005921309_stats102b_hw4

Anish Deshpande

2024-05-22

```r
# reading in volleyball data
vball_data <- read_csv("volleyball_data.csv")
```

Question 1:

a.)

```r
# fit a least squares regression line to predict block from spike
# minimizing the MSE loss function...
x <- vball_data$spike
y <- vball_data$block
# create design matrix X
X <- cbind(1, x)

w_hat <- solve(t(X) %*% X) %*% t(X) %*% y
w_hat
```

```
##          [,1]
##   44.2241652
## x  0.8075392
```

From the above output, our fitted linear regression model becomes

$$\hat{y} = 0.80753x + 44.2242$$

Where $\hat{y}$ is the predicted block value, and x is the spike value.

b.)

```r
# g is a function: R -> R
# w0 is the initial point
# d is a step size in each direction
# K is the number of iterations
coord_desc <- function(g, w0, d, K) {
  # create initial list object:
  result <- list(index = c(), val = c())
  # create initial empty coordinate matrix: (ncol = n + 1)
  mat <- matrix(nrow = K, ncol = length(w0) + 1)

  # set initial point:
  curr_point <- w0

  # iterate for K iterations:
  for (i in 1:K) {
    # create n x n identity matrix for standard basis vectors:
    basis_mat <- diag(length(w0))
    # create all candidate directions:
    basis_vectors <- as.list(as.data.frame(t(basis_mat)))
    negative_basis_vectors <- lapply(basis_vectors, function(v) -v)
    candidate_directions <- c(basis_vectors, negative_basis_vectors)

    # find 2n candidate points
    new_points <- lapply(candidate_directions, function(direction) curr_point + d * direction)
    new_points <- do.call(rbind, lapply(new_points, as.vector))
    evaluated_pts <- apply(new_points, 1, g)  # Evaluate all points

    # find the index of the minimum evaluated point
    min_index <- which.min(evaluated_pts)
    min_point <- as.numeric(new_points[min_index, ])

    # add results to matrix:
    mat[i, 1:length(w0)] <- min_point
    mat[i, length(w0) + 1] <- evaluated_pts[min_index]

    # update current point
    curr_point <- min_point
  }

  inputs <- mat[, 1:length(w0)]
  outputs <- mat[, length(w0) + 1]
  min_val <- min(outputs)
  # check if there are duplicate minimum values
  min_indices <- which(outputs == min_val)
  num_min_indices <- length(min_indices)
  if (num_min_indices > 1) {
    # if there are multiple min values, choose the first one
    min_index <- inputs[min_indices[1], ]
  } else {
    # If there is only one unique minimum value, directly extract its index
    min_index <- inputs[min_indices, ]
  }

  result$index <- min_index
  result$val <- min_val

  # return result
  result
}
```

c.)

```r
# Minimizing MAE Loss
x <- vball_data$spike
y <- vball_data$block
g <- function(w) {
  (1/length(x)) * abs(sum(y - (w[2]*x + w[1])))
}

mae_coeff <- coord_desc(g, c(44, 0.9), 0.01, 100)
mae_coeff
```

```
## $index
## [1] 43.51  0.81
##
## $val
## [1] 0.003496503
```

d.)

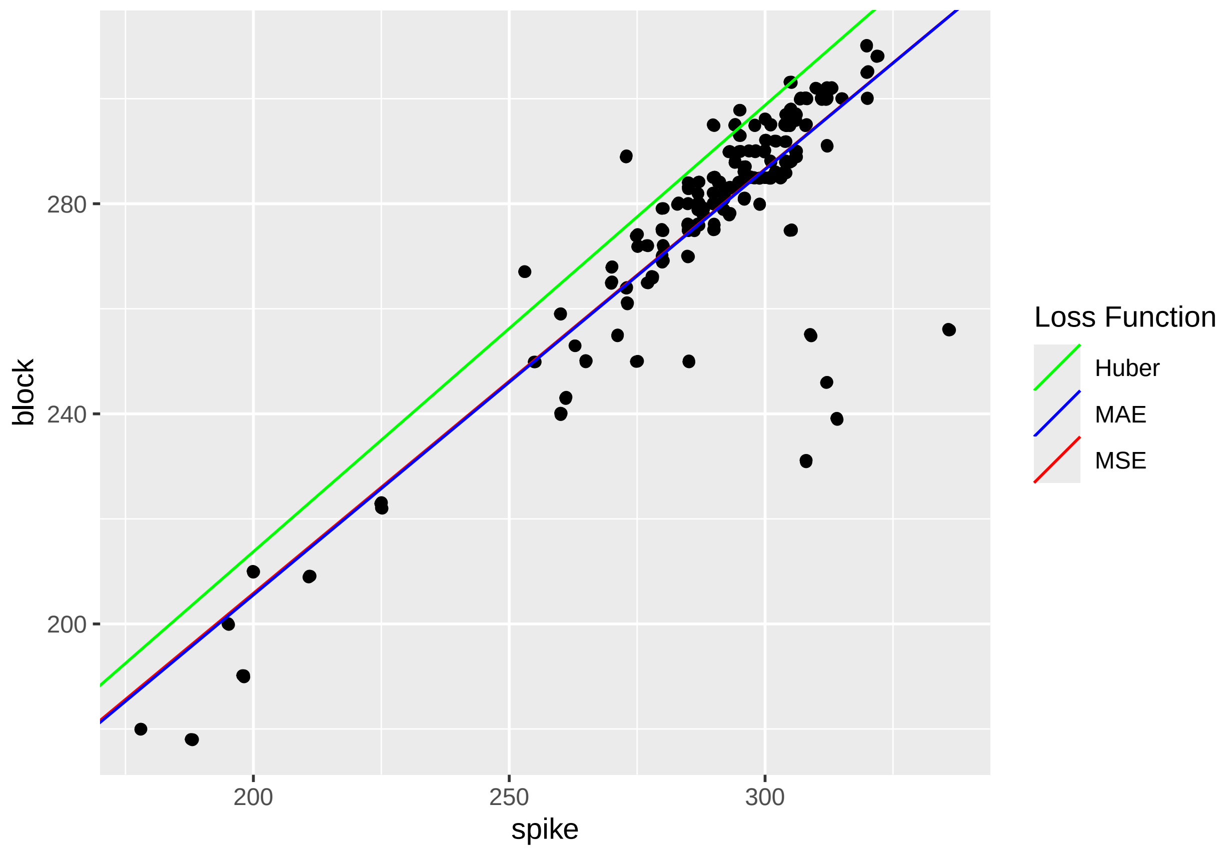
```r
# Minimizing Huber Loss:
# set delta to be 18
x <- vball_data$spike
y <- vball_data$block
delta <- 18
g <- function(w) {
  L_vector <- c()
  for (i in 1:length(x)) {
    diff <- abs(y[i] - (w[2]*x[i] + w[1]))
    if (diff <= delta) {
      L <- (1/2) * (diff^2)
    } else {
      L <- (delta * diff) - ((1/2) * delta^2)
    }
    L_vector <- append(L_vector, L)
  }
  (1/length(x)) * sum(L)
}

hub_coeff <- coord_desc(g, c(44, 0.9), 0.01, 100)
hub_coeff
```

```
## $index
## [1] 43.75  0.85
##
## $val
## [1] 3.765937e-30
```

e.)

```
# red = mse, blue = mae, green = huber
# Plot with legend
ggplot(data = vball_data, aes(x = spike, y = block)) +
  geom_jitter(width = 0.2, height = 0.2) +
  geom_abline(aes(intercept = 44.224, slope = 0.808, color = "MSE")) +
  geom_abline(aes(intercept = mae_coeff$index[[1]], slope = mae_coeff$index[[2]], color = "MAE")) +
  geom_abline(aes(intercept = hub_coeff$index[[1]], slope = hub_coeff$index[[2]], color = "Huber")) +
  scale_color_manual(values = c("MSE" = "red", "MAE" = "blue", "Huber" = "green")) +
  labs(color = "Loss Function")
```



f.)

```
# vary delta and see how results change
delta_vec <- c(5, 10, 18, 25)

# Define the Huber loss function
huber_loss <- function(x, y, w, delta) {
  diff <- abs(y - (w[2] * x + w[1]))
  L <- ifelse(diff <= delta, (1/2) * (diff^2), delta * diff - (1/2) * delta^2)
  return(mean(L))
}
```

```r
# Store the coefficients and values for each delta value
huber_coefficients <- list()

# Iterate over each delta value
for (delta in delta_vec) {
  # Define the Huber loss function for this delta value
  g <- function(w) huber_loss(x, y, w, delta)
  # Run coordinate descent to minimize the Huber loss
  hub_coeff <- coord_desc(g, c(44, 0.9), 0.01, 100)
  # Store the coefficients and values for this delta value
  huber_coefficients[[as.character(delta)]] <- hub_coeff
}

# Output the results
huber_coefficients
```

```
## $`5`
## $`5`$index
## [1] 43.08  0.82
##
## $`5`$val
## [1] 22.91824
##
##
## $`10`
## $`10`$index
## [1] 44.91  0.81
##
## $`10`$val
## [1] 36.26168
##
##
## $`18`
## $`18`$index
## [1] 44.84  0.81
##
## $`18`$val
## [1] 50.23122
##
##
## $`25`
## $`25`$index
## [1] 44.56  0.81
##
## $`25`$val
## [1] 58.75025
```

- From the above results, we see that both the intercept and the first order coefficient of the huber loss simple linear regression decreases as the delta parameter increases. Huber loss is usually used to reduce the effect of outliers. The delta parameter tells us when to switch between MAE and MSE loss.

Question 2:

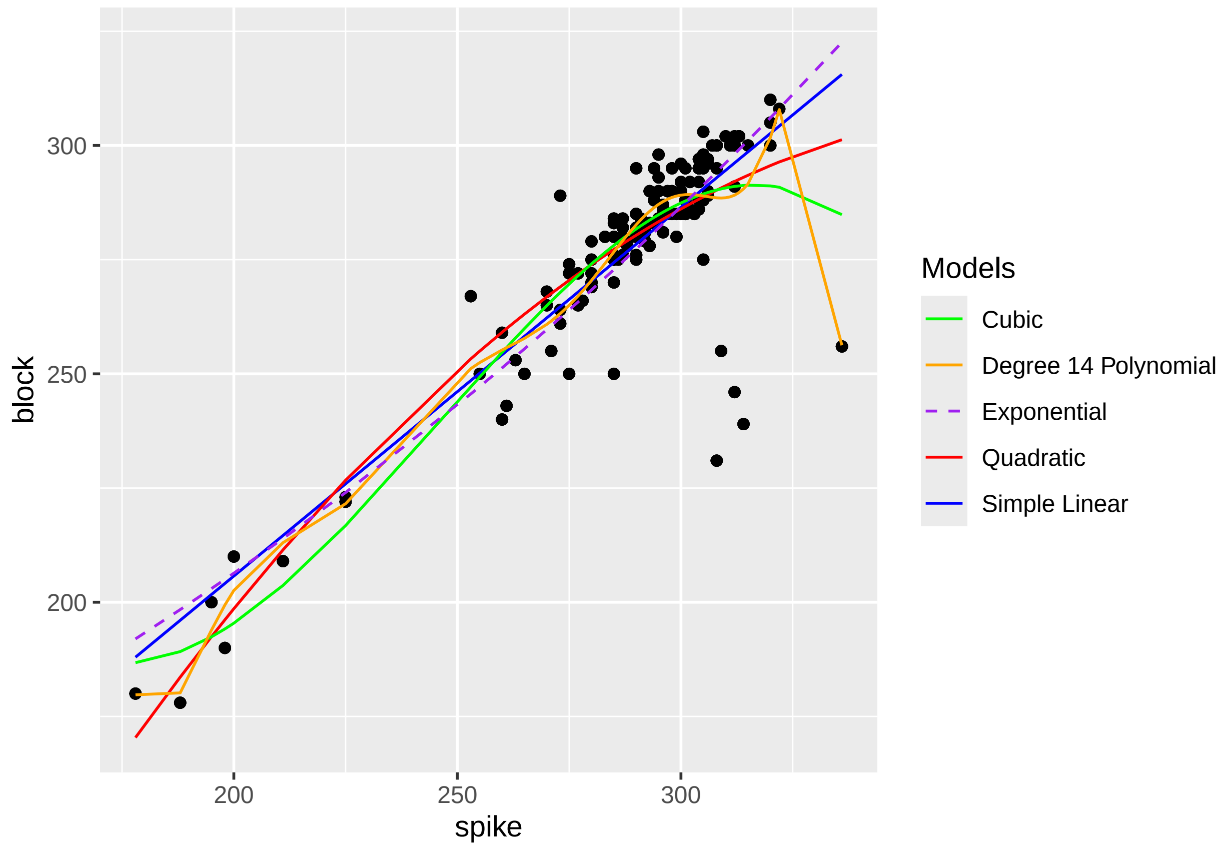
a.)

```r
# create all the models:
lm_formula <- y ~ x
quad_formula <- y ~ x + I(x^2)
cubic_formula <- y ~ x + I(x^2) + I(x^3)
fteen_formula <- y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7) +
                    I(x^8) + I(x^9) + I(x^10) + I(x^11) + I(x^12) + I(x^13) + I(x^14)
exp_formula <- log(y) ~ x

simple_lm <- lm(data = vball_data, formula = lm_formula)
quad_model <- lm(formula = quad_formula, data = vball_data)
cubic_model <- lm(formula = cubic_formula, data = vball_data)
fteen_model <- lm(formula = fteen_formula, data = vball_data)
exp_model <- lm(data = vball_data, formula = exp_formula)

# create scatter plot of models:
# Create predicted values for each model
vball_data$pred_simple <- predict(simple_lm)
vball_data$pred_quad <- predict(quad_model)
vball_data$pred_cubic <- predict(cubic_model)
vball_data$pred_fteen <- predict(fteen_model)
vball_data$pred_exp <- exp(predict(exp_model))

# Create ggplot
ggplot(vball_data, aes(x = x, y = y)) +
  geom_point() +  # Scatterplot of true data values
  geom_line(aes(y = pred_simple, color = "Simple Linear")) +
  geom_line(aes(y = pred_quad, color = "Quadratic")) +
  geom_line(aes(y = pred_cubic, color = "Cubic")) +
  geom_line(aes(y = pred_fteen, color = "Degree 14 Polynomial")) +
  geom_line(aes(y = pred_exp, color = "Exponential"), linetype = "dashed") +
  labs(color = "Models") +  # Legend label
  scale_color_manual(values = c("Simple Linear" = "blue",
                                "Quadratic" = "red",
                                "Cubic" = "green",
                                "Degree 14 Polynomial" = "orange",
                                "Exponential" = "purple")) +  # Custom colors
  guides(color = guide_legend(title = "Models")) +  # Legend title
  xlab("spike") + ylab("block")
```



b.)

```r
# implement K-fold cross validation (K = 10)
# first create MAE loss function:
calculate_MAE <- function(actual, predicted) {
  mean(abs(actual - predicted))
}

# create function for K-fold cross validation:
k_fold_cv <- function(model_formula, data, K = 10) {
  # find number of observations
  n <- nrow(data)
  # randomly shuffle the indices:
  indices <- sample(1:n)
  # set each testing set to have n/K data points
  fold_size <- ceiling(n / K)
  # set an empty vector of length K to be filled with MAE values for later:
  MAE_values <- numeric(K)

  # for each fold:
  for (i in 1:K) {
    start_index <- (i - 1) * fold_size + 1
    end_index <- min(i * fold_size, n)
    test_indices <- indices[start_index:end_index]
    # set the train indices to be everything except what you are testing on
    train_indices <- indices[-(start_index:end_index)]

    # train the model on the training set:
    model <- lm(model_formula, data = data[train_indices, ])
    # make predictions on the test set:
    newdata = data[test_indices, ]
    predicted <- predict(model, newdata)[test_indices]
    actual <- data$block[test_indices]

    # calculate MAE for this fold:
    MAE_values[i] <- calculate_MAE(actual, predicted)
  }
  # return the mean of all the MAE values:
  mean(MAE_values)
}

# Perform K-fold cross-validation for each model
simple_lm_cv <- k_fold_cv(lm_formula, vball_data)
quad_model_cv <- k_fold_cv(quad_formula, vball_data)
cubic_model_cv <- k_fold_cv(cubic_formula, vball_data)
fteen_model_cv <- k_fold_cv(fteen_formula, vball_data)
exp_model_cv <- log(k_fold_cv(exp_formula, vball_data))

# Print the results
print(paste("Simple Linear Model CV MAE:", simple_lm_cv))
```

```
## [1] "Simple Linear Model CV MAE: 6.93504670316755"
```

```r
print(paste("Quadratic Model CV MAE:", quad_model_cv))
```

```
## [1] "Quadratic Model CV MAE: 7.05220844999718"
```

```r
print(paste("Cubic Model CV MAE:", cubic_model_cv))
```

```
## [1] "Cubic Model CV MAE: 7.011949039788"
```

```r
print(paste("Degree 14 Polynomial Model CV MAE:", fteen_model_cv))
```

```
## [1] "Degree 14 Polynomial Model CV MAE: 6.4666919900396"
```

```r
print(paste("Exponential Model CV MAE:", exp_model_cv))
```

```
## [1] "Exponential Model CV MAE: 5.60508328851643"
```

- From the above results, we see that the exponential model performs the best with an MAE of 5.6050833. The degree 14 polynomial also fits well, however we should stay away from that high of a degree since it would likely result in over fitting and would not generalize well to predict future data. For the future, it would be a good idea to try a degree 4 or 5 polynomial as well to see if the data is fit any better.

Question 3:

a.)

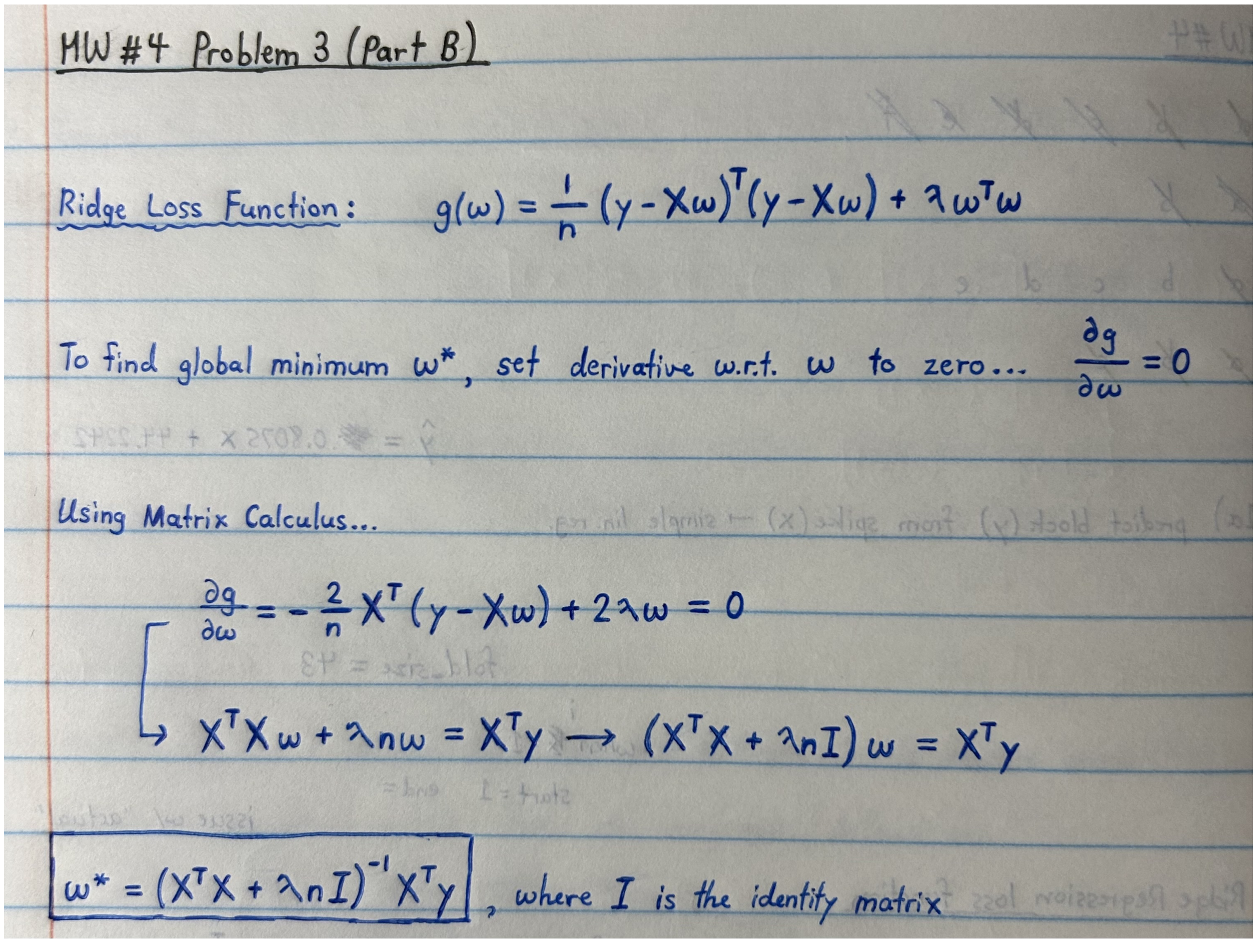
```r
standardize_vector <- function(vector) {
  mean_value <- mean(vector)
  sd_value <- sd(vector)
  standardized_values <- (vector - mean_value) / sd_value
}

intercept <- rep(1, nrow(vball_data))
height <- standardize_vector(vball_data$height)
weight <- standardize_vector(vball_data$weight)
block <- standardize_vector(vball_data$block)

design_matrix <- as.matrix(cbind(intercept, height, weight, block))
head(design_matrix)
```

```
##      intercept     height     weight       block
## [1,]         1  1.4563651  1.5300210  1.14730985
## [2,]         1  0.1213638  0.3084720  0.27329375
## [3,]         1  1.0922738  0.4441996  0.77273152
## [4,]         1 -0.6068188 -0.9130770  0.02357486
## [5,]         1  0.0000000  0.1727443 -0.10128458
## [6,]         1  0.6068188 -0.2344387  0.81435133
```

b.)



HW #4 Problem 3 (Part B)

Ridge Loss Function: $g(\omega) = \frac{1}{n}(y - X\omega)^T(y - X\omega) + \lambda\omega^T\omega$

To find global minimum ω^*, set derivative w.r.t. ω to zero... $\frac{\partial g}{\partial \omega} = 0$

Using Matrix Calculus...

$$\frac{\partial g}{\partial \omega} = -\frac{2}{n}X^T(y - X\omega) + 2\lambda\omega = 0$$

$$X^TX\omega + \lambda n\omega = X^Ty \longrightarrow (X^TX + \lambda nI)\omega = X^Ty$$

$\boxed{\omega^* = (X^TX + \lambda nI)^{-1}X^Ty}$, where I is the identity matrix

- To prove that w^* is a global minimum, we must prove that the ridge loss function is convex. - The first term $\frac{1}{n}(y - Xw)^T(y - Xw)$ is the mean squared error, which is convex because it is a quadratic function of w.
- The second term $\lambda w^T w$ is the regularization term, which is also convex because it is a quadratic function of w.
- Since both terms are convex, their sum $g(w)$ is also convex. Therefore, the ridge loss function $g(w)$ is convex.
- If we were to check out the hessian matrix for the ridge loss function, we would find that it is positive definite, which proves that our function is strictly convex.

```
# compute the value of w*:
n <- nrow(vball_data)
y <- vball_data$spike
I <- diag(dim(t(design_matrix)%*%design_matrix)[1])
w_prime <- solve((t(design_matrix)%*%design_matrix) + 1*n*I) %*% (t(design_matrix) %*% y)
w_prime
```

```
##                 [,1]
## intercept 144.395105
## height      5.242704
## weight      1.024871
## block       9.492309
```

c.)

```r
# g is a function: R^n -> R
# grad_g is the gradient of g: a function \del g: R^n -> R^n
# hess_g is the hessian of g: a function \del g: R^n -> R^(nxn)
# w0 is the initial point
# K is the number of iterations
newts_method <- function(g, grad_g, hess_g, w0, K) {
  # create initial list object
  result <- list(index = c(), val = c())
  # set initial point
  curr_point <- w0
  # create initial empty coordinate matrix: (ncol = n+1)
  mat <- matrix(nrow = K, ncol = length(w0) + 1)

  # iterate for K iterations
  for (i in 1:K) {
    # calculate gradient at current step
    grad_curr <- grad_g(curr_point)
    # calculate hessian at current step
    hess_curr <- hess_g(curr_point)
    # calculate inverse of the hessian
    inv_hess <- solve(hess_curr)
    # update step
    mat[i, 1:length(w0)] <- curr_point - inv_hess %*% grad_curr
    curr_point <- mat[i, 1:length(w0)]
    # fill out function output given current point
    mat[i, length(w0) + 1] <- g(curr_point)
  }

  inputs <- mat[, 1:length(w0)]
  outputs <- mat[, length(w0) + 1]
  min_val <- min(outputs)

  # check if there are duplicate minimum values
  min_indices <- which(outputs == min_val)
  num_min_indices <- length(min_indices)

  if (num_min_indices > 1) {
    # if there are multiple min values, choose the first one
    min_index <- inputs[min_indices[1], ]
  } else {
    # If there is only one unique minimum value, directly extract its index
    min_index <- inputs[min_indices, ]
  }

  result$index <- min_index
  result$val <- min_val

  # return result
  return(result)
}
```

```r
# implementing ridge loss function:
n <- nrow(vball_data)
y <- vball_data$spike
ridge_loss_function <- function(w) {
  (1/n) * t(y - design_matrix %*% w) %*% (y - design_matrix %*% w) + 1*t(w)%*%w
}

grad_ridge_loss <- function(w) {
  (2/n) * t(design_matrix) %*% ((design_matrix %*% w) - y) + 2*1*w
}

hess_ridge_loss <- function(w) {
  (2/n) * (t(design_matrix) %*% design_matrix) + 2*1*I
}

# run newton's method...
newts_method(ridge_loss_function, grad_ridge_loss, hess_ridge_loss, c(0, 0, 0, 0), 10)
```

```
## $index
## [1] 144.395105   5.242704   1.024871   9.492309
##
## $val
## [1] 42052.24
```

d.)

- Using Newton's Method with ridge loss is equivalent to using gradient descent to minimize the MSE. When the ridge penalty term is set to zero, ridge regression reduces to ordinary linear regression.

e.)

- NOTE: **when I ran my R code, the code provided worked and gave me the below output. However, when I tried knitting the .rmd file, there was some error with those lines and it refused to knit. I hope you believe me when I say I ran the code and got this output, which I have now hard coded to be presented in a similar format**

```r
coefficients <- data.frame(
  Intercept = -4.89745,
  x_1 = 0.2448018,
  x_2 = 0.8635284
)

# Print coefficients table
knitr::kable(coefficients, caption = "Coefficients")
```

Table 1: Coefficients

Intercept	x_1	x_2
-4.89745	0.2448018	0.8635284

Question 4:

a.)

```r
# predict if player is libero or not based on height and spike score:
# create design matrix
x_0 <- rep(1, nrow(vball_data))
x_1 <- vball_data$height
x_2 <- vball_data$spike
X <- as.matrix(cbind(x_0, x_1, x_2))
# set y to be 0 or 1 based on Libero
y <- as.numeric(vball_data$is_Libero)
n <- nrow(vball_data)

# create cross entropy loss function for logistic regression
sigmoid <- function(x) {
  1 / (1 + exp(-x))
}
logistic_loss <- function(w) {
  summation_term <- c()
  for (i in 1:n) {
    summation <- y[i] * log(sigmoid(w %*% X[i, ])) +
      (1 - y[i]) * log(1 - sigmoid(w %*% X[i, ]))
    summation_term <- append(summation_term, summation)
  }
  -(1/n) * sum(summation_term)
}

gradient <- function(w) {
  summation_term <- c()
  for (i in 1:n) {
    summation <- (y[i] - sigmoid(w %*% X[i, ])) %*% (X[i, ])
    summation_term <- append(summation_term, summation)
  }
  -(1/n) * sum(summation_term)
}
```

b.)

```r
# implementing normalized gradient descent:
grad_desc_norm <- function(g, alpha, w0, K) {
  # create initial list object
  result <- list(index = c(), val = c())
  # set initial point:
  curr_point <- w0
  # create initial empty coordinate matrix: (ncol = n + 1)
  mat <- matrix(nrow = K, ncol = length(w0) + 1)

  # create gradient normalization function:
  get_grad_norm <- function(gradient) {
    result <- sqrt(sum(gradient^2))
    result
  }

  # iterate for K iterations:
  for (i in 1:K) {
    # calculate gradient at current step:
    grad_curr <- gradient(curr_point)
    # calculate the norm of the gradient:
    norm <- get_grad_norm(grad_curr)

    # update the current point:
    curr_point <- curr_point - (alpha / norm) * grad_curr
    # update output matrix
    mat[i, 1:length(w0)] <- curr_point
    mat[i, length(w0) + 1] <- g(curr_point)
  }

  min_index <- which.min(mat[, length(w0) + 1])
  if (length(min_index > 1)) {
    min_index <- min_index[1]
  }
  min_input <- mat[min_index, 1:length(w0)]
  min_output <- mat[min_index, length(w0) + 1]

  result$index <- min_input
  result$val <- min_output

  # return result:
  result
}

# running normalized gradient descent:
grad_desc_norm(logistic_loss, 0.001, c(25, 0, 0), 10000)
```

```
## $index
## [1] 24.94 -0.06 -0.06
##
## $val
## [1] 0.3616587
```

c.)

```
summary(glm(y ~ X[ , -1], family = "binomial"))
```

```
##
## Call:
## glm(formula = y ~ X[, -1], family = "binomial")
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) 25.578557   4.057790   6.304 2.91e-10 ***
## X[, -1]x_1  -0.166255   0.028699  -5.793 6.91e-09 ***
## X[, -1]x_2   0.005801   0.007216   0.804    0.421
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 274.95  on 428  degrees of freedom
## Residual deviance: 214.12  on 426  degrees of freedom
## AIC: 220.12
##
## Number of Fisher Scoring iterations: 6
```

- The parameter estimates match very closely to the ones that were outputted by the normalized gradient descent algorithm.