Term project (Spring 2016)
IEE 598: Bayesian Statistics

Modification of SLOG algorithm to find Adaptive Lasso coefficients

- Jaideep Khare
ID: 1207906096

Table of Contents

Abstract **3**

Overview of Lasso **4**

Modifications to Lasso **6**

SLOG algorithm **7**

Details of modification **8**

Validation & Conclusion **13**

{parcor} package comparison 14

{glmnet} package comparison 14

References **16**

Appendix: Full code with comments **17**

Abstract

Among the innovations in statistical estimation techniques of the past few decades, it can be argued with some certainty that the Lasso regularization penalty is among the most important ones. A huge number and variety of techniques were born out of further research Lasso estimation method. Lasso is famously good with sparse, high-dimensional data and has found a number of applications in the big data world.

In this project, I attempt to construct a sampler based on a modification of the SLOG algorithm, a deterministic sequence used to estimate Lasso coefficients. The original algorithm, as presented in Rajaratnam (2016), is modified with to loop over lambda, perform cross-validation and run an additional stage that adds penalizing weights based on the data. This modification can be used to estimate Adaptive Lasso coefficients. Finally, we shall compare the performance of this sampler with adaptive lasso R functions.

Overview of Lasso

For the first many decades that regression analysis was performed, the size of a typical problem remained small. Owing to the exponential nature of growth of problem space, regression coefficient estimation was never explored beyond a relatively small number of predictors. Calculations of effects of interaction and higher-order terms was only used when there was reason to suspect these terms in the first place.

Modern computing improved the speed of calculations by orders of magnitude. In the last couple of decades, a large amount of research has been made in dealing with unprecedented scales and size of data.

A typical approach with a large number of variables involves starting with the simplest model and then adding layers of complexity. The Ordinary Least Squares (OLS) regression is usually the starting point for a model fit. It can be proved mathematically that the OLS estimator is the Best Linear Unbiased Estimator. The following is minimized in OLS regression:

$$S(b) = \sum_{i=1}^{n} (y_i - x_i^T b)^2 = (y - Xb)^T (y - Xb),$$

However, OLS fails to perform as well when the number of predictors is large or when the data is sparse, that is, containing a large number of zeros. To address this issue, bias is categorically introduced in the variable by introducing extra terms in the loss function. These extra terms penalize complexity. This process is known as regularization. While this leads to bias in coefficients, the purpose of regularization is to reduce overfitting.

Regularization penalties have been around since before Lasso. Ridge regression (Tikhonov, 1977) is one of the earliest and most popular ones that has found application in a variety of domain. The hinge loss that the Tikhonov regularization loss function provides is a highly desirable property of Ridge regression. Ridge regression coefficients are computed as follows. This regularization has a closed-form solution.

$$\begin{aligned} \hat{\beta}^{\text{ridge}} &= \underset{\beta \in \mathbb{R}^p}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \\ &= \underset{\beta \in \mathbb{R}^p}{\operatorname{argmin}} \underbrace{\|y - X\beta\|_2^2}_{\text{Loss}} + \lambda \underbrace{\|\beta\|_2^2}_{\text{Penalty}} \end{aligned}$$

Lasso (Tibshirani, 1996) is yet another type of regularization penalty. Unlike Ridge Regression, Lasso regularization possesses the unique ability to shrink coefficients to 0. The implication of this property, often referred to as the “oracle property” (Zou, 2005) is one of the biggest strengths of Lasso that make it so desirable for use. Along with continuous variable shrinkage, Lasso also performs automatic model building by reducing coefficients to 0 and thus selecting variables to include in the model.

In today’s terminology, we refer to the Ridge regression penalty as an “L2 norm”, since this penalizing term is equivalent to the Euclidean distance between coefficients. While minimizing the loss function, the L2 norm yields a closed form solution (Tikhonov, 1977) and hence does not require numerical algorithms for computing estimates.

The Lasso estimates, however, on account of being an L1 norm, do not have a closed form solution. Quadratic programming can be used, but it is prohibitively expensive to compute. In the years following Tibshirani’s Lasso formulation, a wide variety of computational algorithms were devised to numerically estimate the Lasso coefficients. Prominent among them are the least angle regression or LARS algorithm (Effron et al, 2004), and the pathwise co-ordinate optimization algorithm (Friedman et al, 2007). Following is the loss function used for Lasso:

$$\min_{\beta} \sum_{i=1}^{n} \rho_{\tau}\left(y_i - x_i'\beta\right) + \lambda \sum_{j=1}^{k} \left|\beta_j\right|.$$

If you compare this expression with the ridge coefficients mentioned earlier, you can see that Lasso utilizes an L1 norm, whereas Ridge works on an L2 norm. This property is what makes Lasso estimates much harder to compute than Ridge coefficients.

The Lasso parameter is typically calculated by the method of cross-validation. This involves separating a test set, training on the rest of the data and then calculating test error. This process is performed for a number of mutually exclusive and exhaustive folds over the data (most default values for R packages fix this number at 10).

Modifications to Lasso

Recently, Lasso and Ridge have been combined in a single regularization penalty called the Elastic Net (Zou & Hastie, 2005). This is a convex combination of Lasso and Ridge penalties.

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} |\mathbf{y} - \mathbf{X}\boldsymbol{\beta}|^2, \qquad \text{subject to } (1-\alpha)\,|\boldsymbol{\beta}|_1 + \alpha\,|\boldsymbol{\beta}|^2 \leqslant t \text{ for some } t.$$

The function in that the loss function is subjected to is called the elastic net penalty. When α = 0, the penalty becomes Ridge regularization. When α = 1, the penalty becomes Lasso. Typically, this penalty is applied with components of both.

Later, another variant with adaptive weights (Zou, 2006) was introduced where each coefficient is penalized inversely by the size of the coefficient.

$$\arg\min_{\boldsymbol{\beta}} \left\| \mathbf{y} - \sum_{j=1}^{p} \mathbf{x}_j \beta_j \right\|^2 + \lambda \sum_{j=1}^{p} w_j |\beta_j|,$$

Here, the w_j's are weights corresponding to the j'th predictor.

In the frequentist sense, a Lasso penalty essentially introduces a bias in regression coefficients to reduce their variance. The value of the penalizing constant determines which size of coefficients will be penalized to what degree.

The same penalty has an interpretation in a Bayesian sense. It was observed that performing regularization via the Lasso point estimate component is the equivalent of introducing a double-exponential prior component, and then adopting a maximum a posteriori Bayesian procedure. (Park & Casella, 2008). The Bayesian Lasso was originally conceived as a quantified way of expressing uncertainty in the regression coefficients, even the ones that are exactly 0. Frequentist statistics used sophisticated approximation techniques to express this uncertainty, but the Bayesian framework gives a natural, intuitive method to do the same.

The following conditionals sample through the Bayesian Lasso:

$$\boldsymbol{\beta} \mid \boldsymbol{\omega}, \boldsymbol{y} \overset{\text{iid}}{\sim} N_p \left[\left(\boldsymbol{X}^T \boldsymbol{X} + \boldsymbol{\Omega}^{-1} \right)^{-1} \boldsymbol{X}^T \boldsymbol{y},\ \sigma^2 \left(\boldsymbol{X}^T \boldsymbol{X} + \boldsymbol{\Omega}^{-1} \right)^{-1} \right],$$

$$\omega_j^{-1} \mid \boldsymbol{\beta}, \boldsymbol{y} \overset{\text{iid}}{\sim} \begin{cases} \text{InverseGaussian}(\lambda/|\beta_j|,\ \lambda/\sigma^2) & \text{if } \beta_j \neq 0, \\ \text{InverseGamma}(1/2,\ \lambda^2/2\sigma^2) & \text{if } \beta_j = 0, \end{cases}$$

The β values generated converge towards Lasso coefficients.

SLOG algorithm

This project attempts to implement the adaptive Lasso using a method prescribed by Rajaratnam et al (2016). This method relies on the foundation set by Park & Casella (2008) and uses their conditionals as a starting point.

Conditional probability distributions are used to express uncertainty in the model coefficients for variables of interest. Rajaratnam et al found that along with expressing uncertainty, the structure of the conditionals can also be exploited to estimate the Lasso point estimate itself. For computing this estimate, they suggested shrinking the Gibbs sampler to the limit of $\sigma^2 \rightarrow 0$. Also, at this limit, the conditional sequence of the Gibbs sampler reduces down to a deterministic sequence. This formalized technique is called Shrinkage via the Limit Of Gibbs sampler, or SLOG.

$$\boldsymbol{b}^{(k+1)} = (\boldsymbol{B}^{(k)})^{1/2} \left[\lambda \boldsymbol{I}_p + (\boldsymbol{B}^{(k)})^{1/2} \boldsymbol{X}^T \boldsymbol{X} (\boldsymbol{B}^{(k)})^{1/2}\right]^{-1} (\boldsymbol{B}^{(k)})^{1/2} \boldsymbol{X}^T \boldsymbol{y},$$

This sequence generates the b-values. As k grows larger, this sequence converges. More importantly, it converges towards Lasso coefficients.

Details of modification

The SLOG algorithm uses a deterministic sequence to generate coefficients to fit data with a regularization penalty. Details of this algorithm are found in Rajaratnam (2016). The paper also describes R code that can be used to generate these coefficients.

For the scope of this project, I have modified the SLOG algorithm so that it closely resembles the estimation process of Adaptive Lasso coefficients.

Step 1: Add loop for running over lambda values as well as a cross-validation loop.

A cross-validation is added, that divides the data into 10 mutually exclusive and exhaustive sets. Model-building is performed on data with one fold held out, and test error is found by predicting on the held-out fold (this method is commonly referred to as leave-one-out-cross-validation.

Implementation of cross-validation:

```
# Create 10 equally size folds

folds <- cut(seq(1,nrow(full.data)),breaks=10,labels=FALSE)

# Run the loop

for (i in 1:10) {
        testIndexes <- which(folds==i,arr.ind=TRUE)
        testData <- full.data[testIndexes, ]
        trainData <- full.data[-testIndexes, ]
        x.values <- data.matrix(full.data[-testIndexes, 1:(ncol(full.data)-1)])
        y.values <- data.matrix(full.data[-testIndexes, ncol(full.data)])

        # Code for SLOG algorithm follows
        …
        …
```

This cross-validation loop is run over a large series of lambda values. The purpose of this is to find an optimum value of lambda which performs with the least mean square error. MSEs are averaged over the cross-validation loop, then the lambda value with the minimum MSE is selected as the optimum lambda

Looping over lambda values

```
# Define lambda sequence to loop over
lambda.seq <- seq (0.01,10,0.01)
l.ada <- NULL
 # Loop over lambda

 for (j in 1:length(lambda.seq)) {
        # Randomize data
        full.data <- cbind(x, y)
        full.data<-full.data[sample(nrow(x)), ]

        # Code for lamba follows
        # …
        # …
        # Inside the CV loop:

        l.ada <- lambda.seq[j]*wts

        # This is the adaptive lasso lambda for this loop
        # "wts" will be defined in the next step
```

Step 2: Find OLS coefficients

The data is fitted to a standard least squares model as a pre-processing step to the modified SLOG algorithm. The idea is to penalize SLOG algorithm inversely with the coefficients of the OLS regression. This implies that smaller coefficients have a larger weight attached to the regularization constant, and vice versa. This problem is a convex optimization problem, which does not suffer from the local minima problem. This lets us obtain the global minima easily, and the sampler converges fast. (Zou, 2005)

Step 3: weight vector inversely proportional to stage-1 coefficients to be used as "lambda" value for stage 2. We are penalizing coefficients with a large value with a smaller weight, and vice versa.

The coefficients found in the previous step are saved to an array. The array is then inverted to form a vector of weights, to be used in the next part of the problem.

Code for computing weights

```
model.stage1 <- lm(y.values~x.values)
coeffs.stage1 <- model.stage1$coefficients[-1]
wts <- 1/abs(as.vector(coeffs.stage1))
```

For future work, we could use glmnet to find ridge regression coefficients for the first stage, and define weights inversely proportional to those coefficients. (Zou, 2005)

Step 4: Implementing SLOG

Now that we have weights from Stage 1 and a λ value penalized with weights, we are set to run the SLOG algorithm as specified in Rajaratnam et al (2016). After initializing the necessary values, a loop is set up with a variable "conv" that is the testing criterion for the stopping rule. This "conv" indicates the shift in b-value in consecutive iterations. Once this value is small enough, the algorithm is stopped. The threshold is defined by the user.

Loop for finding deterministic sequence

```
while(conv==FALSE){
        p <- length(b.cur[vin])
        B.inv  <- diag(l.ada/abs(b.cur[vin] + 0001), nrow = p, ncol = p)
        b.new<-tcrossprod(chol2inv(chol(B.inv+xtx[vin,vin])),t(xty[vin]))
        temp[vin]<-as.vector(b.new)
        temp[abs(temp)<=thresh]<-0
        b.new<-temp
        vin<-which(b.new!=0)
        b.cur <- as.vector(b.new)
        conv<-(sqrt(sum((b.cur-b.old)^2))/sqrt(sum(b.old^2)))<times
        b.old<-b.cur
        k<-k+1
    }
```

The "b.cur" variable in this loop provides the final array of coefficients. Once this loop runs through, these coefficients are saved along with number of iterations. The mean square error is calculated using the following code:

Code for calculating Mean Square Error for each lambda value

```
        response.pred <- NULL
        # Find mean square error of each loop of cross-validation
        for (m in 1:nrow(testData)){
        response.pred[m] <- sum(b.cur*testData[m,1:(ncol(full.data)-1)])
        }

        resid <- full.data[(testIndexes), ncol(full.data)] - response.pred
        mse.cv <- c(mse.cv, mean(resid^2))

} # END OF CROSS-VALIDATION LOOP

# average over cv objects to find values for that particular lambda

mse.lambda[j] <- mean(mse.cv)

} # END OF LOOP OVER ALL VALUES OF LAMBDA
```

Step 5: Select optimum lambda

Finally, after looping through a number of lambda values (1000 in this case), we select the lambda which yields the minimum value of Mean Squared Error. While making this function ready for publication or industry use, it can be expanded easily to let users specify the range of lambda they would like to loop over.

Following is the code for finding optimum lambda:

Code for picking optimum lambda

```
…
…

        mse.lambda[j] <- mean(mse.cv)
        b.lambda <- rbind(b.lambda, rowMeans(B.CUR.CV))
        k.lambda[j] <- mean(K.CUR.CV)

 }      # END OF LOOP OVER ALL VALUES OF LAMBDA

        # Finally, pick the value of b for which MSE is minimum

b.final <- b.lambda[which.min(mse.lambda), ]
k.final <- k.lambda[which.min(mse.lambda)]
lambda.final <- lambda.seq[which.min(mse.lambda)]
return(c(lambda.final, k.final, b.final))

} # END OF FUNCTION
```

Validation & Conclusion

Since this modification tries to mimic the behavior of an adaptive lasso algorithm, validation was performed by comparing results from the algorithm with equivalent results from two well-known R packages

- {parcor}
- {glmnet}

The data chosen to run the algorithm is the canonical Diabetes data. Following is the implementation of this code with diabetes data:

```
> x <- data.matrix((diab.cent[, 1:10]))
> y <- data.matrix((diab.cent[, 11]))
> ada.SLOG.withCV <- adalasso.SLOG.withCV(x = x, y = y, times = 1e-5, thresh = 1e-5)
```

Since we are working with a 0-intercept model, the data was first centered. Test error is found using MSEs averaged over the cross-validation loops. The Adaptive Lasso SLOG algorithm with cross-validation gave the following coefficients as an output

```
> ada.SLOG.withCV.coeffs <- ada.SLOG.withCV[-c(1,2)]
> ada.SLOG.withCV.coeffs
 [1] -0.00142048 -0.10215781  0.32479856  0.18125122 -0.13137796  0.01578451
 [7] -0.06430901  0.06742460  0.32893855  0.03361411
>
```

Upon examining these coefficients, we find that predictor 3 (BMI) and predictor 9 (S5) are significant.

Also found are the final lambda value (the most optimum with the least MSE), and the average number of iterations required before the algorithm converges.

```
> lambda.final <- ada.SLOG.withCV[1]
> lambda.final
[1] 7.73
> k.final <- ada.SLOG.withCV[2]
> k.final
[1] 6.7
>
```

{parcor} package comparison

Comparing this with the adaptive lasso outputs from the parcor package:

```
> library(parcor)
> x <- data.matrix((diab.cent[, 1:10]))
> y <- data.matrix((diab.cent[, 11]))
> adalasso.parcor <- adalasso(X = as.matrix(diab.cent[, 1:10]), y = as.matrix(d
iab.cent[, 11]))
> adalasso.parcor$coefficients.adalasso
          1           2           3           4           5           6
 0.00000000 -0.11891405  0.33943753  0.19284604 -0.25152308  0.09653942
          7           8           9          10
 0.00000000  0.09398438  0.39382817  0.00000000
>
```

We find that the same variables are the most significant, and the magnitudes of other coefficients are also comparable.

An important observation is that the {parcor} adaptive lasso manages to shrink coefficients of predictors 1, 7 and 10 to exactly 0. Although the SLOG modification does shrink these coefficients in magnitude, they are not exactly 0 and thus this method cannot be immediately used for model building.

{glmnet} package comparison

Following is the implementation of the Adaptive Lasso using {glmnet}:

```
> library(glmnet)
> ## The adaptive lasso needs a first stage that is consistent.
> ## Zou (2006) recommends OLS or ridge
> theridge.cv<-cv.glmnet(x,y,alpha=0) ## first stage ridge
> ## Second stage weights from the coefficients of the first stage
> bhat<-as.matrix(coef(theridge.cv,s="lambda.min"))[-1,1] ## coef() is a sparseMatrix
> if(all(bhat==0)){
+ ## if bhat is all zero then assign very close to zero weight to all.
+ ## Amounts to penalizing all of the second stage to zero.
+ bhat<-rep(.Machine$double.eps*2,length(bhat))
+ }
> adpen<-(1/pmax(abs(bhat),.Machine$double.eps)) ## the adaptive lasso weight
> ## Second stage lasso (the adaptive lasso)
> thelasso.cv<-cv.glmnet(x,y,alpha=1,
+ exclude=which(bhat==0),
+ penalty.factor=adpen)
> ## Extract resulting coefs
> adcoef<-coef(thelasso.cv,s="lambda.min")
```

Here are the results obtained via {glmnet}

```
> adcoef
11 x 1 sparse Matrix of class "dgCMatrix"
                         1
(Intercept) -2.908307e-16
AGE          .
SEX         -1.394430e-01
BMI          3.276523e-01
BP           1.950742e-01
S1          -1.039774e-01
S2           .
S3          -1.138751e-01
S4           3.990848e-02
S5           3.306068e-01
S6           3.332449e-02
>
```

This is done via the Ridge Regression method and thus yields slightly different results. The primary significant variables are the same, though, and have comparable magnitudes. The point of this demonstration is that Adaptive Lasso is highly dependent on the manner of calculating weights, and will perform model-building accordingly.

Rajaratnam (2016), however, rigorously proves that coefficients will converge to 0, just like the adaptive lassos of {glmnet} and {parcor} do. Future work could be undertaken in investigating why this is not the case in this particular example, and also in fine-tuning parameters of the function.

As interest in big data grows more and more, there is a huge surge in the quantity and quality of model estimation algorithms high-dimensional, sparse problem spaces. The SLOG algorithm is a robust-to-noise procedure to generate Lasso coefficients without the computational burden of cycling through conditionals. The scale of data does not seem to be slowing down anytime soon; it is innovations such as these that will enable us to make sense of the gargantuan data that today's world generates.

References

- Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, *67*(2), 301-320.
- Zou, H. (2006). The adaptive lasso and its oracle properties. *Journal of the American statistical association*, *101*(476), 1418-1429.
- Park, T., & Casella, G. (2008). The bayesian lasso. *Journal of the American Statistical Association*, *103*(482), 681-686.
- Hans, C. (2009). Bayesian lasso regression. *Biometrika*, *96*(4), 835-845.
- Hans, C. (2010). Model uncertainty and variable selection in Bayesian lasso regression. *Statistics and Computing*, *20*(2), 221-229.
- Tibshirani, R. (1996). Regression shrinkage and selection via the lasso.*Journal of the Royal Statistical Society. Series B (Methodological)*, 267-288.
- Kyung, M., Gill, J., Ghosh, M., & Casella, G. (2010). Penalized regression, standard errors, and Bayesian lassos. *Bayesian Analysis*, *5*(2), 369-411.

Appendix: Full code with comments

Adaptive Lasso using SLOG algorithm: FULL CODE

```
adalasso.SLOG.withCV <- function(x,y,times,thresh,start=NULL){
  # function for implementing Adaptive Lasso using SLOG algorithm
  # x: covariate data
  # y: response data
  # lambda.seq: value of lasso regularization parameter
  # times: convergence criteria - difference between successive coefficient vectors
  # thresh: below this value estimates are set to 0 (runs rSLOG)
  #wts: vector of weights used for adaptive lasso stage

  # First, we define vectors to save values of B, k and Mean Square Error over all values of
lambda chosen

  b.lambda <- NULL
  k.lambda <- NULL
  mse.lambda <- NULL
  # Prepare sequence of lambda to try over

  lambda.seq <- seq (0.01,10,0.01)
  l.ada <- NULL
  # Loop over lambda

  for (j in 1:length(lambda.seq)) {

        # Randomize data
        full.data <- cbind(x, y)
        full.data<-full.data[sample(nrow(x)), ]

        # Create 10 equally size folds
        folds <- cut(seq(1,nrow(full.data)),breaks=10,labels=FALSE)

        # Prepare vectors for storing values from cross-validation
        B.CUR.CV <- NULL
        K.CUR.CV <- NULL
        xty <- NULL
        coeffs.stage1 <- NULL
        mse.cv <- NULL

        # Perform 10 fold cross validation for each value of lambda
        for (i in 1:10) {
```

```
testIndexes <- which(folds==i,arr.ind=TRUE)
    testData <- full.data[testIndexes, ]
    trainData <- full.data[-testIndexes, ]
    x.values <- data.matrix(full.data[-testIndexes, 1:(ncol(full.data)-1)])
    y.values <- data.matrix(full.data[-testIndexes, ncol(full.data)])
    xtx<-crossprod(x.values)
    xty<-crossprod(x.values,y.values)
    p<-length(xty)
    n<-length(y.values)

        # First, we find coefficients using OLS and create a weights vector
        inversely proportional to OLS coefficients.

    model.stage1 <- lm(y.values~x.values)
    coeffs.stage1 <- model.stage1$coefficients[-1]
    wts <- 1/abs(as.vector(coeffs.stage1))

        # This is the SLOG algorithm with the modified lambda value

    l.ada <- lambda.seq[j]*wts
    b.cur <- sign(xty)*l.ada/p
    b.old <- b.cur
    vin<-1:p
    temp<-rep(0,p)
    conv=FALSE
    k<-1

        # Start running the main loop for the SLOG algorithm

    while(conv==FALSE){
    p <- length(b.cur[vin])
    B.inv  <- diag(l.ada/abs(b.cur[vin] + 0001), nrow = p, ncol = p)
    b.new<-tcrossprod(chol2inv(chol(B.inv+xtx[vin,vin])),t(xty[vin]))
    temp[vin]<-as.vector(b.new)
    temp[abs(temp)<=thresh]<-0
    b.new<-temp
    vin<-which(b.new!=0)
    b.cur <- as.vector(b.new)
    conv<-(sqrt(sum((b.cur-b.old)^2))/sqrt(sum(b.old^2)))<times
    b.old<-b.cur
    k<-k+1
    }
        # save values from cross-validation into these objects
```

```
                B.CUR.CV <- cbind(B.CUR.CV, b.cur)
                K.CUR.CV <- c(K.CUR.CV, k)


                response.pred <- NULL
                        # Find mean square error of each loop of cross-validation
                for (m in 1:nrow(testData)){
                response.pred[m] <- sum(b.cur*testData[m,1:(ncol(full.data)-1)])
                }
                resid <- full.data[(testIndexes), ncol(full.data)] - response.pred
                mse.cv <- c(mse.cv, mean(resid^2))

  }                 # END OF CROSS VALIDATION LOOP

                    # average over cv objects to find values for that particular lambda

  mse.lambda[j] <- mean(mse.cv)
  b.lambda <- rbind(b.lambda, rowMeans(B.CUR.CV))
  k.lambda[j] <- mean(K.CUR.CV)

 }                          # END OF LOOP OVER ALL VALUES OF LAMBDA
                            # Finally, pick the value of b for which MSE is minimum

b.final <- b.lambda[which.min(mse.lambda), ]
k.final <- k.lambda[which.min(mse.lambda)]
lambda.final <- lambda.seq[which.min(mse.lambda)]
return(c(lambda.final, k.final, b.final))
}
```