Reversible Jump Markov Chain Monte Carlo Algorithm for Model Selection in Linear Regression

NICK WIBERT, Florida State University

Introduction

Linear regression is a widely studied and implemented statistical method for describing the relationship between some response variable and a set of predictors based on various assumptions of linearity and normality. There are many extensions and adjustments to linear regression that have been proposed over the years to handle the specific challenges of a given application, but one of the more broad topics which is relevant to all applications of linear regression is the topic of variable selection and/or model selection. Given a response and a set of predictors, how can we choose the optimal model? Which predictors should be included or excluded, should these predictors be transformed, etc.?

We restrict our focus here to the specific problem of choosing the optimal subset of predictors for a standard linear model (which we will refer to from here on out as the problem of **model selection**). We will simplify the problem even further by assuming the set of predictors X is already arranged in order of significance for explaining the response y. So, given a set of m predictors, we wish to algorithmically determine the optimal number of predictors n to fit a linear model such that $(n < m)$.

Methodology

Problem Statement

We are interested in the standard linear model described by

$$y = \sum_{i=1}^{n} x_i b_i + \epsilon$$

where $n < m$, x_i represents the predictors, y represents the response, and ϵ is the random noise. We are given k independent measurements which we will denote as $\mathbf{y} \in \mathbb{R}^k$, $\mathbf{X} \in \mathbb{R}^{k \times m}$, and $\epsilon \in \mathbb{R}^k$. Our goal is to estimate the optimal n given these data, which naturally calls for a Bayesian formulation. More specifically, we are looking for a Bayesian solution to the joint estimation of $\{n, b_1, b_2, \dots, b_n\}$.

Approach

Let $\mathbf{X}_n \in \mathbb{R}^{k \times n}$ denote the first n predictors, and $\mathbf{b}_n = (b_1, b_2, \dots, b_n)^T \in \mathbb{R}^n$ represent the n corresponding coefficients for the linear model. We will define the likelihood of our data as well as the priors on $\mathbf{b}_n$ and n as follows:

$$f(y \mid n, \mathbf{b}_n) = \left(\frac{1}{\sqrt{2\pi\sigma_0^2}} \right)^k \exp\left\{ \frac{-1}{2\sigma_0^2} \|\mathbf{y} - \mathbf{X}_n \mathbf{b}_n\|^2 \right\}$$

$$f(\mathbf{b}_n \mid n) = \left(\frac{1}{\sqrt{2\pi\sigma_p^2}} \right)^k \exp\left\{ \frac{-1}{2\sigma_p^2} \|\mathbf{b}_n - \mu_b\|^2 \right\}$$

$$f(n) = \frac{1}{m}$$

Thus, the joint posterior density that we wish to estimate can be described by

$$f(n, \mathbf{b}_n \mid \mathbf{y}) \propto f(\mathbf{y} \mid n, \mathbf{b}_n) f(\mathbf{b}_n \mid n) f(n)$$

We will employ the Reversible Jump Markov Chain Monte Carlo (RJMCMC) algorithm to sample from this posterior density. Since we are interested in estimating the dimension of the coefficient vector $\mathbf{b}_n$, the dimension of the space from which we are sampling will be constantly changing. The RJMCMC algorithm is an extension to the standard MCMC methodology which allows for simulating a posterior distribution on spaces of changing dimensions ("jumping" from dimension to dimension), so it is perfectly suited for the problem of estimating the number of parameters in a linear model. Because RJMCMC is a generalization of the Metropolis-Hastings algorithm, it exploits the characteristics of Markov chains and Monte Carlo sampling such that we can construct a chain which will converge to the true posterior density formulated above. Therefore if we run the algorithm for enough iterations, we will (approximately) be drawing samples from this joint posterior density.

Algorithm

Let $(n, \mathbf{b}_n)$ represent the current sample from the posterior. Furthermore, we define one more likelihood function of a d-dimensional normal random vector $\mathbf{u}$ as

$$h(\mathbf{u}) = \left(\frac{1}{\sqrt{2\pi\sigma_r^2}} \right)^d \exp\left\{ \frac{-1}{2\sigma_r^2} \|\mathbf{u}\|^2 \right\}$$

The RJMCMC algorithm is implemented using the steps below.

Reversible Jump Markov Chain Monte Carlo Algorithm

1) Select a candidate for the number of parameters in the model (call this n^*) by drawing a sample from $f(n)$.
2) If $n^* \geq n$, generate a random vector $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \sigma_r I_{n^*})$ (where $\mathbf{0} \in \mathbb{R}^{n^*}$ and I_{n^*} is the n^*-dimensional identity matrix). Let $\mathbf{u}_1$ denote the first n elements of $\mathbf{u}$, and $\mathbf{u}_2$ denote the remaining $n^* - n$ elements. We calculate a candidate coefficient vector $\mathbf{b}_{n^*}$ using the formula

$$\mathbf{b}_{n^*} = \begin{bmatrix} \mathbf{b}_n \\ 0 \end{bmatrix} + \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix}$$

   Compute the likelihoods of $\mathbf{u}$ and $\mathbf{u}_1$, denoting them as $h_1 = h(\mathbf{u})$ and $h_2 = h(\mathbf{u}_1)$.
3) If $n^* < n$, generate a random vector $\mathbf{u}_1 \sim \mathcal{N}(\mathbf{0}, \sigma_r I_{n^*})$. Let $\mathbf{b}_n^1$ denote the first n^* elements of $\mathbf{b}_n$, and $\mathbf{b}_n^2$ denote the remaining $n - n^*$ elements. Set $\mathbf{u} = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{b}_n^2 \end{bmatrix}$. We calculate a candidate coefficient vector $\mathbf{b}_{n^*}$ using the formula

$$\mathbf{b}_{n^*} = \mathbf{b}_1^n + \mathbf{u}_1$$

   Compute the likelihoods of $\mathbf{u}$ and $\mathbf{u}_1$, denoting them as $h_1 = h(\mathbf{u}_1)$ and $h_2 = h(\mathbf{u})$.
4) Let $E_n = \frac{1}{2\sigma_0^2} \|\mathbf{y} - \mathbf{X}_n \mathbf{b}_n\|^2$. Compute the accept-reject function:

$$\begin{aligned} \rho &= \min\left\{ 1, \frac{f(n^*, \mathbf{b}_{n^*} \mid \mathbf{y}) h_2}{f(n, \mathbf{b}_n \mid \mathbf{y}) h_1} \right\} = \min\left\{ 1, \frac{f(\mathbf{y} \mid n^*, \mathbf{b}_{n^*}) f(\mathbf{b}_{n^*} \mid n^*) h_2}{f(\mathbf{y} \mid n, \mathbf{b}_n) f(\mathbf{b}_n \mid n) h_1} \right\} \\ &= \min\left\{ 1, \frac{1}{h_2} \exp\{-(E_{n^*} - E_n)\} (2\pi\sigma_p^2)^{(n-n^*)/2} \exp\left\{ \frac{-1}{2\sigma_p^2} (\|\mathbf{b}_{n^*} - \mu_b\|^2 - \|\mathbf{b}_n - \mu_b\|^2 \right\} \right\} \end{aligned}$$

5) Generate $U \sim \mathcal{U}[0, 1]$. If $U < \rho$, we accept the candidate, and then set $(n, \mathbf{b}_n) = (n^*, \mathbf{b}_{n^*})$. Otherwise, keep $(n, \mathbf{b}_n)$ and return to Step 1.

Experimental Results

Using Python, we implemented this algorithm on a simulated dataset (See Appendix A for the code used to simulate the data as well as run the algorithm). The simulation was performed using a set of $m = 10$ predictors, each with $k = 10$ independent observations. The std. deviation for the likelihood of $\mathbf{y}$ was set as $\sigma_0 = 0.2$; the std. deviation for the prior on $\mathbf{b}_n$ was set as $\sigma_p = 0.3$; and the std. deviation for the likelihood of the random vector $\mathbf{u}$ was set to be $\sigma_r = 0.2$. The mean for the prior on $\mathbf{b}_n$ was set as an n-dimensional vector of 2's.

To simulate the data, a "true" value for n (let's call this n_0) was first generated from $f(n)$. The predictor set was generated as a k-by-m matrix of samples from a standard normal distribution, then scaled by a factor of 5. The "true" coefficient vector b was generated by generating an n_0-dimensional vector of samples from a standard normal, scaling the vector by σ_p, and then adding μ_b. Finally, the response $\mathbf{y}$ was generated using the standard linear model formula from the Problem Statement, using only the first n_0 predictors.

This process was repeated 10 times, each time resulting in a new draw for the "true" n_0 and a new dataset $(\mathbf{y}, \mathbf{X})$. The RJMCMC algorithm was run for each dataset to obtain $N = 100,000$ samples from the posterior. The histograms in Figure 1 were plotted using the chain of n values produced by the algorithm, each acting as an estimate of the posterior probability of n given the data (i.e. $f(n \mid \mathbf{y})$). Figure 2 displays convergence plots for each of these chains.

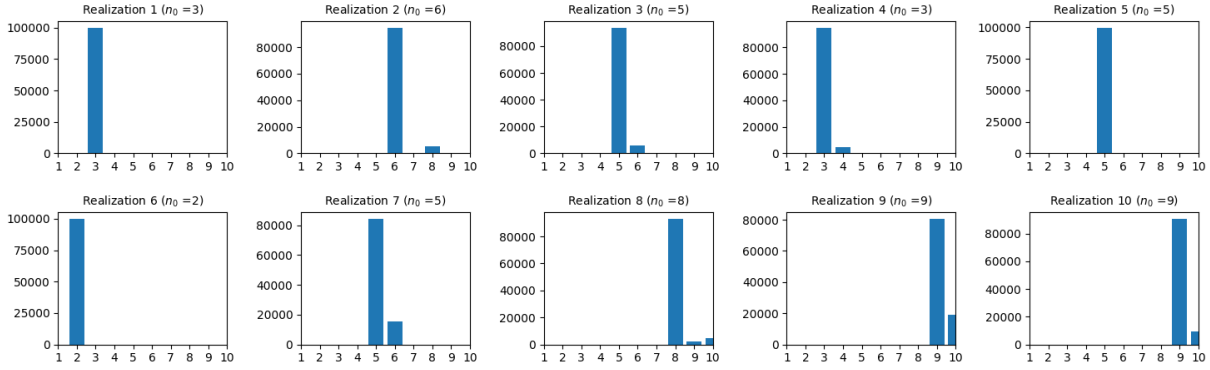

Figure 1: 10 realizations of RJMCMC with $N = 100,000$

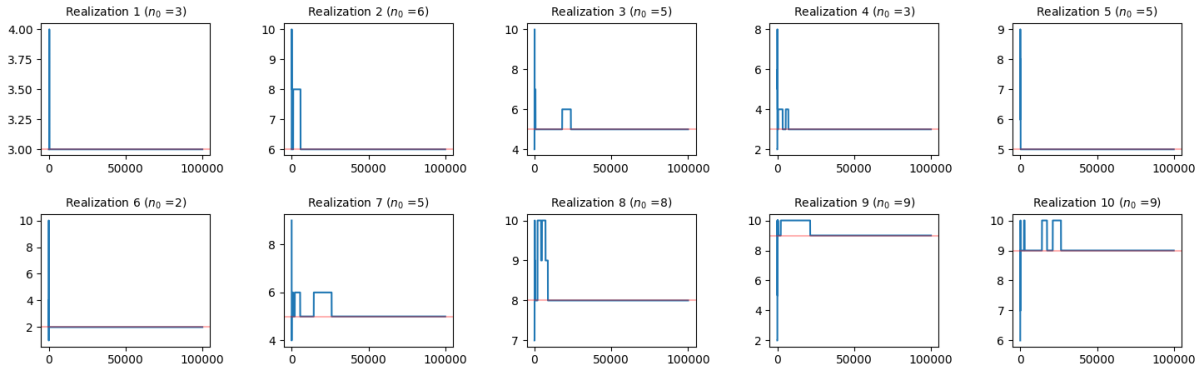

Figure 2: Convergence plots for 10 realizations of RJMCMC

In the plot titles within the figures on the previous page, we see the "true" n_0 which was selected for the given realization. Observe that the mode of each histogram is exactly equal to the optimal number of predictors for the given data. Furthermore, we see clear convergence of the chain of n values to the true n_0 in each realization.

Conclusion

In this experimental setting, we defined $\mathbf{X}$ and $\mathbf{y}$ to have a simple and clearly linear relationship, so we should hope that this algorithm would have no trouble estimating the optimal number of parameters with very high accuracy. The estimates above show that the posterior density $f(n \mid \mathbf{y})$ in this setting is essentially a probability distribution whose support consists of one value ($f(n = n_0 \mid \mathbf{y}) = 1$ and 0 for all other values of n). In fact, if we introduce a sizeable burn-in to our algorithm (say 30,000), it is even more clear (Figure 3).

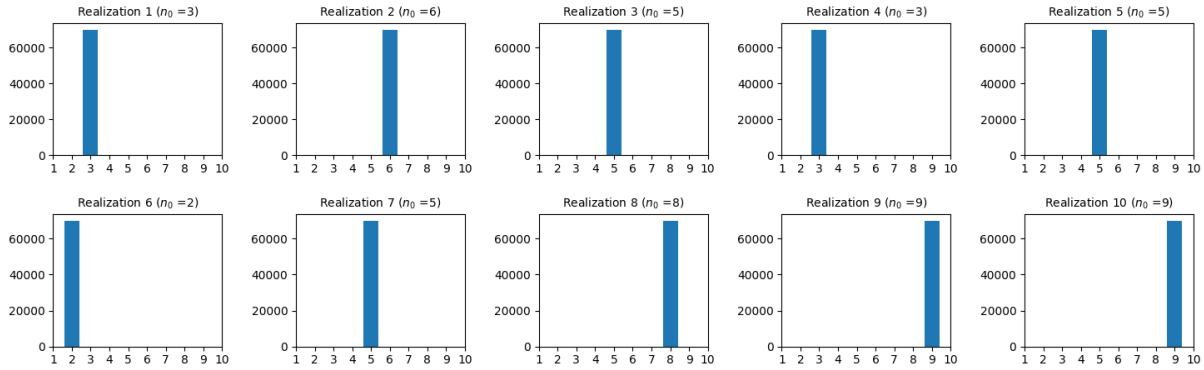

Figure 3: 10 realizations of RJMCMC with $N = 100,000$ and a burn-in of $30,000$

Incorrect estimates for n_0 occur earlier on in the chain before converging to the true n_0. During the simulation process, some histograms would occasionally display the wrong value as the mode; however, in these cases, the true value was always the second most frequent. This is simply a result of the randomness that is part of the nature of Markov chains and Monte Carlo sampling, but if we could in practice run this algorithm infinitely, it would have a 100% success rate. We can combat this variability by increasing the number of iterations (within reason) and introducing a burn-in to throw out the more erratic samples from the beginning of the chain (as seen in the figure above, this increases clarity of our results).

The results above demonstrate RJMCMC as a powerful algorithm for model selection when the optimal number of parameters is unknown. It is important to note that the results shown here were obtained in a very limited setting; not only due to the simplicity of the linear model, but also the assumptions placed upon the data already being ordered. Further research could incorporate the order of these predictors into the experiment to investigate the strength of RJMCMC in a more ambiguous, but still linear setting. The performance of the algorithm in estimating parameter counts for more complicated models or even in a non-linear setting are some other suggested areas of further study.

Appendix A: RJMCMC Simulation in Python

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg
from matplotlib.ticker import MaxNLocator
```

Functions

In [ ]:
```
# Acceptance-rejection
def accept_reject(X, y, b, b_star, h1, h2, L_std, prior_std):
    n = b.shape[0]
    n_star = b_star.shape[0]

    E1 = (1/(2*L_std**2)) * scipy.linalg.norm(y - X[:,:n_star]@b_star)**2
    E2 = (1/(2*L_std**2)) * scipy.linalg.norm(y - X[:,:n]@b)**2
    numerator = (np.exp(-(E1-E2)) * (2*np.pi*prior_std**2)**((n-n_star)/2) \
                 * np.exp((-1/(2*prior_std**2))*(scipy.linalg.norm(b_star - 2*np.ones((n_star,1)))**2 \
                                                - scipy.linalg.norm(b - 2*np.ones((n,1)))**2)) * h2)
    frac = numerator / h1

    return np.min([1., frac])

# Likelihood of u
def h(u, n, var = u_std**2):
    return (1/np.sqrt(2*np.pi*var))**n * np.exp((-1/(2*var))*scipy.linalg.norm(u)**2)
```

Simulation

In [ ]:
```
n0_list = []
posterior_estimates = []

for i in range(10):
    # Parameters for simulation
    m = 10
    n0 = np.ceil(np.random.rand()*m).astype('int')
    n0_list.append(n0)
    k = 10
    L_std = 0.2
    prior_std = 0.3
    prior_mu = 2*np.ones((n0,1))
    b = prior_mu + prior_std*np.random.randn(n0,1)
    u_std = 0.2
    N = 100_000

    # generate X and y
    X = 5*np.random.randn(k,m)
    y = X[:,:n0]@b + L_std*np.random.randn(k,1)

    posterior_n = [np.ceil(np.random.rand()*m).astype('int')]
    n = posterior_n[-1]
    posterior_b = [2*np.ones((n,1)) + prior_std*np.random.randn(n,1)]

    while len(posterior_b) < N:
        # (a) Select candidate n*
        n_star = np.ceil(np.random.rand()*m).astype('int')
        n = posterior_n[-1]
        bn = posterior_b[-1]

        # (b)
        if n_star >= n:
            u = np.random.multivariate_normal(mean=np.zeros(n_star),
                                              cov=(u_std*np.eye(n_star))).reshape(n_star,1)
            u1 = u[:n]
            u2 = u[n:]
            b_star = np.pad(bn, ((0,(n_star - n)),(0,0))) + u

            # Compute likelihoods
            h1 = h(u, n_star)
            h2 = h(u1, n)
```

```
        # (c)
        elif n_star < n:
            u1 = np.random.multivariate_normal(mean=np.zeros(n_star),
                                               cov=(u_std*np.eye(n_star))).reshape(n_star,1)
            b1 = bn[:n_star]
            b2 = bn[n_star:]
            u = np.vstack((u1,b2))
            b_star = b1 + u1

            # Compute likelihoods
            h2 = h(u, n)
            h1 = h(u1, n_star)

        # (d) Compute acceptance-rejection function
        rho = accept_reject(X, y, bn, b_star, h1, h2, L_std, prior_std)

        # (e)
        if np.random.rand() < rho:
            # Accept candidate
            posterior_b.append(b_star)
            posterior_n.append(n_star)
        else:
            # Keep current b_n and n
            posterior_b.append(bn)
            posterior_n.append(n)

    print(f"Realization {i+1} complete.")
    # 'zip' matches each n with its corresponding b_n, then stores them all
    # in a single index corresponding to its realization
    posterior_estimates.append(list(zip(posterior_n, posterior_b)))
```

Plots

In [ ]:
```
# Create 2-by-10 grid for histograms
fig, axs = plt.subplots(2, 5, figsize=(15, 5))
fig.tight_layout(pad=3)

for i in range(10):
    # Extract 'n' chain
    posterior_n = [posterior_estimates[i][j][0] for j in range(N)]
    row = 0 if i < 5 else 1
    col = i if i < 5 else i - 5
    # bar plot of 'n' chain
    axs[row][col].bar(*np.unique(posterior_n, return_counts=True))
    axs[row][col].set_title(f'Realization {i+1} ($n_0$ ={n0_list[i]})', fontsize=10)
    axs[row][col].xaxis.set_major_locator(MaxNLocator(integer=True))
    axs[row][col].set_xlim(1,10)

# save figure as png
plt.savefig("rjmcmc_hist.png")
plt.show();
```

In [ ]:
```
# Create 2-by-10 grid for convergence plots
fig, axs = plt.subplots(2, 5, figsize=(15, 5))
fig.tight_layout(pad=3)

for i in range(10):
    # Extract 'n' chain
    posterior_n = [posterior_estimates[i][j][0] for j in range(N)]
    row = 0 if i < 5 else 1
    col = i if i < 5 else i - 5
    # convergence plot of 'n' chain
    axs[row][col].plot(list(range(N)), posterior_n)
    axs[row][col].axhline(y=n0_list[i], color='red', alpha=0.3)
    axs[row][col].set_title(f'Realization {i+1} ($n_0$ ={n0_list[i]})', fontsize=10)

# save figure as png
plt.savefig("rjmcmc_converge.png")
plt.show();
```