Dr. Bo Ning
University of California, Davis
STA 141C

March 23, 2023

Dear Dr. Ning,

We submit our report concerning the application of Gibbs MCMC to estimating linear regression coefficients for both low- and high-dimensional simulated datasets. In our report we analyze the computational efficiency and predictive power of Gibbs MCMC and compare it to several frequentist estimation methods (OLS, Ridge, and Lasso). We also demonstrate a useful feature of Bayesian estimation - the ability to create credible intervals for the parameter estimates based on their generated posterior distributions.

We find that the Bayesian and frequentist methods have similar predictive power for both low- and high-dimensional data, but that MCMC takes far longer to complete than the frequentist methods when using a reasonable number of iterations. This is due to the fact that it needs to draw a large number of parameter estimates from known distributions over these many iterations. We conclude that MCMC techniques should only be used if obtaining credible intervals is of high value and the posterior distributions are difficult to calculate or draw from.

We were not able to cover MCMC in class due to time constraints. However, we still tried to tie into our project the major themes of the class including analysis of timing and memory use. Since MCMC is an iterative method that is intended to converge, it is similar in this respect to some of the other iterative methods we covered in class like the Jacobi method and power method for singular value decomposition.

We hope that other students may be able to look at our project in the future and be able to supplement their learning of MCMC with it. We believe that knowledge of Bayesian techniques is highly valued in the fields of statistics and data science and our project provides insight into a quintessential application of Bayesian thought.

Thank you for your consideration in reading this report.

Sincerely,

Samuel Van Gorden
Jordan Bowman
Riley Adams

STA 141C Final Project: MCMC with Gibbs Sampling vs Frequentist Methods for Estimating Regression Coefficients on Simulated Data

Jordan Bowman, Samuel Van Gorden, Riley Adams

23 March 2023

Abstract

Within this report, we implement a Gibbs sampler for conducting MCMC calculations to compute the point estimates and distributions of linear regression under a Bayesian framework with non-informative priors. We perform this analysis upon two simulated datasets: the first with $n = 300$ and $p = 6$, including two nonsignificant predictors, and the second featuring a higher dimensional setup with $n = 1050$ and $p = 1000$. Training and test sets of equal size are generated from the same model. We then the coefficient estimates produced by our Gibbs sampler to those from common linear regression functions within `sklearn.linear_model` including `LinearRegression`, `RidgeCV` and `LassoCV` by computing the mean squared error (MSE) of prediction. We also compare runtimes for the higher dimensional model across various methods. Our findings reveal that the Gibbs sampler is significantly more computationally expensive than the `sklearn.linear_model` methods despite yielding similar prediction performance. Ultimately, we conclude that LASSO is the preferred method for our example cases when only point estimates are needed; however, our Gibbs sampling methodology enables the creation of credible intervals for the predictors.

1 Introduction

Linear regression is a fundamental problem in statistical analysis and over the years there have been many methods introduced to produce better predictive results in a frequentist setting. Notable examples include the the Ridge introduced by Hoerl and Kennard (1970) [2] and the LASSO introduced by Tibshirani (1996) [5] among many others. Both of these methodologies in particular function by penalizing the norm the standard OLS solution. Although both the Ridge and LASSO compute different results, the general premise is that we introduce bias and nudge the computed coefficients toward zero in order to, with a properly selected penalty term, drastically reduce the variance of the predicted coefficients and thus the overall mean squared error estimate in the regression model.

The linear regression problem can also be constructed under a Bayesian framework by assuming that the regression coefficients follow some prior distribution. By adjusting these prior, we can impart previous knowledge upon the posterior distribution of our regression coefficient estimates, and, through some measure of center, thereby impact the point estimates of our regression coefficients. In fact, Park and Casella (2008) [3] showed that the LASSO can be formulated in a Bayesian sense with properly constructed Laplace priors however that is outside the scope of this project.

The purpose of this project is to demonstrate the use of Markov Chain Monte Carlo (MCMC) methods for obtaining parameter estimates. Specifically, we will be utilizing Bayesian MCMC with Gibbs sampling in order to obtain posterior distributions of linear regression coefficients and variance under the assumption of homoskedasticity. We use noninformative conjugate priors to generate posteriors for the iterative sampling of our coefficients and variance. Finally, we compare the estimates via MSE of prediction from our Bayesian implementations to those generated using frequentist techniques such as OLS, Ridge and LASSO regression.

The advantage of using Bayesian MCMC over frequentist methods is that having posterior distributions allows us to determine credible intervals. This allows us to have a better understanding of the spread, or variance, of our parameter estimates instead of just getting a point estimate as is the case in frequentist methods. The accuracy of these estimates and credible intervals depend of course on the priors we specify, which is not something we need to worry about when using frequentist methods. Furthermore, MCMC can also be extremely computationally costly compared to frequentist methods.

2 Proposed Methods

2.1 MCMC and Gibbs Sampling

Markov Chain Monte Carlo methods combine stochastic processes (Markov Chains) with random sampling (Monte Carlo methods) to estimate calculations that would otherwise be untenable with classical estimation methods. Monte Carlo algorithms work by randomly sampling from a domain and then calculating the proportion of sampled points that meet some criteria. For example, area under a curve can be estimated by calculating the proportion of points in a rectangle of area A that lie below the curve and then multiplying that proportion by A.

Markov Chains are stochastic processes in which each state in a process depends only on the previous state and none of the earlier ones. They are incorporated with Monte Carlo techniques by allowing each sample taken to be dependent on the previous sample that was taken. In our MCMC implementation, we utilize Markov Chains to select parameter estimation samples that are based on the previously selected samples. In theory, doing so will cause the samples to gradually converge to being drawn from the true parameter distributions.

Gibbs sampling is a special case of Metropolis-Hastings sampling. In Metropolis-Hastings, each parameter estimate is drawn from a posterior distribution based on its prior and the likelihood of our data. The posterior distribution is conditioned on the data and the previously drawn values of every other parameter:

$$\begin{aligned}
&f(\theta_1^{(t)}|\mathbf{X}, \theta_2^{(t-1)}, \theta_3^{(t-1)}, \ldots, \theta_p^{(t-1)})\\
&f(\theta_2^{(t)}|\mathbf{X}, \theta_1^{(t)}, \theta_3^{(t-1)}, \ldots, \theta_p^{(t-1)})\\
&f(\theta_3^{(t)}|\mathbf{X}, \theta_1^{(t)}, \theta_2^{(t)}, \ldots, \theta_p^{(t-1)})\\
&\qquad\vdots\\
&f(\theta_p^{(t)}|\mathbf{X}, \theta_1^{(t)}, \theta_2^{(t)}, \ldots, \theta_{p-1}^{(t)})
\end{aligned}$$

where $\theta_i^{(j)}$ refers to the j^{th} sample of the i^{th} coefficient and $\mathbf{X}$ is our dataset. Each $\theta_i^{(0)}$ must be initialized to a reasonable starting value.

Normally, in Metropolis-Hastings each sample is only accepted with a probability given by

$$\min\{1, \frac{f(\boldsymbol{\theta}^{(t)}|\boldsymbol{\theta}^{(t-1)})f(\boldsymbol{\theta}^{(t-1)})}{f(\boldsymbol{\theta}^{(t-1)}|\boldsymbol{\theta}^{(t)})f(\boldsymbol{\theta}^{(t)})}\} \tag{1}$$

to that of the previous parameter draws. However, with Gibbs sampling the second component of the min operator takes value 1, and thus we always accept each draw. We are able to use Gibbs sampling in our project because we have known full conditional posteriors given our choice of priors [1].

MCMC involves repeating the above process for a certain number of iterations and discarding the first n iterations as a burn-in phase (when the process has not yet converged). We save the parameter values estimated in each iteration, minus the burn-in, and use these to construct a joint distribution of all parameters. From here we can find point estimates using measures of centrality—such as the mean or median–and credible intervals given these estimates and the parameter distribution.

2.2 Simple Bayesian Linear Regression

First we consider a standard linear regression model as given by

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{2}$$

where response vector $\mathbf{y} \in \mathbb{R}^n$, design matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, coefficients $\boldsymbol{\beta} \in \mathbb{R}^p$ and error term $\boldsymbol{\epsilon} \in \mathbb{R}^n$. Recall that each $\epsilon_i \overset{\text{iid}}{\sim} N(0, \sigma^2)$. In Bayesian linear regression we assume, as done by Scanlon (2021) [4] that $\boldsymbol{\beta}$ is produced by a multivariate normal distribution as follows:

$$\boldsymbol{\beta} \sim N_p(\boldsymbol{\mu}_\beta, \boldsymbol{\Sigma}_\beta). \tag{3}$$

Futhermore, we assume that the variance of the error term σ^2 results from an inverse-gamma distribution as follows:

$$\sigma^2 \sim IG(\alpha_0, \beta_0). \tag{4}$$

These are the prior distributions of $\boldsymbol{\beta}$ and σ^2. Note that these priors are conjugate priors and so the full conditional posterior distributions belong to the same multivariate normal and inverse-gamma distributions respectively. As derived by Scanlon (2021) [4], the posterior distribution for $\boldsymbol{\beta}$ is

$$[\boldsymbol{\beta}|\cdot] \sim N_p(\mathbf{A}_\beta^{-1}\mathbf{b}_\beta, \mathbf{A}_\beta^{-1}), \tag{5}$$

where

$$\mathbf{A}_\beta = \frac{1}{\sigma^2}\mathbf{X}^\top\mathbf{X} + \boldsymbol{\Sigma}_\beta^{-1}, \tag{6}$$

and

$$\mathbf{b}_\beta = \frac{1}{\sigma^2}\mathbf{X}^\top\mathbf{y} + \boldsymbol{\Sigma}_\beta^{-1}\boldsymbol{\mu}_\beta, \tag{7}$$

and the posterior distribution for σ^2 is

$$\left[\sigma^2|\cdot\right] \sim IG\left(\alpha_0 + \frac{n}{2}, \beta_0 + \frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right). \tag{8}$$

We can then use these posterior distributions to construct a Gibbs sampler to compute our regression coefficients.

As one final note, notice for a truly non-informative prior on $\boldsymbol{\beta}$ such that $\boldsymbol{\Sigma}_\beta^{-1} \to \mathbb{0}_{p \times p}$, which for a diagonal $\boldsymbol{\Sigma}_\beta = \text{diag}(s_1^2, ..., s_p^2)$ means each $s_j^2 \to \infty$, implies that

$$[\boldsymbol{\beta}|\cdot] \sim N_p\left((\mathbf{X}^\top\mathbf{X})\mathbf{X}\mathbf{y}, \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}\right). \tag{9}$$

Recall that this is the mean and variance estimate of the OLS solution of the linear regression coefficients $\hat{\boldsymbol{\beta}}_{\text{OLS}}$ for a given σ^2.

2.3 Gibbs Sampler Implementation of Bayesian Linear Regression

For a large number of iterations K, we will create a Gibbs sampler in Python that iteratively samples from each of the posterior distributions described in the previous section for each iteration $k = 1, ..., K$. This will lead to the following being computed at each iteration k:

$$\mathbf{A}_\beta = \frac{1}{\sigma^{2(k-1)}}\mathbf{X}^\top\mathbf{X} + \boldsymbol{\Sigma}_\beta^{-1}, \tag{10}$$

$$\mathbf{b}_\beta = \frac{1}{\sigma^{2(k-1)}}\mathbf{X}^\top\mathbf{y} + \boldsymbol{\Sigma}_\beta^{-1}\boldsymbol{\mu}_\beta, \tag{11}$$

which will then be used to used to sample the current iteration of

$$\left[\boldsymbol{\beta}^{(k)}|\cdot\right] \sim N_p(\mathbf{A}_\beta^{-1}\mathbf{b}_\beta, \mathbf{A}_\beta^{-1}), \tag{12}$$

and this allows for the sampling of

$$\left[\sigma^{2(k)}|\cdot\right] \sim IG\left(\alpha_0 + \frac{n}{2}, \beta_0 + \frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^{(k)})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^{(k)})\right). \tag{13}$$

We must also set initial values of α_0, β_0, $\boldsymbol{\mu}_\beta$ and $\boldsymbol{\Sigma}_\beta$. Scanlon (2021) uses $\alpha_0 = \beta_0 = 0.01$, $\boldsymbol{\mu}_\beta = \mathbb{0}_p$ and $\boldsymbol{\Sigma}_\beta = 100\mathbf{I}_p$ for a sample model in which $p = 3$ [4]. This assumes identically and independently distributed β_i which conveniently allows for a trivial computation of $\boldsymbol{\Sigma}_\beta^{-1} = \frac{1}{100}\mathbf{I}_p$. We have used the same setup in the construction of our Gibbs sampler found in A.2.

To improve computational efficiency, we have also identified some matrix computations that are repeated at every iteration and thus only must be computed once. These are $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$. Unfortunately, $\mathbf{A}_\beta^{-1}$ must be computed at every iteration k which is computationally expensive on the order of $O(p^3)$ time.

2.4 Frequentist Methods OLS/Ridge/Lasso

For a linear model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$ as defined previously, the oridnary least squares (OLS) solution to linear regression is the solution to the following minimization problem:

$$\hat{\boldsymbol{\beta}}_{OLS} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}\|_2^2. \tag{14}$$

This problem has a closed form solution that can be computed directly as $\hat{\boldsymbol{\beta}}_{OLS} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$.

In the case of Ridge regression we add a scaled squared ℓ_2-norm $\lambda\|\boldsymbol{\beta}\|_2^2$ penalty term to this minimization. Thus the solution to the Ridge regression optimization problem is defined as:

$$\hat{\beta}_{Ridge} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_2^2. \tag{15}$$

Again, we have a closed form solution to this problem that is $\hat{\boldsymbol{\beta}}_{Ridge} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_p)^{-1}\mathbf{X}^\top\mathbf{y}$.

For LASSO, this penalty term is instead the the scaled ℓ_1-norm $\lambda\|\boldsymbol{\beta}\|_1$, which means that

$$\hat{\beta}_{Lasso} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1. \tag{16}$$

However, unlike OLS and Ridge, Lasso does not have a closed form solution and so the solution must be computed through some form of iterative algorithm.

For Ridge and LASSO, an optimal λ of the can be approximated through k-folds cross-validation. The `sklearn.linear_model.LassoCV` and `sklearn.linear_model.LassoCV` functions automatically perform this cross-validation step with a 5-fold setup when fitting a model. Although likely not the most efficient setup, we will choose to use these functions for this project for the sake of brevity.

As a final remark, we will also be using the `sklearn.linear_model.LinearRegression` function. This function is far more computationally costly than computing the OLS solution through Cholesky decomposition for instance. However, as only one computation of the linear model was needed for each model, it seemed sensible to use this function.

3 Data Simulation

We have created two simulated samples to test our Gibbs sampler against. Both of these models are uncorrelated sparse models, that is they all have zero true covariance values and there are nonsignificant predictors. Each of these models are used to generate equally sized test and training sets.

3.1 Small Model

Our first model is a low p large n model. That is to say, it has a small number of predictors p and a comparatively large number of samples n. In this case the model $p = 6$ and $n = 300$. The response vector $\mathbf{y} \in \mathbb{R}^n$ is sampled as

$$\mathbf{y} = 7\mathbf{x}_1 + 1\mathbf{x}_2 + 0\mathbf{x}_3 + 4\mathbf{x}_4 + 3\mathbf{x}_5 + 0\mathbf{x}_6 + \boldsymbol{\epsilon} \tag{17}$$

where each $\mathbf{X}_j \in \mathbb{R}^n$ is a vector of n observations for $j = 1, ..., j$. Further, every $\epsilon_i \overset{\text{iid}}{\sim} N(0, 9)$ and each $x_{i,j}$ is i.i.d. under some uniform distribution for each value of j. These uniform distributions were chosen arbitrarily and are as follows

$$
\begin{aligned}
x_{i,1} &\sim U(0,30)\\
x_{i,2} &\sim U(20,50)\\
x_{i,3} &\sim U(15,18)\\
x_{i,4} &\sim U(0,50)\\
x_{i,5} &\sim U(0,10)\\
x_{i,6} &\sim U(25,27)
\end{aligned}
$$

for each $i = 1, ..., n$. In this model the regression coefficients were also chosen arbitrarily. The code for this small model generation is found in A.3.

3.2 Large Model

Our second model has a large p and a relatively small n, however $n > p$ still holds. Accordingly, we have selected $p = 1000$ and $n = 1050$ for this model. These p and n values are not on the scale of some of the datasets that may be encountered in the modern day, however they were approaching the limits of our computational abilities due to the time required to run our Gibbs sampler defined above. The response variable $\mathbf{y} \in \mathbb{R}^n$ was constructed as follows

$$\mathbf{y} = \mathbf{X}\mathbf{G}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{18}$$

with design matrix $\mathbf{X} \in \mathbb{R}^{n\times p}$, coefficients $\boldsymbol{\beta} \in \mathbb{R}^n$, error vector $\boldsymbol{\epsilon} \sim N_p(\mathbb{0}_p, \sigma^2\mathbf{I}_p)$ and "trigger" matrix $\mathbf{G} = \text{diag}(g_1, ..., g_p)$ where each $g_j \in \{0, 1\}$. Every $g_j \sim \text{Bernoulli}(p = 0.5)$ which will then randomly set an expected 50% of the β_j values to zero. We use this construction as an easy way of creating sparsity considering the number of parameters we are generating. Further, for this model we are sampling every $x_{ij} \sim U(0, 100)$ and $\boldsymbol{\beta} \in N_{1000}(\boldsymbol{\mu}_\beta, \boldsymbol{\Sigma}_\beta)$ with $\mu_j \sim U(-10, 10)$ and $\boldsymbol{\Sigma} = \text{diag}(\sigma^2_{\beta_1}, ..., \sigma^2_{\beta_p})$ such that every $\sigma^2_{\beta_j} \sim U(0, 20)$. These parameters and distributions were largely chosen arbitrarily in order to produce a random $\mathbf{X}$ and $\boldsymbol{\beta}$. The code for the generation of this larger model is found in A.4.

4 Results & Comparisons

For both the models, our Gibbs sampler was run such that $K = 5000$. We are also taking the first 50% of the data, i.e. the first 2500 samples, as a burn-in period.

4.1 Small Model

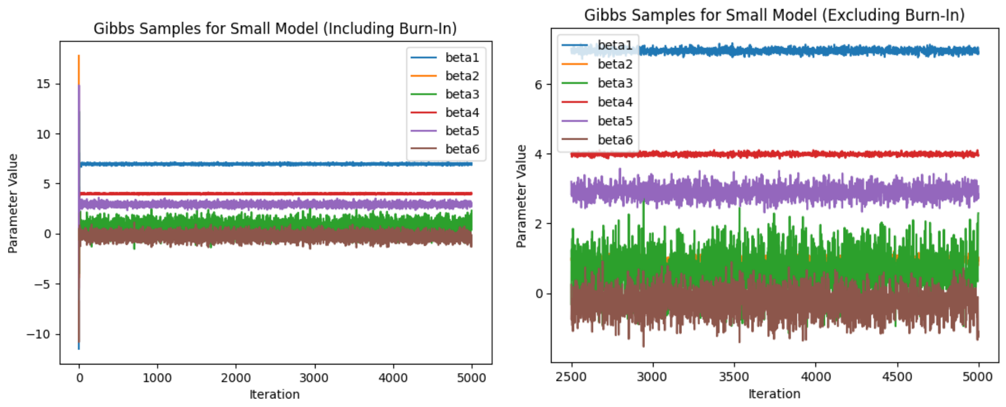

Figure 1: Parameter values for 5,000 iterations of Gibbs Samples with "burn-in" samples included (left) and without "burn-in" samples (right). Notice that with these priors, the algorithm stabilizes extremely quickly.

	True Value	OLS	Ridge	Lasso	Gibbs
$\hat{\beta}_1$	7.0	6.945047	6.942184	6.938708	6.943393
$\hat{\beta}_2$	1.0	0.970877	0.970560	0.961613	0.968138
$\hat{\beta}_3$	0.0	0.833671	0.796489	0.000000	0.581158
$\hat{\beta}_4$	4.0	3.980728	3.980158	3.977486	3.981427
$\hat{\beta}_5$	3.0	2.920467	2.910189	2.814856	2.911239
$\hat{\beta}_6$	0.0	0.491147	0.443575	0.000000	-0.253574
MSE		84.387188	84.344407	83.401259	83.736562

Table 1: Predicted $\hat{\beta}_j$ for each method and MSE of prediction on the test data set. For this comparison $\lambda_{Ridge} = 10.0$ and $\lambda_{LASSO} = 0.92396$. The point estimate for the Gibbs method is a mean of the distribution of the sampled betas after the removal of the burn-in. These distributions are shown as follows Figure 2

.

The point estimates we get for the coefficients are very close to the true values for all methods. Lasso appears to be the best method for detecting parameters with no influence which is what we would expect since one property of LASSO is that it will 'nudge' variables close to zero to be zero. This ultimately leads to the lowest MSE among the methods.

Examining the distributions in Figure 2, we see that all parameter estimates from the other methods used, including the true values, are clearly within a 95% probability interval produced by Gibbs sampling. The only exception to this is that the distribution for the sampled Gibbs $\hat{\beta}_6$ does not appear to include the OLS and Ridge solution. On the whole, it seems that Gibbs sampling estimates of $\hat{\beta}_3$ and $\hat{\beta}_6$ are the furthest away from the true values. This indicates that with nonsignificant betas our Gibbs sampler implementation begins to struggle.

Further, we note that based on the MSEs of prediction seen in Table 1 it seems that our prior assumptions actually did provide some degree improvement to our regression model in this case.

4.2 Large Model

	OLS	Ridge	Lasso	Gibbs
Runtime (s)	0.4880088	0.603109	5.262211	3162.128275
MSE_{pred}	2.791440e+07	2.791382e+07	2.769916e+07	2.791400e+07

Table 2: Runtime for each method and MSE of prediction on the test data set.

The results presented in Table 2 provide a comparison of the four methods we evaluated for estimating regression coefficients: Ordinary Least Squares (OLS), Ridge, Lasso, and Gibbs sampling using the Markov Chain Monte Carlo (MCMC) algorithm. The performance of each method is evaluated based on two metrics: runtime and mean squared error (MSE) of prediction on the test dataset.

From the table, it is evident that the MSE of prediction is of the same order of magnitude for all four methods. Lasso has the lowest MSE, suggesting that it provides the best model fit among the methods compared. In terms of runtime, OLS and Ridge are the fastest methods, with OLS being slightly faster than Ridge. Lasso takes over 10 times longer to execute than OLS, however, we must keep in mind that we ran the LASSO regression with built in 5-fold cross-validation. So the computation speed might actually be much faster than OLS and Ridge.

Examining the Gibbs MCMC algorithm, it is apparent that it has the longest runtime by far, taking approximately 600 times longer than Lasso. This suggests that the algorithm might not be the most efficient choice for this particular problem, especially when runtime is a concern. Gibbs sampling may only be more suitable for high-dimensional data when obtaining credible intervals is crucial, and when the time available for computation is not a limiting factor.

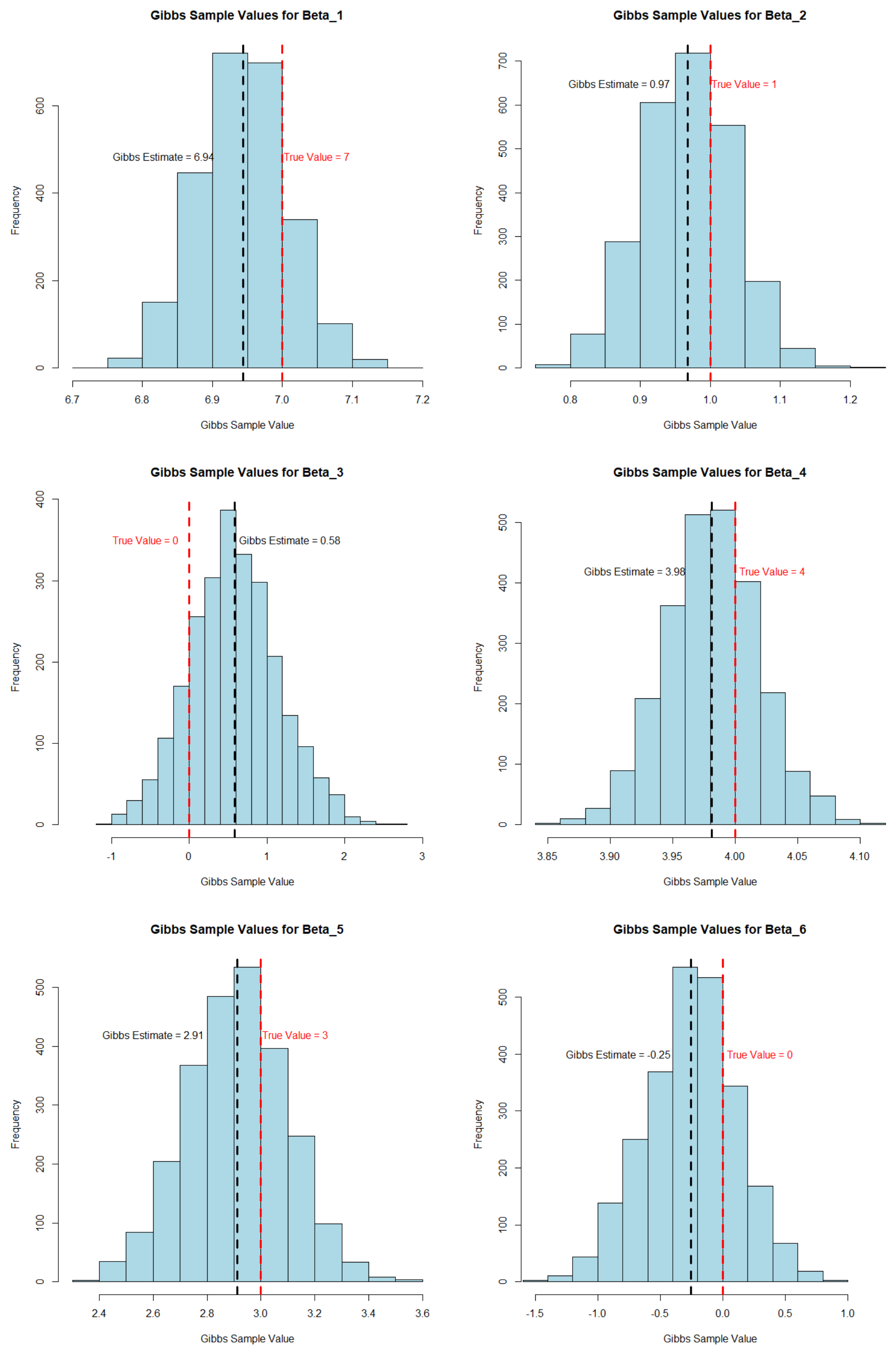

Figure 2: Each plot shows the MCMC samples that were generated for the corresponding coefficient after the first 2500 (50% of the samples) of burn-in were removed. The distributions appear Normal which is expected due to the use of Normal conjugate prior.

Overall, the Lasso method seems to provide the best trade-off between model accuracy and computational efficiency. It achieves the lowest MSE of prediction and takes a reasonable amount of time compared to the other methods. The choice of the most appropriate method ultimately depends on the specific requirements of the analysis, such as the desired level of accuracy and the available computational resources. For this reason, it seems that LASSO is the most effective method to try when p is close to n if point estimates are all that are needed.

In conclusion, the results of this comparison highlight the limitations of the Gibbs MCMC algorithm in terms of computational efficiency for this particular problem. Although it provides a comparable model fit in terms of MSE, its substantially longer runtime makes it less attractive when time is a constraint. However, the Gibbs MCMC algorithm can still be a valuable tool in situations where high-dimensional data is involved and obtaining credible intervals is crucial, provided that the available computational resources and time are sufficient. As always, the choice of the most appropriate method will depend on the specific requirements of the analysis and the characteristics of the dataset and problem at hand.

5 Conclusion

Our results demonstrated that Gibbs MCMC and OLS, Ridge, and Lasso yield comparable estimates and prediction errors for both low- and high-dimensional datasets. However, the difference in between the Bayesian and frequentist methods became substantial when scaling up from a 300x6 dataset to a 1050x1000 dataset. Consequently, we conclude that MCMC Gibbs is likely not the most suitable method for estimating a model with a very large number of parameters, especially when they approach the number of observations.

In future research, it would be valuable to explore potential modifications to our MCMC algorithm that could enhance its performance on high-dimensional datasets. Gibbs sampling, which is already an improvement over the traditional Metropolis-Hastings algorithm when the full conditionals are known, may offer some avenues for optimization. Alternatively, we could consider non-MCMC Bayesian techniques that do not require a significant number of iterations but still provide posterior distributions for obtaining credible intervals for our estimates. However, addressing computationally challenging posterior distributions would necessitate additional considerations.

References

[1] Patrick Breheny. MCMC Methods: Gibbs and Metropolis - University of Iowa.

[2] Arthur E. Hoerl and Robert W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 42(1):80–86, 2000.

[3] Trevor Park and George Casella. The Bayesian Lasso. *Journal of the American Statistical Association*, 103(482):681–686, 2008.

[4] Thuy Scanlon. Evaluating the Efficiency of Markov Chain Monte Carlo Algorithms. Master's thesis, University of Arkansas, Jul 2021.

[5] Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

A Appendix

A.1 Discussing Code Optimization

The Gibbs algorithm itself is an effective and relatively fast algorithm to perform our MCMC sampling in the case of Bayesian regression. However it is not easy to optimize.

In the `beta_prop = np_rng.multivariate_normal(A_inv@b, A_inv)` line of `Gsblr` in A.2, notice that not only do we need to compute $\mathbf{A}^{-1}\mathbf{b}$ but we also need to compute $\mathbf{A}^{-1}$ by itself. If it were only required that we compute $\mathbf{A}^{-1}\mathbf{b}$ within this algorithm, we could instead solve the following linear equation:

$$\mathbf{Az} = \mathbf{b}.$$

This problem can be computed quite readily through an LU, Cholesky or other decomposition since A will be positive definite by construction. For instance if we were to solve this linear equation through LU decomposition such that $\mathbf{A} = \mathbf{PLU}$ it would be as follows:

$$\begin{aligned} \mathbf{Az} &= \mathbf{b}, \\ \mathbf{PLUz} &= \mathbf{b}, \\ \mathbf{LUz} &= \mathbf{P}^{-1}\mathbf{b}, \\ \mathbf{LUz} &= \mathbf{P}^{\top}\mathbf{b}. \end{aligned}$$

Therefore if we let $\mathbf{w} = \mathbf{Uz}$, we could solve for $\mathbf{w}$ in $\mathbf{Lw} = \mathbf{P}^{\top}\mathbf{b}$ by forward substitution and then solve $\mathbf{w} = \mathbf{Uz}$ for $\mathbf{z}$ by backward substitution. Since $\mathbf{z} = \mathbf{A}^{-1}\mathbf{b}$, this would be considerably faster than taking the inverse and performing matrix multiplication afterwards, as was discussed in the beginning of this course. However, we also need $\mathbf{A}^{-1}$ by itself so this methodology cannot be used.

Nevertheless we were able to save some computational time by computing the matrix multiplications that would remain constant for every iteration of the Gibbs sampler before the loop began so that they were not unnecessarily repeated. Although the majority of the computational complexity in this algorithm came from the matrix inversion, this small optimization did save a nontrivial amount of time.

As mentioned previously in Section 2.4 we also would've been able to decrease the computational time in calculating the OLS solution by using a decomposition like the LU method mentioned above. The setup would be the same however now $\mathbf{A}$ would be replaced with $(\mathbf{X}^{\top}\mathbf{X})^{-1}$, $\mathbf{b}$ would be replaced with $\mathbf{X}^{\top}\mathbf{Y}$ and $\mathbf{z}$ would be $\boldsymbol{\beta}$.

A.2 Gibbs Sampler (gsblr.py)

```
import pandas as pd
import numpy as np
import scipy as sp
from scipy import stats
import math as m

def gibbs_bayesian_linreg(X, y, niter, np_rng=None):
    # Check that input arrays have the correct dimensions
    assert len(y) == X.shape[0]
    assert np_rng is not None

    # Get the number of data points (n) and the number of features (p)
    n = X.shape[0]
    p = X.shape[1]

    # Create empty arrays to store estimates of beta coefficients and sigma2 values
    beta_estimates = np.empty((p, niter))
    sigma2_estimates = np.empty(niter)

    # Set initial values for beta's mean and variance
    ibm = np.zeros(p)
    ibv = 100*np.eye(p)

    # Set initial values for the shape and scale parameters of the inverse gamma distributi
    a_0 = 0.01
    bs_0 = 0.01
    # Sample initial value of beta and sigma2.
    beta_estimates[:,0] = np_rng.multivariate_normal(ibm, ibv)
    sigma2_estimates[0] = sp.stats.invgamma.rvs(a=a_0, scale=bs_0)

    # Precompute values for matrix multiplications to improve efficiency
    XtX = X.T @ X
    Xty = X.T @ y
    bvi = 0.01*np.eye(p)
    bvi_m = bvi@ibm

    # Iterate through Gibbs sampling
    for i in range(1, niter):
        sigma2_inv = 1 / sigma2_estimates[i-1]

        # Compute A and b matrices for the current iteration
        A = XtX * sigma2_inv + bvi
        b = X.T@y * sigma2_inv + bvi_m
        A_inv = np.linalg.inv(A)

        # Generate a new proposal for beta
        beta_prop = np_rng.multivariate_normal(A_inv@b, A_inv)

        # Generate a new proposal for sigma2
        diff = np.reshape(y - X@beta_prop, (n, 1))
        shape = a_0 + n/2
        scale = bs_0 + 1/2 * diff.T@diff
        sigma2_prop = sp.stats.invgamma.rvs(a=shape, scale=scale)

        # Store the new proposals in the arrays
        beta_estimates[:, i] = beta_prop
        sigma2_estimates[i] = sigma2_prop

    return beta_estimates, sigma2_estimates

class Gsblr:
    def __init__(self, rseed=None, burn_prop=0.5):
        # Ensure the burn-in proportion is within the valid range
        assert burn_prop > 0 and burn_prop < 1
        self.burn_prop = burn_prop
        self.samples = None
        self.burn_samples = None
        self.coef = None
        # Set up random number generator
        self.rng = np.random.default_rng(seed=rseed)
        np.random.seed(seed=rseed)

    def fit(self, X, y, niter=5000):
        # Perform Gibbs sampling for the given data and number of iterations
        beta, sigma2 = gibbs_bayesian_linreg(X=X, y=y, niter=niter, np_rng=self.rng)

        # Store the sampled values in a DataFrame
        gibbs_dict = {'beta'+str(i):beta[i-1] for i in range(1, beta.shape[0]+1)}
        gibbs_dict.update({'sigma2':sigma2})
        self.samples = pd.DataFrame(gibbs_dict)

        # Compute the number of burn-in iterations
        burn_num = int(np.floor(self.samples.shape[0] * self.burn_prop))
        # Store samples after removing burn-in iterations
        self.burn_samples = self.samples.iloc[burn_num:]
        # Compute and store the coefficients (mean of the remaining samples)
        self.coef = self.samples.iloc[burn_num:].mean(axis=0)

    def get_samples(self, var=False, remove_burn=True):
        # Return the samples with different options:
        # - var: include sigma2 in the output
        # - remove_burn: remove the burn-in iterations
        if not var and remove_burn:
            return self.burn_samples.iloc[:,:-1]
        elif not var and not remove_burn:
            return self.samples.iloc[:,:-1]
        elif var and remove_burn:
            return self.burn_samples
        else:
            return self.samples

    def get_coef(self, var=False):
        # Return the coefficients (mean of the samples) with or without sigma2
        if not var:
            return self.coef[:-1]
        else:
            return self.coef

    def predict(self, X_test):
        # Predict the target variable for the given test data
        betas = self.get_coef().to_numpy()
        betas.reshape(X_test.shape[1], 1)
        y_hat = X_test@betas
        return y_hat
```

A.3 Small Model Generation (small_model.py)

```
'''
Simulates a model:
y = B.T @ X + e
y = 7X_i1 + 1X_i2 + 0X_i3 + 4X_i4 + 3X_i5 + 0X_i6 + e_i

e ~ N(0,9)
b1 = 7, b2 = 1, b3 = 0, b4 = 4, b5 = 3, b6 = 0

Each X_i is uniformly distributed with some set of parameters.
'''

# Import modules.
import numpy as np

# Create instance of generator.
rng = np.random.default_rng(141)
```

```
# Define n, p.
n = 300
p = 6

# Define betas.
betas = np.array([7, 1, 0, 4, 3, 0])

# Function to sample x, y.
def sampler(betas = betas, n = n):

    # sample each x vector.
    x1 = rng.uniform(0,30,n)
    x2 = rng.uniform(20,50,n)
    x3 = rng.uniform(15,18,n)
    x4 = rng.uniform(0,50,n)
    x5 = rng.uniform(0,10,n)
    x6 = rng.uniform(25,27,n)

    # Concatenate x vectors into X matrix.
    X = np.column_stack((x1, x2, x3, x4, x5, x6))

    # Sample epsilon.
    e = rng.normal(0,9,n)

    # Generate y.
    y = X @ betas[:, None] + e[:, None]

    # output X,y
    return X, y

# Generate training, testing data.
X_train, y_train = sampler()
X_test, y_test = sampler()

# Save data.
np.savetxt('data/small_model/betas.txt', betas, delimiter=',')
np.savetxt('data/small_model/X_train.txt', X_train, delimiter=',')
np.savetxt('data/small_model/y_train.txt', y_train, delimiter=',')
np.savetxt('data/small_model/X_test.txt', X_test, delimiter=',')
np.savetxt('data/small_model/y_test.txt', y_test, delimiter=',')
```

A.4 Large Model Generation (large_model.py)

```
# import modules
import numpy as np

# create instance of generator
rng = np.random.default_rng(141)

# define n, p
n = 1050
p = 1000
```

```
## generate betas (vector of length p)
# covariance matrix
betas_cov = np.diag(rng.uniform(0,20, p))
# means vector
betas_means = rng.uniform(-10,10,p)
# draw betas
betas = rng.multivariate_normal(betas_means, betas_cov)

# Function to sample X, y
def sampler(betas=betas, n=n, p=p):
    ## generate X (n rows, p columns)
    X = rng.uniform(0,100,(n,p))

    ## generate G
    # a matrix with 1's and 0's on diag
    # 0's make corresponding betas insignificant
    G = np.diag(rng.binomial(n=1, p=0.5, size=p))

    ## generate error terms
    # covariance matrix
    err_cov = 25 * np.identity(n)
    # means vector
    err_mean = np.zeros(n)
    # error terms
    err = rng.multivariate_normal(err_mean, err_cov)

    ## generate Y
    y = X @ (G @ betas) + err

    # output X, y, G
    return X, y, G

# Generate training, testing data.
X_train, y_train, G_train = sampler()
X_test, y_test, G_test = sampler()

## store simulated data
# beta
np.savetxt('data/large_model/betas.txt', betas, delimiter=',')
np.savetxt('data/large_model/betas_cov.txt', betas_cov, delimiter=',')
np.savetxt('data/large_model/betas_means.txt', betas_means, delimiter=',')
# train
np.savetxt('data/large_model/G_train.txt', G_train, delimiter=',')
np.savetxt('data/large_model/X_train.txt', X_train, delimiter=',')
np.savetxt('data/large_model/y_train.txt', y_train, delimiter=',')
# test
np.savetxt('data/large_model/G_test.txt', G_test, delimiter=',')
np.savetxt('data/large_model/X_test.txt', X_test, delimiter=',')
np.savetxt('data/large_model/y_test.txt', y_test, delimiter=',')
```

A.5 Model Computations

A.5.1 Small Model Computations (sml_models.py)

```
'''
```

```
Please note that while the original code was in a Jupyter Notebook, we have condensed it i
'''

# import modules
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV
from sklearn.metrics import mean_squared_error
import gsblr

# import data
X_train = np.loadtxt('data/small_model/X_train.txt', delimiter=',')
y_train = np.loadtxt('data/small_model/y_train.txt', delimiter=',')
X_test = np.loadtxt('data/small_model/X_test.txt', delimiter=',')
y_test = np.loadtxt('data/small_model/y_test.txt', delimiter=',')
betas = np.loadtxt('data/small_model/betas.txt', delimiter=',')

## LINEAR REGRESSION (OLS) ------------------------------------------------
# fit linear regression model
linreg_model = LinearRegression().fit(X_train, y_train)
# linear regression coefficients
linreg_coef = linreg_model.coef_
# predict with linreg model
linreg_pred = linreg_model.predict(X_test)
# MSE for linreg model
linreg_mse = mean_squared_error(y_test, linreg_pred)
## -----------------------------------------------------------------------

## RIDGE -----------------------------------------------------------------
# fit ridge regression model with cross validation
ridge_model = RidgeCV().fit(X_train, y_train)
# ridge "alpha" parameter (lambda)
ridge_model.alpha_
# ridge coefficients
ridge_coef = ridge_model.coef_
# predict with ridge model
ridge_pred = ridge_model.predict(X_test)
# MSE for ridge model
ridge_mse = mean_squared_error(y_test, ridge_pred)
## -----------------------------------------------------------------------

## LASSO -----------------------------------------------------------------
# fit lasso regression model with cross validation
lasso_model = LassoCV(random_state=141).fit(X_train, y_train)
# lasso "alpha" parameter (lambda)
lasso_model.alpha_
# lasso coefficients
lasso_coef = lasso_model.coef_
# predict with lasso model
lasso_pred = lasso_model.predict(X_test)
# MSE for lasso model
lasso_mse = mean_squared_error(y_test, lasso_pred)
## -----------------------------------------------------------------------
## GIBBS ------------------------------------------------------------
# initialize gibbs sampler
gibbs = gsblr.Gsblr(rseed=141)
# fit gibbs sampler
gibbs.fit(X_train, y_train)
# gibbs coefficients
gibbs_coef = gibbs.get_coef().values
# predict with gibbs
gibbs_pred = gibbs.predict(X_test)
# MSE for gibbs
gibbs_mse = mean_squared_error(y_test, gibbs_pred)
## ------------------------------------------------------------------

## DATA SUMMARY ------------------------------------------------------
# create dataframe of all regression coefficients
coef_df = pd.DataFrame({'true_coef': betas,
                        'linreg_coef': linreg_coef,
                        'ridge_coef': ridge_coef,
                        'lasso_coef': lasso_coef,
                        'gibbs_coef': gibbs_coef}
                        )

coef_df
# create dataframe of MSE for each model
mse_df = pd.Series(data=[linreg_mse, ridge_mse, lasso_mse, gibbs_mse],
                   index= ['linreg_mse', 'ridge_mse', 'lasso_mse', 'gibbs_mse'])

mse_df

# dataframe of gibbs samples (excluding burned samples)
gibbs_samples = gibbs.get_samples()

# dataframe of gibbs samples (including burned samples)
gibbs_samples_all = gibbs.get_samples(remove_burn=False)

# save to csv
gibbs_samples.to_csv('gibbs_samples.csv')
gibbs_samples_all.to_csv('gibbs_samples_all.csv')
coef_df.to_csv('coefs_data.csv')
## ------------------------------------------------------------------
```

A.5.2 Large Model Computations (lrg_models.py)

```
'''
Please note that while the original code was in a Jupyter Notebook, we have condensed it i
'''

# import modules
import numpy as np
import pandas as pd
import time
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV
from sklearn.metrics import mean_squared_error
import gsblr
```

```
# import data
X_train = np.loadtxt('data/large_model/X_train.txt', delimiter=',')
y_train = np.loadtxt('data/large_model/y_train.txt', delimiter=',')
X_test = np.loadtxt('data/large_model/X_test.txt', delimiter=',')
y_test = np.loadtxt('data/large_model/y_test.txt', delimiter=',')

## LINEAR REGRESSION -----------------------------------------------
# begin timer
start = time.time()
# fit linear regression model
linreg_model = LinearRegression().fit(X_train, y_train)
# end timer
linreg_time = time.time() - start
# linear regression coefficients
linreg_coef = linreg_model.coef_
# predict with linreg model
linreg_pred = linreg_model.predict(X_test)
# MSE for linreg model
linreg_mse = mean_squared_error(y_test, linreg_pred)
## -----------------------------------------------------------------

## RIDGE -----------------------------------------------------------
# begin timer
start = time.time()
# fit ridge regression model with cross validation
ridge_model = RidgeCV().fit(X_train, y_train)
# end timer
ridge_time = time.time() - start
# ridge "alpha" parameter (lambda)
ridge_model.alpha_
# ridge coefficients
ridge_coef = ridge_model.coef_
# predict with ridge model
ridge_pred = ridge_model.predict(X_test)
# MSE for ridge model
ridge_mse = mean_squared_error(y_test, ridge_pred)
## -----------------------------------------------------------------

## LASSO -----------------------------------------------------------
# begin timer
start = time.time()
# fit lasso regression model with cross validation
lasso_model = LassoCV(random_state=141).fit(X_train, y_train)
# end timer
lasso_time = time.time() - start
# lasso "alpha" parameter (lambda)
lasso_model.alpha_
# lasso coefficients
lasso_coef = lasso_model.coef_
# predict with lasso model
lasso_pred = lasso_model.predict(X_test)
# MSE for lasso model
lasso_mse = mean_squared_error(y_test, lasso_pred)
## -----------------------------------------------------------------
## GIBBS ------------------------------------------------------------
# 1) Iterations: 5000, Burn proportion: 0.5
# begin timer
start = time.time()
# initialize gibbs sampler
gibbs = gsblr.Gsblr(rseed=141)
# fit gibbs sampler
gibbs.fit(X_train, y_train)
# end timer
gibbs_time = time.time() - start
# gibbs coefficients
gibbs_coef = gibbs.get_coef().values
# predict with gibbs
gibbs_pred = gibbs.predict(X_test)
# MSE for gibbs
gibbs_mse = mean_squared_error(y_test, gibbs_pred)

# 2) Iterations: 100, Burn proportion: 0.3
# begin timer
start = time.time()
#initialize gibbs sampler
gibbs_2 = gsblr.Gsblr(rseed=141, burn_prop=0.3)
# fit gibbs_2 sampler
gibbs_2.fit(X_train, y_train, niter= 100)
# end time
gibbs_time_2 = time.time() - start
# predict with gibbs_2
gibbs_2_pred = gibbs_2.predict(X_test)
# MSE for gibbs_2
gibbs_2_mse = mean_squared_error(y_test, gibbs_2_pred)
# results for gibbs_2
pd.Series({'Runtime': gibbs_time_2,
           'MSE': gibbs_2_mse})
## -----------------------------------------------------------------

## DATA SUMMARY -----------------------------------------------------
# create datafram to summarize results
lrg_results = pd.DataFrame({
    'Method': ['Linreg', 'Ridge', 'LASSO', 'Gibbs'],
    'MSE': [linreg_mse, ridge_mse, lasso_mse, gibbs_mse],
    'runtime': [linreg_time, ridge_time, lasso_time, gibbs_time]
})
# view results
lrg_results
# save results
lrg_results.to_csv('lrg_model_results.csv')
```

A.6 Graphics Generation

A.6.1 Histograms of Gibbs Estimates (histograms.R)

```
# libraries
library(readr)
```

```
# read in data
gibbs_samples <- read_csv("gibbs_samples.csv")[,-1]

# histogram for beta1
hist(gibbs_samples$beta1,
     col = 'lightblue',
     labels = FALSE,
     main = "Gibbs Sample Values for Beta 1",
     xlab = "Gibbs Sample Value")
abline(v = coefs_data$true_coef[1],
       col = "red",
       lwd = 3,
       lty = 2)
text(x=7.05, y= 485, col = 'red', "True Value = 7")
abline(v = coefs_data$gibbs_coef[1],
       col = "black",
       lwd = 3,
       lty = 2)
text(x= 6.83, y=485, col = 'black', lwd = 10, labels = "Gibbs Estimate = 6.94")


# histogram for beta2
hist(gibbs_samples$beta2,
     col = 'lightblue',
     labels = FALSE,
     main = "Gibbs Sample Values for Beta 2",
     xlab = "Gibbs Sample Value")
abline(v = coefs_data$true_coef[2],
       col = "red",
       lwd = 3,
       lty = 2)
text(x=1.05, y= 650, col = 'red', "True Value = 1")
abline(v = coefs_data$gibbs_coef[2],
       col = "black",
       lwd = 3,
       lty = 2)
text(x= 0.87, y=650, col = 'black', labels = "Gibbs Estimate = 0.97")


# histogram for beta3
hist(gibbs_samples$beta3,
     col = 'lightblue',
     labels = FALSE,
     main = "Gibbs Sample Values for Beta 3",
     xlim = c(-1.5,3),
     breaks = 18,
     xlab = "Gibbs Sample Value")
abline(v = coefs_data$true_coef[3],
       col = "red",
       lwd = 3,
       lty = 2)
text(x=-0.55, y= 350, col = 'red', "True Value = 0")
abline(v = coefs_data$gibbs_coef[3],
       col = "black",
        lwd = 3,
        lty = 2)
text(x= 1.3, y=350, col = 'black', labels = "Gibbs Estimate = 0.58")


# histogram for beta4
hist(gibbs_samples$beta4,
     col = 'lightblue',
     labels = FALSE,
     main = "Gibbs Sample Values for Beta 4",
     xlab = "Gibbs Sample Value")
abline(v = coefs_data$true_coef[4],
        col = "red",
        lwd = 3,
        lty = 2)
text(x=4.03, y= 420, col = 'red', "True Value = 4")
abline(v = coefs_data$gibbs_coef[4],
        col = "black",
        lwd = 3,
        lty = 2)
text(x= 3.92, y=420, col = 'black', labels = "Gibbs Estimate = 3.98")


# histogram for beta5
hist(gibbs_samples$beta5,
     col = 'lightblue',
     labels = FALSE,
     main = "Gibbs Sample Values for Beta 5",
     xlab = "Gibbs Sample Value")
abline(v = coefs_data$true_coef[5],
        col = "red",
        lwd = 3,
        lty = 2)
text(x=3.13, y= 420, col = 'red', "True Value = 3")
abline(v = coefs_data$gibbs_coef[5],
        col = "black",
        lwd = 3,
        lty = 2)
text(x= 2.6, y=420, col = 'black', labels = "Gibbs Estimate = 2.91")


# histogram for beta6
hist(gibbs_samples$beta6,
     col = 'lightblue',
     labels = FALSE,
     main = "Gibbs Sample Values for Beta 6",
     xlim = c(-1.5,1.3),
     breaks = 10,
     xlab = "Gibbs Sample Value")
abline(v = coefs_data$true_coef[6],
        col = "red",
        lwd = 3,
        lty = 2)
text(x=0.3, y= 400, col = 'red', "True Value = 0")
```

```
abline(v = coefs_data$gibbs_coef[6],
        col = "black",
        lwd = 3,
        lty = 2)
text(x= -0.83, y=400, col = 'black', labels = "Gibbs Estimate = -0.25")
```

A.6.2 Gibbs Samples Plots (gibbs_samples_plot.py)

```
'''
Please note that while the original code was in a Jupyter Notebook, we have condensed it i
'''

# import modules
import numpy as np
import pandas as pd

# read in data
gibbs_samples = pd.read_csv('../gibbs_samples.csv', index_col=0)
gibbs_samples_all = pd.read_csv('../gibbs_samples_all.csv', index_col=0)

# create plot with burn-in
ax = gibbs_samples_all.plot(title="Gibbs Samples for Small Model (Including Burn-In)")
ax.set_xlabel("Iteration")
ax.set_ylabel("Parameter Value")

# create plot without burn-in
ax = gibbs_samples.plot(title="Gibbs Samples for Small Model (Excluding Burn-In)")
ax.set_xlabel("Iteration")
ax.set_ylabel("Parameter Value")
```