Sean Steinle

STAT778

An Adaptive Metropolis-Hastings Algorithm

Background:

This project is meant to replicate and extend the work of Holden et al. 2009 [1]. In their work, the authors introduced an adaptive technique for Metropolis-Hastings sampling.

This adaptive technique aims to improve the speed at which the sampling algorithm converges by leveraging samples from earlier in the chain. To this end, the authors' modification to the algorithm is to keep track of a vector of samples, Y, and to leverage them when generating new samples and when determining whether a sample should be accepted.

One drawback of employing adaptivity is that keeping track of Y may absorb a significant memory overhead. Adaptivity also requires certain conditions like the Doebin condition, which states that the proposal distribution has heavier tails than the target distribution.

A final note about the paper: the authors of the paper do not make it explicit as to exactly how Y should factor into sample generation or the acceptance calculation. They provided some commentary on how this phenomenon might differ when the proposal distribution is parametric or not parametric, but the lack of specificity on this point was quite difficult for me to overcome.

Implementation:

In this project, I implemented a Metropolis-Hastings sampler with a Bayesian linear model whose posterior served as the proposal distribution. The model's likelihood was calculated by using the Gaussian probability density function in conjunction with observed data—the distribution was centered on the *ax+b* with spread *sigma*. The prior was derived from a multivariate Gaussian for *a* and *b* and a Gamma distribution for *sigma*. Of the authors' four examples, this setting is most similar to the fourth.

After implementing an ordinary Metropolis-Hastings sampler, I added the adaptive functionality. Specifically, I modified the algorithm to maintain a history vector, Y, and I calculated the prior with respect to the average of the accepted samples as opposed to the proposed sample itself.

Simulation Study:

To test my algorithm in a controlled environment, I attempted to recover my parameters from synthetically generated data. To do this I generated a 2-dimensional matrix with a single uniform input and a normally distributed response with parameters *c, d*. Because the input *x* was uniform, a successful sampler would converge to the *a* = 0, *b* = *c*, and *sigma* = *d*. In my test case, *c* = 50 and *d* = 10.

Convergence to the correct parameters was important, but the speed of the convergence is what adaptivity is interested in. To this end, I ran the algorithm from 5 different starting points. If the adaptivity I implemented was successful, then the adaptive version of the algorithm should converge faster from any of the starting points than the non-adaptive version would converge. For my test case, I used the following starting points (*a*,*b*,*sigma*): (0,0,1), (0,10,2), (0,20,4), (0,30,6), (0,40,8), (0,50,10).

To measure convergence, I attempted to use the autocorrelation function (ACF) from the statsmodels Python package. Unfortunately, there was no clear, parameter-agnostic threshold for which I could measure convergence. Additionally, it was clear that finding the correct *n_lags* hyperparameter required further finetuning, so I visually inspected parameter estimation plots for convergence instead. That said, I still generate the ACF plots for completeness.

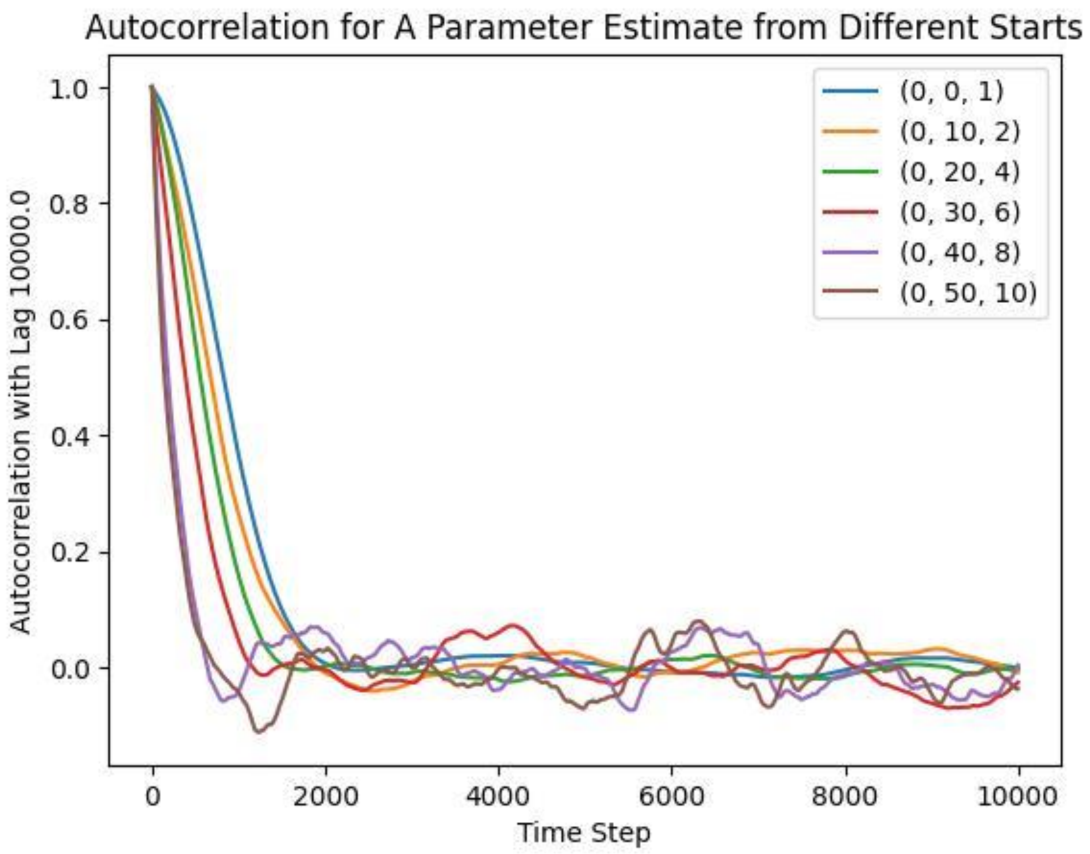

Figure 1: Autocorrelation function as a potential convergence indicator

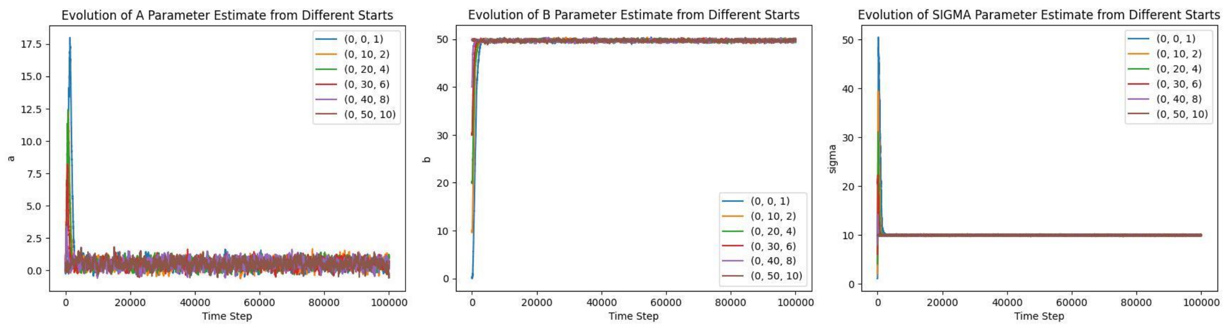

Figure 2: Parameter estimations using the non-adaptive algorithm

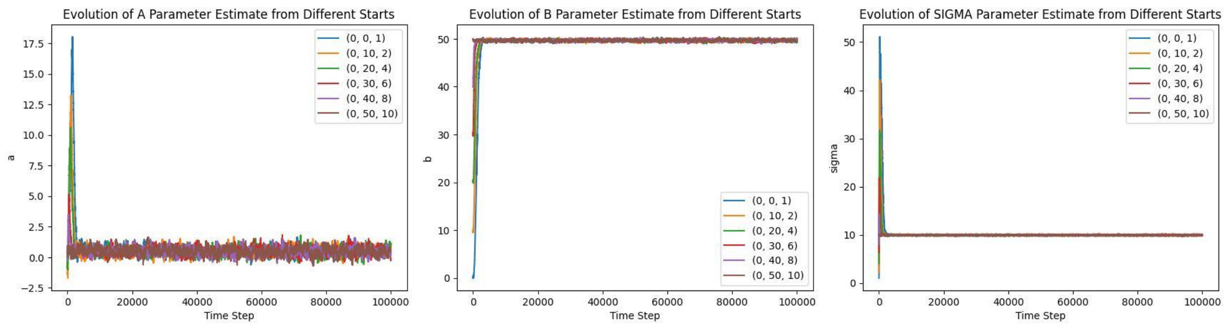

Figure 3: Parameter estimates using the adaptive algorithm

As you can see, both algorithms recover the correct parameters successfully. In fact, the charts are virtually indistinguishable. Upon further inspection, it appears that the rates are which each algorithm converged are also indistinguishable. This may be the case for one of two reasons: either the target distribution in this case is so simple to estimate that the performance gains are unobservable or there are no performance gains for the adaptive algorithm.

Real-World Application:

For a real-world application of this algorithm, I wanted to see if coaches could improve their chances of a successful play by making modifications before the play. To measure whether a play is successful or not, I use the favorite statistic of the NFL advanced metrics community, expected points added (EPA). I used the nflfastR library [2] to gather NFL statistics from the 2023-2024 season for analysis.

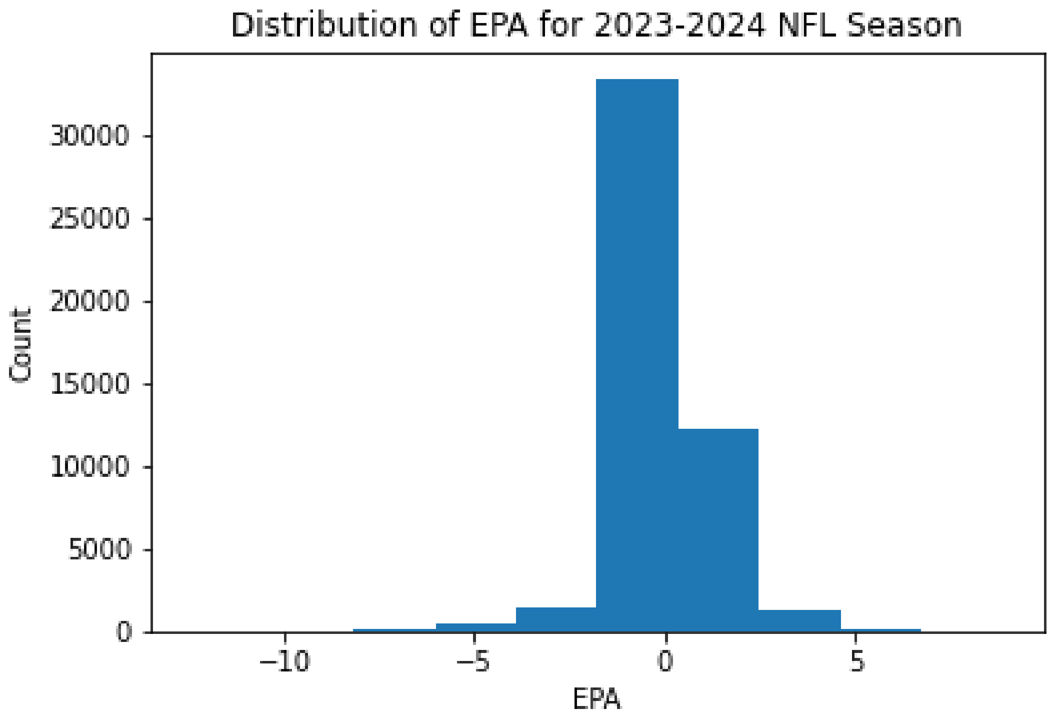

Figure 4: Distribution of EPA metric

I first manually inspected pre-snap variables which I expected to be predictive of EPA—distance to first down and yardline. Unfortunately, neither of these variables were correlated with EPA at all (r = <0.05).

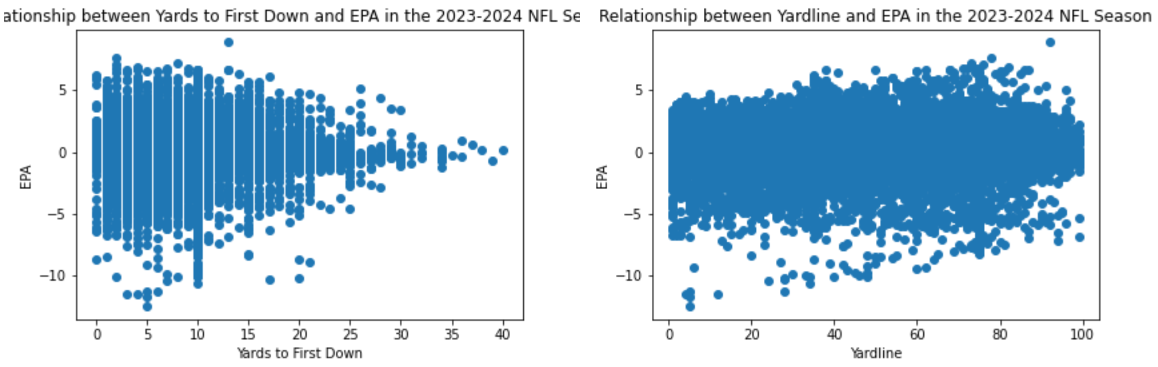

Figure 5: Yards to first down and yardline against EPA

I thus pivoted my strategy. My new goal was to ascertain as to whether the ensemble of pre-snap variables was predictive of EPA. If it was, then I would try to find a latent variable which could act as a single-dimensional predictor. Unfortunately no combination of pre-snap variables was very useful.

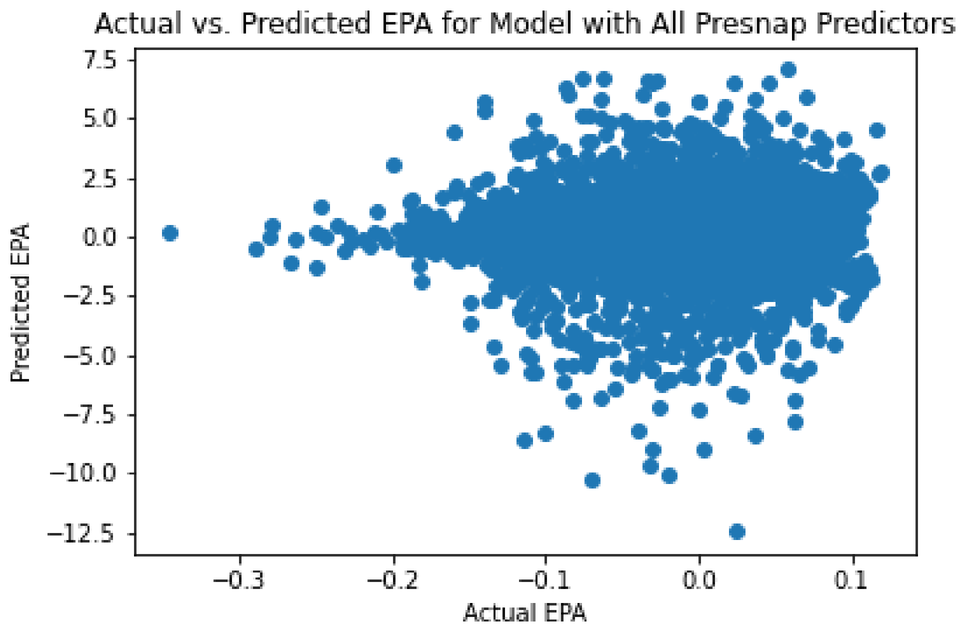

Figure 6: Predicted EPA vs. Actual EPA for model with all pre-snap predictors

I should note that while this is a disappointing result, it's not a particularly surprising one: if there was a silver bullet to creating perfect pre-snap adjustments, it would have been uncovered and exposed by the very sharp analytical market of NFL data analytics teams. Because EPA cannot be effectively predicted given only pre-snap statistics, I modified my research question to ask how important is the number of passing yards in a play in influencing the play's expected points added?

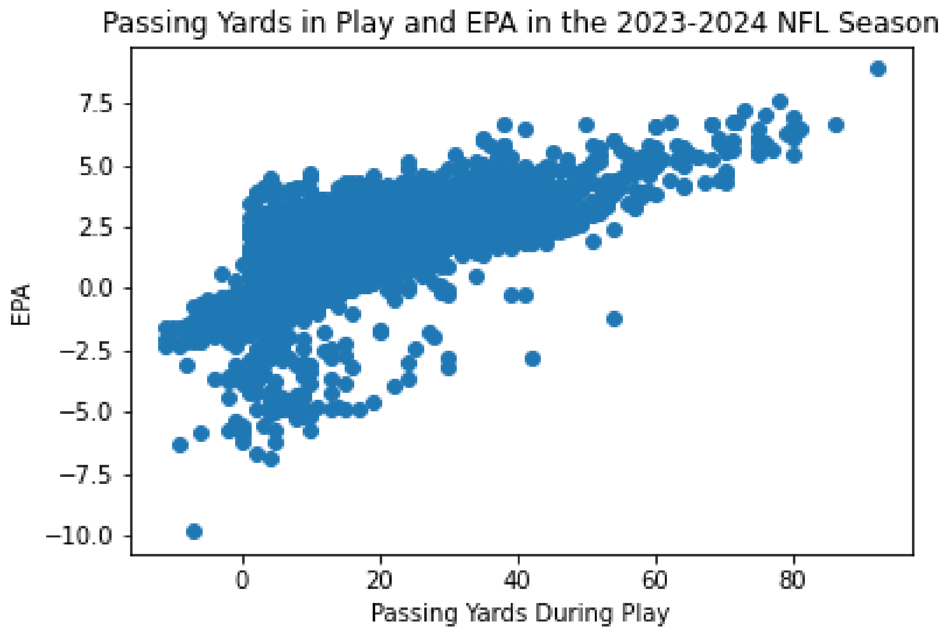

Figure 7: Passing yards against EPA

Initially, I naively estimated the parameters could be (*a*,*b*,*sigma*) = (5,20,1.5), so I tried starting the algorithm from four initial parameter locations: (0,0,1), (5,20,1.5), (10,40,2) and (20,80,5). Running both algorithms on this parameter set revealed an interesting result: the adaptive algorithm failed to converge to reasonable parameters for (10,40,2) and (20,80,5). Given that the non-adaptive algorithm converged from all four starting parameter sets, this is an alarming sign. There is a small chance that the adaptivity was tuned incorrectly, as the authors warn, but it is much more likely that my adaptivity was insufficiently specified.

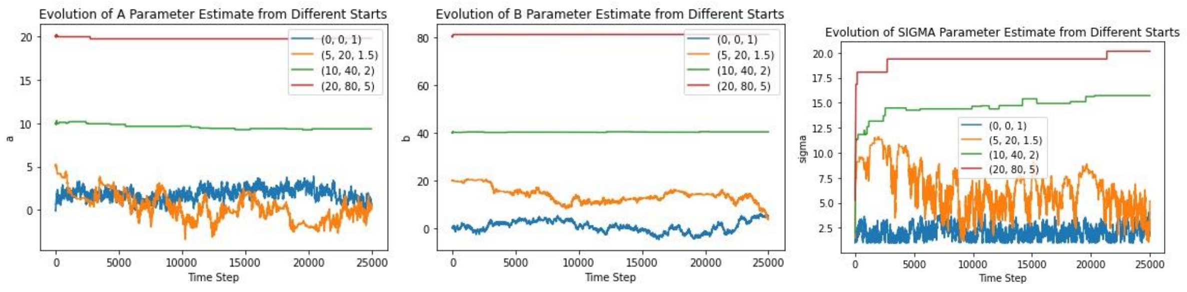

Figure 8: Parameter estimates using the adaptive algorithm

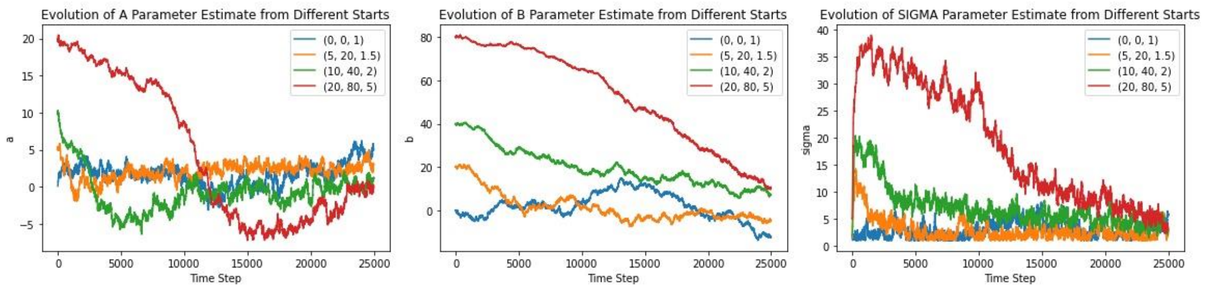

Figure 9: Parameter estimates using the non-adaptive algorithm

Conclusion:

Although the adaptivity component of my sampling algorithm ultimately fell short, this exercise was incredibly helpful in understanding the essence of the Metropolis-Hastings algorithm and the technique of Monte Carlo Markov Chains generally. It forced me to grapple with difficult questions along the way like how to measure convergence, how to tune hyperparameters you know very little about, and how to design a purely experimental statistical study.

Citations:

1. Holden, Lars, Ragnar Hauge, and Marit Holden. "Adaptive independent metropolis-hastings." (2009): 395-413.

2. Carl S, Baldwin B (2024). *nflfastR: Functions to Efficiently Access NFL Play by Play Data*. R package version 4.6.1.9008, https://github.com/nflverse/nflfastR, https://www.nflfastr.com/.