State-Space Exploration Twist on Simulated Tempering

Samuel Leung

Background

Bayesian inference of a parameter of interest involves deriving a posterior distribution based on prior belief and observed data (likelihood). In many cases, direct computation of the posterior is not possible. Fortunately, Monte-Carlo Markov Chain (MCMC) algorithms make it possible to sample from the posterior, recreating the posterior distribution fully as the number of samples approach infinity. A majority of MCMC methods are based on the Metropolis-Hastings algorithm, but for this project we explore a possible adaptation of another MCMC sampling method: simulated tempering. Simulated tempering generates and samples from a series of Markov Chains based on user-defined "temperatures" in an effort to explore a desired state space. The generated Markov Chains are able to communicate and swap temperatures with adjacent Markov Chains, which allows the algorithm to traverse the state space more efficiently and helps prevent the algorithm from being stuck in local minimums. As an alternative way to decide which Markov Chain to sample from, we design a Markov Chain on the state space which satisfies local balance and maximizes the expected distance between state spaces. For this project, we restrict our exploration to be on problems with discrete state space.

The Logic

Consider a problem with discrete state space $X_i; i \in \{1, \cdots, n\}$, where $X_i \sim \pi_i(X_i)$. With traditional simulated tempering, given that we are at state X_i, we consider sampling from only the next neighboring state distributions π_{i-1} or π_{i+1} (neighboring in integer space). Rather than only considering these distributions, we can consider choosing to sample from non-neighboring π_i's (ie. consider jumping to different index i's).

As mentioned in the project proposal, we utilize linear optimization to solve for the matrix of transition probabilities K_{ij} under the constraint of global balance (for my project, the rows represent the current index i, and the columns represent the possible next index j). Mathematically, the constraints used are:

$$\pi(i) = \sum_{j \in \{1,\cdots,n\}} \pi(j)K(i|j); \;\; \forall i, \sum_j K(i|j) = 1$$

Also mentioned in the proposal, we want to optimize the transition matrix to maximize the expected distance. Mathematically, the objective function to maximize is:

$$f_{\text{obj}} = \sum_{i,j \in \{1,\cdots,n\}} \pi(i)K(j|i) \cdot d(i,j)$$

This maximization will be dependent on the choice of distance measure between state spaces $d(i,j)$. Many of the common distance metrics, such as Euclidean and Manhatten distances, are reduced to just the absolute distance in one dimension, so we choose this to be one of the distances to try. To place more weight on farther states, we also compare squared and cubed distances.

To ensure that the algorithm is behaving properly and that the proper transition matrix is produced, the optimized kernels should also pass certain tests before being used. Namely, we want to make sure that the kernels produced still respect the constraints listed from linear optimization,

so we check that global balance still holds. Given that global balance holds, meaning that kernel K is $\pi-$invariant, we also note that as the steps of the Markov Chain n approaches infinity, the limit of K^n converges to a matrix with the initial distribution π_0 as its rows. In more mathematical notation, given that K is $\pi-$invariant:

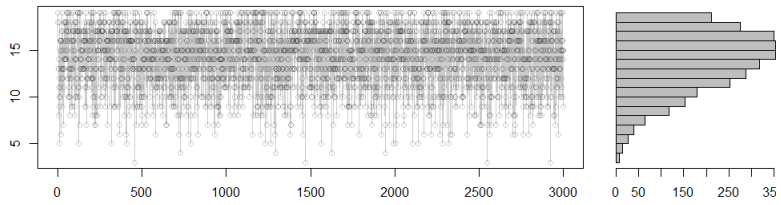
$$\lim_{n\to\infty} K^n = \begin{pmatrix} \pi_0^T \\ \vdots \\ \pi_0^T \end{pmatrix}$$

By definition, this result also satisfies global balance, but should not be optimized towards maximizing the objective function. Therefore, we can also test that for whatever K we solve for, the expected distance for that K should be greater than this $\pi-$stacked kernel, namely $f_{\text{obj}}(K) > f_{\text{obj}}(K_{\pi-\text{stacked}})$. In code, this check is done directly after the kernel is generated. If the kernel doesn't satisfy the tests explained above, sampling does not occur.

Findings

We test our implementation on the rockets example introduced in class, defined the same way as its described in excercise 2. Using the method described in the previous section, we first generate a sample of the indexes over 3000 iterations, and show the trace and histogram plots. All algorithms were initialized with starting index of 2, out of 21 different state spaces and 19 possible starting indices (we don't start with indexes 1 or 21, as those correspond with posterior probabilities of zero):

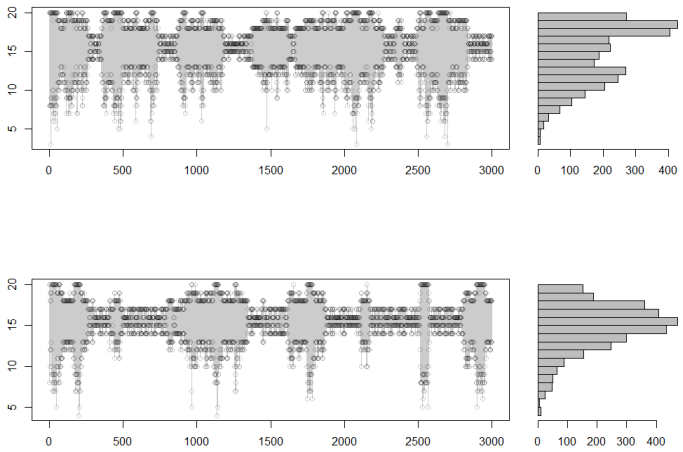
Figure 1. Trace Plot and Histogram of samples; kernel maximized under absolute distance



From Figure 1, we see that the algorithm seems to traverse the entire state space, without staying at the same index for long. We also see a tendency for the algorithm to jump to indices far from its current position, which is what we want to see. To further investigate how the specified distance function affects index traversal, we look next at the trace and histogram plots for squared distance ($d(i,j) = (i-j)^2$) and cubed distance ($d(i,j) = |i-j|^3$):

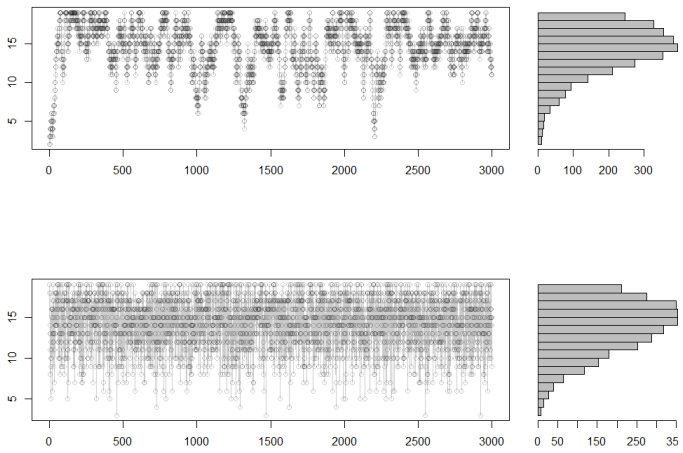
Comparing the trace plots in Figure 2 to the trace plot generated from absolute distance, we see that with greater weight placed on the distance between indices, the chain does not seem to mix as well. Under squared and cubed distances, larger jumps are taken more often compared to absolute distance. On the contrary, squared and cubed distance also seems to get stuck and sample only a few indices over longer periods of time, whereas absolute distance does not appear to have this problem. Judging from the histograms produced by each distance metric, the frequency of how many times an index is similar to one another. But as more weight is placed on how far apart the indices are, it appears that the histogram trends towards being more uniform and symmetric.

Figure 2. Trace Plot and Histogram of samples; kernel maximized under squared distance (top) and cubed distance (bottom)



We now compare the trace and histogram plots for the random walk over the indices, as well as simple IID sampling from the initial posterior through Figure 3. We want to compare with the random walk as it acts as a baseline for traversal of the state space while only considering neighboring states, and the IID sampling lets us see what the state space distribution actually looks like. The random walk was implemented using Metropolis-Hastings, where only adjacent indices are proposed (specific implementation can be found in this file of my github repo).

Figure 3. Trace Plot and Histogram of samples from Random Walk (top) and IID sampling (bottom)



Looking at the trace plots, nothing out of the ordinary stands out. The trace plot and histogram produced from absolute distance looks the most similar to that from IID sampling out of all plots produced. Both methods sample less from indexes with lower corresponding posterior probabilities, by construction. In the context of simulated tempering, where we want to explore the posterior state space by sampling from auxiliary distributions. If we replace sampling scheme of simulated tempering with either IID sampling or from an absolute-distance maximized kernel, then we may

run into the risk of under-coverage when trying to estimate the posterior distribution, as certain auxiliary distributions would be much harder to reach and sample from.

In simulated tempering, the mixing time is an important metric that represents how efficiently the state space is explored. One way we can get a measure of the mixing time is by estimating the recurrence time, or how long it takes for the Markov Chain to return to its initial point. From the histograms above, we see that the frequency of visits is not uniform across the indices, so we visualize the average recurrence time over 100 iterations. In addition to this, from testing at various initialization, occasionally the chain fails to return to its original index. We also opt to visualize how many times the chain misses the initial index, over the 100 iterations.

Figure 4. Histograms of average recurrence times, for Absolute Distance (top), Squared Distance (middle), and Cubed Distance (bottom), colored by posterior probabilities.

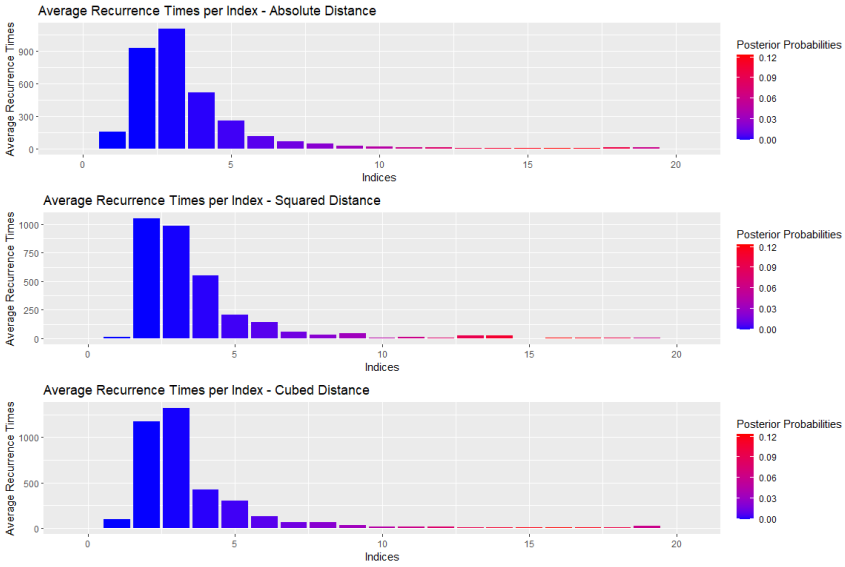
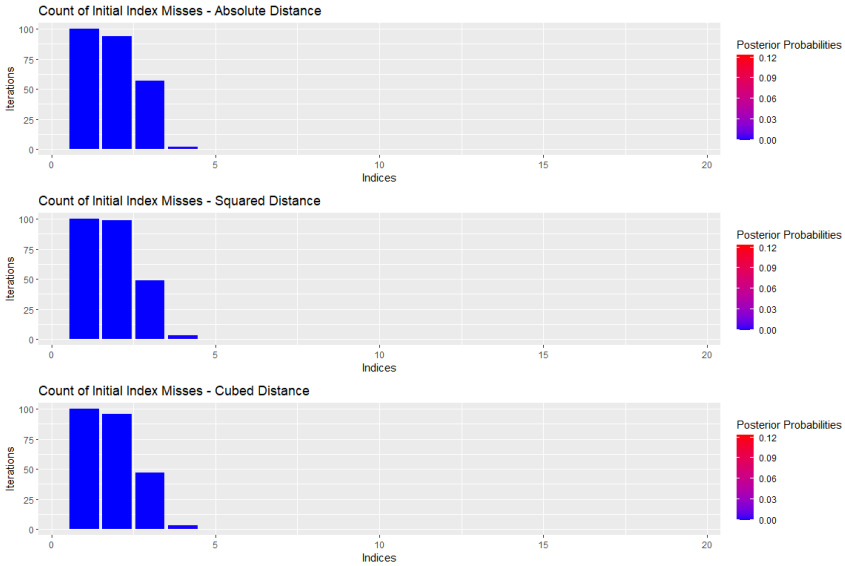


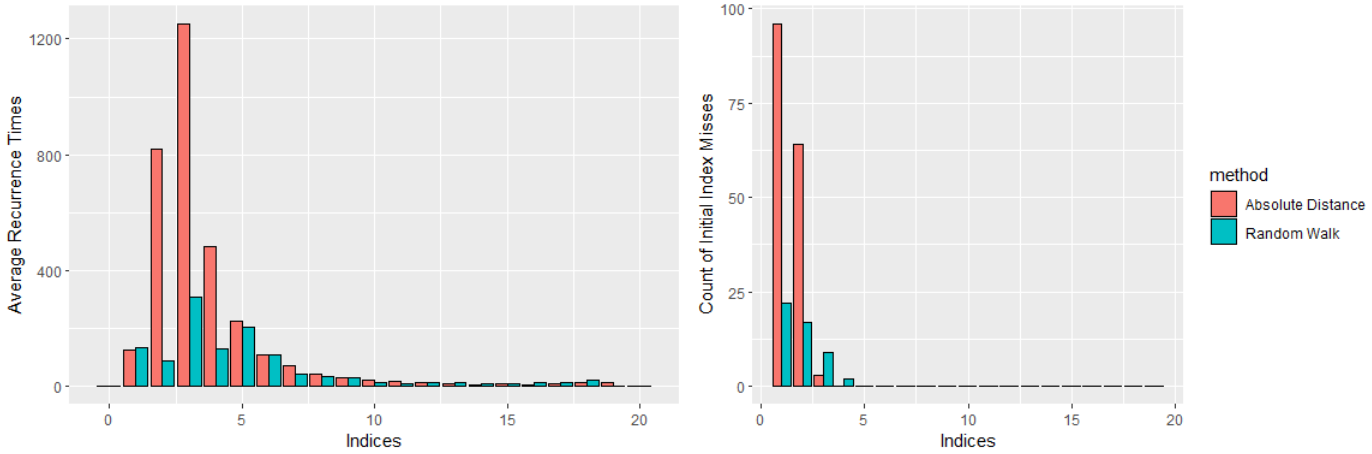
Figure 5. Histograms of missed returns to initial point, for Absolute Distance (top), Squared Distance (middle), and Cubed Distance (bottom), colored by posterior probabilities.



Across all distance metrics, we see that indices corresponding to low posterior probabilities tend

to have high average recurrence times and high number of misses to return to its initial index, which indicates poor mixing times for the corresponding indexed posterior. Lastly, we want to compare how mixing times with our proposed algorithm measures up to mixing times from random walk. To avoid over-plotting, we only compare absolute distance with random walk mixing times:

Figure 6. Histograms of average recurrence times (left) and return-to-initial index misses (right), for absolute distance and random walk methods



On average, the random walk algorithm has lower recurrence time and return-to-initial index misses, indicating that random walk seems to have better mixing time over absolute distance optimized kernel sampling.

Conclusion and Limitations

In this project, we explore how an expected distance maximized kernel performs in terms of how effectively it traverses a discrete state space. From the trace plots, we see that when using absolute distance, the chain seems to mix well, but as heavier emphasis is placed on the distance between indexes, the trace plots show that the sampling process oscillates between a small range of indices for longer periods of time before being able to escape and traverse to other state spaces. When considering mixing times, an important metric for simulated tempering, we found that indices with lower posterior probabilities have poor recurrence times, and tend to be unable to return to its starting index. When compared to random walk, on average the random walk returns to its original starting state faster and misses less frequently.

The results shown in this report are only for the dummy example of the rockets dataset shown in class. For all the conclusions listed in this report, they should be taken with a grain of salt, as more formal testing should be done before any definitive statements about performances between algorithms are made. Adapting simulated tempering to use this algorithm of moving between state spaces would be a natural continuation of the work shown here, and would provide further insight on whether or not this modification is useful. Benchmarking the modified simulated tempering method against different datasets would also be useful in assessing performance given varying factors, such as problems with much larger state spaces.

Link to Github repository: github.com/sleung124/Stat447-Final-Project