Exercise 7 - Comparing penalized regression estimators

Konstantinos Vakalopoulos 12223236

2023-11-29

Contents

Task 1

Task 1.1

Write your own function for the lasso using the shooting algorithm. Give default values for the tolerance limit and the maximum number of iterations. Do not forget to compute the coefficient for the intercept.

We are going to implement the lasso regression algorithm using the shooting algorithm according to the slides 32-34. First we introduce the soft function which corresponds to so-called soft thresholding.

```
#softmax function
softmax_R <- function(x, y){
  sign(x) * pmax(abs(x) - y, 0)
}
```

Then, in the next code block, the shooting algorithm is presented. Regarding the tolerance limit and the maximum number of iterations, 10^{-5} limit and 10000 iterations are taken intuitively. The function takes as input the X matrix, the y (response) variable, a lambda value (here we take lambda equals to 10), the tolerance limit and the maximum iterations. The output of the function is a list that contains the beta coefficients (including the intercept), the number of iterations and a Boolean variable in case the tolerance limit was exceeded.

It is worth noting that in order to avoid long execution times of the function, only a single for loop was used instead of two, as the second one was replaced by matrix multiplication using the crossprod() function.

```
lasso_shoot_algorithm <- function(X,y,lambda = 10,epsilon = 1e-5, max_iter = 10000){
  # Convert into a matrix (in case we didn't before the function was called)
  X <- as.matrix(X)

  p <- ncol(X)

  # Initialize the beta coeffs
  beta <- numeric(p)

  converged <- FALSE
  iteration <- 0

  # Matrix multiplication
  XX <- crossprod(X, X)
  Xy <- crossprod(X, y)

  while (!converged & (iteration < max_iter)){
    # Assign the previous betas to compare them with the new ones
    beta_prev <- beta
    for (j in 1:p){
      # Calculation of alpha
      a <- 2 * XX[j,j]
      # Calculation of c
      c <- 2 * (Xy[j] - sum(XX[j,] %*% beta) + beta[j] * XX[j,j])
      # calculation of beta using the soft function
      beta[j] <- softmax_R(c/a, lambda/a)
    }
    iteration <- iteration + 1
    converged <- sum(abs(beta - beta_prev)) < epsilon
  }
```

```
  # Add the intercept
  beta <- c(mean(y),beta)
  beta_coef <- list(beta = beta, n_iter = iteration, converged = converged)
  return(beta_coef)
}
```

For demonstration purposes, the artificial data set from the slides was used in order to perform the shooting algorithm.

```
# Create artificial data
set.seed(12223236)
n <- 500
p <- 20
X <- matrix(rnorm(n*p), ncol=p)
X <- scale(X, center=TRUE, scale=FALSE)
eps <- rnorm(n, sd=5)
beta <- 3:5
y <- 2 + X[,1:3] %*% beta + eps
```

Then, we execute the function. Below, the beta coefficients are presented along with the execution time.

```
startTime <- Sys.time()
lasso_shoot_algorithm(X,y)
```

```
## $beta
##  [1]  1.99458017  3.27456371  4.32608054  4.99592562 -0.34033616  0.21922159
##  [7] -0.17695649 -0.12018672  0.11462585 -0.17333884  0.09091399 -0.49405470
## [13] -0.28374160  0.03853852 -0.17936291  0.01650682 -0.02507751  0.05190201
## [19] -0.09721631  0.40326100  0.12191055
##
## $n_iter
## [1] 8
##
## $converged
## [1] TRUE
```

```
endTime <- Sys.time()
# prints recorded time
print(endTime - startTime)
```

```
## Time difference of 0.06309199 secs
```

Task 1.2

Write a function which computes the lasso using your algorithm for a vector of λs and which returns the matrix of coefficients and the corresponding λ values.

In this part of the exercise, we are going to call for each lambda value the above shooting algorithm in order to return the matrix of coefficients and the corresponding λ values. Our function, now, takes as input the a sequence of lambda values, the X matrix and the y variable.

```r
lasso_shoot_algorithm_multiple_lambdas <- function(X,y, lambda){

  p <- ncol(X)
  # the dimension is length(lambda) for the rows and for the columns p+1
  # for all the coefs plus the intercept
  beta_coef <- matrix(0, nrow = length(lambda), ncol = p+1)

  for (k in 1:length(lambda)){
    betas <- lasso_shoot_algorithm(X,y,lambda[k])
    beta_coef[k,] <- betas$beta
  }
  return(beta_coef)
}
```

For 20 different values of lambda, the result will be:

```r
lambda.grid <- 10^seq(-2,10, length=20)
# Print the 12 rows
lasso_implementation <- lasso_shoot_algorithm_multiple_lambdas(X,y,lambda.grid)
# Change the rownames
# The names indicate the lambda values
rownames(lasso_implementation) <- as.character(lambda.grid)
head(lasso_implementation,12)
```

```
##                       [,1]     [,2]      [,3]      [,4]       [,5]      [,6]
## 0.01                1.99458 3.287742 4.3326499 5.0051579 -0.3492746 0.2279083
## 0.0428133239871939  1.99458 3.287698 4.3326283 5.0051275 -0.3492452 0.2278798
## 0.183298071083244   1.99458 3.287513 4.3325360 5.0049977 -0.3491195 0.2277576
## 0.784759970351461   1.99458 3.286720 4.3321404 5.0044419 -0.3485814 0.2272346
## 3.35981828628378    1.99458 3.283323 4.3304471 5.0020621 -0.3462774 0.2249955
## 14.3844988828766    1.99458 3.268780 4.3231973 4.9918737 -0.3364132 0.2154091
## 61.5848211066026    1.99458 3.208734 4.2956345 4.9457325 -0.2926828 0.1755899
## 263.665089873036    1.99458 2.969178 4.1502955 4.7549504 -0.1154166 0.0000000
## 1128.83789168469    1.99458 2.128243 3.4434726 3.9808234  0.0000000 0.0000000
## 4832.93023857175    1.99458 0.000000 0.3013725 0.5945352  0.0000000 0.0000000
## 20691.3808111479    1.99458 0.000000 0.0000000 0.0000000  0.0000000 0.0000000
## 88586.6790410081    1.99458 0.000000 0.0000000 0.0000000  0.0000000 0.0000000
##                         [,7]        [,8]       [,9]       [,10]       [,11]
## 0.01                -0.1903308 -0.12896191 0.12660002 -0.1850396 0.10177250
## 0.0428133239871939  -0.1902868 -0.12893308 0.12656069 -0.1850012 0.10173683
## 0.183298071083244   -0.1900988 -0.12880968 0.12639230 -0.1848367 0.10158413
## 0.784759970351461   -0.1892935 -0.12828136 0.12567138 -0.1841322 0.10093038
## 3.35981828628378    -0.1858461 -0.12601943 0.12258488 -0.1811162 0.09813145
## 14.3844988828766    -0.1710867 -0.11633538 0.10937051 -0.1682035 0.08614832
## 61.5848211066026    -0.1084479 -0.07898649 0.05007973 -0.1148521 0.03423964
## 263.665089873036     0.0000000  0.00000000 0.00000000  0.0000000 0.00000000
## 1128.83789168469     0.0000000  0.00000000 0.00000000  0.0000000 0.00000000
## 4832.93023857175     0.0000000  0.00000000 0.00000000  0.0000000 0.00000000
## 20691.3808111479     0.0000000  0.00000000 0.00000000  0.0000000 0.00000000
## 88586.6790410081     0.0000000  0.00000000 0.00000000  0.0000000 0.00000000
##                        [,12]      [,13]      [,14]      [,15]      [,16]
## 0.01                -0.5034386 -0.3007792 0.04947372 -0.1887404 0.02536968
## 0.0428133239871939  -0.5034078 -0.3007232 0.04943780 -0.1887096 0.02534057
```

```
## 0.183298071083244  -0.5032758 -0.3004836 0.04928403 -0.1885777 0.02521594
## 0.784759970351461  -0.5027109 -0.2994579 0.04862566 -0.1880131 0.02468234
## 3.35981828628378   -0.5002920 -0.2950662 0.04580696 -0.1855959 0.02239781
## 14.3844988828766   -0.4899362 -0.2762640 0.03373918 -0.1752472 0.01261701
## 61.5848211066026   -0.4480323 -0.1940224 0.00000000 -0.1298927 0.00000000
## 263.665089873036   -0.2814826  0.0000000 0.00000000  0.0000000 0.00000000
## 1128.83789168469    0.0000000  0.0000000 0.00000000  0.0000000 0.00000000
## 4832.93023857175    0.0000000  0.0000000 0.00000000  0.0000000 0.00000000
## 20691.3808111479    0.0000000  0.0000000 0.00000000  0.0000000 0.00000000
## 88586.6790410081    0.0000000  0.0000000 0.00000000  0.0000000 0.00000000
##                          [,17]      [,18]       [,19]     [,20]      [,21]
## 0.01                -0.03810129 0.06186102 -0.10998459 0.4177174 0.13511577
## 0.0428133239871939 -0.03805851 0.06182831 -0.10994265 0.4176700 0.13507239
## 0.183298071083244  -0.03787537 0.06168826 -0.10976310 0.4174667 0.13488669
## 0.784759970351461  -0.03709125 0.06108866 -0.10899436 0.4165963 0.13409165
## 3.35981828628378   -0.03373419 0.05852159 -0.10570317 0.4128699 0.13068783
## 14.3844988828766   -0.01936151 0.04753111 -0.09161245 0.3969162 0.11611494
## 61.5848211066026    0.00000000 0.00000000 -0.02934688 0.3328189 0.05555498
## 263.665089873036    0.00000000 0.00000000  0.00000000 0.1129440 0.00000000
## 1128.83789168469    0.00000000 0.00000000  0.00000000 0.0000000 0.00000000
## 4832.93023857175    0.00000000 0.00000000  0.00000000 0.0000000 0.00000000
## 20691.3808111479    0.00000000 0.00000000  0.00000000 0.0000000 0.00000000
## 88586.6790410081    0.00000000 0.00000000  0.00000000 0.0000000 0.00000000
```

As we could see, while the lambda is increasing and so as the penalty, the beta coefficients become zero.

Task 1.3

Compare the performance and output of your functions against the lasso implementation from glmnet.

First, we import the glmnet library. Afterwards, using the microbenchmark function, we compare the performance of out function against the lasso implementation from glmnet.

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-7
```

```
microbenchmark::microbenchmark("Own Implementation of Lasso" =
                     lasso_shoot_algorithm_multiple_lambdas(X,y,lambda.grid),
                              "Glmnet Lasso Implementation" =
                     glmnet(x=X, y=y, alpha=1, lambda=lambda.grid))
```

```
## Unit: milliseconds
##                          expr     min       lq      mean  median      uq     max
##  Own Implementation of Lasso 21.3952 22.39375 24.160865 23.2520 24.9148 38.8320
##  Glmnet Lasso Implementation  1.5309  1.72805  2.108218  1.9262  2.1202 10.1999
##  neval
##    100
##    100
```

As is evident from the table, the implementation using the glmnet library is computationally much faster. Also, the output of the glmnet will be:

```
model_glmnet_impl <- glmnet(x=X, y=y, alpha=1, lambda=lambda.grid)
model_glmnet_impl$beta
```

```
## 20 x 20 sparse Matrix of class "dgCMatrix"

##   [[ suppressing 20 column names 's0', 's1', 's2' ... ]]

##
## V1  . . . . . . . . . . . . . . . . . . . 0.06626123 2.483601  3.06097762  3.233527317
## V2  . . . . . . . . . . . . . . . . . . . 1.47399077 3.707624  4.20921625  4.305071913
## V3  . . . . . . . . . . . . . . . . . . . 2.04033646 4.298203  4.82774813  4.965431088
## V4  . . . . . . . . . . . . . . . . . . .          .         .        -0.18460566 -0.310689766
## V5  . . . . . . . . . . . . . . . . . . .          .         .         0.07489271  0.191991335
## V6  . . . . . . . . . . . . . . . . . . .          .         .         .          -0.133132418
## V7  . . . . . . . . . . . . . . . . . . .          .         .         .          -0.091268140
## V8  . . . . . . . . . . . . . . . . . . .          .         .         .           0.075001846
## V9  . . . . . . . . . . . . . . . . . . .          .         .         .          -0.136228590
## V10 . . . . . . . . . . . . . . . . . . .          .         .         .           0.056131197
## V11 . . . . . . . . . . . . . . . . . . .          .         .        -0.33532241 -0.461888561
## V12 . . . . . . . . . . . . . . . . . . .          .         .        -0.02775483 -0.228984721
## V13 . . . . . . . . . . . . . . . . . . .          .         .         .           0.003337591
## V14 . . . . . . . . . . . . . . . . . . .          .         .        -0.01063797 -0.147096902
## V15 . . . . . . . . . . . . . . . . . . .          .         .         .           .
## V16 . . . . . . . . . . . . . . . . . . .          .         .         .           .
## V17 . . . . . . . . . . . . . . . . . . .          .         .         .           0.018092668
## V18 . . . . . . . . . . . . . . . . . . .          .         .         .          -0.054439695
## V19 . . . . . . . . . . . . . . . . . . .          .         .         0.19237948  0.356873425
## V20 . . . . . . . . . . . . . . . . . . .          .         .         .           0.078062101
##
## V1   3.27485008
## V2   4.32582463
## V3   4.99610842
## V4  -0.34038111
## V5   0.21949020
## V6  -0.17700803
## V7  -0.11981398
## V8   0.11492874
## V9  -0.17347761
## V10  0.09114963
## V11 -0.49339586
## V12 -0.28419635
## V13  0.03850360
## V14 -0.17915721
## V15  0.01560906
## V16 -0.02512229
## V17  0.05209506
## V18 -0.09733460
## V19  0.40303999
## V20  0.12171421
```

The dots in the table above are translated as zeros, which mean that while the lambda increases, the beta coefficients are getting close to zero or they are actually zero. Now, comparing the two cases for the initial values of lambda, the results are approximately the same. However, as lambda increases, the results differ. It is also worth noting that in the results of the glmnet function, the intercept is not included. In our application, we include the intercept.

Task 1.4

Write a function to perform 10-fold cross-validation for the lasso using MSE as the performance measure. The object should be similarly to the cv.glmnet give the same plot and return the λ which minimizes the Root Mean Squared Error and Mean Squared Error, respectively.

The below function performs 10-fold cross-validation for the lasso using MSE as the performance measure. Takes as input the X matrix, the y variable, the sequence of lambda, the performance measure (default value is MSE) and the number of folds. The output of the function is an object similar to cv.glmnet function. The different parts of the object were calculated according to https://www.rdocumentation.org/packages/glmnet/versions/4.1-8/topics/cv.glmnet or by typing the command ?cv.glmnet.

The object will contain:

1. **cvm:** The mean cross-validated error, a vector of length "length(lambda)".
2. **cvsd:** Estimate of the standard error of cvm.
3. **lambda:** The values of lambda used in the fits.
4. **cvup:** Upper curve = cvm + cvsd.
5. **cvlo:** Lower curve = cvm - cvsd.
6. **name:** A text string indicating the type of measure (for plotting purposes).
7. **lambda:** The values of lambda used in the fits.
8. **index:** A one-column matrix with the indices of "lambda.min" and "lambda.1se" in the sequence of coefficients, fits, etc.
9. **lambda.min:** Value of lambda that gives the minimum cvm.
10. **lambda.1se:** Largest value of lambda such that the error is within 1 standard error of the minimum.

```
cross_validation <- function(X,y,lambda.grid = c(1,10),type.measure = "mse", num_folds = 10){
  X <- as.matrix(X)
  n <- length(lambda.grid)
  #Initializations for the object
  cvm <- numeric(n)
  cvsd <- numeric(n)
  cvup <- numeric(n)
  cvlo <- numeric(n)
  nzero <- numeric(n)
  lambda.min <- 0
  lambda.1se <- 0
  index <- matrix(c(NA,NA),nrow=2)

  for (i in 1:n){
    fold_size <- nrow(X) / num_folds
    cv.error <- numeric(num_folds)
    betas <- matrix(0,nrow = num_folds, ncol = ncol(X))
    set.seed(12223236)
    for (fold in 1:num_folds){

      # Create the validation sets
      validation_indices <- ((fold - 1) * fold_size + 1):(fold * fold_size)
```

```
    validation_set_X <- X[validation_indices, ]
    validation_set_y <- y[validation_indices]

    # Create the training sets
    training_indices <- setdiff(1:nrow(X), validation_indices)
    training_set_X <- X[training_indices, ]
    training_set_y <- y[training_indices]

    # Calculate the betas using our own lasso implementation
    coeff <- lasso_shoot_algorithm(training_set_X, training_set_y,lambda.grid[i])

    # The prediction is done by multiplying the validation X set with the betas
    # In the validation set we add a column of 1s for the intercept
    pred <- cbind(rep(1,length(validation_indices)),validation_set_X) %*% coeff$beta

    # We also want to return the betas
    # So we remove the intercept
    beta.coef <- coeff$beta[-1]
    betas[fold,] <- beta.coef

    # Calculate the corresponding performance measure
    if (type.measure == "mse"){
      cv.error[fold] <- mean((validation_set_y-pred)^2)
      name = "Mean Squared Error"
    }
    else if (type.measure == "rmse"){
      cv.error[fold] <- sqrt(mean((validation_set_y-pred)^2))
      name = "Root Mean Squared Error"
    }
  }
  # Calculations based on the cv.glmnet object
  cvm[i] <- mean(cv.error)
  cvsd[i] <- sd(cv.error)
  nzero[i] <- sum(rowMeans(betas)!=0)

}
# Calculations based on the cv.glmnet object
cvup <- cvm + cvsd
cvlo <- cvm - cvsd

# Calculations based on the cv.glmnet object
df <- data.frame(cvm=cvm,cvsd=cvsd,lambda.grid = lambda.grid,id=1:n)
index[1,]<- which.min(df$cvm)
lambda.min <- lambda.grid[index[1,]]
df_filter <- df[min(df$cvm)+df$cvsd[index[1,]]>=df$cvm,]
df_filter <- dplyr::filter(df_filter,cvm==max(cvm))
index[2,] <- df_filter$id
lambda.1se <- lambda.grid[index[2,]]

output <- list(cvm = cvm,
               cvsd = cvsd,
               name=name,
               cvup = cvup,
```

```
                cvlo = cvlo,
                index=index,
                lambda.1se=lambda.1se,
                lambda.min=lambda.min,
                lambda=lambda.grid,
                nzero=nzero)

  attr(output,"class") <- "cv.glmnet"
  return(output)
}
```

Therefor, if we run the above function using as performance measure the MSE, the result will be:

```
lambda.grid <- 10^seq(-2,10, length=20)
res.MSE <- cross_validation(X,y,lambda.grid)
res.MSE
```

```
##
## Call:  NULL
##
## Measure: Mean Squared Error
##
##     Lambda Index Measure    SE Nonzero
## min  263.7     8   28.30 5.872      10
## 1se 1128.8     9   32.44 7.738      10
```

For the performance measure RMSE:

```
res.RMSE <- cross_validation(X,y,lambda.grid, type.measure = "rmse")
res.RMSE
```

```
##
## Call:  NULL
##
## Measure: Root Mean Squared Error
##
##     Lambda Index Measure     SE Nonzero
## min  263.7     8   5.295 0.5459      10
## 1se 1128.8     9   5.662 0.6525      10
```

The λ which minimizes the Root Mean Squared Error and Mean Squared Error, respectively:

```
cat("For the Mean Squared Error: ", res.MSE$lambda.min)
```

```
## For the Mean Squared Error:  263.6651
```

```
cat("For the Root Mean Squared Error: ", res.RMSE$lambda.min)
```

```
## For the Root Mean Squared Error:  263.6651
```

Below, two plots are presented, one regarding our implementation and the second for the cv.glmnet. The plots show the MSE for lambda value.

```
plot(res.MSE)
```

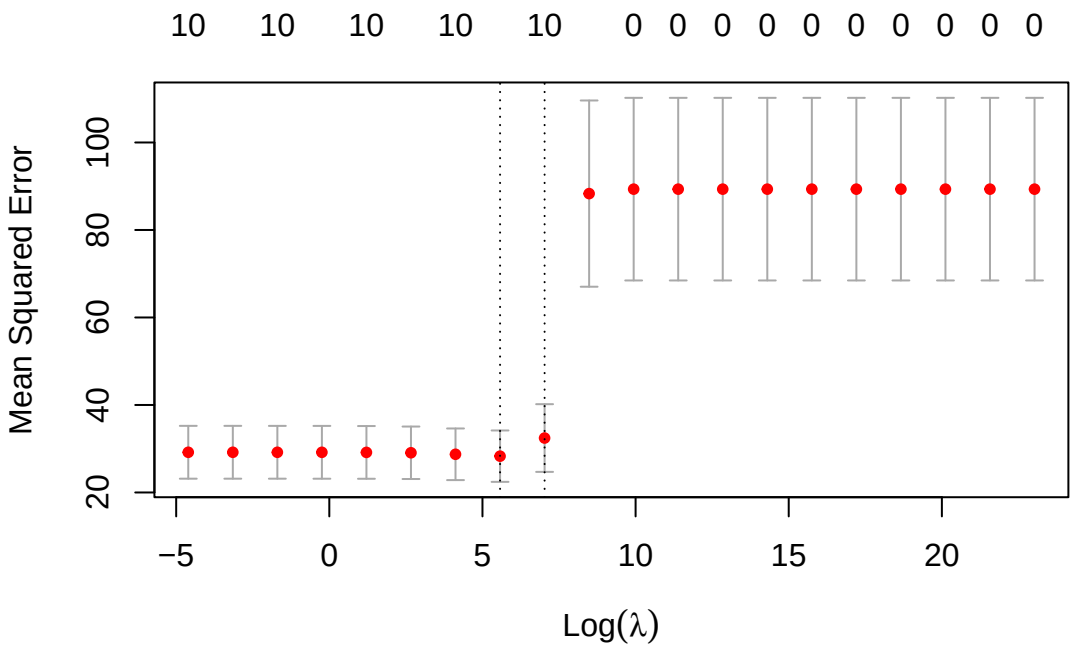

Figure 1: Own Implementation of Lasso algorithm

```
plot(cv.glmnet(X,y,nfolds = 10,lambda = lambda.grid))
```

Task 2

We will work with the Hitters data in the ISLR package. Take the salary variable as the response variable and create the model matrix x based on all other variables in the data set. Then divide the data into training and testing data with a ratio of 70:30.

First, we install the package "ISLR" and import the same library. From the package "ISLR", the "Hitters" data set is introduced. Using the commands str() and head(), we get an idea about the type of the variables and the data itself.

```
# uncomment for installation
# install.packages("ISLR")
library(ISLR)
data("Hitters")
str(Hitters)
```

```
## 'data.frame':    322 obs. of  20 variables:
##  $ AtBat    : int  293 315 479 496 321 594 185 298 323 401 ...
##  $ Hits     : int  66 81 130 141 87 169 37 73 81 92 ...
```

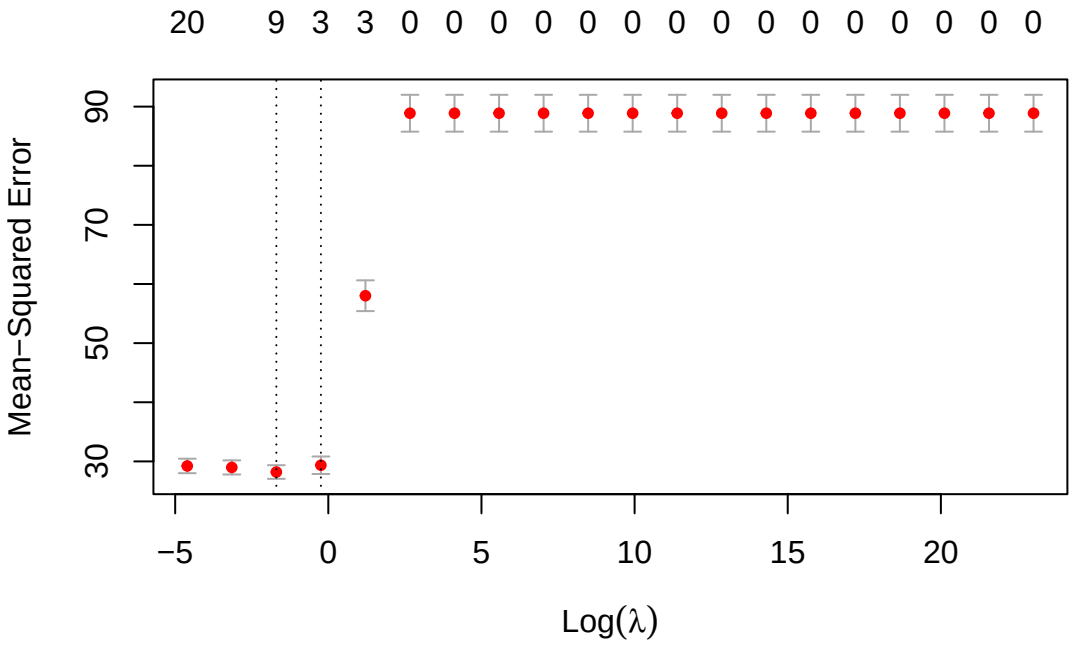

Figure 2: Glmnet Implementation of Lasso algorithm

```
##  $ HmRun    : int  1 7 18 20 10 4 1 0 6 17 ...
##  $ Runs     : int  30 24 66 65 39 74 23 24 26 49 ...
##  $ RBI      : int  29 38 72 78 42 51 8 24 32 66 ...
##  $ Walks    : int  14 39 76 37 30 35 21 7 8 65 ...
##  $ Years    : int  1 14 3 11 2 11 2 3 2 13 ...
##  $ CAtBat   : int  293 3449 1624 5628 396 4408 214 509 341 5206 ...
##  $ CHits    : int  66 835 457 1575 101 1133 42 108 86 1332 ...
##  $ CHmRun   : int  1 69 63 225 12 19 1 0 6 253 ...
##  $ CRuns    : int  30 321 224 828 48 501 30 41 32 784 ...
##  $ CRBI     : int  29 414 266 838 46 336 9 37 34 890 ...
##  $ CWalks   : int  14 375 263 354 33 194 24 12 8 866 ...
##  $ League   : Factor w/ 2 levels "A","N": 1 2 1 2 2 1 2 1 2 1 ...
##  $ Division : Factor w/ 2 levels "E","W": 1 2 2 1 1 2 1 2 2 1 ...
##  $ PutOuts  : int  446 632 880 200 805 282 76 121 143 0 ...
##  $ Assists  : int  33 43 82 11 40 421 127 283 290 0 ...
##  $ Errors   : int  20 10 14 3 4 25 7 9 19 0 ...
##  $ Salary   : num  NA 475 480 500 91.5 750 70 100 75 1100 ...
##  $ NewLeague: Factor w/ 2 levels "A","N": 1 2 1 2 2 1 1 1 2 1 ...
```

```
head(Hitters)
```

```
##                   AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun
## -Andy Allanson      293   66     1   30  29    14     1    293    66      1
## -Alan Ashby         315   81     7   24  38    39    14   3449   835     69
## -Alvin Davis        479  130    18   66  72    76     3   1624   457     63
## -Andre Dawson       496  141    20   65  78    37    11   5628  1575    225
```

```
## -Andres Galarraga   321   87    10   39  42    30     2    396   101     12
## -Alfredo Griffin    594  169     4   74  51    35    11   4408  1133     19
##                   CRuns CRBI CWalks League Division PutOuts Assists Errors
## -Andy Allanson       30   29     14      A        E     446      33     20
## -Alan Ashby         321  414    375      N        W     632      43     10
## -Alvin Davis        224  266    263      A        W     880      82     14
## -Andre Dawson       828  838    354      N        E     200      11      3
## -Andres Galarraga    48   46     33      N        E     805      40      4
## -Alfredo Griffin    501  336    194      A        W     282     421     25
##                   Salary NewLeague
## -Andy Allanson        NA         A
## -Alan Ashby        475.0         N
## -Alvin Davis       480.0         A
## -Andre Dawson      500.0         N
## -Andres Galarraga   91.5         N
## -Alfredo Griffin   750.0         A
```

As it is observed, all the variables are integers except of 3, which are factors. Therefore, data preprocessing is needed in order to proceed to the next tasks.

First, we remove the NAs observations and we assign to the X, all the variables except the response variable, which is the Salary variable. The response is assigned to the y. Then, we convert our factor variables to integer and we split the data into training and testing with a ratio of 70:30. To avoid data leakage, we center our integer variables for the train and the test set, seperately, because it is needed for the lasso algorithm. Finally, we convert the training and testing data into a matrix.

```
df_hitters <- Hitters

# Remove the NAs
df_hitters <- na.omit(df_hitters)

y <- df_hitters$Salary
X <- df_hitters[,!names(df_hitters) %in% "Salary"]

#Convert the factor variables to integer
indx <- sapply(X, is.factor)
X[indx] <- lapply(X[indx], function(x) as.integer(x))

# X <- as.matrix(X)

# Split the data into training and testing data with a ratio of 70:30
set.seed(12223236)
n <- nrow(X)
train <- sample(1:n,round(n*0.7))
test <- (1:n)[-train]

train_X <- X[train,]
train_y <- y[train]

#center the integer variables for the train
indx <- sapply(X, is.integer)
train_X[indx] <- lapply(train_X[indx], function(x) scale(x,center = TRUE, scale = FALSE))

train_X <- as.matrix(train_X)
```

```
test_X <- X[test,]
test_y <- y[test]

#center the integer variables for the test
indx <- sapply(X, is.integer)
test_X[indx] <- lapply(test_X[indx], function(x) scale(x,center = TRUE, scale = FALSE))
test_X <- as.matrix(test_X)
```

Task 2.1

Use your lasso function to decide which lambda is best here. Plot also the whole path for the coefficients.

For the same sequence of lambdas, as before, we use our lasso function.

```
lambda.grid <- 10^seq(-2,10, length=20)
# Own lasso implementation
startTime <- Sys.time()
own.res <- cross_validation(train_X, train_y, lambda.grid)
endTime <- Sys.time()
# prints recorded time
print(endTime - startTime)
```

```
## Time difference of 2.124457 mins
```

Therefore the best lambda is:

```
cat("The best lambda that minimizes the MSE is: ", own.res$lambda.min)
```

```
## The best lambda that minimizes the MSE is:  263.6651
```

In order to plot the whole path of the coefficients, we are going to call the function from task 1.2 using our lambda sequence. We remove the intercept and then we convert the beta matrix to a data frame. Finally, we change the names of the columns.

```
beta.df <- lasso_shoot_algorithm_multiple_lambdas(train_X,train_y,lambda.grid)[,-1]
beta.df <- as.data.frame(beta.df)

beta_vector <- character(ncol(X))
for (i in 1:length(beta_vector)) {
  beta_vector[i] <- paste0("Beta_", i)
}
# Change the names
colnames(beta.df) <- beta_vector
# We add as a column the lambda values
beta.df <- cbind(beta.df, lambda.grid)
```

Using the ggplot2 and tidyverse libraries, we plot the whole path for the coefficients.

```
library(ggplot2)
library(dplyr)
library(tidyr)

beta.df <- beta.df %>%
  gather(key = "variable", value = "value", -lambda.grid)
```

```
ggplot(beta.df, aes(x = log(lambda.grid), y = value)) +
  geom_line(aes(color = variable), size = 0.8) +
  ggtitle("Path for the Coefficients") +
  xlab("Log(lambda)") +
  ylab("Beta Values") +
  labs(color = "Beta Coefficients")
```

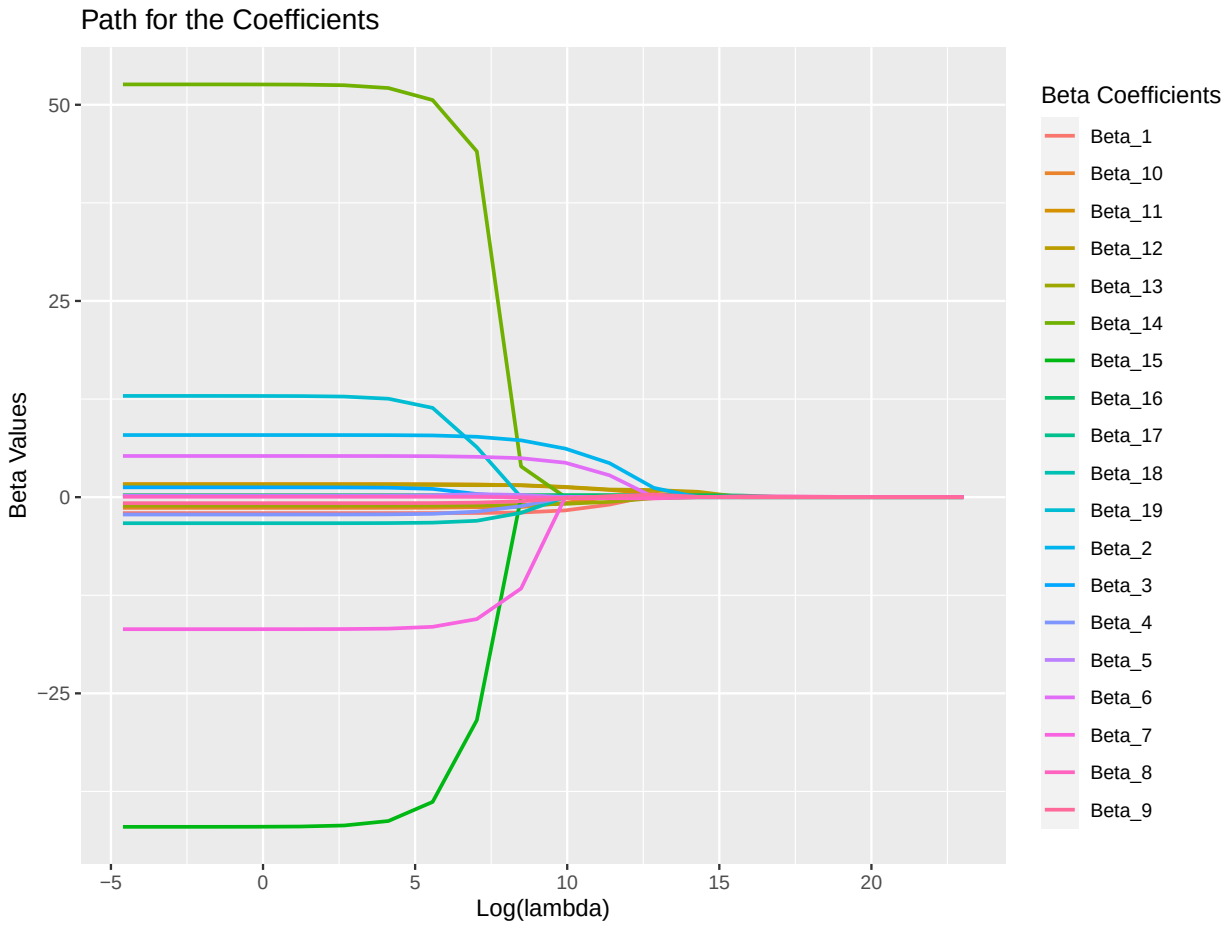

Figure 3: Coefficients Path

Task 2.2

Compare your fit against the lasso implementation from glmnet.

When we refer to fit, we, basically, test our model not to the testing data but to the training data. Therefore, for the glmnet implementation:

```
cv.glmnet.res <- cv.glmnet(train_X,train_y,nfolds = 10,lambda = lambda.grid)
res.glmnet.lasso <- glmnet(train_X,train_y, alpha=1, lambda = cv.glmnet.res$lambda.min)
```

More specifically, we first call the cv.glmnet in order to find the minimum lambda and based on that lambda we call the glmnet function. In order to implement the lasso algorithm, we assign 1 to the parameter alpha. The MSE and RMSE on the training data will be:

```
# GLMNET implementation
pred.lasso <- predict(res.glmnet.lasso, newx = train_X, s = cv.glmnet.res$lambda.min)
cat("The MSE is: ", mean((train_y-pred.lasso)^2))
```

```
## The MSE is:  93606.95
```

```
cat("\nThe RMSE is: ", sqrt(mean((train_y-pred.lasso)^2)))
```

```
##
## The RMSE is:  305.9525
```

Regarding our implementation, we perform the same procedure. We use the function from task 1.1 using as input the training data and the minimum lambda from the 10 fold cross validation.

```
# OWN LASSO implementation
res.own.lasso <- lasso_shoot_algorithm(train_X, train_y,own.res$lambda.min)
pred.own <- cbind(rep(1,nrow(train_X)),train_X) %*% res.own.lasso$beta

cat("The MSE is: ", mean((train_y-pred.own)^2))
```

```
## The MSE is:  82786.55
```

```
cat("\nThe RMSE is: ", sqrt(mean((train_y-pred.own)^2)))
```

```
##
## The RMSE is:  287.7265
```

As it is observed, the results in both cases are relatively close but not identical.

Task 2.3

Fit also a ridge regression and a least squares regression for the data (you can use here glmnet).

To implement ridge regression, we have to change the alpha parameter from 1 to 0. Also, for the least squares regression, we use the lm() function. Also, for the least squares regression we use the not centered data. Therefore:

```
# Ridge model
ridge.regression <- glmnet(train_X,train_y, alpha=0)
# LS model
ls.regression <- lm(Salary~.,data=df_hitters[train,])
```

Task 2.4

Compute the lasso, ridge regression and ls regression predictions for the testing data. Which method gives the better predictions? Interpret all three models and argue about their performances.

The table below presents the results for the three models. The performance measures are the RMSE and MSE.

```
# Lasso
pred.lasso <- predict(res.glmnet.lasso, newx = test_X, s = cv.glmnet.res$lambda.min)

# Ridge
pred.ridge<- predict(ridge.regression, newx = test_X)

# Least squares (best model)
pred.ls <- predict(ls.regression, newdata = df_hitters[test,])

data <- data.frame(
  Method = c("Lasso", "Ridge", "Least Squares"),
  MeanSquaredError = c(mean((test_y-pred.lasso)^2),
                       mean((test_y-pred.ridge)^2),
                       mean((test_y-pred.ls)^2)),
  RootMeanSquaredError = c(sqrt(mean((test_y-pred.lasso)^2)),
                           sqrt(mean((test_y-pred.ridge)^2)),
                           sqrt(mean((test_y-pred.ls)^2)))
)

knitr::kable(data, format = "markdown")
```

Method	MeanSquaredError	RootMeanSquaredError
Lasso	141008.1	375.5104
Ridge	153760.3	392.1228
Least Squares	132595.5	364.1366

According to the results, the best model, in terms of performance, is the least squares model. It makes sense that is the best because it takes into account all the variables in the data set. As mentioned below, in the cases of ridge and lasso model, some form of feature selection occurs as the lambda parameter takes larger values. Thus, the lasso and ridge regression models are simpler versions of the LS model, which tend to overfit.

Regarding the interpretation of the models, the ridge model is mentioned in task 3 and lasso model is mentioned in task 1.3 and task 3. As for the least squares model, all the coefficients are used to predict the response variable. There is no penalty term in the residual sum of squares, and thus, it is evident that the LS model tends to overfit the data.

Task 3

Explain the notion of regularized regression, shrinkage and how Ridge regression and LASSO regression differ.

The biggest problem in least squares regression is the multicollinearity, which is a statistical concept where several independent variables in a model are correlated. In LS regression, When the independent variables are correlated, the values of the beta coeffecients tend to have very large values. In order, then, to avoid

these large values, we have to shrink the beta coefficients by imposing a penalty on their size. This penalty applies in the residual sum of squares by adding a penalty/complexity parameter λ. This procedure is called Regularized regression, also known as shrinkage method.

There are two shrinkage methods, the ridge and lasso regression. In ridge regression our goal is to minimize the regular linear regression cost function along with a "L2 regularization" term. On the other hand, lasso regression minimizes the regular linear regression cost function along with an "L1 regularization" term. A significant difference between ridge and lasso regression, is that in ridge regression, while the lambda parameter tends to infinity, the beta coefficients tend to zero but they are never zero. However, lasso regression can lead to some coefficients becoming exactly zero, which can be used, effectively, to select a subset of features.