Ridge Regression and Estimation of Associated Test Error

ANKIT TEWARI & ROBERT CARULLA

Introduction

In this session, we are going to write R functions for implementing the ridge regression by selecting the appropriate penalization parameter λ choice based on the minimization of the Mean squared prediction error (PMSE) in a validation set (MSPEval (λ)), K-fold Cross-Validation Approach using K=5 and 10 and the information of λ we had previously available using the Leave-one-out Cross-Validation and Generalized Cross Validation approach

The Ridge Regression estimator is defined as follows-

$$\hat{\beta}_{ridge} = (X^T X + \lambda I_P)^{-1} X^T Y$$

and We define the matrix H_λ as following-

$$H_\lambda = X(X^T X + \lambda I_P)^{-1} X^T$$

We will using this notation for developing insight about fitting the ridge regression model on the `prostate_cancer` dataset.

When using a model, it is necessary to estimate how much test error can be generated when using that statistical model for predictions on test dataset. Also, in order to decide how much flexible we must keep our model (that is, for example, the value of K in K-Nearest Neighbour Regression), we can use the method of cross validation which comes in several versions that we will discuss and implement turn by turn. These methods can be particularly useful considering the two scanrios described below-

- Model Assessment: Measuring and evaluating the model's performance
- Model Selection: Identifying the appropriate level of flexibility for our model

The method of cross-validation is based on the idea of the validation set approach to estimate the test error associated with a machine learning (or equivalently, a statistical learning) model. The objective of the cross-validation approach is to overcome the two major issues that arise with the validation set approach namely, ***large variance of estimates of test error*** and ***overestimation of the test error***.

Leave-one-out cross-validation (LOOCV)

The LOOCV estimate of the test MSE is the average of the n different MSEs obtained by leaving each of the n different observations in the training set, one at a time. It is defned as-

$$CV_{(n)} = \frac{1}{n} \sum_{i=1}^{n} MSE_{(i)}$$

where, $MSE_{(i)}$ is the *mean squared error* associated with the prediction $\hat{y_i}$ for the input (x_i, y_i).

The major advantages that LOOCV has over the Validation Set approach is that it has ***lower bias than the validation set approach*** for the same set of n observations. Secondly, since the amount of data used in training is far more than the validation set approach, the ***overestimation of test error*** associated with LOOCV is lower than the validation set approach. Finally, there is no randomness in the training/validation set splits.

k-*Fold Cross-Validation Method*

The estimate of the MSE_{test} is obtained using the following k-fold cross-validation estimator-

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_{(i)}$$

Observe that, LOOCV method that we had just discussed is a special case of the k-fold cross-validation method where we had set k=n. So, what it means is, just like we fit the model on $(k-1)$ folds and estimate the test error using the k-th fold in case of k-folds cross-validation, in case of LOOCV, we fit the model using the $(n-1)$ folds and estimate the test error using the 1 fold left out in training the data.

Methods

We define the function `ridge.reg` for computing the $\hat{\beta}_{ridge}$. This function returns the matrix of fitted values of $\hat{\beta}_{ridge}$ for each of the values of λ provided for choosing the tuning paramter. In case, we have already selected the tuning parameter, we can sepcify only the chosen value of lambda as paramter to the function which in such case returns a vector of fitted $\hat{\beta}_{ridge}$. The follwoing code defines the function-

```r
ridge.reg <- function(x_train, y_train, lambda.v){
  n <- nrow(x_train)
  p <- ncol(x_train)
  X <- scale(x_train, center = TRUE, scale = TRUE)
  Y <- scale(y_train, center = TRUE, scale = FALSE)
  n.lambdas = length(lambda.v)

  XtX <- t(X)%*%X
  # matrix of coefficients (25 different values for each of the p variable)
  # this matrix stores different values of parameters along 25 row for each p variable
  beta.path <- matrix(0,nrow=n.lambdas, ncol=p)

  # matrix H_lambda
  diag.H.lambda <- matrix(0,nrow=n.lambdas, ncol=n)


  for (l in 1:n.lambdas){
    lambda <- lambda.v[l]

    H.lambda.aux <- t(solve(XtX + lambda*diag(1,p))) %*% t(X)
    beta.path[l,] <-  H.lambda.aux %*% Y
    H.lambda <- X %*% H.lambda.aux
    diag.H.lambda[l,] <- diag(H.lambda)
  }
  fitted_coef <- as.matrix(beta.path)
  return(fitted_coef)
}
```

Now, in order to use the function for fitting the desired values of λ for using them to find the appropriate tuning parameter, we define a function which choses the sequence of λ which is the most appropriate for the dataset under consideration. In order to care for the specific sequence of λ which will be most suited for the dataset, this generic function takes into account the training dataset under consideration and uses the (X_train, Y_train) as the parameter.

```r
compute_lambda_sequence <- function(x, y){
  ## Standardize variables: (need to use n instead of (n-1) as denominator)
  n <- 100
  mysd <- function(y) sqrt(sum((y-mean(y))^2)/length(y))
  sx <- scale(x, scale = apply(x, 2, mysd))
  sx <- as.matrix(sx, ncol = 20, nrow = 100)
  sy <- as.vector(scale(y, scale = mysd(y)))
  ## Calculate lambda path (first get lambda_max):
  lambda_max <- max(abs(colSums(sx*sy)))/n
  epsilon <- .0001
  K <- 25
  lambdapath <- round(exp(seq(log(lambda_max), log(lambda_max*epsilon),
                              length.out = K)), digits = 10)
  lambdapath
}
```

Now, since our objective also inlcuded the comparison of different methods (other than the Predictive Mean Squared Error) for assessing the estimate of the test error as accurately as possible, we consider writing the fucntion for computing the Predictive Mean Squared Error using the function `compute_PMSE` defined below-

```r
compute_PMSE <-function(X_train, Y_train, X_validate, lambda.v){

  # Implementation of RIdge Regression using user defined function
  ridge.prostate.user <- ridge.reg(X_train, Y_train, lambda.v)

  # Ridge regression using standard package MASS for comparison
  library(MASS)
  ridge.prostate.mass <- lm.ridge(Y_train ~ X_train, lambda = lambda.v)

  #obtaine the predicted values of Y for different lambdas
  pred_mass <- as.matrix(X_validate%*%as.matrix(ridge.prostate.mass$coef))
  pred_user <- as.matrix(X_validate%*%t(as.matrix(ridge.prostate.user)))

  PMSE_user <- numeric(length = ncol(pred_user))
  PMSE_mass <- numeric(length = ncol(pred_mass))

  for (i in 1:length(lambda.v)) {
    PMSE_user[i] <- mean((as.vector(pred_user[,i]) - Y_validate)^2)
    PMSE_mass[i] <- mean((as.vector(pred_mass[,i]) - Y_validate)^2)
  }
  result <- data.frame(User = PMSE_user, Mass = PMSE_mass, lambda= lambda.v)
  return(result)

}
```

Also, we consider the K-fold cross-validation approach for estimating the test error. Therefore, we define the function for computing the K-Fold Cross Validation Error assocaited with Ridge Regression as follows-

```r
K_fold_CV <- function(X, Y, lambda.v, K=5) {

  # Randomly splitting training data (X , Y)
  dataset <- as.data.frame(cbind(X, Y))
  dataset <- dataset[sample(nrow(dataset)),]
  # Generate the folds (that is, divide the randomly split data into k parts)
  folds <- cut(seq(1, nrow(dataset)), breaks = K, labels = F)
  # Define the vector of MSE_lambda for each lambda
  MSE_lambda <- numeric(length = length(lambda.v))
  # for each lambda
  for ( j in 1:length(lambda.v)) {
    # Define the vector of PMSE for each K
    MSE <- numeric(length = K)
    for (i  in 1:K){

    # creation of training and validation data for K-th fold
    validation_indices <- which(folds == i, arr.ind = TRUE)
    X_valid <- as.matrix(dataset[validation_indices,  1:8])
    X_train <- as.matrix(dataset[-validation_indices, 1:8])
    Y_valid <- as.matrix(dataset[validation_indices,  9])
    Y_train <- as.matrix(dataset[-validation_indices, 9])

    # fit the ridge regression model on K-th fold training data
    ridge.prostate.model <- ridge.reg(X_train, Y_train, lambda.v = lambda.v[j])

    # predict using the ridge reression model on the K-th fold validation set
    prediction <- as.matrix(X_valid%*%t(ridge.prostate.model))

    # evaluation of the K-th MSE
    MSE[i] <- mean((prediction - matrix(rep(Y_valid, ncol(prediction)),
                                        ncol=ncol(prediction),
                                        nrow = nrow(prediction)))^2)
    }
    MSE_lambda[j] <- mean(MSE)


  }
  result <- data.frame(cvm= MSE_lambda, lambda= lambda.v)
  return(result)

}
```

Let us now begin with creating the datasets which we can use for fitting our. First, we will read the `prostte_cancer` dataset from our directory. The dataset is stored in the file `Prostate cancer data Fitxer.txt`. We then save this dataset as `prostate` for the skae of simplicity. The following code achieves this purpose-

```
library(readr)
Prostate_cancer_data_Fitxer <- read_delim("Session 2/Prostate cancer data Fitxer.txt",
    "\t", escape_double = FALSE, trim_ws = TRUE)
```

```
## Parsed with column specification:
## cols(
##   No = col_integer(),
##   lcavol = col_double(),
##   lweight = col_double(),
##   age = col_integer(),
##   lbph = col_double(),
##   svi = col_integer(),
##   lcp = col_double(),
##   gleason = col_integer(),
##   pgg45 = col_integer(),
##   lpsa = col_double(),
##   train = col_logical()
## )
```

```
#View(Prostate_cancer_data_Fitxer)

prostate <- Prostate_cancer_data_Fitxer
train.sample <- which(prostate$train==TRUE)
use.only <- train.sample
# creating the training dataset as (X,Y)
Y <- scale( prostate$lpsa[use.only], center=TRUE, scale=FALSE)
X <- scale( as.matrix(prostate[use.only,1:8]), center=TRUE, scale=TRUE)
```

In order to test the function and find the associated test error, we define the validation set consisting of the data not used in the training set. The following code illustrates it-

```
prostate <- Prostate_cancer_data_Fitxer
validation.sample <- which(prostate$train==FALSE)
use.only <- validation.sample
# creating X_validate and Y_validate as validation dataset
Y_validate <- scale( prostate$lpsa[use.only], center=TRUE, scale=FALSE)
X_validate <- scale( as.matrix(prostate[use.only,1:8]), center=TRUE, scale=TRUE)
```

Now, using the fucntion that we had defined previously for computing the values of λ sequence scientifically, we store those values in the variable `lambda.v.f`

```
lambda.v.f <- compute_lambda_sequence(X,Y)
```

Then, we fit the ridge regression model on the prostate cancer data and store the obtained coefficients. Note that, we have used our own user defined function as well as the fucntion `lm.ridge` from the `MASS` package just for evaluating how good our function is fitting the ridge coefficients. The following code demonstrates it-

```
# Implementation of RIdge Regression using user defined function
ridge.prostate.user <- ridge.reg(X, Y, lambda.v = lambda.v.f)
# Ridge regression using standard package MASS for comparison
library(MASS)
ridge.prostate.mass <- lm.ridge(Y ~ X, lambda = lambda.v.f)
```

Finally, we obtain the predicted values of the target variable using the two functions discussed previously. The code below illustrates it using the variables `pred_mass` and `pred_user`.

```
pred_mass <- X_validate%*%as.matrix(ridge.prostate.mass$coef)
pred_user <- X_validate%*%t(as.matrix(ridge.prostate.user))
#print(cbind(pred_mass[,1], pred_user[,1]))
```

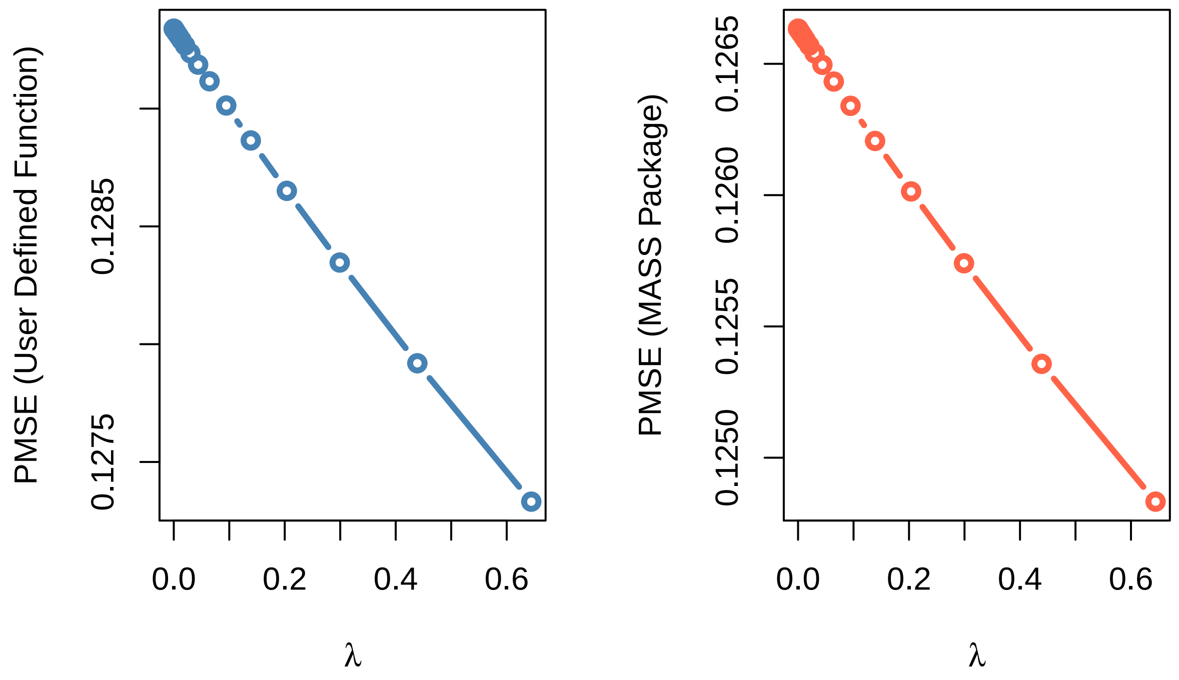
Results

In this section, we will describe the results that we have obtained for assessment of test errors using different strategies namely, Predictive Mean Squared Errors (PMSE) evaluated on validation set and Cross-Validation methods.

Predictive Mean Squared Errors (PMSE)

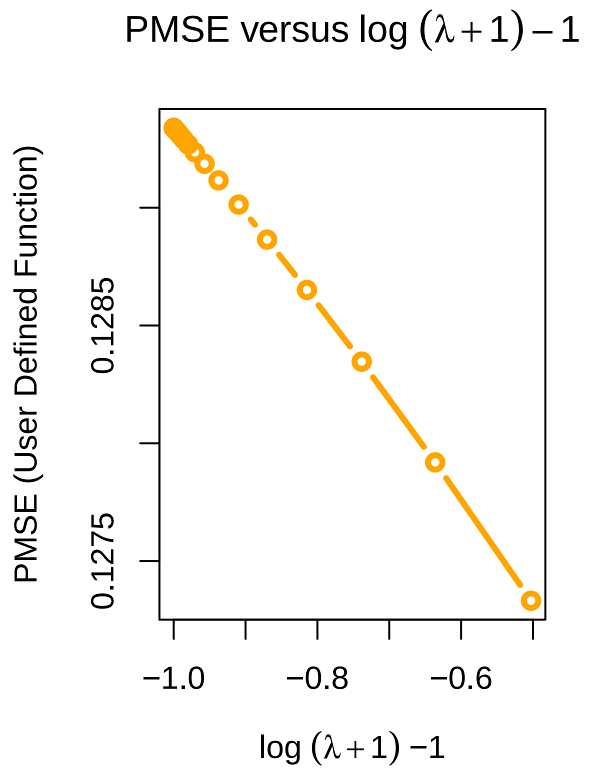
First, let us explore the results of the PMSE. We have plotted the PMSE values against the λ, This plot below clearly suggests that the values of PMSE are decreasing with increasing values of λ. The result on the left consists of ridge regression coefficients estimated using our own user defined function and on the right, we have defined the results using coefficients of ridge regression estimated using the ridge regression function `lm.rige` of `MASS` package.

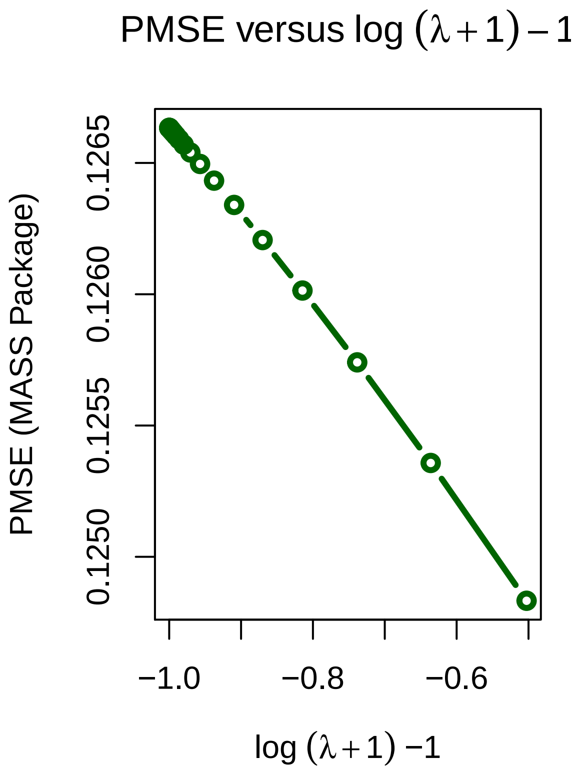
```r
z_lambda <- compute_PMSE(X, Y, X_validate, lambda.v.f)
par( mfrow=c(1,2) )
plot(z_lambda$lambda, z_lambda$User, type = 'b', col="steelblue", lwd=3,
     xlab = expression(lambda),
     ylab = "PMSE (User Defined Function)")
plot(z_lambda$lambda, z_lambda$Mass, type = 'b', col="tomato", lwd=3,
     xlab = expression(lambda),
     ylab = "PMSE (MASS Package)")
```



Similarily, we can also observe the variation in values of PMSE versus $log(\lambda + 1) - 1$ for ridge regression coefficients estimated using our own user defined function on the left and on the right, we have defined the results using coefficients of ridge regression estimated using the ridge regression function `lm.rige` of `MASS` package. This experiment also tells how good estimates of ridge regression coefficients are for our own function in comparison with the standard packages available.

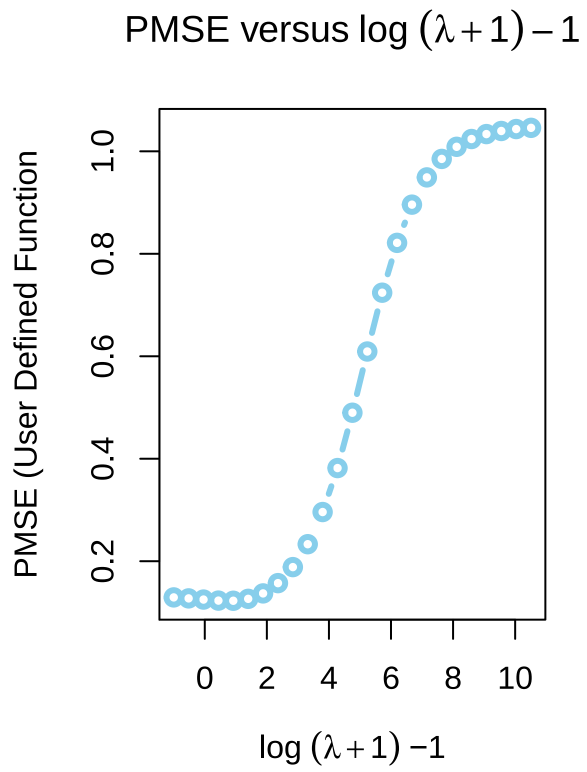
```r
par(mfrow=c(1,2))
plot((log(z_lambda$lambda +1)-1), z_lambda$User, type = 'b', col="orange",
     lwd=3, main = expression("PMSE versus"~"log"~(lambda+1)-1),
     xlab = expression("log"~(lambda + 1)~"-1"),
     ylab = "PMSE (User Defined Function)")
plot((log(z_lambda$lambda +1)-1), z_lambda$Mass, type = 'b', col="darkgreen",
     lwd=3, main = expression("PMSE versus"~"log"~(lambda+1)-1),
     xlab = expression("log"~(lambda + 1)~"-1"),
     ylab = "PMSE (MASS Package)")
```

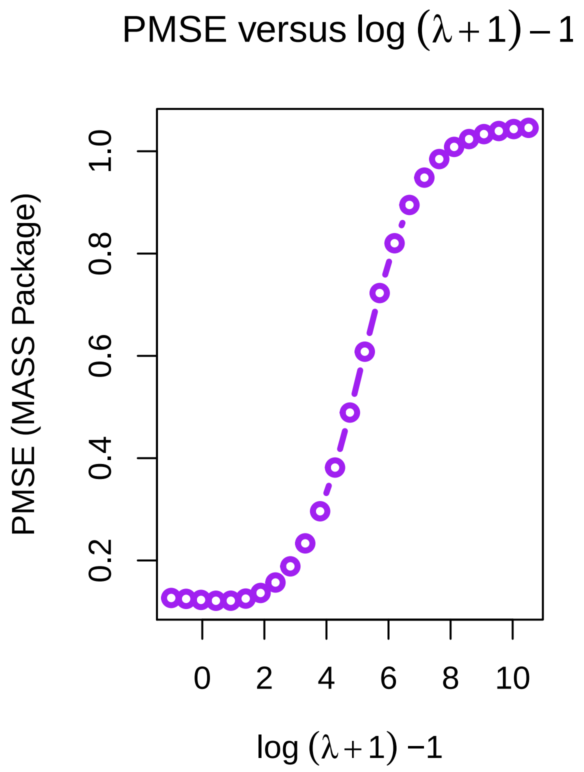




Now, If instead of chosing the sequence of λ using our scientifically defined function `compute_lambda_sequence`, we had used any arbitray sequence of values of λ, the plot of Mean Squared Error versus $log(\lambda + 1) - 1$ for the two methods would have looked like someting below-

```
lambda.max <- 1e5
n.lambdas <- 25
lambdas <- exp(seq(0,log(lambda.max+1),length=n.lambdas))-1

z <- compute_PMSE(X, Y, X_validate, lambdas)
par(mfrow=c(1,2))
plot(log(z$lambda+1)-1, z$User, type = 'b', col="skyblue",
     lwd=3, main = expression("PMSE versus"~"log"~(lambda+1)-1),
     xlab = expression("log"~(lambda + 1)~"-1"),
     ylab = "PMSE (User Defined Function")
plot(log(z$lambda+1)-1, z$Mass, type = 'b', col="purple",
     lwd=3, main = expression("PMSE versus"~"log"~(lambda+1)-1),
     xlab = expression("log"~(lambda + 1)~"-1"),
     ylab = "PMSE (MASS Package)")
```



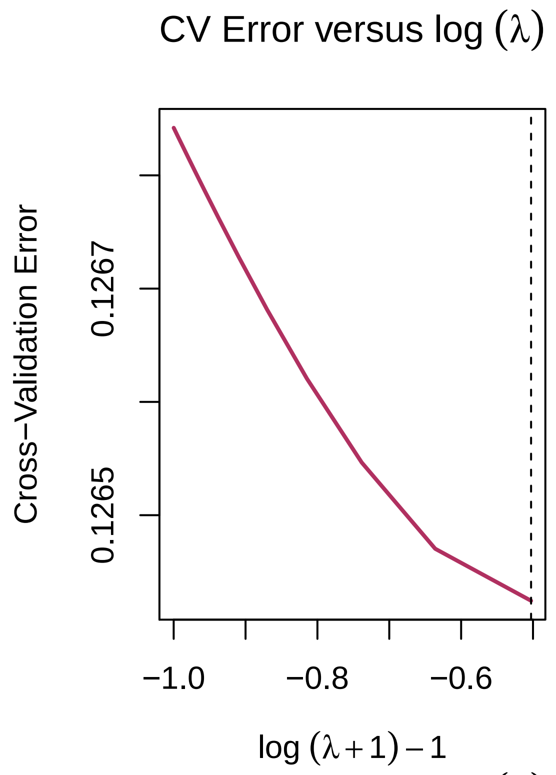


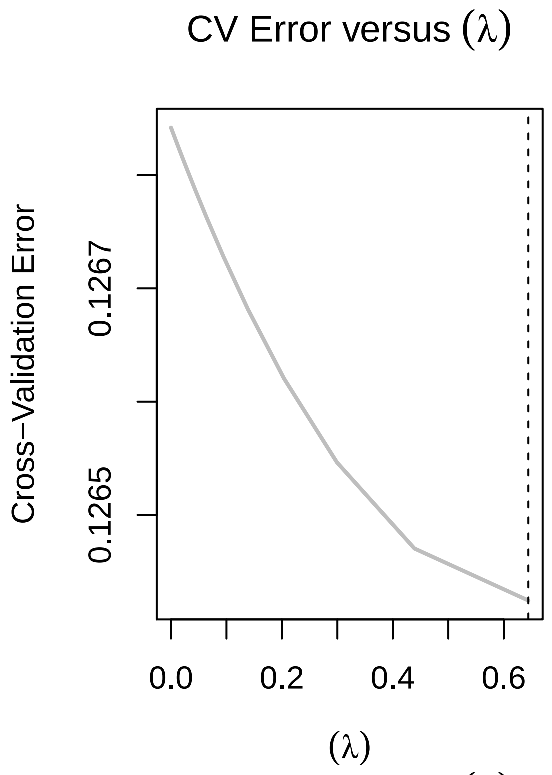
This clearly signifies the imprtance of correct sequence of λ. As we observe, the PMSE first decreases then attains a constant for larger values of $log(\lambda + 1) - 1$. Obviously, we are not very interested in computing the PMSE for such values of λ for which the decrease in PMSE is very less in comparison to the computing power spent.

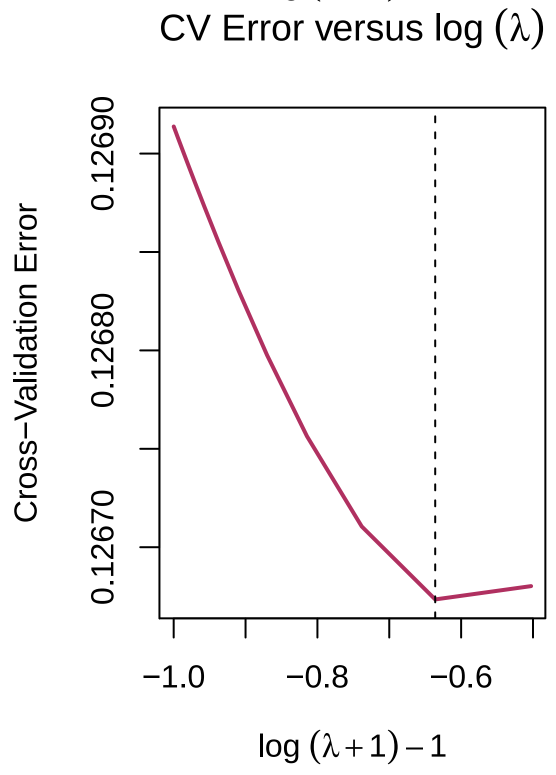
Cross-Validation

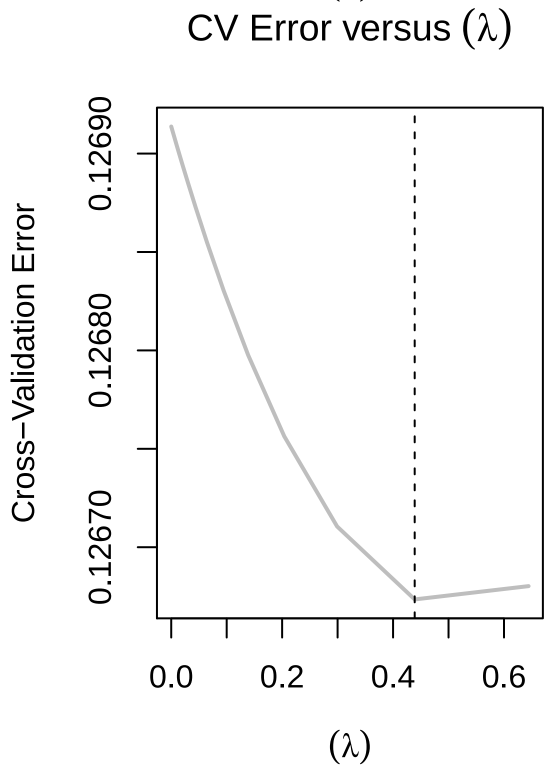
In order to be as abstract as possible, we performed cross validation 4 times with K=10 folds on the `prostate cancer` dataset and the results are provided with the visualization below-

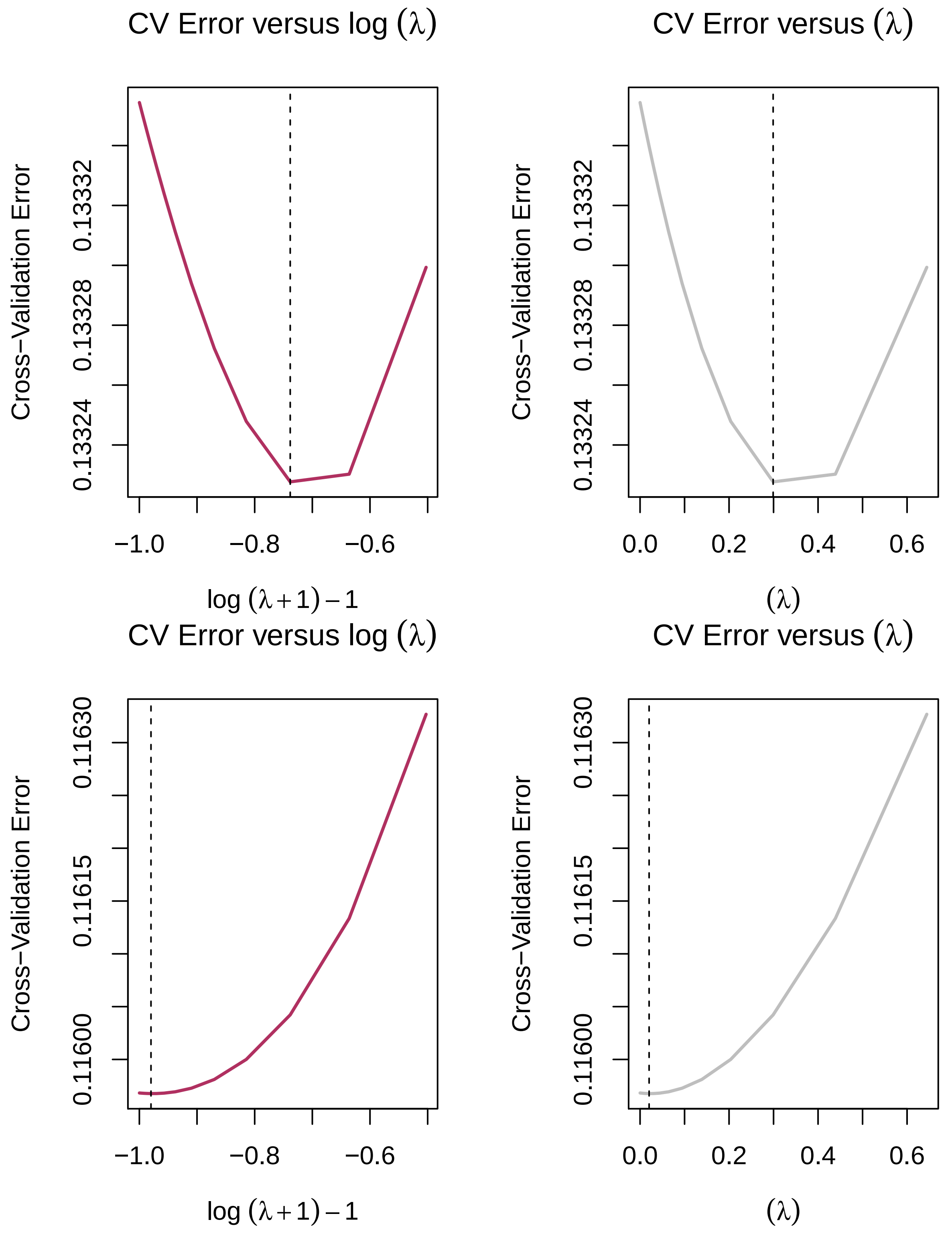
```r
par(mfrow=c(4,2))
CV_error <- numeric(4)
tuning_parameter <- numeric(4)
for(i in 1:4){

  W <- K_fold_CV(X, Y, lambda.v.f, K=10)
  CV_error[i] <- W$cvm[which.min(W$cvm)]
  tuning_parameter[i] <- W$lambda[which.min(W$cvm)]

  par(mfrow=c(1,2))
  plot(log(W$lambda + 1)-1, W$cvm, type = 'l', col="maroon", pch=1, lwd=2,
     xlab = expression("log"~(lambda+1)-1), ylab = "Cross-Validation Error",
     main = expression("CV Error versus log"~(lambda)))
  abline(v =(log(W$lambda+1)-1)[which.min(W$cvm)], lty=2)
  plot(W$lambda, W$cvm, type = 'l', col="grey", pch=1, lwd=2,
     xlab = expression((lambda)), ylab = "Cross-Validation Error" ,
     main = expression("CV Error versus"~(lambda)))
  abline(v =W$lambda[which.min(W$cvm)], lty=2)
}
```


CV Error versus log (λ)
Cross−Validation Error
0.1265
0.1267
−1.0
−0.8
−0.6
log (λ + 1) − 1


CV Error versus (λ)
Cross−Validation Error
0.1265
0.1267
0.0
0.2
0.4
0.6
(λ)


CV Error versus log (λ)
Cross−Validation Error
0.12670
0.12680
0.12690
−1.0
−0.8
−0.6
log (λ + 1) − 1


CV Error versus (λ)
Cross−Validation Error
0.12670
0.12680
0.12690
0.0
0.2
0.4
0.6
(λ)



Clearly, this randomness in the curves of `Cross-Validation Error` versus log (λ) and λ respectively for each of the four iteration is attributed to the random selection of training and validation datasets. This verifies that the minimum cross validation error varies in each of these four iterations. The corresponding minimun cross-validation errors in these four iterations are provided below-

```r
for(i in 1:4){
  print(paste("Iteration", i, ":", "CV Error",round(CV_error[i],5),
              "|    Tuning Parameter :", tuning_parameter[i]))
}
```

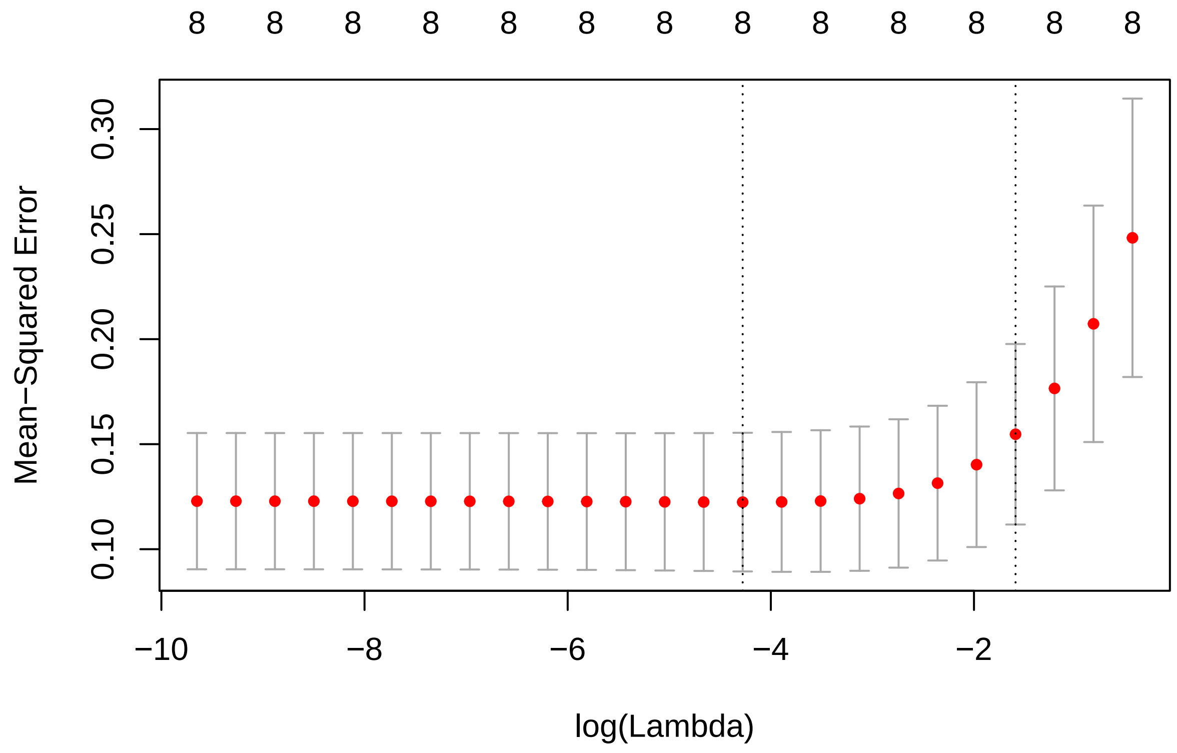
```
## [1] "Iteration 1 : CV Error 0.12642 |    Tuning Parameter : 0.6443233424"
## [1] "Iteration 2 : CV Error 0.12667 |    Tuning Parameter : 0.4389723831"
## [1] "Iteration 3 : CV Error 0.13323 |    Tuning Parameter : 0.2990684031"
## [1] "Iteration 4 : CV Error 0.11597 |    Tuning Parameter : 0.0203752931"
```

In order to compare how well our function for computing the cross validation error associated with ridge regression is performing, let us compare the cross-validation error approximated by our function above with the cross validation error that we obtain if we use the standard cross validation function `cv.glmnet` of the `glmnet` package. For this, we set the number of folds=10 and use the same sequence of lambda values as we had used in our own function. By setting alpha=0, we are telling the function to run the cross validation function for ridge regression. The code below demonstrates thw use of glmnet package for computing cross validation error associated ridge regression-

```r
library(glmnet)
```

```
## Loading required package: Matrix

## Loading required package: foreach

## Loaded glmnet 2.0-13
```

```r
cv.out=cv.glmnet(X,Y,alpha=0, nfolds = 10, lambda = lambda.v.f)
plot(cv.out)
```



```r
bestlam = cv.out$lambda.min
print(paste("Minimum Cross Validation Error:", cv.out$cvm[which.min(cv.out$cvm)],
            " | Tuning Parameter:", bestlam))
```

```
## [1] "Minimum Cross Validation Error: 0.122411313833964  | Tuning Parameter: 0.0138815256"
```

We can observe from the output of the above statement that the Minimum Cross-Validation Error identitified by our function was very close to that identified by the glmnet's function. Also, the value of tuning paramter is also similar to the one predicted in our experimentation.

Comparing the results of cross-validation errors

If we use the Predictive Mean Squared Error as a measure of estimating the test error, then we may get following statistics as the best estimate of our tuning paramter λ associated with the Minimum Cross-Validation Error.

```
print("Estimates based on PMSE")
```

```
## [1] "Estimates based on PMSE"
```

```
print(paste("Minimum Cross Validation Error:", min(z_lambda$User),
            " | Tuning Parameter:", z_lambda$lambda[which.min(z_lambda$User)]))
```

```
## [1] "Minimum Cross Validation Error: 0.127331545158687  | Tuning Parameter: 0.6443233424"
```

Now, Consider the following piece of code which makes the use of five fold cross validation on the training prostate cancer dataset which we had been using-

```
five_fold_CV <- K_fold_CV(X, Y, lambda.v.f, K=5)
five_fold_error <- five_fold_CV$cvm[which.min(five_fold_CV$cvm)]
five_fold_tuning_parameter <- five_fold_CV$lambda[which.min(five_fold_CV$cvm)]
print(paste("Minimum Cross Validation Error:", five_fold_error,
            " | Tuning Parameter:", five_fold_tuning_parameter))
```

```
## [1] "Minimum Cross Validation Error: 0.116287005617498  | Tuning Parameter: 0.0644323342"
```

This piece of code suggests the use of above mentioned statistics for choosing the best parameter values. If instead, we compare the results with the ten fold cross-validation performed on the same dataset using the same sequence of λ values for deciding the tuning parameter, we obtain the following statistics-

```
print(paste("Minimum Cross Validation Error:", min(CV_error),
            " | Tuning Parameter:", tuning_parameter[which.min(CV_error)]))
```

```
## [1] "Minimum Cross Validation Error: 0.115967676591156  | Tuning Parameter: 0.0203752931"
```

Further, if we compare the values of tuning paramter=0.01420817 based on Leave-one-out Cross validation where Cross-Validation error was approximately 0.117208701515139 and and Generalized Cross Validation for which tuning parameter was approximately 0.0142081721 with a cross-validation error of 0.120951237087727, then we can easily see that the ten fold cross validation is performing best (although ten five cross validation was almost the same) in terms of the least Cross-Validation Error.

fitting the ridge regression model on Boston Corrected Dataset

```
load("~/Documents/Statistical_Learning_Theory/Session 2/boston.Rdata")
head(cbind(boston.utm, boston.c))
```

```
##        x       y       TOWN TOWNNO TRACT      LON     LAT MEDV CMEDV
## 1 338.73 4679.73     Nahant      0  2011 -70.9550 42.2550 24.0  24.0
## 2 339.23 4683.33 Swampscott      1  2021 -70.9500 42.2875 21.6  21.6
## 3 340.37 4682.80 Swampscott      1  2022 -70.9360 42.2830 34.7  34.7
## 4 341.05 4683.89 Marblehead      2  2031 -70.9280 42.2930 33.4  33.4
## 5 341.56 4684.44 Marblehead      2  2032 -70.9220 42.2980 36.2  36.2
## 6 342.03 4685.09 Marblehead      2  2033 -70.9165 42.3040 28.7  28.7
##      CRIM ZN INDUS CHAS   NOX    RM  AGE    DIS RAD TAX PTRATIO      B
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12
##   LSTAT
## 1  4.98
## 2  9.14
## 3  4.03
## 4  2.94
## 5  5.33
## 6  5.21
```

```r
x <- data.matrix(cbind(boston.c[,-c(1,6)], boston.utm[,1:2]))
y <- data.matrix(boston.c$MEDV)
ridge.mod.boston <- ridge.reg(x, y, lambda.v.f)
print(ridge.mod.boston[,1])
```

```
##  [1] -0.1329331 -0.1329752 -0.1329980 -0.1330093 -0.1330125 -0.1330092
##  [7] -0.1329992 -0.1329815 -0.1329533 -0.1329110 -0.1328485 -0.1327575
## [13] -0.1326263 -0.1324393 -0.1321765 -0.1318142 -0.1313275 -0.1306959
## [19] -0.1299118 -0.1289900 -0.1279727 -0.1269247 -0.1259180 -0.1250130
## [25] -0.1242460
```

Discussions

Following is a set of practical advise that was considered as important while using `glmnet()`-

- It is advisable to use the function `data.matrix` instead of `as.matrix` for converting the dataframe of input and output dataset into a matrix of input and output data respectively. It is so becuase the package `glmnet()` prefers the `data.matrix` over `as.matrix` for some reasons. So, wwhenever we encounter an error similar to `Error in cbind2(1, newx) %*% nbeta : invalid class 'NA' to dup_mMatrix_as_dgeMatrix`, It is important to see if we have used `data.matrix` for conversion of dataframe to matrix or not.
- Often, we observed that it was not feasible to keep the value of K very high in deciding how many folds to choose for K-fold cross validation method as it increases the computation time significantly on high dimension/large data sets. Recommended is 5 or 10. Although, experiments suggested that any value of K between 5 and 10 approximately worked fine.
- It was also recommended to perform the scaling of the variables prior to applying these methods for estimating the test error as such a method yielded optimal results.

Conclusions

The ridge regression proved to be highly effective in case of having data such that $X^T X^{-1}$ was difficult in terms of numerical computation, thereby creating obstacles in the way of linear mdoel fitting. A number of methods were explored in order to estimate the prediction accuracy of the model. We look towards understanding the `bootstrap` which is also a resampling method in order to gain more insight about how we can improve the accuracy of prediction of our model.

References

- Lecture by Aarti Singh- Carnegie Melon University- Machine Learning