms**	1min 48s ± 303 ms
Batchwise (CPU)	2.09 s ± 7.72 ms	43.5 s ± 244 ms
Batchwise (GPU)	6.72 s ± 16.5 ms	**9.91 s ± 6.15 ms**

Table 1: Execution time comparison of the Metropolis algorithm with $t_{\max} = 10,000$, $N = 1000$ and $M = 5000$ using the standard one run at a time implementation and batchwise computations. Using one run at a time (`batch_size=None` in the code), the execution time of the second column (100 runs) is obtained with a for loop of 100 iterations, while batchwise computations can achieve the same result using one iteration with `batch_size=100`. Results are obtained using an Intel Core i5 CPU and a Geforce GTX 1080 GPU. The best execution time of each column is shown in bold.

run at a time method, especially when executing the program on GPU (this may be explained by the fact that we first need to transfer the parameters from CPU to GPU). However, this overhead is largely compensated by the speed-up gained using parallel computation when the number of runs becomes large (e.g. we observe a 10x speed-up over 100 runs using batchwise computation on GPU compared to the standard method on CPU).

2.2 Reconstruction Error

Given the $m \times n$ matrix W and the vector of m observations Y, the aim of the project is to reconstruct the unknown ground-truth vector X. The reconstruction error is defined as follows:

$$e(\hat{x}, X) = \frac{1}{4n}\|\hat{x} - X\|^2$$

We can easily see that $e(\hat{x}, X)$ corresponds to the fraction of entries on which $\hat{x}$ differs from X. Indeed:

$$\|\hat{x} - X\|^2 = \sum_{i=1}^{n}(\hat{x_i} - X_i)^2 = \sum_{i:\hat{x_i}=X_i} \underbrace{(\hat{x_i} - X_i)^2}_{=0} + \sum_{i:\hat{x_i}\neq X_i} \underbrace{(\hat{x_i} - X_i)^2}_{=4} = 4 \cdot \#\{i : \hat{x_i} \neq X_i\}$$

which comes from the fact that since $S = \{-1, 1\}^n, \hat{x_i} \neq X_i \iff (\hat{x_i} = 1 \text{ and } X_i = -1)$ or $(\hat{x_i} = -1 \text{ and } X_i = 1)$.

2.3 Cooling Strategies

For the cooling strategy, we considered three different approaches which are briefly described in the following. These strategies are written as three different classes in the file **scheduler.py**.

2.3.1 Fixed Temperature Strategy

As a matter of fact, this method can not be considered as a "cooling" strategy! It considers a fixed temperature, and as a result fixed β, for the whole process. The only free parameter of this method is its initial temperature.

We consider this approach as the baseline strategy, and in our first few experiments, we observed that it is by far outperformed by the other strategies. Therefore, we do not report the results corresponding to this method in the following parts.

2.3.2 Fixed Step Size Strategy

By this strategy, after every N_S number of steps, the temperature of the system is decreased by a factor γ^{-1}. Both N_S and γ are fixed for whole process. Therefore, considering the initial temperature, this strategy has 3 free parameters.

2.3.3 Adaptive Step Size Strategy

By this strategy, which is conceptually similar to the previous one, after every N_S number of steps, a specific condition about the process is checked. Then, based on the result, the temperature of the system is decreased by a factor γ^{-1}. By considering this conditioning procedure, we want to achieve two goals at the same time:

1. Preventing the chain to be stuck in a local minimum, which can happen by a rapid decrease in temperature.

2. Having a fast rate of convergence, which can be achieved only by a rapid decrease in temperature.

To have a balance in this trade-off, we decided to decrease the temperature of the system if and only if there is not "enough progress" in the process of minimization. Therefore, after every N_S number of steps, a window with the length of N_W preceding the current time is considered. Then, the average decrease of energy function in this window is compared to the standard deviation of energy function in the same window. If the ratio of the average progress to the standard deviation is less than a constant η, the temperature of the system is decreased by a factor γ^{-1}. Mathematically, it can be written as below:

$$\forall n \in \mathbb{N}, t = nN_S : \frac{H^{(t)} - H^{(t-N_W)}}{\sigma(H^{(t):(t-N_W)})} < \eta \iff \beta^{(t+1)} = \gamma\beta^{(t)}$$

where $H^{(t)}$ is the energy of the system at time t.

N_S, N_W, η, and γ are fixed for whole process. Therefore, considering the initial temperature, this strategy has 5 free parameters.

3 Results

3.1 Metropolis Hasting vs. Glauber

For several values of α, we ran the simulation separately for Metropolis Hasting and Glauber algorithms. In both cases, other setting were fixed: Adaptive Step Size Cooling, $N = 1000$, $\beta^{(0)} = 0.5$, $N_S = 5000$, $N_W = 1000$, $\gamma = 1.05$, $\eta = 4\sqrt{3}$. The simulation code for this part is written in the file **run_two_alg.py**, and the figures are generated by codes in the file **Two_Algorithm.ipynb**.

Results are shown in Fig. 1 and 2. An interesting, more general animation is produced for this part, which is attached with the name **Met_vs_Gla_Animation.gif**. Overall, results can be summarized in some bullet points:

- For the case $\alpha > 1.3$, there is no difference in the final estimation error.
- Metropolis Hasting algorithm has smaller final estimation error the case $\alpha < 1.3$.
- Glauber algorithm is faster in convergence for all α values.

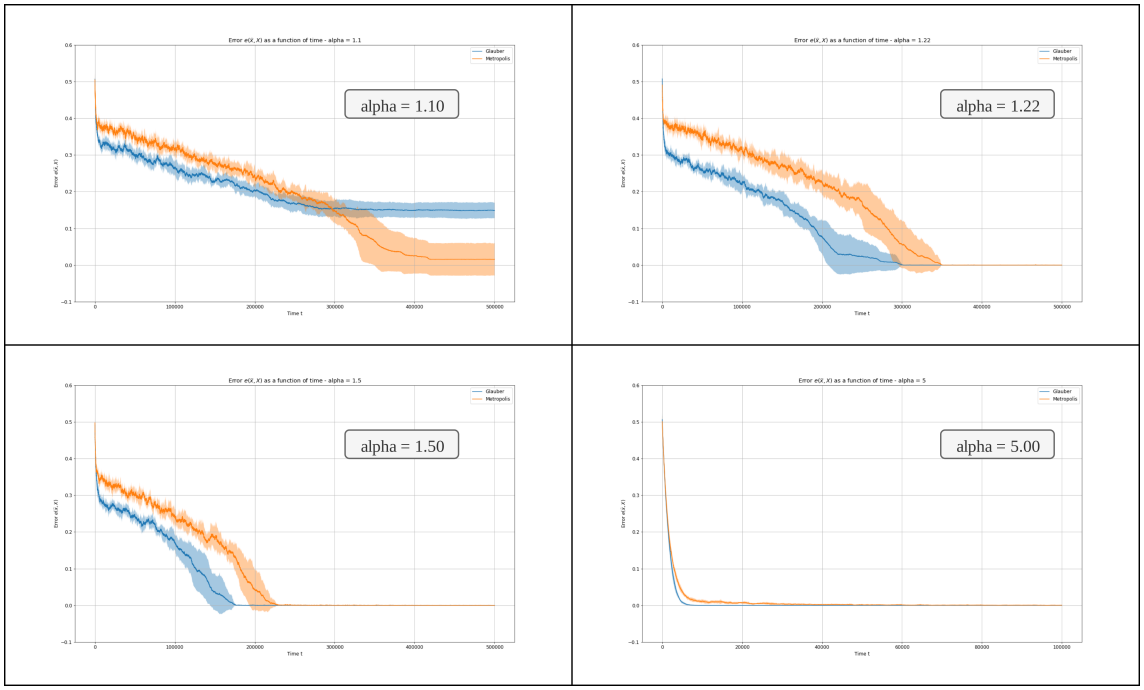


Figure 1: The average of estimation error ($\pm$ standard deviation) for both the Glauber (Blue) and Metropolis Hasting (Orange) algorithms over time - each plot is corresponding to a specific α

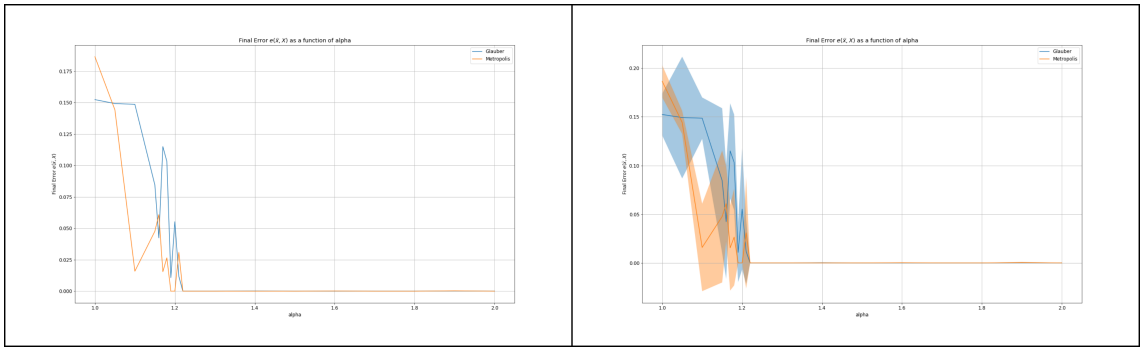


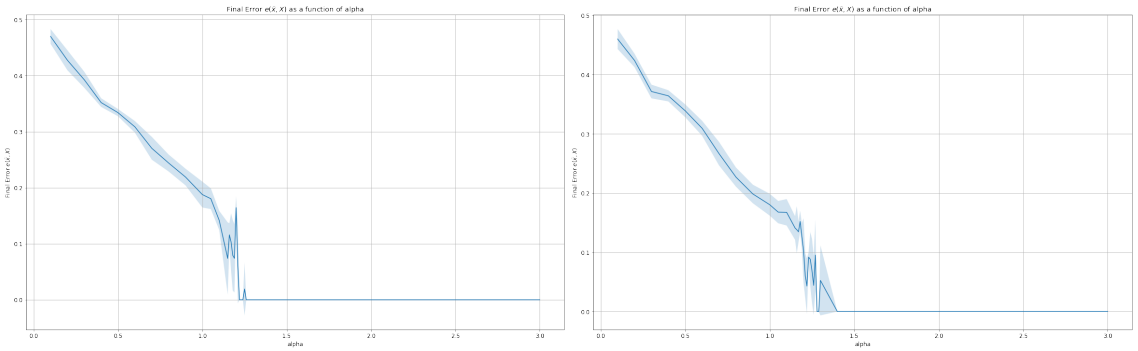
Figure 2: The average of final estimation error for both the Glauber (Blue) and Metropolis Hasting (Orange) algorithms over α - standard deviation is also shown in the right figure

3.2 Adaptive vs. Fixed Step Size Cooling Strategies

In this section, we are going to compare two cooling strategies(β Schedulers) for the Metropolis-Hastings Algorithm (The result for the Glauber algorithm is also similar)

For several values of α, we ran the Metropolis-Hasting algorithm separately for Fixed Step Size and Adaptive Step Size strategies. In both cases, other setting were fixed: $N = 1000$, $\beta^{(0)} = 0.4$, $N_S = 4000$, $N_W = 2000$, $\gamma = 1.1$, $\eta = 2\sqrt{3}$.

In the following, we analyze the critical value of alpha ,α_c (look at the section 3.3), and also the convergence time for Fixed Step Size and Adaptive Step Size strategies (As Fixed Temp Strategy does not have a good performance we don't analyze it here). The simulation codes for this part are written in **Metropolis_AdaptiveScheduler.ipynb** and **Metropolis_StepScheduler.ipynb** files.

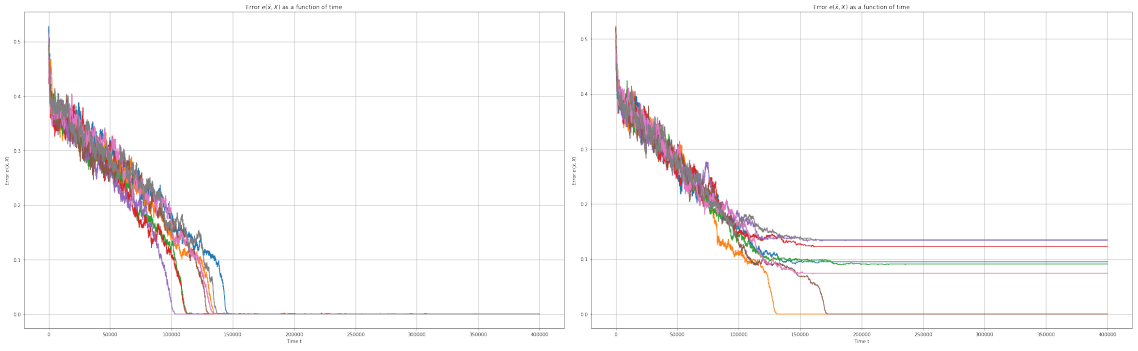


(a) (b)

Figure 3: Average of final estimation error for Adaptive(a) and Fixed Step Size(b) Schedulers over α - standard deviation is also shown

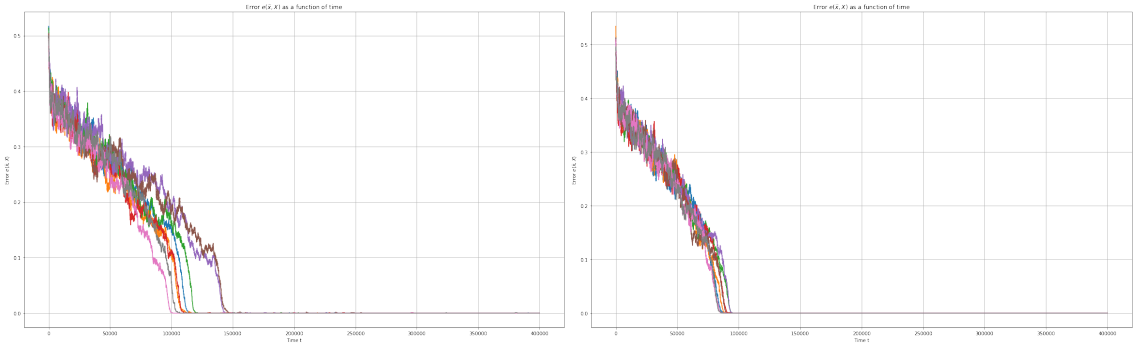
The critical value of alpha for Adaptive Scheduler is 1.21 and for Fixed Step Size is 1.40, so when the value of α is between 1.21 and 1.40 error of Adaptive Scheduler converges to zero but error of Fixed Step Size Scheduler does not converge to zero! Indeed if we look at the results Adaptive Scheduler performs better that Fixed Step Size Scheduler for all $\alpha < 1.4$

In the following, we analyze the convergence rate of these two cooling schedulers. The simulation codes for this part is written in **TWO_schedulers.ipynb**.



(a) (b)

Figure 4: Estimation error for 8 runs of Metropolis-Hasting algorithm for $\alpha = 1.3$ with (a)Adaptive Scheduler and (b)Fixed Step Size Scheduler



(a) (b)

Figure 5: Estimation error for 8 runs of Metropolis-Hasting algorithm for $\alpha = 1.5$ with (a)Adaptive Scheduler and (b)Fixed Step Size Scheduler

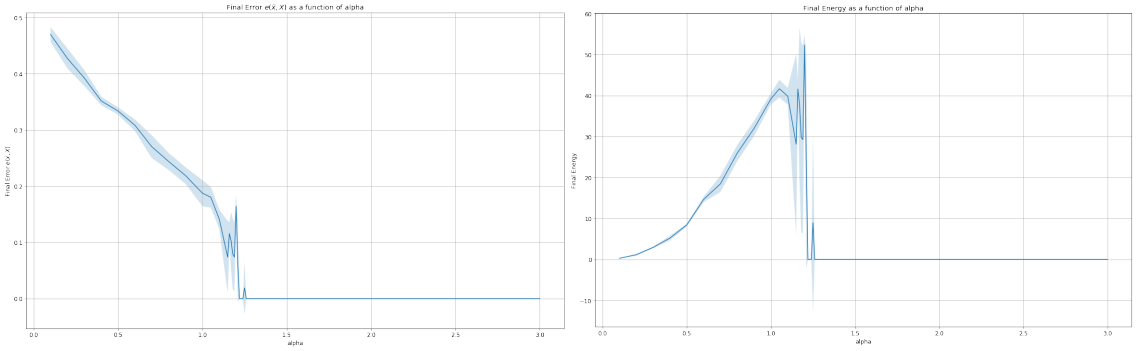
As we can see in these plots the Fixed Step Size Scheduler converge faster than Adaptive Scheduler

for $\alpha = 1.5$(generally for $\alpha \geq 1.4$) but for $\alpha < 1.4$ Adaptive Scheduler performs better.

3.3 Detailed Analysis of Phase Transition for the Final Model

Final Model: By the analyses in section 3.1 and 3.2 we conclude that the best method is to use the Metropolis-Hastings Algorithm with Adaptive Step Size strategy(to maximize the range of α which error converges to zero). We also checked for the parameters of the Adaptive Step Size strategy and find the best parameters as follows: $\beta^{(0)} = 0.4$, $N_S = 4000$, $N_W = 2000$, $\gamma = 1.1$, $\eta = 2\sqrt{3}$

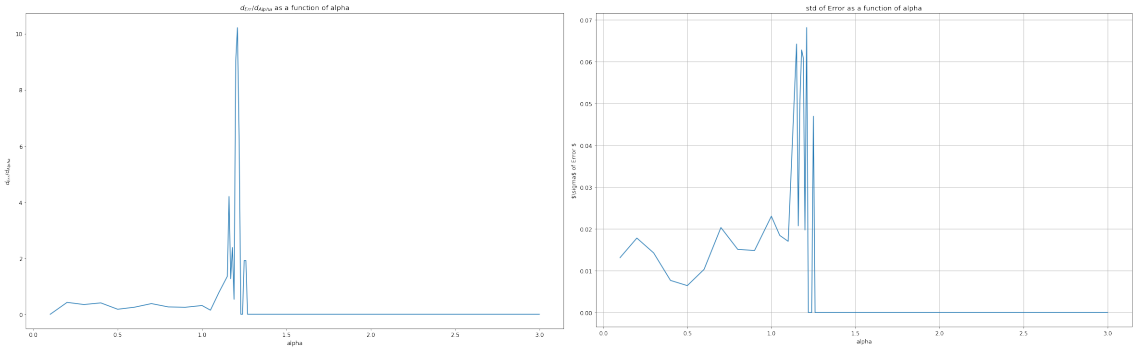
Critical alpha and Phase transition: In the following plot, we can see a phase transition behavior, which means there exist a value of alpha, α_c, that the behavior of error and energy functions are different for $\alpha < \alpha_c$ and $\alpha \geq \alpha_c$ (for $\alpha \geq \alpha_c$ error and energy are zero). In such a phase transition, if we look at the derivative of the error or energy function with respect to alpha we can see it somehow diverges at α_c (Figure 7a).



(a) (b)

Figure 6: Phase Transition: for $\alpha \geq \alpha_c$ (a)error and (b)energy are zero

Another way of finding phase transition behavior is looking at the variance which also diverges near the α_c (Figure 7b).



(a) (b)

Figure 7: Phase Transition: (a)derivative of error with respect to alpha, and (b)variance of error peak at $\alpha = \alpha_c$

α_c for Metropolis-Hastings Algorithm with Adaptive Step Size strategy is 1.21

α_c depends on the algorithm, cooling scheduler, and the parameters but it is usually between 1.2 and 1.4

α_{random} : for $\alpha \leq 0.1$ the error is 0.50 and it is simmilar to pure random guess.

4 Conclusion

In this project, we have applied the MCMC method to a generalized linear estimation problem. Our main contributions are the followings. First, we have written an efficient and parallelized implementation of the MCMC method, enabling simulations to be run on a GPU. We have moreover designed an adaptive cooling strategy that outperform the naive constant temperature baseline and our fixed step size cooling strategy. By comparing the performance of the Metropolis and Glauber algorithms, we are now able to choose the best performing one for the final competition. Finally, we have precisely studied the phase transition in our algorithm's performance based on the value of α.