STATS 790 Assignment 2

Yiran Zhang
400119421

12 February, 2023

Question 1

(a) In linear regression, the assumed model is $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \epsilon$, it can be written in matrix form as $\overrightarrow{Y} = X\overrightarrow{\beta} + \overrightarrow{\epsilon}$, where $\overrightarrow{\beta}$ is the vector of coefficients (dimension is p $\times$ 1, or (p+1) $\times$ 1 if consider β_0), Y is the n $\times$ 1 vector of responses, X is an n $\times$ p matrix of predictors (or n $\times$ (p+1) if add one column of 1's for β_0), it is also assumed to be full rank. The estimator of $\overrightarrow{Y}$ is $\hat{\overrightarrow{Y}} = X\overrightarrow{\beta}$.

Naive Linear Algebra Use the least square estimation:

$$\begin{aligned}
RSS(\overrightarrow{\beta}) &= \sum_{i=1}^{n} \epsilon_i^2 \\
&= \sum_{i=1}^{n} (y_i - \hat{y_i})^2 \\
&= (\overrightarrow{Y} - \hat{\overrightarrow{Y}})^T(\overrightarrow{Y} - \hat{\overrightarrow{Y}}) \\
&= (\overrightarrow{Y} - X\overrightarrow{\beta})^T(\overrightarrow{Y} - X\overrightarrow{\beta})
\end{aligned}$$

We want to minimize the residual sum of square value, therefore we take the first derivative of it and set it to 0.

$$\begin{aligned}
\frac{\partial RSS(\overrightarrow{\beta})}{\partial \overrightarrow{\beta}} &= \frac{\partial(\overrightarrow{Y} - X\overrightarrow{\beta})^T(\overrightarrow{Y} - X\overrightarrow{\beta})}{\partial \overrightarrow{\beta}} \\
&= -2X^T\overrightarrow{Y} + 2X^TX\overrightarrow{\beta} \\
&= 0
\end{aligned}$$

Therefore, $X^T\overrightarrow{Y} = X^TX\overrightarrow{\beta}$, solve for $\overrightarrow{\beta}$ and get that the coefficients are $\overrightarrow{\beta} = (X^TX)^{-1}X^T\overrightarrow{Y}$. We can also compute the second partial derivative to check if it is minimum: $\frac{\partial^2 RSS(\overrightarrow{\beta})}{\partial \overrightarrow{\beta}^2} = 2X^TX > 0$. Hence the least square is minimized when $\overrightarrow{\beta} = (X^TX)^{-1}X^T\overrightarrow{Y}$.

QR Decomposition Based on the definition of QR Decomposition, since X is defined as full rank, it can be written as the form X=QR, Q is an n $\times$ (p+1) orthogonal matrix (i.e., $Q^T = Q^{-1}$), and R is an (p+1) $\times$ (p+1) invertible upper triangular matrix (Anton & Rorres, 2013). Then we can substitute X in the above equation with Q and R.

$$\begin{aligned} X^T X \overrightarrow{\beta} &= X^T \overrightarrow{Y} \\ (QR)^T QR \overrightarrow{\beta} &= (QR)^T \overrightarrow{Y} \\ R^T Q^T QR \overrightarrow{\beta} &= R^T Q^T \overrightarrow{Y} \\ R^T R \overrightarrow{\beta} &= R^T Q^T \overrightarrow{Y} \\ R \overrightarrow{\beta} &= Q^T \overrightarrow{Y} \end{aligned}$$

To obtain $\overrightarrow{\beta}$ estimate, use backward substitution since R is upper triangular, and $Q^T \overrightarrow{Y}$ is a (p+1) $\times$ 1 vector.

SVD X is full rank, therefore we can apply singular value decomposition to X by writing it in the form $X = U\Sigma V^T$, where U is n $\times$ n orthogonal matrix ($U^T U = I$), Σ is n $\times$ (p+1) matrix, V is (p+1) $\times$ (p+1) orthogonal matrix ($V^T V = I$) (Anton & Rorres, 2013).

Then, solve for $X\overrightarrow{\beta} = \overrightarrow{Y}$

$$\begin{aligned} U\Sigma V^T \overrightarrow{\beta} &= \overrightarrow{Y} \\ U^T U\Sigma V^T \overrightarrow{\beta} &= U^T \overrightarrow{Y} \\ \Sigma V^T \overrightarrow{\beta} &= U^T \overrightarrow{Y} \\ V^T \overrightarrow{\beta} &= \Sigma^{-1} U^T \overrightarrow{Y} \\ V V^T \overrightarrow{\beta} &= V\Sigma^{-1} U^T \overrightarrow{Y} \\ \overrightarrow{\beta} &= V\Sigma^{-1} U^T \overrightarrow{Y} \end{aligned}$$

The coefficients are $\overrightarrow{\beta} = V\Sigma^{-1}U^T\overrightarrow{Y}$, note that Σ^{-1} is a pseudo inverse of Σ.

Cholesky Decomposition Cholesky decomposition states that a positive definite matrix can be written as the product of a lower triangular matrix and its transpose. X is full rank, hence $X^T X$ is positive definite, then it can be written as $X^T X = LL^T$ where L is lower triangular.

$$
\begin{aligned}
X\overrightarrow{\beta} &= \overrightarrow{Y} \\
X^TX\overrightarrow{\beta} &= X^T\overrightarrow{Y} \\
LL^T\overrightarrow{\beta} &= X^T\overrightarrow{Y} \\
L(L^T\overrightarrow{\beta}) &= X^T\overrightarrow{Y}
\end{aligned}
$$

To solve for $\overrightarrow{\beta}$, we can solve for $L^T\overrightarrow{\beta}$ first from the equation $L(L^T\overrightarrow{\beta}) = X^T\overrightarrow{Y}$ using forward substitution, then solve for $\overrightarrow{\beta}$ from $L^T\overrightarrow{\beta}$ using backward substitution.

(b) Assume that the predictors X and responses Y are in separate matrices.

Naive Linear Algebra:

```r
beta_naive <- function(X,Y){
  # X is the matrix of predictors.
  # Y is the vector of responses
  X_new <- cbind(1, X) # add a column of 1 for beta_0
  solve(t(X_new) %*% X_new) %*% t(X_new) %*% Y # beta estimator
}
```

QR Decomposition:

```r
beta_qr <- function(X,Y){
  # X is the matrix of predictors.
  # Y is the vector of responses
  X_new <- cbind(1, X) # add a column of 1 for beta_0
  qr_decomp <- qr(X_new) # Obtain the QR decomposition of X
  Q <- qr.Q(qr_decomp) # Q matrix
  R <- qr.R(qr_decomp) # R matrix
  backsolve(R, t(Q) %*% Y)
}
```

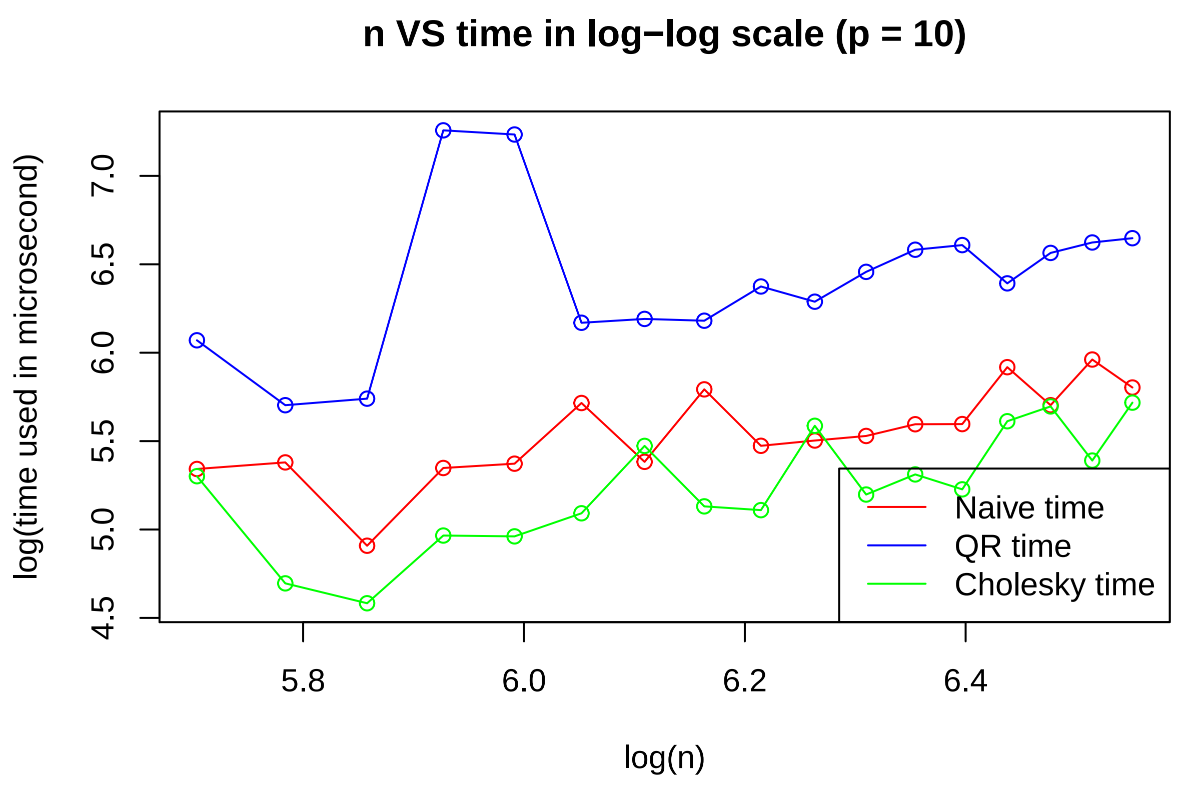
Cholesky Decomposition:

```r
beta_chol <- function(X,Y){
  # X is the matrix of predictors.
  # Y is the vector of responses
  X_new <- cbind(1, X) # add a column of 1 for beta_0
  chol_target <- t(X_new)%*%X_new # prepare X^T*X for Cholesky Decomposition
  L <- t(chol(chol_target)) # lower triangular matrix L
  # solve for L^T*beta, L(L^T*beta) = X^T*Y
  L_trans_beta <- forwardsolve(L, t(X_new) %*% Y, upper.tri = FALSE)
  backsolve(t(L), L_trans_beta, upper.tri = TRUE) # solve for beta
}
```

Benchmark:

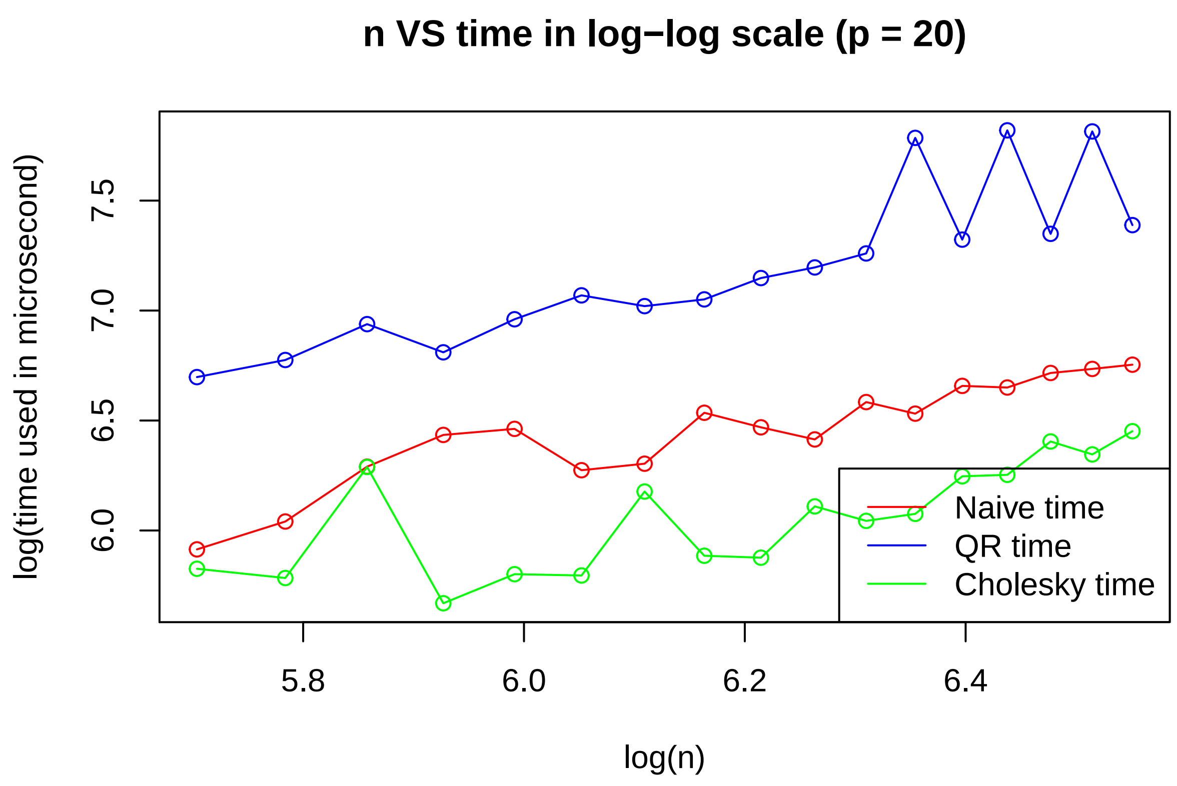
1. Use p = 10 and a range of n values.

```r
# Import benchmark library
library(microbenchmark)

p <- 10
n_vec <- round(100*seq(3, 7, by = 0.25)) # a list of n values

# Store the time for each n value
output_table <- matrix(NA, ncol = 4, nrow = length(n_vec))
colnames(output_table) <- c("n", "naive_time", "qr_time", "cholesky_time")
index <- 1

# Run benchmark for different n values
for (n in n_vec){
  X <- matrix(rnorm(n*p), nrow=n, ncol=p) # Simulate X matrix
  Y <- matrix(rnorm(n), ncol=1) # Simulate Y vector
  # Benchmark the three algorithms
  benchmark_result <- microbenchmark(beta_naive(X, Y),
                                     beta_qr(X, Y),
                                     beta_chol(X, Y))
  result <- summary(benchmark_result)
  # Store the mean computational time in the matrix
  output_table[index, ] = c(n, result$mean[1],
                             result$mean[2],
                             result$mean[3])
  index <- index + 1
}

# To plot in log-log scale, take the log of entire table
target <- log(output_table)

# Find limit of y axis for better visualization
ylim_min <- min(target[,2], target[,3], target[,4])
ylim_max <- max(target[,2], target[,3], target[,4])

# Create line graph
plot(target[, 1], target[, 2],
     type = "o", ylim=c(ylim_min, ylim_max), col='red',
     xlab = 'log(n)', ylab = 'log(time used in microsecond)',
     main = 'n VS time in log-log scale (p = 10)')
lines(target[, 1], target[, 3], type = "o", col = 'blue')
lines(target[, 1], target[, 4], type = "o", col = 'green')
legend(x = 'bottomright',
       legend=c("Naive time", "QR time", "Cholesky time"),
       col=c("red", "blue", "green"), lty = 1)
```



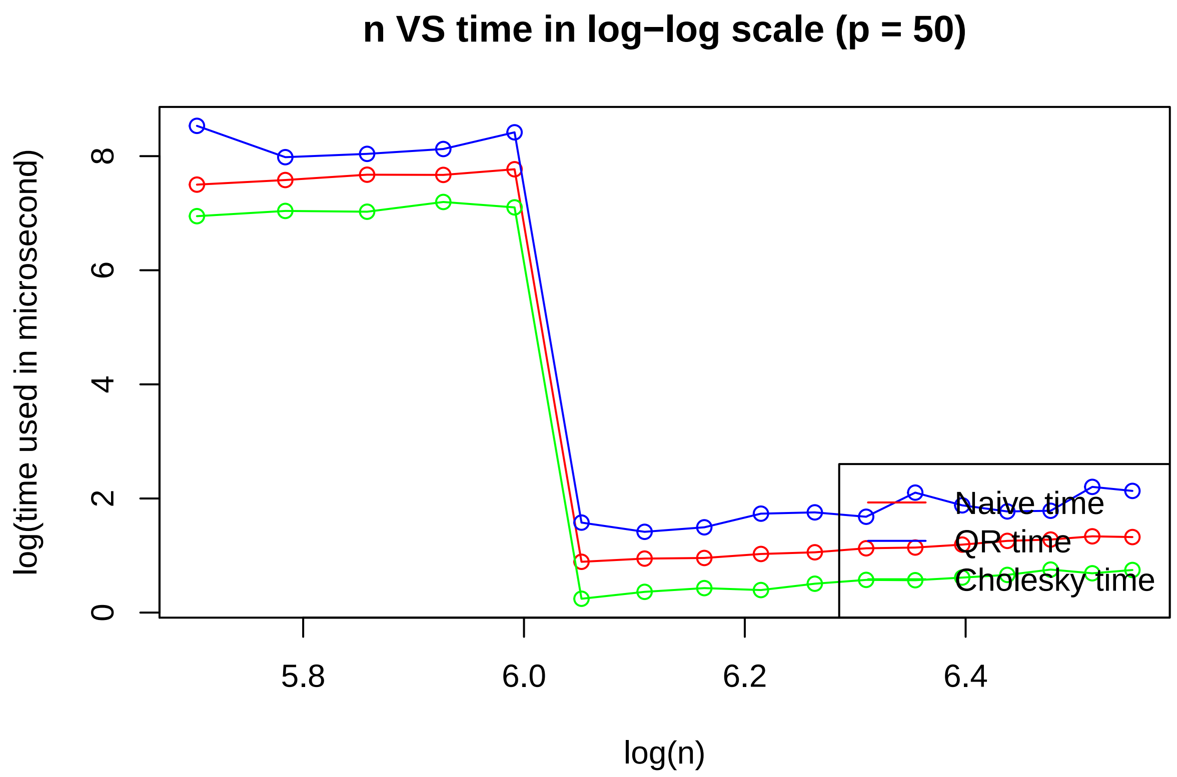
2. Use p = 20 and a range of n values.

```
p <- 20
n_vec <- round(100*seq(3, 7, by = 0.25)) # a list of n values

# Store the time for each n value
output_table <- matrix(NA, ncol = 4, nrow = length(n_vec))
colnames(output_table) <- c("n", "naive_time", "qr_time", "cholesky_time")
index <- 1

# Run benchmark for different n values
for (n in n_vec){
  X <- matrix(rnorm(n*p), nrow=n, ncol=p) # Simulate X matrix
  Y <- matrix(rnorm(n), ncol=1) # Simulate Y vector
  # Benchmark the three algorithms
  benchmark_result <- microbenchmark(beta_naive(X, Y),
                                     beta_qr(X, Y),
                                    beta_chol(X, Y))
  result <- summary(benchmark_result)
  # Store the mean computational time in the matrix
  output_table[index, ] = c(n, result$mean[1],
                           result$mean[2],
                           result$mean[3])
  index <- index + 1
}

# To plot in log-log scale, take the log of entire table
target <- log(output_table)

# Find limit of y axis for better visualization
ylim_min <- min(target[,2], target[,3], target[,4])
ylim_max <- max(target[,2], target[,3], target[,4])

# Create line graph
plot(target[, 1], target[, 2],
     type = "o", ylim=c(ylim_min, ylim_max), col='red',
     xlab = 'log(n)', ylab = 'log(time used in microsecond)',
     main = 'n VS time in log-log scale (p = 20)')
lines(target[, 1], target[, 3], type = "o", col = 'blue')
lines(target[, 1], target[, 4], type = "o", col = 'green')
legend(x = 'bottomright',
       legend=c("Naive time", "QR time", "Cholesky time"),
       col=c("red", "blue", "green"), lty = 1)
```



3. Use p = 50 and a range of n values.

```
p <- 50
n_vec <- round(100*seq(3, 7, by = 0.25)) # a list of n values

# Store the time for each n value
output_table <- matrix(NA, ncol = 4, nrow = length(n_vec))
colnames(output_table) <- c("n", "naive_time", "qr_time", "cholesky_time")
index <- 1

# Run benchmark for different n values
for (n in n_vec){
  X <- matrix(rnorm(n*p), nrow=n, ncol=p) # Simulate X matrix
  Y <- matrix(rnorm(n), ncol=1) # Simulate Y vector
  # Benchmark the three algorithms
  benchmark_result <- microbenchmark(beta_naive(X, Y),
                                     beta_qr(X, Y),
                                    beta_chol(X, Y))
  result <- summary(benchmark_result)
  # Store the mean computational time in the matrix
  output_table[index, ] = c(n, result$mean[1],
                             result$mean[2],
                             result$mean[3])
  index <- index + 1
}

# To plot in log-log scale, take the log of entire table
target <- log(output_table)

# Find limit of y axis for better visualization
ylim_min <- min(target[,2], target[,3], target[,4])
ylim_max <- max(target[,2], target[,3], target[,4])

# Create line graph
plot(target[, 1], target[, 2],
     type = "o", ylim=c(ylim_min, ylim_max), col='red',
     xlab = 'log(n)', ylab = 'log(time used in microsecond)',
     main = 'n VS time in log-log scale (p = 50)')
lines(target[, 1], target[, 3], type = "o", col = 'blue')
lines(target[, 1], target[, 4], type = "o", col = 'green')
legend(x = 'bottomright',
       legend=c("Naive time", "QR time", "Cholesky time"),
       col=c("red", "blue", "green"), lty = 1)
```



We benchmark the three algorithms using microbenchmark command for p = 10, 20, 50 respectively and using n from the range (300, 700). The log-log scale plots above shows that the algorithm using QR Decomposition has the highest computational time, linear regression with naive linear algebra has the second highest computational time, and linear regression with Cholesky Decomposition has the lowest computational time. The general trends show that the computational time increases as n increases, and increases as p increases.

(c) Time complexity for naive linear algebra is said to be $O(np^2 + p^3)$ (RUser4512, 2018) , for QR decomposition is $O(np^2)$ (Ratz & Gander, 2021), for SVD is $O(n^2p + p^3)$ (Kambhampati, 2001), and for Choleskey decomposition is $O(n^3)$ (Rana, 2022).

With the assumption that $n > p$, the ranking for time complexity for these four algorithms should be Cholesky Decomposition > SVD > Naive Linear Algbera > QR Decomposition. This result differs from the results in part (b), where Cholesky Decomposition in fact has the lowest cost and QR Decomposition has the highest.

Question 2

The formula for ridge regression is $\hat{\beta}^{ridge} = argmin[\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{i=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2]$. In matrix form $\hat{\beta}^{ridge} = (\overrightarrow{Y} - X\overrightarrow{\beta})^T(\overrightarrow{Y} - X\overrightarrow{\beta}) + \lambda \overrightarrow{\beta}^T \overrightarrow{\beta}$ (Hastie et al., 2009).

Minimize the RSS function by taking the first derivative and set to 0:

$$\begin{aligned}
\frac{\partial RSS(\overrightarrow{\beta}^{ridge})}{\partial \overrightarrow{\beta}^{ridge}} &= \frac{\partial (\overrightarrow{Y} - X\overrightarrow{\beta})^T(\overrightarrow{Y} - X\overrightarrow{\beta}) + \lambda \overrightarrow{\beta}^T \overrightarrow{\beta}}{\partial \overrightarrow{\beta}} \\
&= -2X^T\overrightarrow{Y} + 2X^TX\overrightarrow{\beta} + 2\lambda\overrightarrow{\beta} \\
&= 0
\end{aligned}$$

$\overrightarrow{\beta}^{ridge} = (X^TX + \lambda I_p)^{-1}X^T\overrightarrow{Y}$, we will implement this estimation into the R code below.

```r
ridge_augmented <- function(X, Y, lambda=0){
  X <- cbind(1, X) # add one column of 1
  p <- ncol(X)
  solve(t(X) %*% X + lambda * diag(p)) %*% t(X) %*% Y # compute the coefficients
}
```

Apply the above algorithm and the glmnet package algorithm to the prostate cancer dataset to compare the output and timing.

```r
# Import cancer dataset.
cancer <- read.table('https://hastie.su.domains/ElemStatLearn/datasets/prostate.data')

# Split train and test set.
train_cancer <- cancer[cancer$train == TRUE, ]
test_cancer <- cancer[cancer$train == FALSE, ]

# Split the design matrix and response vector for train and test set.
x_train <- as.matrix(scale(train_cancer[,1:8])) # scale x as required.
y_train <- train_cancer$lpsa

x_test <- as.matrix(scale(test_cancer[,1:8]))
y_test <- test_cancer$lpsa
```

```r
# Calculate the coefficients for the newly implemented ridge regression
my_ridge <- ridge_augmented(x_train, y_train, lambda = 0.087)

# Augment the design matrix for test set for prediction
x_test_aug <- cbind(1, x_test)

# Predict on the test set
my_ridge_pred <- x_test_aug %*% my_ridge

# Calculate the mean squared error
mse_my_ridge <- mean((my_ridge_pred - y_test)^2)
mse_my_ridge
```

```
## [1] 0.5488212
```

```r
# Naive implementation by glmnet package in R
library(glmnet)

# Build ridge regression model
ridge_naive <- glmnet(x_train, y_train, alpha=0)

# Predict on the test set
naive_pred <- predict(ridge_naive, x_test)

# Calculate mean squared error
mse_naive <- mean((naive_pred - y_test)^2)
mse_naive
```

```
## [1] 0.7605016
```

```r
# Compare the timing between two methods
a <-microbenchmark(my_ridge, ridge_naive, times = 10000)
```

Based on the above outputs, the new implemented ridge regression has less mean squared error than the build-in ridge function in glmnet. The microbenchmark function is ran multiple times on both functions and sometimes the new ridge regression function is faster, sometimes the build-in function is faster, but generally they have similar computational time.

Question 3

ESL 3.6 Given prior $\overrightarrow{\beta} \sim N(0, \tau I)$, then $\pi(\overrightarrow{\beta}) \propto exp(-\frac{1}{2\tau}\overrightarrow{\beta}^T\overrightarrow{\beta})$. Given the sampling model is also Gaussian, $\overrightarrow{Y}|\overrightarrow{\beta} \sim N(X\overrightarrow{\beta}, \sigma^2 I)$, then $\pi(\overrightarrow{Y}|\overrightarrow{\beta}) \propto exp(-\frac{1}{2\sigma^2}(\overrightarrow{Y} - X\overrightarrow{\beta})^T(\overrightarrow{Y} - X\overrightarrow{\beta}))$.

From Bayesian inference, we calculate the posterior distribution to be:

$$\begin{aligned}
\pi(\overrightarrow{\beta}|\overrightarrow{Y}) &\propto \pi(\overrightarrow{\beta})\pi(\overrightarrow{Y}|\overrightarrow{\beta}) \\
&= exp(-\frac{1}{2\tau}\overrightarrow{\beta}^T\overrightarrow{\beta} - \frac{1}{2\sigma^2}(\overrightarrow{Y} - X\overrightarrow{\beta})^T(\overrightarrow{Y} - X\overrightarrow{\beta})) \\
&= exp\{-\frac{1}{2}(\frac{\overrightarrow{\beta}^T\overrightarrow{\beta}}{\tau} + \frac{\overrightarrow{Y}^T\overrightarrow{Y} - 2\overrightarrow{Y}^TX\overrightarrow{\beta} + \overrightarrow{\beta}^TX^TX\overrightarrow{\beta}}{\sigma^2})\} \\
&\propto exp\{-\frac{1}{2}[(\frac{1}{\tau}I + \frac{X^TX}{\sigma^2})\overrightarrow{\beta}^T\overrightarrow{\beta} - \frac{2X^T\overrightarrow{Y}}{\sigma^2}\overrightarrow{\beta}]\} \\
&= exp\{-\frac{1}{2}(\frac{1}{\tau}I + \frac{X^TX}{\sigma^2})[\overrightarrow{\beta}^T\overrightarrow{\beta} - 2\frac{\frac{X^T\overrightarrow{Y}}{\sigma^2}}{\frac{1}{\tau}I + \frac{X^TX}{\sigma^2}}\overrightarrow{\beta}\}] \\
&\propto exp\{-\frac{1}{2}(\frac{1}{\tau}I + \frac{X^TX}{\sigma^2})(\overrightarrow{\beta} - E[\overrightarrow{\beta}|\overrightarrow{Y}])^2\}
\end{aligned}$$

The posterior distribution is also Gaussian, with mean equal to $\frac{\frac{X^T\overrightarrow{Y}}{\sigma^2}}{\frac{1}{\tau}I + \frac{X^TX}{\sigma^2}}$.

Simplify it and we get $(X^TX + \frac{\sigma^2}{\tau}I)^{-1}X^T\overrightarrow{Y}$, if taking the regularization parameter $\lambda = \frac{\sigma^2}{\tau}$, then the mean of the posterior is $(X^TX + \lambda I)^{-1}X^T\overrightarrow{Y}$, which is exactly the formula for ridge regression estimate. Moreover, since the posterior is Gaussian, the mode is the same as the mean, which is also the ridge regression estimate.

ESL 3.19 From ESL Chapter 3 equation 3.44, we learned that the formula for ridge regression estimate is $\overrightarrow{\beta}^{ridge} = (X^TX + \lambda I)^{-1}X^T\overrightarrow{Y}$. Apply SVD on the design matrix X, $X = U\Sigma V^T$, substitute it into $\overrightarrow{\beta}^{ridge}$ and get:

$$\begin{aligned}
\overrightarrow{\beta}^{ridge} &= ((U\Sigma V^T)^TU\Sigma V^T + \lambda I)^{-1}(U\Sigma V^T)^T\overrightarrow{Y} \\
&= (V\Sigma^TU^TU\Sigma V^T + \lambda I)^{-1}V\Sigma U^T\overrightarrow{Y} \\
&= (V\Sigma^2V^T + \lambda I)^{-1}V\Sigma U^T\overrightarrow{Y} \\
&= V(\Sigma^2 + \lambda I)^{-1}V^TV\Sigma U^T\overrightarrow{Y} \\
&= V(\Sigma^2 + \lambda I)^{-1}\Sigma U^T\overrightarrow{Y}
\end{aligned}$$

$$
\begin{aligned}
||\overrightarrow{\beta}^{ridge}||^2 &= \overrightarrow{\beta}^{T ridge}\overrightarrow{\beta}^{ridge} \\
&= \overrightarrow{Y}^T U\Sigma^T(\Sigma^2+\lambda I)^{-1}V^T V(\Sigma^2+\lambda I)^{-1}\Sigma U^T\overrightarrow{Y} \\
&= \overrightarrow{Y}^T U\Sigma^T(\Sigma^2+\lambda I)^{-2}\Sigma U^T\overrightarrow{Y} \\
&= (U^T\overrightarrow{Y})^T\Sigma^T(\Sigma^2+\lambda I)^{-2}\Sigma(U^T\overrightarrow{Y}) \\
&= \sum_{j=1}^{p}(\overrightarrow{u_j}\overrightarrow{Y})^2\frac{\sigma_j^2}{(\sigma_j^2+\lambda)^2}
\end{aligned}
$$

When $\lambda \to 0$, the term $\frac{\sigma_j^2}{(\sigma_j^2+\lambda)^2}$ increases as the denominator decreases, which leads to $||\overrightarrow{\beta}^{ridge}||^2$ increases. $||\overrightarrow{\beta}^{ridge}||$ is the square root of $||\overrightarrow{\beta}^{ridge}||^2$, therefore it also increases when $\lambda \to 0$. This intuitively makes sense, because when λ decreases, the penalty on the coefficients weakens, resulting in larger coefficients.

In the lasso case, $\overrightarrow{\beta}^{lasso} = argmin[\frac{1}{2}\sum_{i=1}^{n}(y_i-\beta_0-\sum_{j=1}^{p}x_{ij}\beta_j)^2+\lambda\sum_{j=1}^{p}|\beta_j|]$, the $\overrightarrow{\beta}^{lasso}$ here is defined to be the β values that minimizes the above equation. When λ decreases and goes to 0, the term $\lambda\sum_{j=1}^{p}|\beta_j|$ decreases, therefore $\sum_{j=1}^{p}|\beta_j|$ values need to increase to remain in the same minimized equation value, which results in increasing $\overrightarrow{\beta}^{lasso}$. Therefore, $||\overrightarrow{\beta}^{lasso}||$ also increases when $\lambda \to 0$.

ESL 3.28 The original lasso regression estimate is $\overrightarrow{\beta}^{lasso} = argmin_\beta\sum_{i=1}^{n}(y_i-\beta_0-\sum_{k=1}^{p}x_{ik}\beta_k)^2$ subject to $\sum_{k=1}^{p}|\beta_k| \le t$ (Hastie et al., 2009). When we augment the design matrix with one variable $X_j^* = X_j$, denote the new design matrix $X^* = (X, X_j^*)$, then the new lasso regression estimate is:

$\overrightarrow{\beta^*}^{lasso} = argmin_\beta\sum_{i=1}^{n}(y_i-\beta_0-\sum_{k=1}^{p}x_{ik}\beta_k-x_{ij}\beta_j^*)^2$, subject to $\sum_{k=1}^{p}|\beta_k|+|\beta_j^*| \le t$

The value of t is not changed, then $\sum_{k=1}^{p}|\beta_k|+|\beta_j^*| = \sum_{k\neq j}^{p}|\beta_k|+|\beta_j|+|\beta_j^*| \le t$, to remain with the same t value, $|\beta_j|+|\beta_j^*| = |\beta_j^{lasso}| = a$, and since both β_j and β_j^* are coefficients for an identical variable $X_j^* = X_j$, $\beta_j = \beta_j^* = \frac{a}{2}$.

To conclude, augment the matrix with one identical variable will result in the lasso coefficient of X_j and X_j^* to be both $\frac{a}{2}$ (half of the original value), using the same t value.

ESL 3.30

$$
\begin{aligned}
& argmin_\beta||\overrightarrow{Y} - X\overrightarrow{\beta}||_2^2 + \lambda(\alpha||\overrightarrow{\beta}||_2^2 + (1-\alpha)||\overrightarrow{\beta}||_1) \\
&= argmin_\beta||\overrightarrow{Y} - X\overrightarrow{\beta}||_2^2 + \lambda\alpha||\overrightarrow{\beta}||_2^2 + \lambda(1-\alpha)||\overrightarrow{\beta}||_1 \\
&= argmin_\beta|| \begin{pmatrix} \overrightarrow{Y} - X\overrightarrow{\beta} \\ \sqrt{\lambda\alpha}\overrightarrow{\beta} \end{pmatrix} ||_2^2 + \lambda(1-\alpha)||\overrightarrow{\beta}||_1 \\
&= argmin_\beta|| \begin{pmatrix} \overrightarrow{Y} \\ 0 \end{pmatrix} - \begin{pmatrix} X\overrightarrow{\beta} \\ -\sqrt{\lambda\alpha}\overrightarrow{\beta} \end{pmatrix} ||_2^2 + \lambda(1-\alpha)||\overrightarrow{\beta}||_1 \\
&= argmin_\beta|| \begin{pmatrix} \overrightarrow{Y} \\ 0 \end{pmatrix} - \begin{pmatrix} X\overrightarrow{\beta} \\ \sqrt{\lambda\alpha}\overrightarrow{\beta} \end{pmatrix} ||_2^2 + \lambda(1-\alpha)||\overrightarrow{\beta}||_1 \\
&= argmin_\beta|| \begin{pmatrix} \overrightarrow{Y} \\ 0 \end{pmatrix} - \begin{pmatrix} X \\ \sqrt{\lambda\alpha} \end{pmatrix} \overrightarrow{\beta}||_2^2 + \lambda(1-\alpha)||\overrightarrow{\beta}||_1
\end{aligned}
$$

So if make $\overrightarrow{Y_1} = \begin{pmatrix} \overrightarrow{Y} \\ 0 \end{pmatrix}$, and $X_1 = \begin{pmatrix} X \\ \sqrt{\lambda\alpha} \end{pmatrix}$, then the RSS formula will equal to the first part of the above formula: $||\overrightarrow{Y} - X\overrightarrow{\beta}||_2^2 + \lambda\alpha||\overrightarrow{\beta}||_2^2$. When we apply the lasso regression estimate formula to this $\overrightarrow{Y_1}$ and X_1, we get:

$\overrightarrow{\beta}^{lasso} = argmin_\beta||\overrightarrow{Y_1} - X_1\overrightarrow{\beta}||_2^2 + \lambda_1||\overrightarrow{\beta}||_1 = ||\overrightarrow{Y} - X\overrightarrow{\beta}||_2^2 + \lambda\alpha||\overrightarrow{\beta}||_2^2 + \lambda_1||\overrightarrow{\beta}||_1$, where $\lambda_1 = \lambda(1-\alpha)$. In other words, the elastic-net optimization problem can be turned into a lasso problem.

Reference

- Hastie, T., Tibshirani, R., Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition. Springer.
- Anton, H., & Rorres, C. (2013). Elementary Linear Algebra: Applications Version. Wiley.
- Kambhampati, S. (2001). SVD computation complexity (m^2 n + n^3). https://rakaposhi.eas.asu.edu/s01-cse494-mailarchive/msg00028.html
- Rana, S. (2022). Cholesky Decomposition : Matrix Decomposition. GeeksforGeeks. https://www.geeksforgeeks.org/cholesky-decomposition-matrix-decomposition/
- Ratz, A. V., & Gander, W. (2021). Can QR Decomposition Be Actually Faster? Schwarz-Rutishauser Algorithm. Towards Data Science. https://towardsdatascience.com/can-qr-decomposition-be-actually-faster-schwarz-rutishauser-algorithm-a32c0cde8b9b
- RUser4512. (2018). Computational complexity of machine learning algorithms – The Kernel Trip. The Kernel Trip. https://www.thekerneltrip.com/machine/learning/computational-complexity-learning-algorithms/