STAT542 HW2

Name: Chun Yin Ricky Chue (chue2@illinois.edu)

Question 1.

Firstly, load in thebitcoin dataset, and split the training and testing datasets. Training data is any data prior to 1/1/2017. We test the fitted linear model with data from 1/1/2017 to 9/12/2017.

```
library(lubridate)
MyData <- read.csv(file="bitcoin_dataset.csv", header=TRUE, sep=",")
MyData$Date_Order <- as.numeric(MyData$Date)        ### Converting date to numeric orders.
bit_train = MyData[MyData[, "Date_Order"] <= 2644,] ### Training dataset
bit_test = MyData[MyData[, "Date_Order"] >= 2645,]  ### Testing dataset
```

For the first analysis, we ignore the variable `btc_trade_volume` as a predictor.

```
drop_cols <- c("Date","btc_trade_volume","btc_market_price")     ### Columns to drop
bit_train.x <- bit_train[ , !(names(bit_train) %in% drop_cols)]   ### Predictors
bit_train.y <- bit_train$btc_market_price                         ### Responses
bit_test.x <- bit_test[ , !(names(bit_test) %in% drop_cols)]
bit_test.y <- bit_test$btc_market_price
predictor_num <- ncol(bit_train.x)                                ### Number of predictors.

### Further check if there is any missing values in the "cleaned" dataset.
### These should not return any entries if the dataset is clean.
which(is.na(bit_train.x), arr.ind=TRUE)
which(is.na(bit_train.y), arr.ind=TRUE)
which(is.na(bit_test.x), arr.ind=TRUE)
which(is.na(bit_test.y), arr.ind=TRUE)
```

Part a.

Fit thebest subset selection to the dataset.

```
library(leaps)
```

```
## Warning: package 'leaps' was built under R version 3.3.2
```

```
### Best subset selection by exhaustion.
RSSleaps=regsubsets(as.matrix(bit_train.x),bit_train.y,nvmax = predictor_num)
sumleaps=summary(RSSleaps,matrix=T)
```

There are $P = 22$ predictors in the dataset. The code was searching for the best model of each size (different number of predictors, $p = 1, ..., 22$) of linear regression by exhaustion, and find out the most indicative predictor given p. For example, the following code shows the most indicative predictors when we constrain $p = 1$, 2, 3 and 4. i.e. we only use 1, 2, 3 and 4 predictors for linear regression to predict the outcome variable \$Y = \$ `btc_market_price`. (Here we are not printing the entire table.)

```
head(sumleaps$outmat, 4)
```

As a summary, we state the most indicative predictors for the best models for $p = 1$, 2, 3 and 4.
$p = 1$: `btc_market_cap`
$p = 2$: `btc_market_cap`, `btc_difficulty`

$p = 3$: btc_market_cap, btc_hash_rate, btc_miners_revenue
$p = 4$: btc_market_cap, btc_hash_rate, btc_miners_revenue, btc_cost_per_transaction

For completeness, we also state the 'weak' predictors for the best models for $p = $19, 20, 21. i.e. the predictors that are not included in the predictive model.

```
tail(sumleaps$outmat, 4)
```

The predictors that are not considered are:
$p = 19$: btc_cost_per_transaction_percent, btc_median_confirmation_time, btc_estimated_transaction_volume
$p = 20$: btc_cost_per_transaction_percent, btc_median_confirmation_time
$p = 21$: btc_cost_per_transaction_percent

For $p = 22$, we verify that all predictors are kept in the linear fitting.

Part b.

Calculate the C_p, AIC and BIC criteria. The model with the lowest value would be chosen as the best model.

```
bit_training = data.frame(cbind(bit_train.x, "Y" = bit_train.y))
lmfit=lm(Y~.,data = bit_training)
msize=apply(sumleaps$which,1,sum)
n=dim(bit_training)[1]

# Rescale Cp, AIC, BIC to (0,1), for illustrative purposes.
inrange <- function(x) { (x - min(x)) / (max(x) - min(x)) }

Cp = sumleaps$cp; Cp = inrange(Cp);
AIC = n*log(sumleaps$rss/n) + 2*msize; AIC = inrange(AIC);
BIC = sumleaps$bic; BIC = inrange(BIC);
```

The size of the best models for C_P:

```
minpos_Cp = which.min(Cp)
minpos_Cp
```

```
## [1] 15
```

The size of the best models for AIC:

```
minpos_AIC = which.min(AIC)
minpos_AIC
```

```
## 15
## 15
```

The size of the best models for BIC:

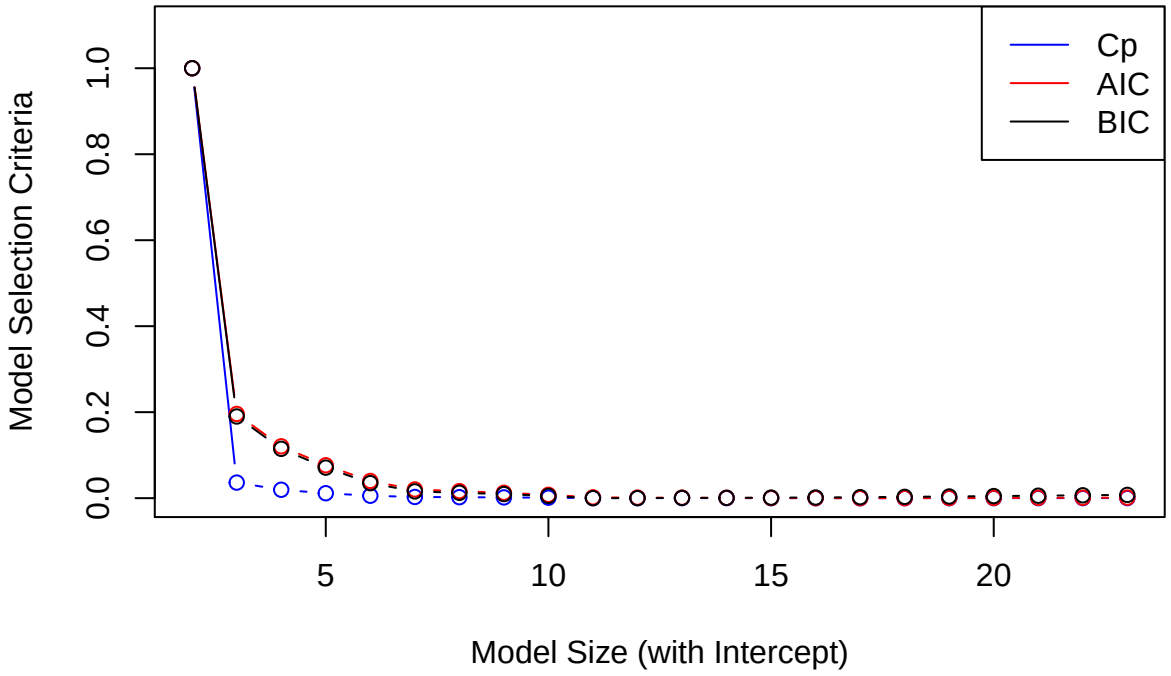
```
minpos_BIC = which.min(BIC)
minpos_BIC
```

```
## [1] 10
```

Therefore, the C_P and AIC criteria selects the best model with 15 predictors, whereas the BIC selects a simpler model with 10 predictors.

Visualizing C_P, AIC and BIC, the minimum values of these criterion would indicate the best model. As we can see, C_P and AIC criteria are giving very similar predictions.

```
plot(range(msize), c(0, 1.1), type="n", xlab="Model Size (with Intercept)",
     ylab="Model Selection Criteria")
points(msize, Cp, col="blue", type="b")
points(msize, AIC, col="red", type="b")
points(msize, BIC, col="black", type="b")
legend("topright", lty=rep(1,3), col=c("blue", "red", "black"),
       legend=c("Cp", "AIC", "BIC"))
```



To fit the best models to the testing dataset, with BIC criterion: The following shows the indicative predictors, and the fitted coefficients.

```
coefBIC_x <- coef(RSSleaps, minpos_BIC)
```

Then, we apply the fitted model to the testing data. Firstly we write the design matrix $\mathbf{M_{test}}$ for the testing data.

```
bit_testing = data.frame(cbind(bit_test.x, "Y" = bit_test.y))
test_m <- model.matrix(Y ~ ., data = bit_testing)        ### Design matrix for the testing data
```

Now, we select the intercept and the indicative predictors under the BIC criterion, and calculate the fitted values of the testing data $\widehat{Y} = \mathbf{M}_{test,BIC} x_{test}$. And calculate the prediction error $n_{test}^{-1} \sum_{i \in test} (\widehat{Y}_i - Y_i)^2$

```
predBIC_y <- test_m[ , names(coefBIC_x)] %*% coefBIC_x
predBIC_err <- mean((bit_test.y - predBIC_y)^2)
predBIC_err
```

```
## [1] 42784.07
```

The best models for the C_P and AIC criteria:

```
coefCpAIC_x <- coef(RSSleaps, minpos_Cp)
```

Similarly, we select the intercept and the indicative predictors under the C_P and AIC criteria, and calculate the fitted values of the testing data $\widehat{Y} = \mathbf{M}_{test,Cp/AIC} x_{test}$. And calculate the prediction error $n_{test}^{-1} \sum_{i \in test} (\widehat{Y}_i -$

$Y_i)^2$

```
predCpAIC_y <- test_m[ , names(coefCpAIC_x)] %*% coefCpAIC_x
predCpAIC_err <- mean((bit_test.y - predCpAIC_y)^2)
predCpAIC_err
```

```
## [1] 51854.26
```

Part c.

```
bit_train.logy = log10(1 + bit_train.y)
bit_test.logy = log10(1 + bit_test.y)
```

Fit the best subset selection to the dataset, with output $Y \to \log_{10}(1+Y)$.

```
### Best subset selection by exhaustion.
RSSlogleaps=regsubsets(as.matrix(bit_train.x),bit_train.logy,nvmax = predictor_num)
sumlogleaps=summary(RSSlogleaps,matrix=T)
```

Similarly, we show the most indicative predictors when we constrain $p = 1$, 2, 3 and 4. i.e. we only use 1, 2, 3 and 4 predictors for linear regression to predict the outcome variable $\log_{10}(1+Y)$.

```
head(sumlogleaps$outmat, 4)
```

As a summary, we state the most indicative predictors for the best models for $p = 1$, 2, 3 and 4.
$p = 1$: btc_total_bitcoins
$p = 2$: btc_cost_per_transaction, Date_Order
$p = 3$: btc_cost_per_transaction, Date_Order, btc_blocks_size
$p = 4$: btc_cost_per_transaction, Date_Order, btc_blocks_size, btc_n_transactions_total

where Data_Order is a newly defined predictor, i.e. the number of days after 2009-10-05.
We also state the 'weak' predictors for the best models for $p = $ 19, 20, 21. i.e. the predictors that are not included in the predictive model.

```
tail(sumlogleaps$outmat, 4)
```

The predictors that are not considered are:
$p = 19$: btc_n_transactions_excluding_popular, btc_n_transactions_excluding_chains_longer_than_100, btc_n_orphaned_blocks
$p = 20$: btc_n_transactions_excluding_popular, btc_n_transactions_excluding_chains_longer_than_100
$p = 21$: btc_n_transactions_excluding_popular

For $p = 22$, we verify that all predictors are kept in the linear fitting.

Calculate the C_p, AIC and BIC criteria.

```
bit_logtraining = data.frame(cbind(bit_train.x, "Y" = bit_train.logy))
lmlogfit=lm(Y~.,data = bit_logtraining)
mlogsize=apply(sumlogleaps$which,1,sum)
n=dim(bit_logtraining)[1]

logCp = sumlogleaps$cp; logCp = inrange(logCp);
logAIC = n*log(sumlogleaps$rss/n) + 2*mlogsize; logAIC = inrange(logAIC);
logBIC = sumlogleaps$bic; logBIC = inrange(logBIC);
```

The size of the best models for C_P:

```
minpos_logCp = which.min(logCp)
minpos_logCp
```

```
## [1] 17
```

The size of the best models for AIC:

```
minpos_logAIC = which.min(logAIC)
minpos_logAIC
```

```
## 17
## 17
```

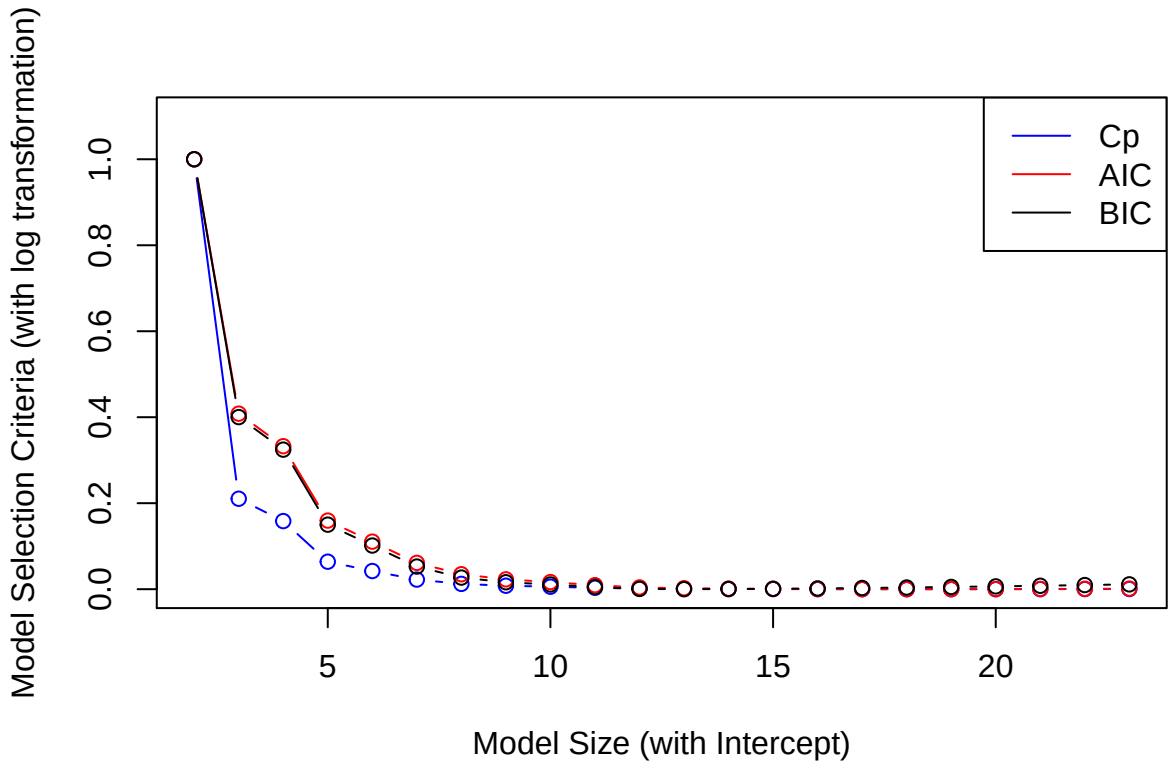
The size of the best models for BIC:

```
minpos_logBIC = which.min(logBIC)
minpos_logBIC
```

```
## [1] 12
```

Therefore, the C_P and AIC criteria selects the best model with 17 predictors, whereas the BIC selects a simpler model with 12 predictors.
Visualizing C_P, AIC and BIC, the minimum values of these criterion would indicate the best model.

```
plot(range(mlogsize), c(0, 1.1), type="n", xlab="Model Size (with Intercept)",
     ylab="Model Selection Criteria (with log transformation)")
points(mlogsize, logCp, col="blue", type="b")
points(mlogsize, logAIC, col="red", type="b")
points(mlogsize, logBIC, col="black", type="b")
legend("topright", lty=rep(1,3), col=c("blue", "red", "black"),
       legend=c("Cp", "AIC", "BIC"))
```



To fit the best models to the testing dataset, with BIC criterion: The following shows the indicative predictors, and the fitted coefficients.

```
coeflogBIC_x <- coef(RSSlogleaps, minpos_logBIC)
```

Then, we apply the fitted model to the testing data. Firstly we write the design matrix $\mathbf{M_{test}}$ for the testing data.

```
bit_logtest = data.frame(cbind(bit_test.x, "Y" = bit_test.logy))
testlog_m <- model.matrix(Y ~ ., data = bit_logtest)        ### Design matrix for test data
```

Now, we select the intercept and the indicative predictors under the BIC criterion, and calculate the fitted values of the testing data $\widehat{Y} = \mathbf{M}_{test,BIC} x_{test}$. As this outcome has been logarithmically transformed, we need to transform the predicted value back by $Y_i = 10^{Y_{log,i}} - 1$. And calculate the prediction error $n_{test}^{-1} \sum_{i \in test} (\widehat{Y}_i - Y_i)^2$

```
predlogBIC_y <- test_m[ , names(coeflogBIC_x)] %*% coeflogBIC_x
predBIC_y_transback <- 10**predlogBIC_y - 1
predBIC_logerr <- mean((bit_test.y - predBIC_y_transback)^2)
predBIC_logerr
```

```
## [1] 12425968
```

The best models for the C_P and AIC criteria:

```
coeflogCpAIC_x <- coef(RSSlogleaps, minpos_logCp)
```

Similarly, the prediction error $n_{test}^{-1} \sum_{i \in test} (\widehat{Y}_i - Y_i)^2$

```
predlogCpAIC_y <- test_m[ , names(coeflogCpAIC_x)] %*% coeflogCpAIC_x
predCpAIC_y_transback <- 10**predlogCpAIC_y - 1
predlogCpAIC_err <- mean((bit_test.y - predCpAIC_y_transback)^2)
predlogCpAIC_err
```

```
## [1] 6119170
```

Question 2.

Part I.

Writing my own Lasso fitting code. I am using the same data generator as the sample code `HW2.r`.

```
library(MASS)
library(glmnet)
set.seed(1)
N = 400; P = 20
Beta = c(1:5/5, rep(0, P-5)); Beta0 = 0.5
V = matrix(0.5, P, P); diag(V) = 1
X = as.matrix(mvrnorm(N, mu = 3*runif(P)-1, Sigma = V))
X = sweep(X, 2, 1:10/5, "*")
y = Beta0 + X %*% Beta + rnorm(N)
lm(y ~ X)
```

Here, we scale X and Y, which is necessary for Lasso implementation.
We scale X as $x_{scaled,j} = (x_{unscaled,j} - \bar{x})/\sigma_{x_j}$, where $\bar{x}$ and σ_{x_j} are the mean and sample standard deviation of X and x_j respectively. We scale Y by simply subtracting its mean, i.e. $y_{scaled} = y_{unscaled} - \bar{y}$

```
x_center = colMeans(X)          ### Mean of X
x_scale = apply(X, 2, sd)       ### Sample standard deviation of X
X2 = scale(X)                   ### Scaling X

bhat = rep(0, ncol(X2)) # initialize beta estimated
ymean = mean(y)                 ### Mean of Y
y2 = y - ymean                  ### Scaled Y
```

Soft thresholding function definition

```
soft_th <- function(b, pen) {
    result <- numeric(length(b))
    result[which(b > pen)] <- b[which(b > pen)] - pen
    result[which(b < -pen)] <- b[which(b < -pen)] + pen
    return(result)
}
```

Initialize λ values as for the Lasso penality terms.

```
lambda = exp(seq(log(max(abs(cov(X2, y2)))), log(0.001), length.out = 50))
```

Lasso Fitting function.
We use coordinate descent for the update of the β's.
The code goes from β_1, update its value while keeping the other β's fixed. Then it updates β_2, keeping others fixed, and so forth to β_p, where p is the number of predictors of the linear model.At each step, the objective function is to be updated correspondingly as well.
After that, the code iterates the process until the optimized solution is achieved, or if the number of iteration exceeds 500.

```
LassoFit <- function(myX, myY, mybeta, mylambda, tol = 1e-10, maxitr = 500) {
  mybeta <- as.matrix(mybeta)        ### Define a matrix for beta.
  obj <- numeric(length=(maxitr+1))  ### Store up the objective functions
  betalst <- list(length(maxitr+1))  ### Store up the beta values.
  betalst[[1]] <- mybeta

  for (k in 1:maxitr) {
```

```
    ### Update each of the j-th element respectively.
    for (j in 1:ncol(myX)) {
      ### Residual, just update the j-th element in beta.
      r = myY - myX[,-j] %*% mybeta[-j]
      mybeta[j] = (1/norm(as.matrix(myX[,j]), "F")^2) *
                  soft_th((t(r) %*% myX[,j]), length(myY) * mylambda)
    }
    ### Get the updated beta vector and store it in the betalst matrix.
    betalst[[k+1]] = mybeta

    ### Update the objective function.
    obj[k] = (1/2)*(1/length(myY))*norm(myY - myX%*%mybeta,"F")^2 + mylambda*sum(abs(mybeta))

    if (norm(betalst[[k]] - mybeta,"F") < tol) break;
    if (k > 10) {
      if (sum(abs(obj[(k-9):k] - mean(obj[(k-9):k]))) < tol) break;
    }
  }
  mybeta = list(obj = obj[1:k], beta = mybeta)
  return (mybeta)
}
```

Initiate a matrix `beta_all` that records the fitted beta for each lambda value, and a matrix `beta0_all` to store the intercept of each lambda value (after rescaling back to the original scale).

```
beta_all = matrix(NA, ncol(X), length(lambda))
beta0_all = rep(NA, length(lambda))
```

Scaling the β's back to the original scale. The scaling can be written in the mathematical expression: $\widehat{Y}_{scaled} = \widehat{Y}_{unscaled} - \bar{y} = \widehat{\beta_0} + \sum_{j=1}^{k} \widehat{\beta}_j(\frac{x_j - \bar{x_j}}{\sigma_{x_j}})$

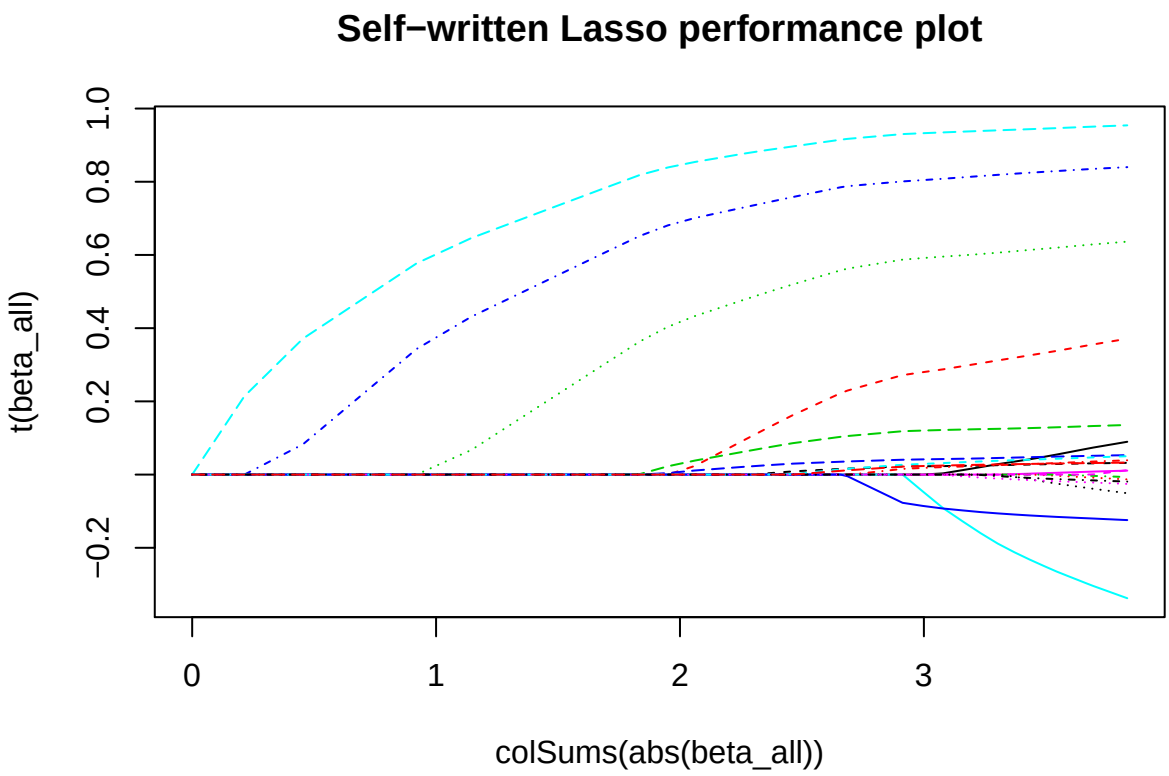
$\widehat{Y}_{unscaled} = (\bar{y} - \sum_{j=1}^{k} \widehat{\beta}_j \frac{\bar{x_j}}{\sigma_{x_j}}) + \sum_{j=1}^{k} \frac{\widehat{\beta}_j}{\sigma_{x_j}} x_j$. Hence, $\widehat{\beta_0}_{,unscaled} = \bar{y} - \sum_{j=1}^{k} \widehat{\beta}_j \frac{\bar{x_j}}{\sigma_{x_j}}$ and $\widehat{\beta}_{j,unscaled} = \frac{\widehat{\beta}_j}{\sigma_{x_j}}$

```
for (i in 1:length(lambda)) {
  bhat = rep(0, ncol(X2)) # initialize it
    bhat = LassoFit(X2, y2, bhat, lambda[i])

    beta_all[, i] = bhat$beta / x_scale
    beta0_all[i] = ymean - sum(bhat$beta * x_center / x_scale)
}
```

Visualize the result. The plot is showing the values of all the β's as a function of `colSums(abs(beta_all))`, which is related to the penality term λ. `colSums(abs(beta_all))` increases as the level of penality decreases (i.e. λ decreases.). Each line represents different predictors.
As expected, when λ increases, i.e. `colSums(abs(beta_all))` decreases, the predictors would be shrinked to zero. Some are shrunk faster by the fact that they reach zero at larger `colSums(abs(beta_all))` than the others.

```
matplot(colSums(abs(beta_all)), t(beta_all), type="l", main = "Self-written Lasso performance plot")
```



For sanity check, we calculate the difference between the answers generated by this self-written Lasso algorithm with the standard R package `glmnet`, and they are typically less than 0.01, indicating the code is correct.

```
max(abs(beta_all - glmnet(X, y, lambda = lambda)$beta))
```

```
## [1] 0.002327794
```

```
max(abs(beta0_all - glmnet(X, y, lambda = lambda)$a0))
```

```
## [1] 0.00134999
```

Part II.

Similar to part I, we scale the testing and training data by subtracting the mean, then divide by the unbiased sample standard deviation. As we see the range of the response variable Y is large, we scale Y as well for prediction.

```
bit_train_scale.x = scale(bit_train.x)
bit_train_scale.y = scale(bit_train.y)
bit_test_scale.x = scale(bit_test.x)
bit_test_scale.y = scale(bit_test.y)
n = length(bit_test.y)
testy_sd = sd(bit_test.y)        ### Sample standard deviation of test response
testy_mean = mean(bit_test.y)    ### Mean of outcome of test data
testx_mean = colMeans(bit_test.x)
testx_sd = apply(bit_test.x, 2, sd)
```

Perform Lasso fitting. This time, as we have scaled the response variable Y, we need to rescale Y back in a slightly different way.
$\widehat{Y}_{scaled} = \frac{\widehat{Y}_{unscaled} - \bar{y}}{\sigma_y} = \widehat{\beta_0} + \sum_{j=1}^{k} \widehat{\beta}_j (\frac{x_j - \bar{x_j}}{\sigma_{x_j}})$ where σ_y and σ_{x_j} are the sample standard deviation of testing Ys and x_is respectively.
After manipulation of equations, the predicted testing response $Y_{unscaled} = (\bar{y} - \sigma_y \sum_{j=1}^{k} \widehat{\beta}_j (\frac{\bar{x_j}}{\sigma_{x_j}})) + \sigma_y \sum_{j=1}^{k} (\frac{\widehat{\beta}_j}{\sigma_{x_j}} x_j)$

```
lambda = exp(seq(log(0.1), log(0.001), length.out = 50))
lassofitnum = length(lambda)      ### Number of lasso fits.
beta_all = matrix(NA, ncol(bit_train.x), lassofitnum)
# this vecter stores the intercept of each lambda value
beta0_all = rep(NA, lassofitnum)

for (i in 1:lassofitnum) {
    # if your function is correct, this will run pretty fast
  bhat = rep(0, ncol(bit_train.x)) # initialize it
    bhat = LassoFit(bit_train_scale.x, bit_train_scale.y, bhat, lambda[i])

    beta_all[, i] = testy_sd * bhat$beta / testx_sd
    beta0_all[i] = testy_mean - testy_sd * sum(bhat$beta * testx_mean / testx_sd)
}
```

```
predlasso_y <- matrix(, nrow = n, ncol = lassofitnum)

bit_lassotest = data.frame(cbind(bit_test.x, "Y23" = bit_test.y))
### Design matrix for the testing data
testlasso_m <- model.matrix(Y23 ~ ., data = bit_lassotest)

betabind = rbind("intercept" = beta0_all, beta_all)

predlasso_y = testlasso_m %*% betabind
```

So now we can make a fair comparison with the testing data. Calculate the testing errors for different λ's.

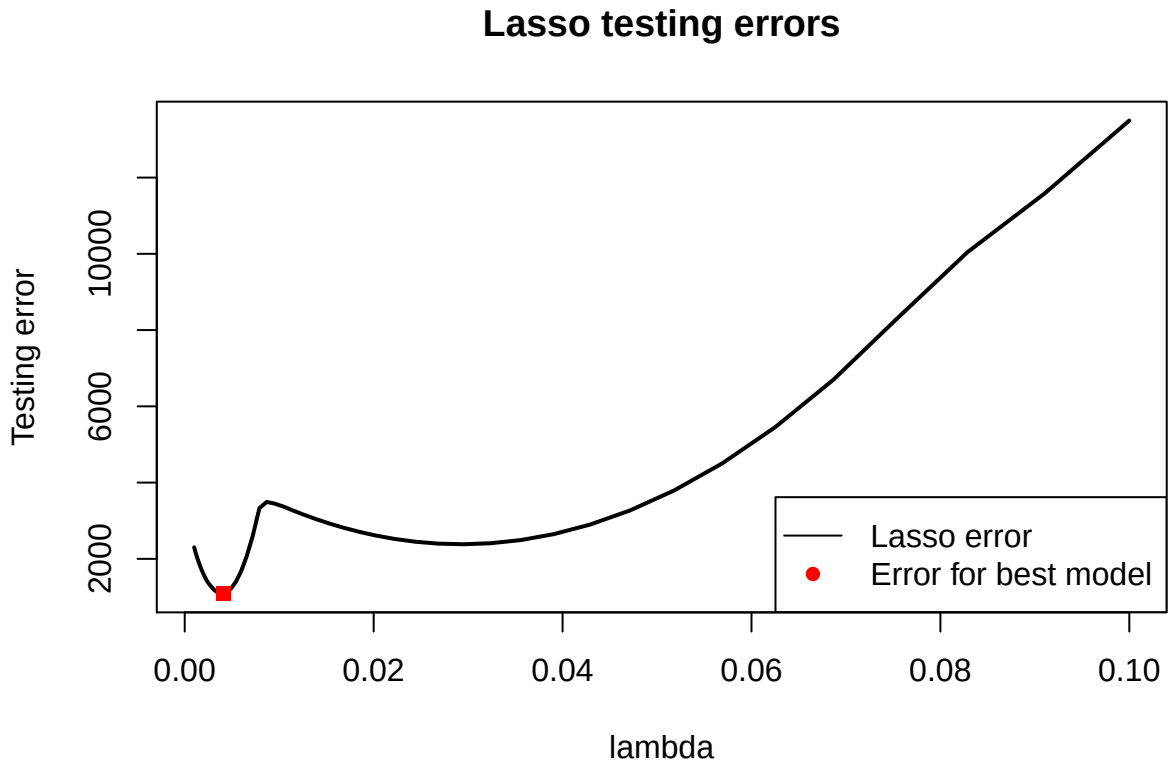
```
predlasso_err <- matrix(, nrow = 1, ncol = lassofitnum)
for (i in 1:lassofitnum)
  predlasso_err[,i] <- mean((bit_test.y - predlasso_y[,i])^2)

predlasso_err = as.vector(predlasso_err)
```

```
### Find the minimum testing error, and the lambda corresponding to the least error.
minlasso_err = min(predlasso_err)
minlasso_errpos = which.min(predlasso_err)
lasso_minerr = lambda[minlasso_errpos]
```

We report the testing errors of Lasso fitting by visualization. Testing error $= n_{test}^{-1} \sum_{i \in test} (\widehat{Y}_i - Y_i)^2$.

```
plot(lambda, predlasso_err, type="l", col="black", xlab="lambda",
     ylab="Testing error", lwd = 2, main = "Lasso testing errors")
points(lasso_minerr, minlasso_err, col = "red", type = "o", lwd = 4, pch = 15)
legend("bottomright", lty = c(1,0), pch = c(NA,16), col = c("black","red"),
       legend=c("Lasso error", "Error for best model"))
```



The best model is achieved when $\lambda = 0.003986997$, where the prediction error $n_{test}^{-1} \sum_{i \in test} (\widehat{Y}_i - Y_i)^2 = 1089.19$.