# Exercise 7 - Comparing penalized regression estimators

Bosse Behrens, st.id 12347333

2024W

## Task 1

### part 1

We implement Lasso using the shooting algorithm. As input we take a matrix X with the observated values of the predictors, a vactor y that contain the observations for the target variable, the Regularization parameter lambda, the tolerance limit epsilon with default value $10^{-6}$, the maximum iterations with default value 10000 and a factor that is the factor in front of the $a_j$ and $c_j$ values. This factor is 2 in the equations for the shooting algorithm we were provided, but we implement it to later show the difference with the glmnet lasso function. Since we don't want to standardize the data manually first, we first include the centering and scaling in the function. To do so we calculate columnwise average and standard deviation to then scale the provided data with it by centering around the means and then sclaing by the deviation. The target column y also gets centered around its mean. Then we compute the design and hat matrix to calculate the OLS estimators for the initial $\beta$ values. We also compute the vector a that contain the $a_j$ values for the shooting algorithm. Then we start iterating, by calculating the new $c_j$ values with the old beta and then updating the new beta using the ancillary soft function. The iterations stop if either the absolute sum of the difference between two consecutive beta vectors is less than the specified tolerance or the for loop ends by reaching the maximum iterations.

```
#setting default values for the tolerance limit epsilon and the maximum
#iterations max_iter
lasso_shooting <- function(X, y, lambda, epsilon = 1e-10, max_iter = 10000, factor = 2) {


  n <- nrow(X)
  m <- ncol(X)

  #standardizing the data
  X_mean <- colMeans(X)
  X_sd <- apply(X, 2, sd)
  X_std <- scale(X, center = X_mean, scale = X_sd)

  #scaling target variable
  y_mean <- mean(y)
  y_centered <- y - y_mean

  #computing initial beta values by calculating OLS estimators
  XtX <- t(X_std) %*% X_std
  Xty <- t(X_std) %*% y_centered
  beta_hat_0 <- solve(XtX, Xty)

  beta_old <- beta_hat_0
  beta_new <- beta_old
```

```
  #1/n translates to glmnets lasso implementation
  if (factor == "normal"){
    factor <- 1/n
  }

  a <- (factor) * colSums(X_std^2) #vector a with a_j values for shooting algorithm

  for (iter in 1:max_iter) {
    for (j in 1:m) {

      #calculating the new c_j values in every iteration with the new betas
      bracket_j <- y_centered - X_std %*% beta_new + beta_new[j] * X_std[, j]
      c_j <- (factor) * sum(X_std[, j] * bracket_j)

      #updating beta using the ancillary soft function
      beta_new_j <- c_j/a[j]
      delta <- lambda/a[j]
      beta_new[j] <- sign(beta_new_j) * max(abs(beta_new_j) - delta, 0) #x_+
    }

    #stop iterating if algorithm converged (difference in beta values less than
    # tolerance epsilon)
    if (sum(abs(beta_new - beta_old)) < epsilon) {
      break
    }

    #updating old beta as well for next iteraiton
    beta_old <- beta_new
  }

  #if max iterations have been iterated through, print message
  if (iter == max_iter){
      print("Max iterations reached, failed to converge")
  }

  #transform beta back so it can be used to predict on unscaled data
  beta_final <- beta_new / X_sd

  #calcualte the intercept
  intercept <- y_mean - sum(beta_final * X_mean)

  #final beta valeus includiong intercept
  beta_full <- c(Intercept = intercept, beta_final)

  return(beta_full)
}
```

**part 2**

We now use our function from part 1 to implement another one that also takes as input a vector of $\lambda$ values and will return a matrix with the coefficients for every *lambda* used. To do so we use the same inputs as in the function from part 1 except the lambdas which is now as mentioned a vector of values. Then the empty matrix to store the $\lambda$ and coefficients is created and at last the lasso function iterated over in a loop for every $\lambda$ value in the input vector.

```
lasso_shooting_lambda <- function(X, y, lambdas, epsilon = 1e-7,
                                  max_iter = 1000, factor = 2) {

  m=ncol(X)

  #creating the empty matrix frame for the coefficients for
  #different lambda vlaues
  coef_matrix <- matrix(0, nrow = m + 1, ncol = length(lambdas))
  rownames(coef_matrix) <- c("Intercept", paste0("X", 1:m))
  colnames(coef_matrix) <- paste0("Lambda_", lambdas)

  #iterating through the lambdas
  l <- length(lambdas)
  for (i in l:1) {

    #calling lasso function woith right lambda as input
    lambda <- lambdas[i]
    beta <- lasso_shooting(X, y, lambda, epsilon = epsilon,
                           max_iter = max_iter, factor = factor)

    #storing coefficients
    coef_matrix[,i] <- beta

  }

  return(coef_matrix)
}
```

**part 3**

We generate a matrix X of observations for the predictor variables, a vector y of values for the target variable and a sequence of lambda values that get exponentially larger.

```
set.seed(12347333)
n <- 200
m <- 5
X <- matrix(rnorm(n*m),nrow=200)
b <- c(1,2,3,4)
y <- X[,2:5] %*% b + rnorm(n, sd=1)
lambdas <- c(0.0001, 0.001, 0.01, 0.1, 1)
lambda_value <- 3
```

First we want to compare our lasso implementation and glmnet's with only one value for $\lambda$ provided. We write a function that does just this by computing the coeffivcients for both and then the abolute difference.

```
library(glmnet)
```

```
## Lade nötiges Paket: Matrix
```

```
## Loaded glmnet 4.1-8
```

```
single_lambda_evaluation <- function(X, y, lambda, factor = 2){
  beta_shooting <- lasso_shooting(X, y, lambda = lambda_value, factor = factor)

  lasso_glmnet <- glmnet(x=X, y=y, alpha=1,lambda = lambda_value)
```

```
  beta_glmnet <- as.numeric(coef(lasso_glmnet))

  beta_difference <- abs(beta_shooting - beta_glmnet)

  results_beta_single <- cbind(beta_shooting, beta_glmnet, beta_difference)

  print(results_beta_single)
}
```

First we compare the two outputs.

```
single_lambda_evaluation(X,y,lambda_value,2)
```

```
##           beta_shooting beta_glmnet beta_difference
## Intercept   0.001457419 -0.03437287      0.03583029
##            -0.065111443  0.00000000      0.06511144
##             0.851101784  0.00000000      0.85110178
##             2.079650741  0.00000000      2.07965074
##             2.928232161  0.16785542      2.76037674
##             3.989766738  1.00433859      2.98542815
```

As we can see there is a are significant differences in the coefficients. Also the custom lasso fails to actually shrink any coefficients to even close to zero. The differences stem from the way both functions work. Glmnet's lasso minimizes an object function that has a different pre-factor, which normalizes the $\lambda$ value in regard to the size of the data. In our own implementation this would translate to a factor of $\frac{1}{n}$ instead of 2. We therefore compare them again with this other value.

```
single_lambda_evaluation(X,y,lambda_value,factor="normal")
```

```
##           beta_shooting beta_glmnet beta_difference
## Intercept   -0.03443811 -0.03437287    6.524603e-05
##              0.00000000  0.00000000    0.000000e+00
##              0.00000000  0.00000000    0.000000e+00
##              0.00000000  0.00000000    0.000000e+00
##              0.16044680  0.16785542    7.408627e-03
##              0.99678016  1.00433859    7.558432e-03
```

As we can see, now the coefficients do not differ significantly. Therefore it probably has to do with adjusting the lambda accordingly to input size, but in general the optimal lambda should be searched for anyway and not just some arbitrary random value taken as we just did.\ Now we also want to compare the performance when using some vector of different lambdas on the second implemented function to glmnets. To do so we again compute the matrices of coefficients for both and the the absolute difference.

```
library(glmnet)

beta_shooting <- lasso_shooting_lambda(X, y, lambdas = lambdas)

lasso_glmnet <- glmnet(x = X, y = y, alpha = 1, lambda = lambdas)
beta_glmnet <- as.matrix(coef(lasso_glmnet))

beta_difference <- beta_shooting - beta_glmnet

t_beta_shooting <- t(beta_shooting)
t_beta_glmnet   <- t(beta_glmnet)
t_beta_difference <- t(beta_difference)

colnames(t_beta_shooting) <- paste0(colnames(t_beta_shooting), "_shooting")
```

```r
colnames(t_beta_glmnet)   <- paste0(colnames(t_beta_glmnet), "_glmnet")
colnames(t_beta_difference) <- paste0(colnames(t_beta_difference), "_diff")

results_beta_single <- as.data.frame(
  cbind(t_beta_shooting, t_beta_glmnet, t_beta_difference)
)

results_transposed <- as.data.frame(t(results_beta_single))
print(results_transposed)
```

```
##                     Lambda_1e-04 Lambda_0.001  Lambda_0.01    Lambda_0.1
## Intercept_shooting   0.00243458  0.002434287  0.002431356  2.402040e-03
## X1_shooting         -0.07232470 -0.072322542 -0.072300902 -7.208450e-02
## X2_shooting          0.85792694  0.857924892  0.857904416  8.576997e-01
## X3_shooting          2.08653835  2.086536282  2.086515618  2.086309e+00
## X4_shooting          2.93572877  2.935726526  2.935704036  2.935479e+00
## X5_shooting          3.99710306  3.997100857  3.997078847  3.996859e+00
## (Intercept)_glmnet  -0.02370547 -0.007885672  0.001136593  2.307758e-03
## V1_glmnet            0.00000000  0.000000000 -0.062744505 -7.139005e-02
## V2_glmnet            0.00000000  0.762947607  0.848841557  8.570153e-01
## V3_glmnet            1.21913886  1.995984127  2.077369623  2.085616e+00
## V4_glmnet            2.02901631  2.838505934  2.925763538  2.934741e+00
## V5_glmnet            2.97926632  3.898150242  3.987344671  3.996126e+00
## Intercept_diff       0.02614005  0.010319959  0.001294763  9.428144e-05
## X1_diff             -0.07232470 -0.072322542 -0.009556396 -6.944485e-04
## X2_diff              0.85792694  0.094977285  0.009062859  6.843091e-04
## X3_diff              0.86739948  0.090552154  0.009145995  6.928438e-04
## X4_diff              0.90671247  0.097220592  0.009940498  7.382246e-04
## X5_diff              1.01783673  0.098950614  0.009734176  7.328336e-04
##                        Lambda_1
## Intercept_shooting  0.0021088816
## X1_shooting        -0.0699204457
## X2_shooting         0.8556520392
## X3_shooting         2.0842426319
## X4_shooting         2.9332300707
## X5_shooting         3.9946577807
## (Intercept)_glmnet  0.0024219006
## V1_glmnet          -0.0722315355
## V2_glmnet           0.8578357420
## V3_glmnet           2.0864484770
## V4_glmnet           2.9356296339
## V5_glmnet           3.9970055191
## Intercept_diff     -0.0003130189
## X1_diff             0.0023110898
## X2_diff            -0.0021837028
## X3_diff            -0.0022058451
## X4_diff            -0.0023995632
## X5_diff            -0.0023477384
```

Here we see that there are again differences, but they are not consistent for different values of lambda. The smallest differences in coefficients are at the 0.01 $\lambda$ value, while for larger and smaller values they differ more again. We now want to check what the actual optimal $\lambda$ in this case would be by using cv.glmnet.

```r
cvgl <- cv.glmnet(X,y,alpha=1)
print(cvgl$lambda.min)
```

```
## [1] 0.01630247
```

As we can see the optimal value for $\lambda$ computed by crossvalidating is close to 0.01, which was also the value where both glmnets lasso and our own were the most similar in terms of coefficients. This is to be expected since both use different optimization algorithms and therefore have different behaviors in convergence and numeric precision. Also there if a difference for values of lambda since for large regularization strength they are behaving more differently due to the different algorithms, while for small values both are more similar to normal linear regression and therefore do not differ as much. On the optimal value of $\lambda$ though both will have a balance between regularization and best model fit, which tends to be a stable and consistent solution and therefore not depending on the algorithm used for implementation.

**part 4**

Now we write a funciton that uses our lasso implementation, takes again a sequence of lambdas as input and performs K-fold crossvalidation (K as input for the fold) on every input lambda using our lasso implementation. It then gives out a plot just like the cv.glmnet function that shows the mean MSE plotted vs the lambda values and a vertical line for the optimal lambda that minimizes the MSE. the function also returns the mean RMSEs and MSEs and the optimal lambda for both (which is the same since RMSE is just the squareroot of MSE).

```r
cv_lasso_shooting <- function(X, y, lambdas, K = 10, epsilon = 1e-7,
                              max_iter = 10000, factor = 2) {

  set.seed(12347333)

  n <- nrow(X)

  #splitting up the indices to the folds
  folds <- sample(rep(1:K, length.out = n))

  #creating empty matrix to store mse values
  mse_matrix <- matrix(NA, nrow = length(lambdas), ncol = K)


  for (fold in 1:K) {

    #setting test/train indices according to each fold of K
    test_idx <- which(folds == fold)
    train_idx <- setdiff(1:n, test_idx)

    #splitting data
    X_train <- X[train_idx, , drop=FALSE]
    y_train <- y[train_idx]
    X_test <- X[test_idx, , drop=FALSE]
    y_test <- y[test_idx]

    #using our lasso function to get the coefficients
    beta_matrix <- lasso_shooting_lambda(X_train, y_train, lambdas,
                                         epsilon = epsilon, max_iter = max_iter,
                                         factor = factor)

    #adding 1 for itnercept to design matrix
```

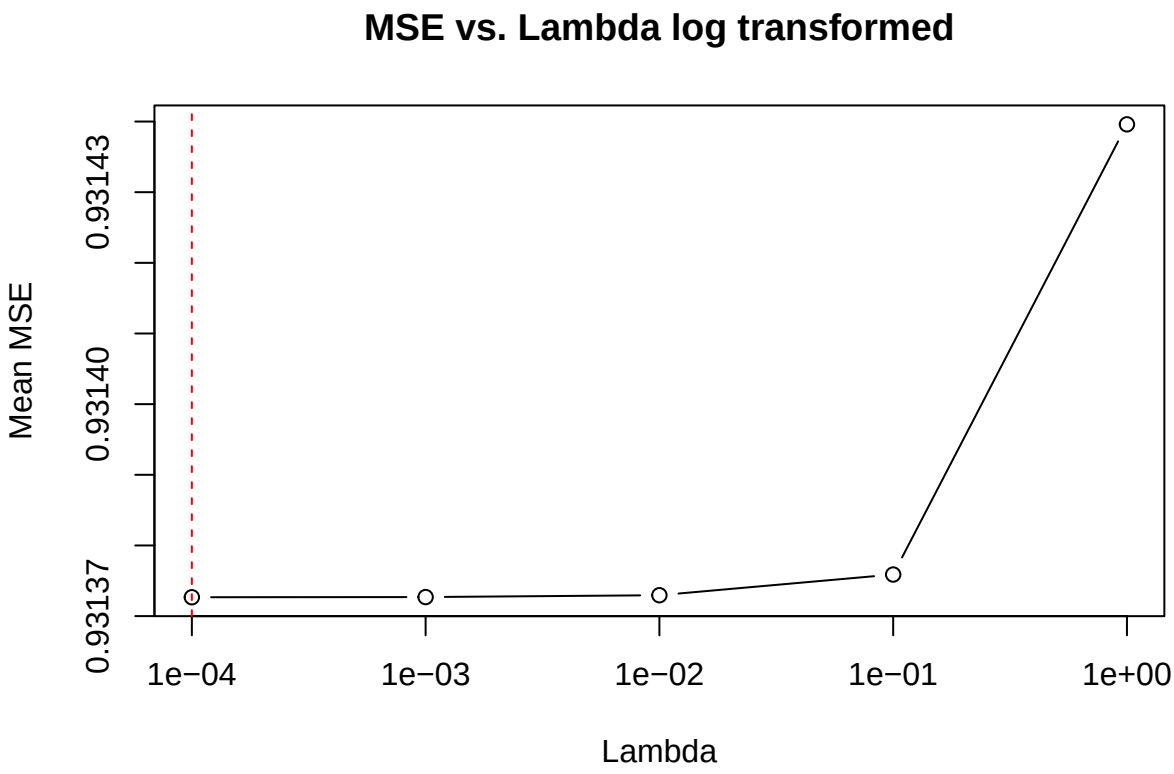
```
    X_test_with_intercept <- cbind(1, X_test)

    #making the predictions
    y_pred_matrix <- X_test_with_intercept %*% beta_matrix

    #getting the MSE for the fold in the iteration
    mse_fold <- colMeans((y_test - y_pred_matrix)^2)

    #saving MSE
    mse_matrix[, fold] <- mse_fold
  }

  #computing MSE and RMSE mean
  mean_mse <- rowMeans(mse_matrix, na.rm = TRUE)
  mean_rmse <- sqrt(mean_mse)

  #getting the optimal lambda vlaues for minimizing MSE/RMSE
  lambda.min.mse <- lambdas[which.min(mean_mse)]
  lambda.min.rmse <- lambdas[which.min(mean_rmse)]


  #storing results
  result <- list(
    lambda = lambdas,
    mean.mse = mean_mse,
    mean.rmse = mean_rmse,
    lambda.min.mse = lambda.min.mse,
    lambda.min.rmse = lambda.min.rmse
  )

  #plotting the spcified plot just as in cv.glmnet with log transform for better
  #readability
  plot(lambdas, mean_mse, type = "b",
       xlab = "Lambda", ylab = "Mean MSE", main = "MSE vs. Lambda log transformed",
       xlim = range(lambdas), log = "x")
  abline(v = lambda.min.mse, col = "red", lty = 2)


  return(result)

}
```

Now we generate a sequence of lambdas to test on.

```
lambdas <- c(0.0001, 0.001, 0.01, 0.1, 1)
```

Using our new function with the previously generated data.

```
results <- cv_lasso_shooting(X,y,lambdas, factor = 2)
```




```
print(results)
```

```
## $lambda
## [1] 1e-04 1e-03 1e-02 1e-01 1e+00
##
## $mean.mse
## [1] 0.9313727 0.9313727 0.9313730 0.9313759 0.9314396
##
## $mean.rmse
## [1] 0.9650765 0.9650765 0.9650767 0.9650782 0.9651112
##
## $lambda.min.mse
## [1] 1e-04
##
## $lambda.min.rmse
## [1] 1e-04
```

The plot shows us now the optimal $\lambda$ value in our tested sequence is 0.0001. WHile this differs from part 3 where using cv.glmnet the optimal value was around 0.16, this is still only a very small deviation and can happen due to different implementations.

## Task 2

### part 1

First we load the data.

```r
library(ISLR)
data(Hitters)
```

Now we want to split the data into train/test, but we first have to do some preprocessing. There are some missing value sin the target column Salaray and we simply delete these observations. Then there are three categorical columns, which we use one-hot encoding for. In one of these this encoding creates 2 new variables even though there ar eonly two different categories present, so we manually delete one again. Then we are done with preprocessing and split the data into train and test sets.

```r
set.seed(12347333)
library(dplyr)
```

```
##
## Attache Paket: 'dplyr'
```

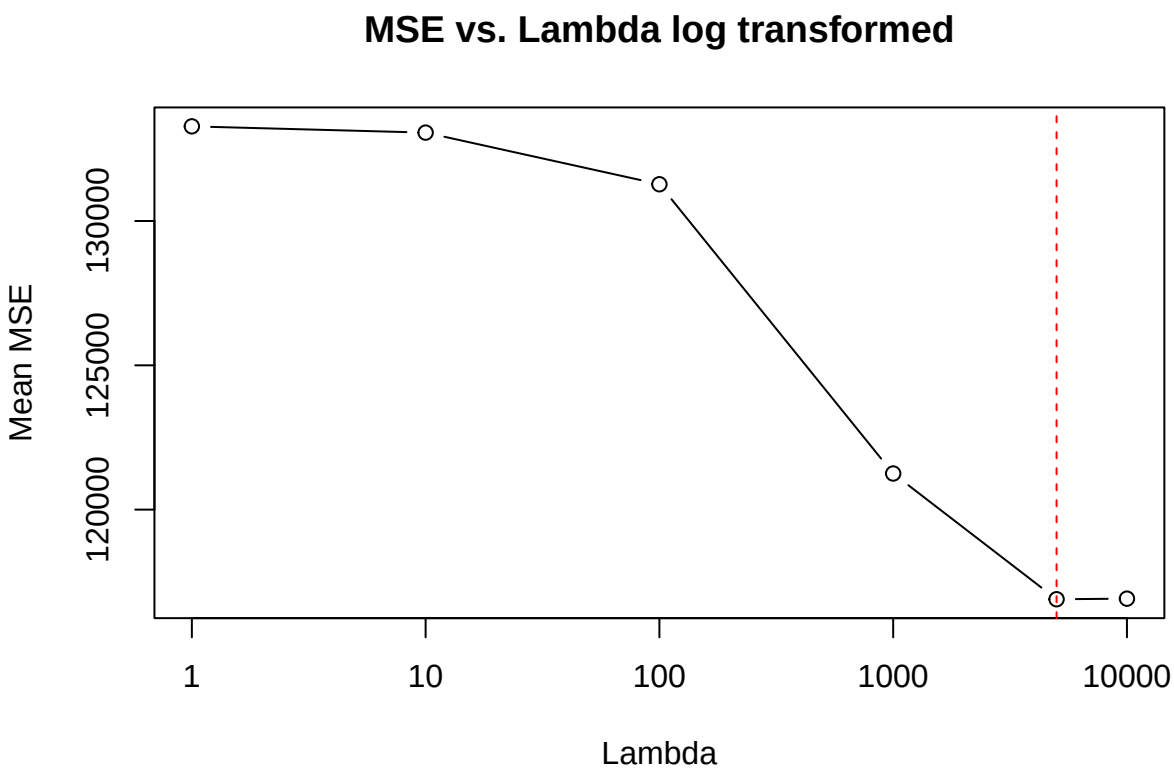
```
## Die folgenden Objekte sind maskiert von 'package:stats':
##
##     filter, lag
```

```
## Die folgenden Objekte sind maskiert von 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
Hitters <- Hitters %>% filter(!is.na(Salary))

X <- Hitters %>% select(-Salary)

X <- model.matrix(~ . - 1, data = X)

X <- X[, colnames(X) != "LeagueN"]

X <- as.matrix(X)

n <- nrow(X)
train <- sample(1:n, round(0.7 * n))
test <-(1:n)[-train]

X_train <- X[train,]
y_train <- Hitters[train, "Salary"]
X_test <- X[test,]
y_test <- Hitters[test, "Salary"]
```

We now test our cv lasso implementation by generating a sequence with a wide range of lambda values to plug into the function.

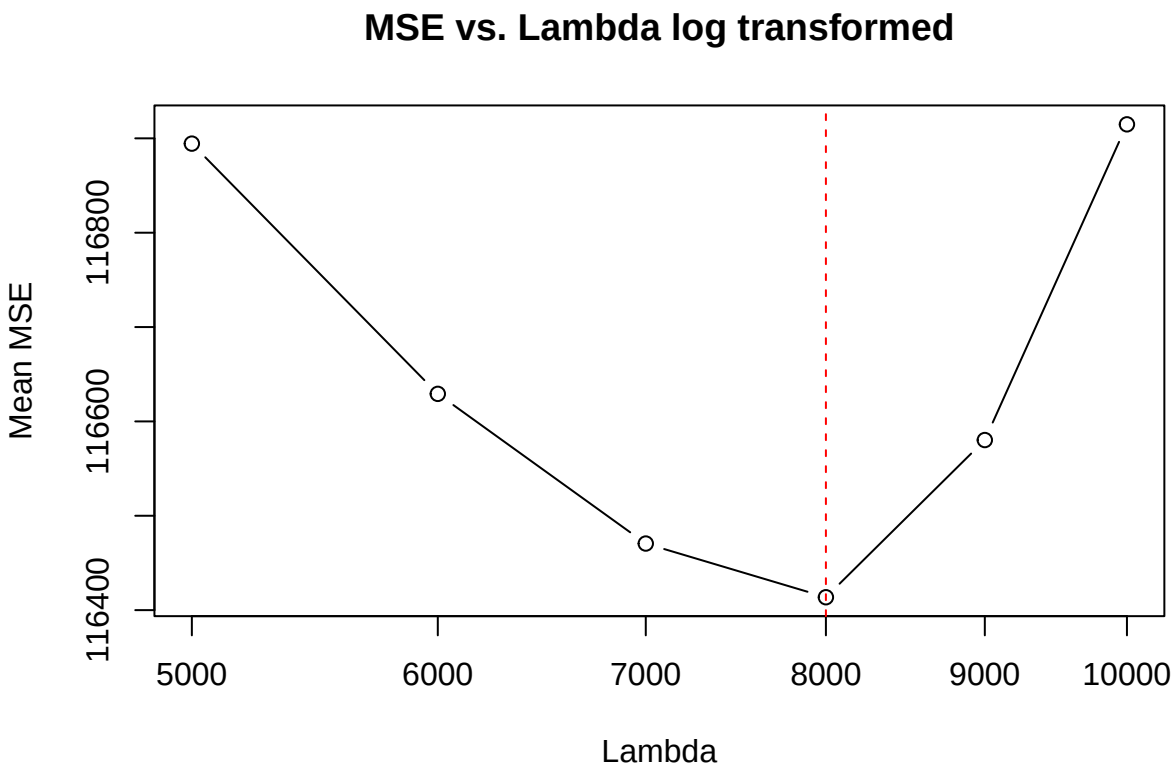
```r
lambdas <- c(1, 10, 100, 1000, 5000, 10000)
cv_lasso_shooting(X_train,y_train,lambdas, factor = 2)
```




```
## $lambda
## [1]     1    10   100  1000  5000 10000
##
## $mean.mse
## [1] 133280.4 133063.5 131274.1 121251.2 116894.4 116914.8
##
## $mean.rmse
## [1] 365.0758 364.7786 362.3176 348.2114 341.8983 341.9281
##
## $lambda.min.mse
## [1] 5000
##
## $lambda.min.rmse
## [1] 5000
```

We see that the optimal value would be here at 5000, but we want to narrow taht down further. Looking at the plot, the optimal value is probably somewhere between 5000 and 10000, so we generate a new sequence.

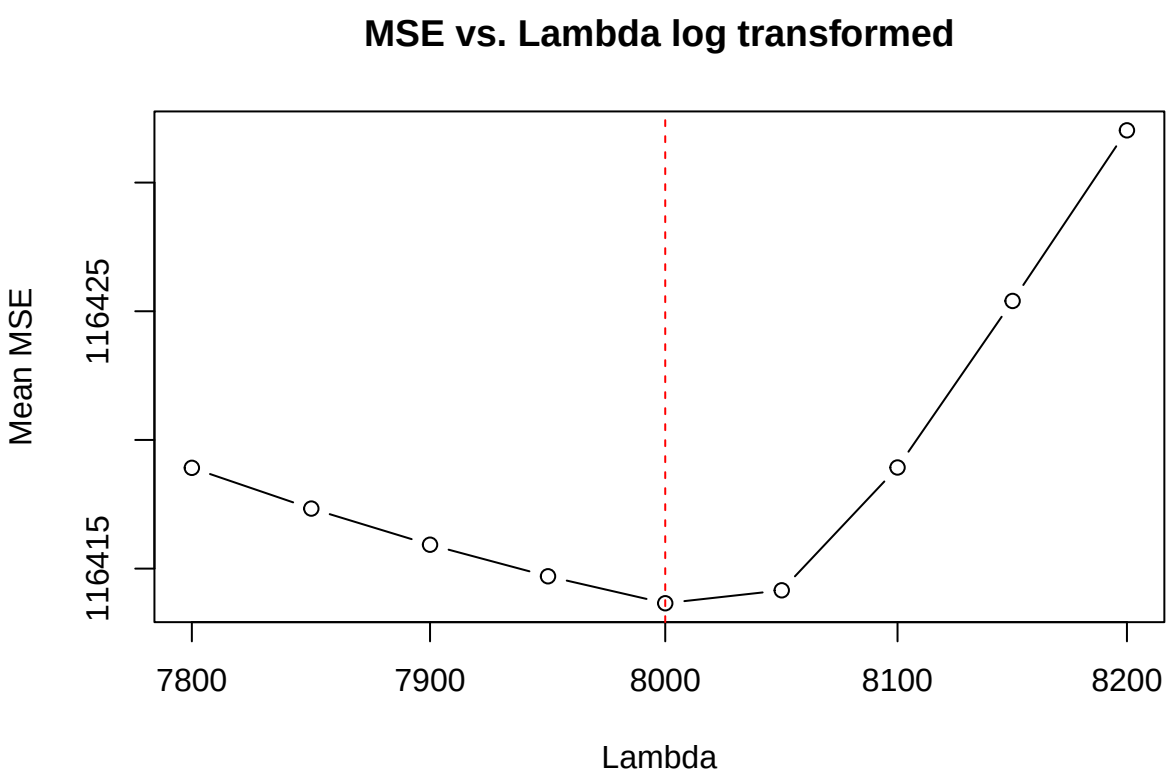
```
lambdas <- c(5000, 6000, 7000, 8000, 9000, 10000)
cv_lasso_shooting(X_train,y_train,lambdas, factor = 2)
```




```
## $lambda
## [1]  5000  6000  7000  8000  9000 10000
##
## $mean.mse
## [1] 116894.4 116629.2 116470.6 116413.7 116580.2 116914.8
##
## $mean.rmse
## [1] 341.8983 341.5102 341.2778 341.1945 341.4385 341.9281
##
## $lambda.min.mse
## [1] 8000
##
## $lambda.min.rmse
## [1] 8000
```

Here the optimal value is at 8000, so we again generate a new sequence to narrow that down further.

```
lambdas <- seq(7800, 8200, 50)
result_hitters <- cv_lasso_shooting(X_train,y_train,lambdas, factor = 2)
```




In this the optimal values is again at 8000, so we take this as our final optimal value.

```
min_lam <- result_hitters$lambda.min.mse
print(min_lam)
```

```
## [1] 8000
```

**part 2**

Now that we have our (near) optimal value for $\lambda$, we use this to train a lasso model with out original lasso implementation. We aöso use cv.glmnet to find the optimal $\lambda$ value ro use in the glmnet lasso function, which we also use to train a second mdoel. Then we use both models to predict the target variable Salary for the test data.

```
lasso_custom <- lasso_shooting(X_train, y_train, lambda = 8000)

X_test_interc <- cbind(Intercept = rep(1, nrow(X_test)), X_test)
y_pred_custom <- X_test_interc %*% lasso_custom

lasso_opt <- cv.glmnet(X_train, y_train, alpha = 1)
lambda_las_opt <- lasso_opt$lambda.min
lasso_glmnet <- glmnet(X_train, y_train, alpha = 1, lambda = lambda_las_opt)

y_pred_glm_lasso <- predict(lasso_glmnet, X_test)
```

Creating a function for computing the RMSE.

```
rmse <- function(actual, predicted){
  rmse <- sqrt(mean((actual - predicted)^2))
```

```
  rmse
}
```

Now that we have our predictions, we plot them each against the actual values in the test data and also compute the RMSE.

```
cat("RMSE of custom Lasso with optimal lambda: ", rmse(y_test, y_pred_custom))
```

```
## RMSE of custom Lasso with optimal lambda:  344.9991
```

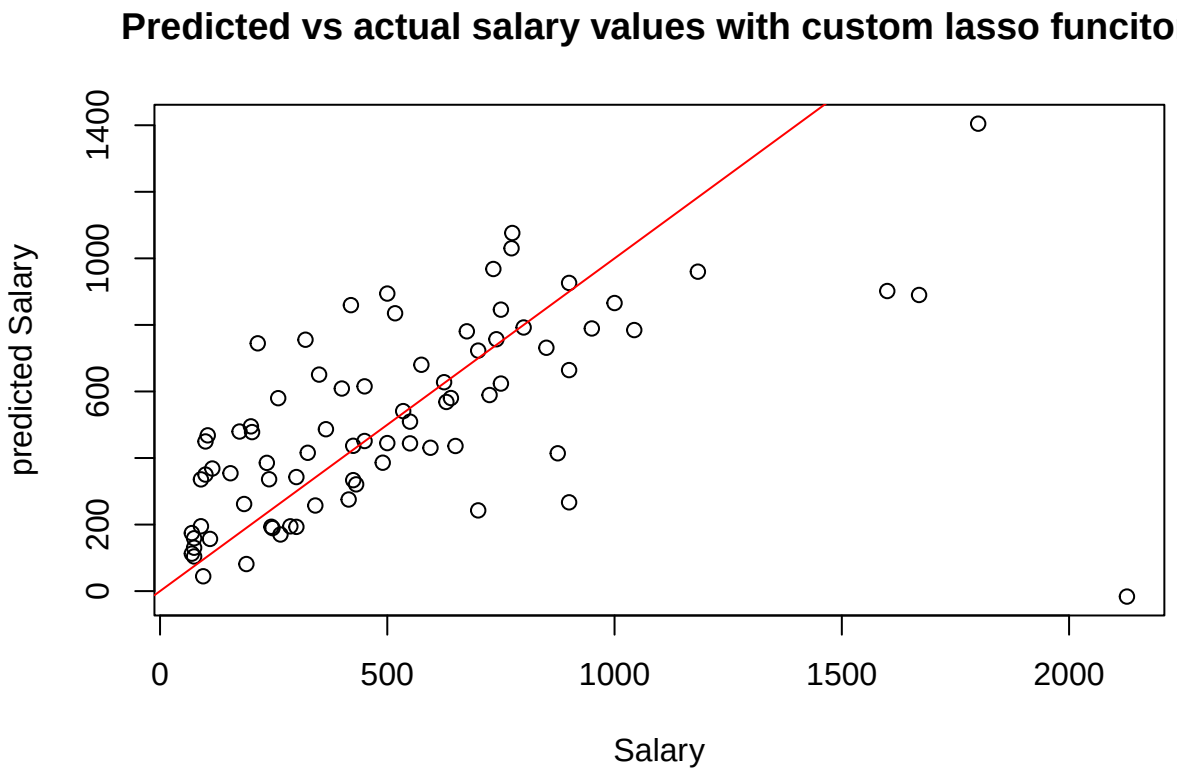
```
cat("RMSE of glmnet Lasso with optimal lambda: ", rmse(y_test, y_pred_glm_lasso))
```

```
## RMSE of glmnet Lasso with optimal lambda:  335.5087
```

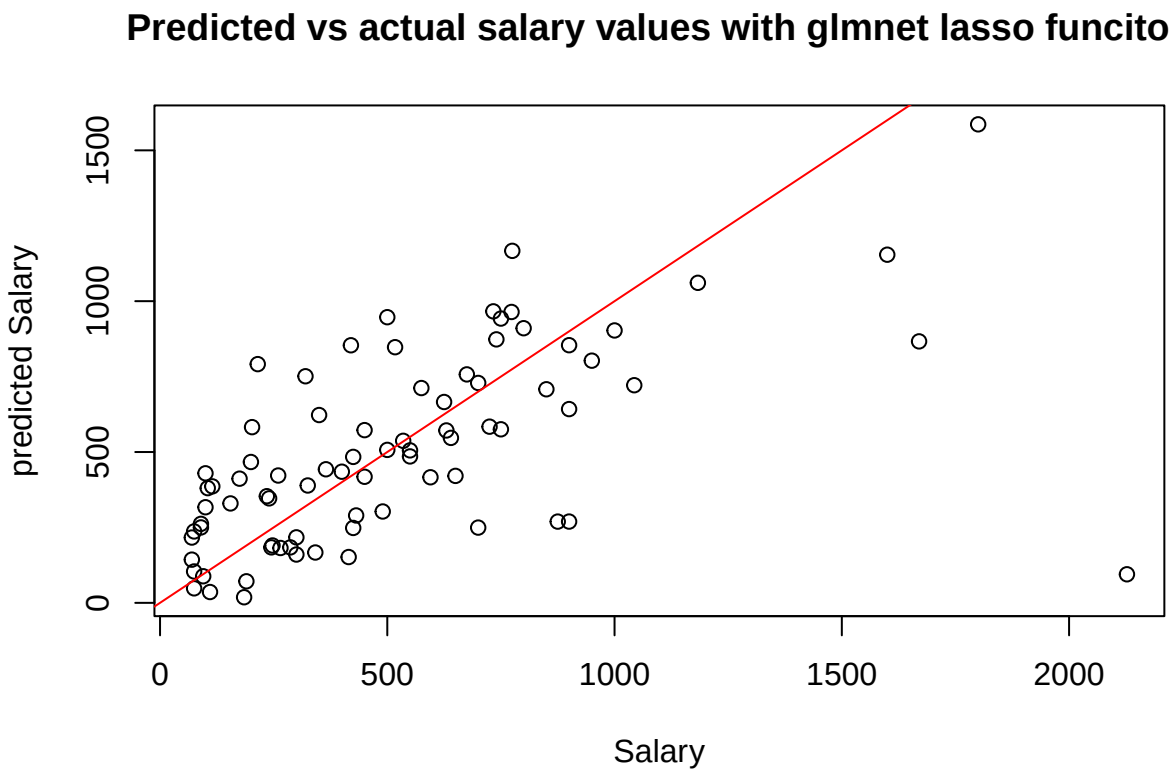
Creating a fucntion to plot results.

```
plot_results <- function(y_test, y_pred, main){
  plot(y_test, y_pred, xlab = "Salary", ylab = "predicted Salary",
     main = main)
  abline(0,1,col="red")
}
```

```
plot_results(y_test, y_pred_custom,
             "Predicted vs actual salary values with custom lasso funciton")
```




```
plot_results(y_test, y_pred_glm_lasso,
             "Predicted vs actual salary values with glmnet lasso funciton")
```




As we can see, the RMSE is slightly smaller for glmnets lasso function, but relatively the difference is very low. Also the plots of predictions vs actual test data look very similar. From this we conclude that our implementation of Lasso also works well.

**part 3**

We now also fit a Ridge model (for which we first use cv.glment again to get the optimal $\lambda$) and a least squares model.

```
#ridge model
ridge_model <- glmnet(X_train, y_train, alpha = 0)

ridge_cv <- cv.glmnet(X_train, y_train, alpha = 0)
lambda_opt_ridge <- ridge_cv$lambda.min

ridge <- glmnet(X_train, y_train, alpha = 0, lambda = lambda_opt_ridge)

#least squares model, where lambda is simply 0
least_squares <- glmnet(X_train, y_train, alpha = 0, lambda = 0)
```

**part 4**

Now we use the mdoels form part 3 to again predict the target variable for the test data.

```
y_pred_glm_ridge <- predict(ridge, X_test)
y_pred_glm_ls <- predict(least_squares, X_test)
```

In the end we compute the RMSE for every model's predictions and again plot the residual plot for each.

```
cat("RMSE of custom Lasso with optimal lambda: ", rmse(y_test, y_pred_custom))
```

```
## RMSE of custom Lasso with optimal lambda:  344.9991
```

```
cat("RMSE of glmnet Lasso with optimal lambda: ", rmse(y_test, y_pred_glm_lasso))
```

```
## RMSE of glmnet Lasso with optimal lambda:  335.5087
```

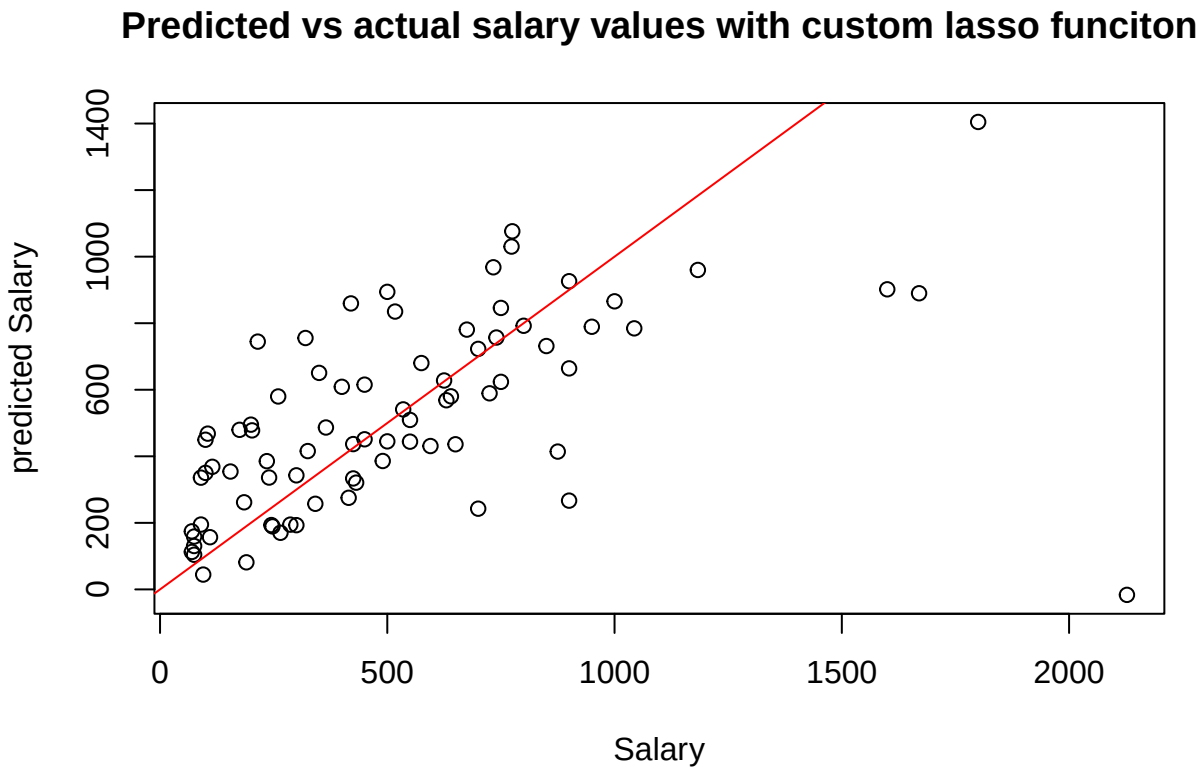
```
cat("RMSE of Ridge with optimal lambda: ", rmse(y_test, y_pred_glm_ridge))
```

```
## RMSE of Ridge with optimal lambda:  337.2844
```

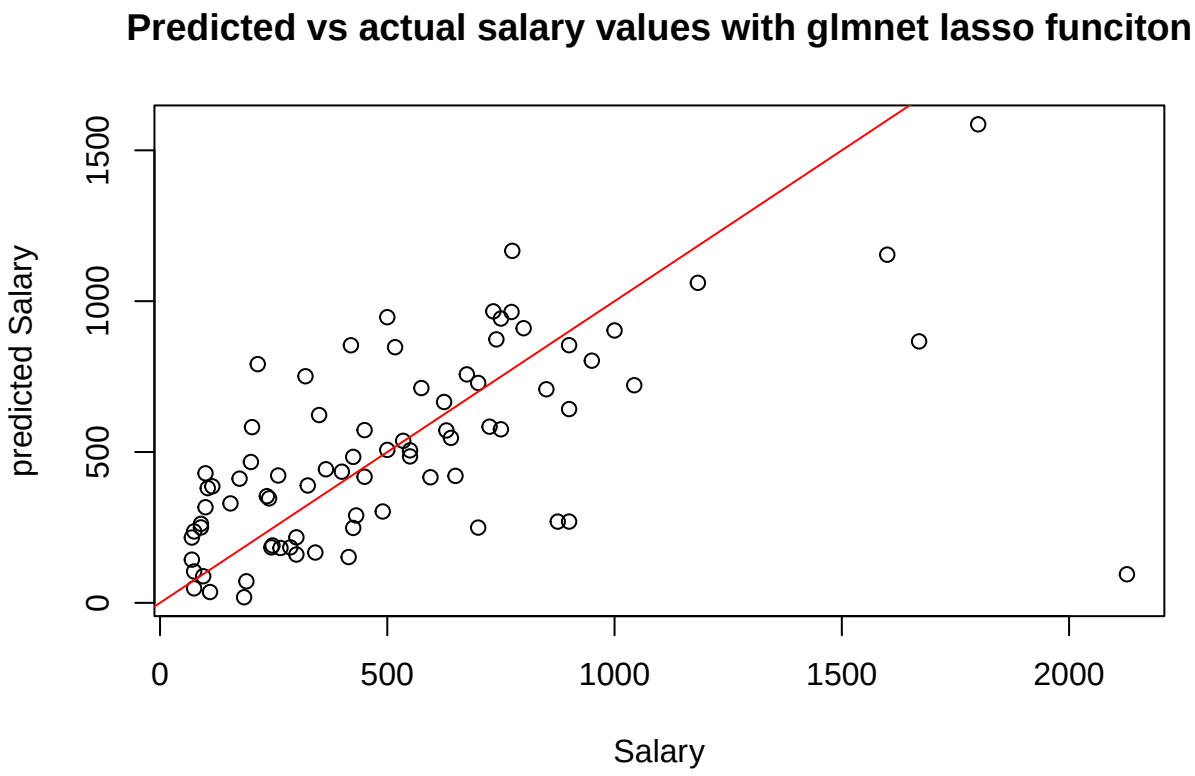
```
cat("RMSE of Least Squares: ", rmse(y_test, y_pred_glm_ls))
```

```
## RMSE of Least Squares:  334.7246
```

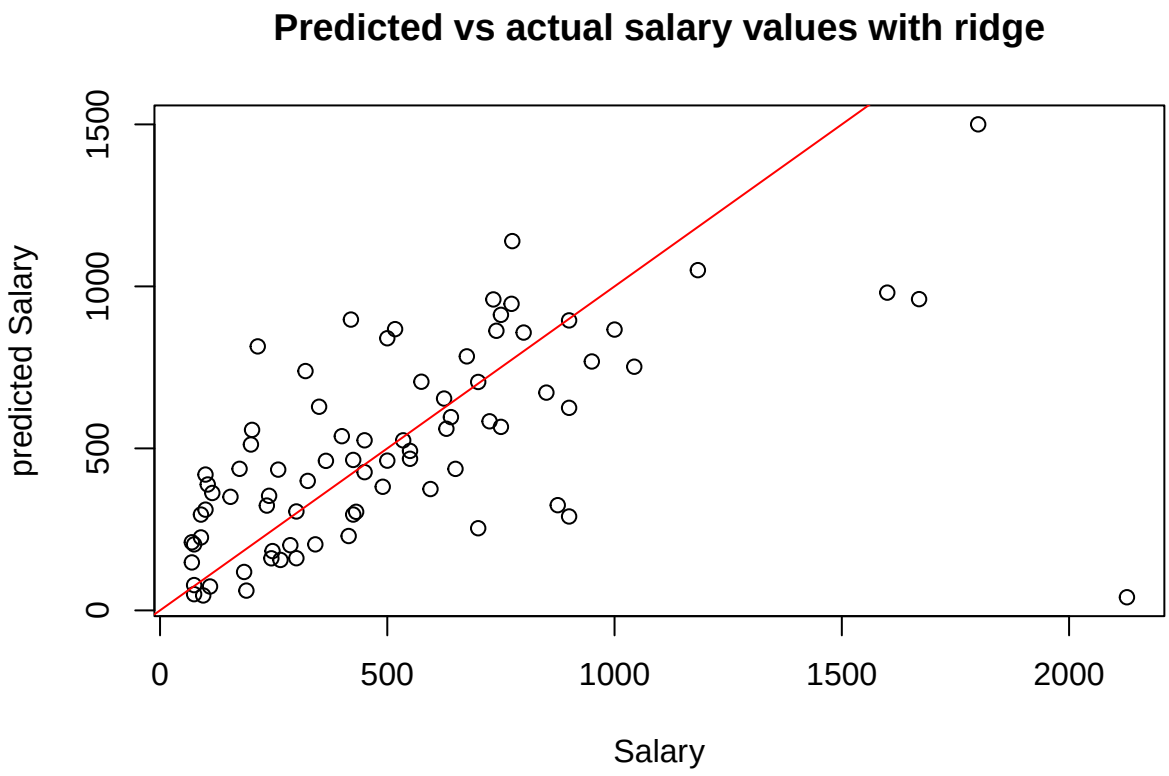
```
plot_results(y_test, y_pred_custom,
             "Predicted vs actual salary values with custom lasso funciton")
```



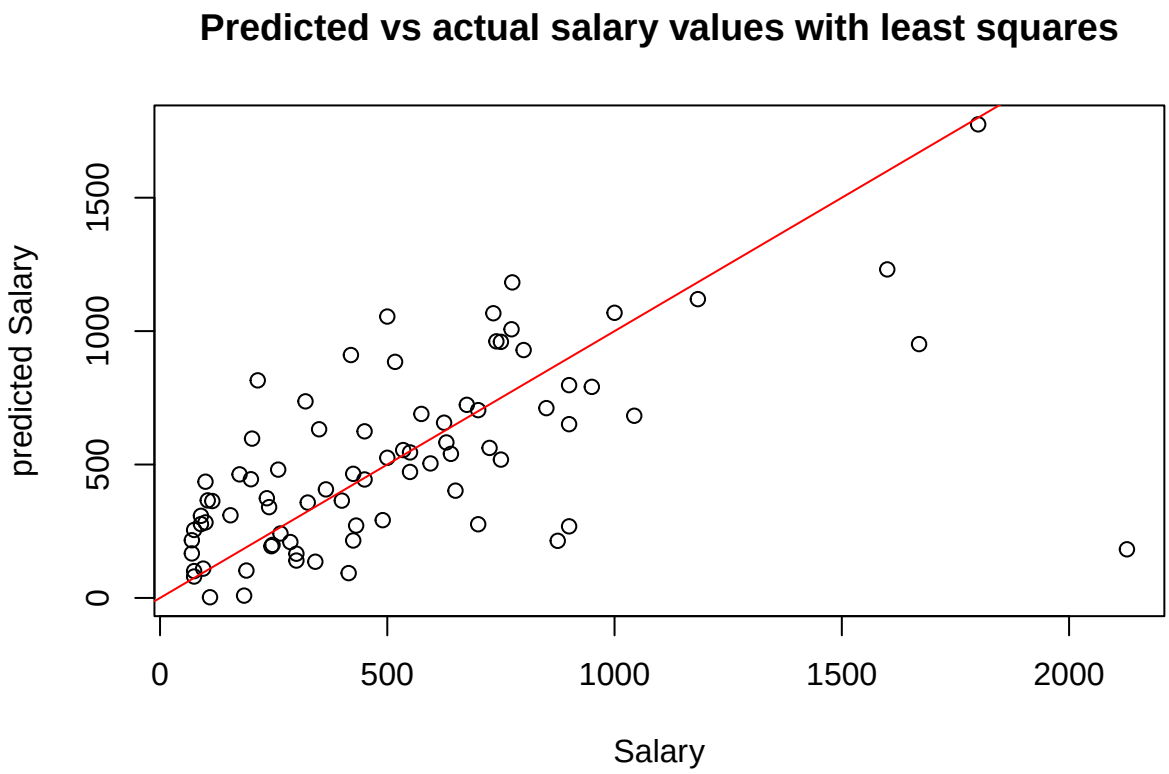

```
plot_results(y_test, y_pred_glm_lasso,
             "Predicted vs actual salary values with glmnet lasso funciton")
```




```
plot_results(y_test, y_pred_glm_ridge,
             "Predicted vs actual salary values with ridge")
```




```
plot_results(y_test, y_pred_glm_ls,
             "Predicted vs actual salary values with least squares")
```




Again the RMSE is very similar for all 4 models, this time time in the order of lowest to highest being : Least Squares, Ridge, Lasso (glmnet), Lasso (custom). The plots also look very similar, even though some slight differences can be observed, especially in the outliers. All have very similar perrformance. The residuals for the bulk of the data seem to be randomly distributed around the $x = y$ line, while also all four models seem to perform poorer on the outliers that are at the high end of salary. The RMSE is relatively high in regard to the scale of the target variable, so all 4 models perform overall mediocre. To get good prediciotns some other model class than linear models should probably be used, but if having to choose from these 4 it would be the simple least squares linear models not only because it has the lowest RMSE, but also because all 4 models do not really differ significantly in performance and therefore one should simply select the most simple model.

### Task 3

The point of regularized regression is improve regression models by generalizing them through penalties on the coefficients. This is done and can improve models with collinearity in the predictors, very high dimensional datasets that have more predictors than samples and also when a model is simply overfitting due to high variance in the data. The principle of the regularization is to add the aforementioned penalty as a term to the object funciton to be minimized which will then constrain the coefficients by shrinking them towards zero. This reduces their variance but can lead to a small bias. This tradeoff between a small bias but much less significance is the key goal to improve the performance of the model. The shrinkage mean shrinking the coefficients towards zero, which reduces their magnitude and can stabilize the model. \ In Lasso regression the added penalty term is L1, which is $\lambda \sum_{j=1}^{m} |\beta_j|$ (therefore the name lasso, least absolute shrinkage). This penalty term can shrink coefficients to zero and therefor perform feature selection. $\lambda$ is the regularization strength which controls how strong the influence of the penalty is, with a value of zero being equal to normal least squares regression. Lassso therefore also produces simpler models by with less predictors. This works in general well in higher dimensional data where many predictors are just random noise and not useful for the model. On the other hand it is not performing well when many predictors are highly correlated, since it will then shrink some predictors arbitrarily to zero from some correlation group. If the dimension is also too

high, it can only select at most as many predictors as there are samples, which can be bad in some cases.\ In Ridge regression the added penalty term is L2, which is $\lambda \sum_{j=1}^{m} \beta_j^2$. $\lambda$ is again the regularization strength. In Ridge coefficients get shrunk towards zero, but never to actual zero and thus no feature selection is done. This distributes the weight more between highly correlated predictors. It is useful in high dimesnional cases with few samples where Least Squares does not provide unique solutions and Lasso can give at most the sample size number of estimators. On the other hand Ridge can never delete irrelevant predictors that are just random noise, which is in some cases detrimental when only a subset of all variables is significant.