Stats 102B HW 4

Bryan Mui - UID 506021334

Due Wed, June 4, 11:00 pm

```
set.seed(777)
library(tidyverse)
library(xtable)
library(ggplot2)
train <- read_csv("train_data.csv")
```

```
Rows: 600 Columns: 601
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (601): X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X15,...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
val <- read_csv("validation_data.csv")
```

```
Rows: 200 Columns: 601
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (601): X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X15,...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Problem 1

Consider the function

$$f(x) = \frac{1}{4}x^4 - x^2 + 2x$$

Part (α)

Using the pure version of Newton's algorithm report x_k for $k = 20$ (after running the algorithm for 20 iterations) based on the following 5 initial points:

1. $x_0 = -1$
2. $x_0 = 0$
3. $x_0 = 0.1$
4. $x_0 = 1$
5. $x_0 = 2$

Newton's pure algorithm is as follows:

1. Select $x_0 \in \mathbb{R}^n$
2. While stopping criterion $>$ tolerance do:
    1. $x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$
    2. Calculate value of stopping criterion($|f(x_{k+1}) - f(x_k)| \leq \epsilon$)

Gradient: $\nabla f(x) = \frac{\partial}{\partial x} = f'(x) = x^3 - 2x + 2$

Hessian: $\nabla^2 f(x) = \frac{\partial^2}{\partial x^2} = f''(x) = 3x^2 - 2$

```
# params
max_iter <- 20
starting_points <- c(-1, 0, 0.1, 1, 2)
stopping_tol <- 1e-6


# algorithm
newton_pure_alg <- function(max_iter, starting_point, stopping_tol) {
  beta <- starting_point
  iterations_ran <- 0
  betas_vec <- c(beta)

  obj <- function(x) {
  return(1/4 * x^4 - x^2 + 2*x)
  }
  grad <- function(x) {
    x^3 - 2*x + 2
  }
  hessian <- function(x) {
    3*x^2 - 2
  }

  # Training loop
  for(i in 1:max_iter) {
    beta_new <- beta - (grad(beta) / hessian(beta))
    betas_vec[i+1] <- beta_new
    if(abs(beta_new - beta) <= stopping_tol) { break }
```

```
    beta <- beta_new
  }
  iterations_ran <- i
  return(list(iterations=iterations_ran, betas=betas_vec))
}

# running the alg
cat("Newton's Algorithm(Pure) For Different Starting Points: \n")
```

```
Newton's Algorithm(Pure) For Different Starting Points:
```

```
for (starting_point in starting_points) {
  result <- newton_pure_alg(max_iter, starting_point, stopping_tol)
  cat("\nStarting Point:", starting_point, "\nIterations:", result$iterations, "\nBetas:\n")
  print(result$betas)
  cat("\n", "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~", "\n")
}
```

```
Starting Point: -1
Iterations: 8
Betas:
[1] -1.000000 -4.000000 -2.826087 -2.146719 -1.842326 -1.772848 -1.769301
[8] -1.769292 -1.769292

 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Starting Point: 0
Iterations: 20
Betas:
 [1] 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Starting Point: 0.1
Iterations: 20
Betas:
 [1] 1.000000e-01 1.014213e+00 7.965577e-02 1.009099e+00 5.222653e-02
 [6] 1.003965e+00 2.332944e-02 1.000804e+00 4.806795e-03 1.000035e+00
[11] 2.072525e-04 1.000000e+00 3.865288e-07 1.000000e+00 1.345590e-12
[16] 1.000000e+00 0.000000e+00 1.000000e+00 0.000000e+00 1.000000e+00
[21] 0.000000e+00

 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Starting Point: 1
Iterations: 20
Betas:
 [1] 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Starting Point: 2
```

```
Iterations: 9
Betas:
 [1]  2.0000000  1.4000000  0.8989691 -1.2887793 -2.1057673 -1.8292000
 [7] -1.7717158 -1.7692966 -1.7692924 -1.7692924

 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Part (i) What do you observe?

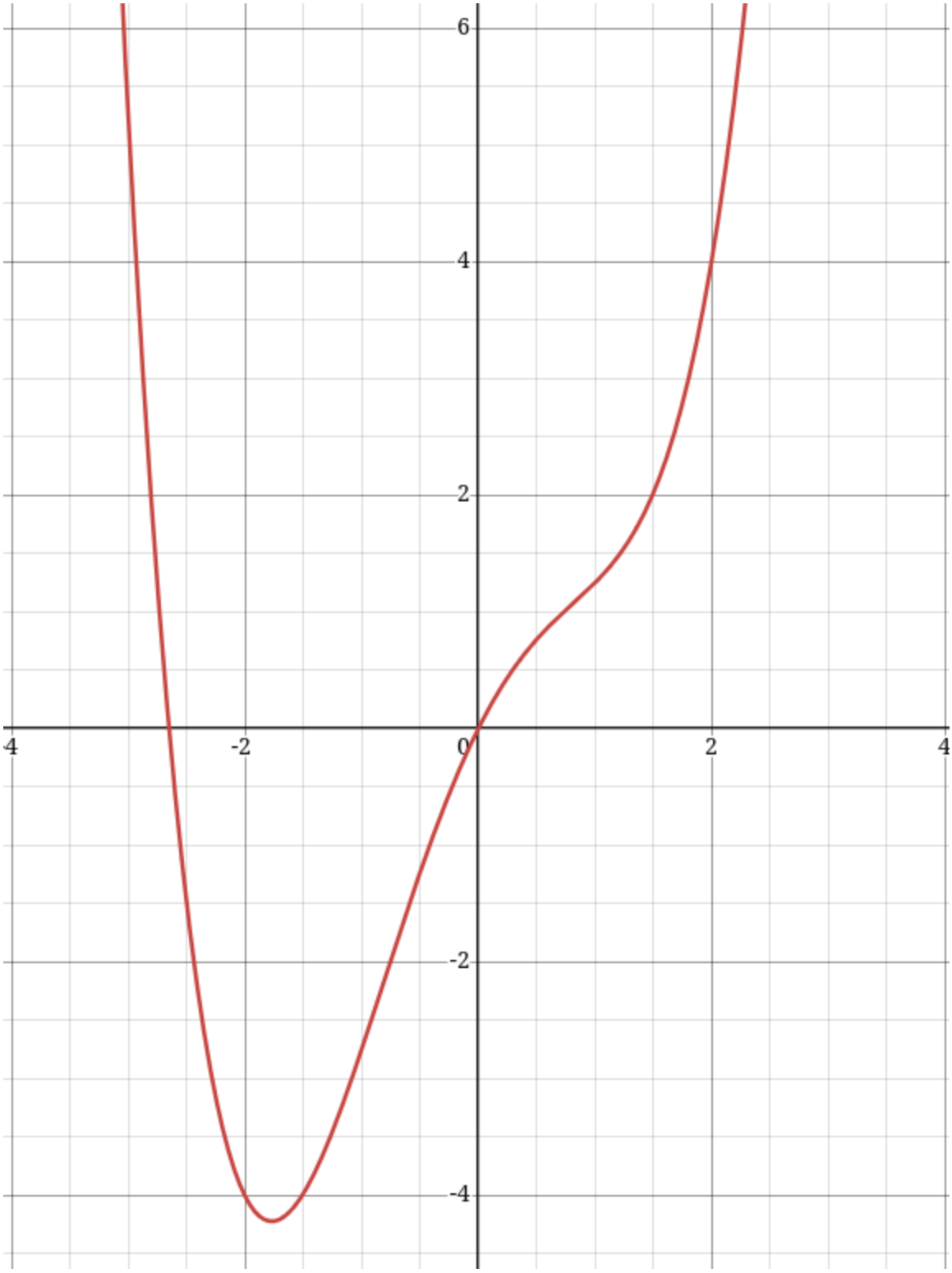


Figure 1: Plot of the Objective Function f(x)

I observe that given the code output, the algorithms that did not converge oscillate between 0 1 0 1, which means that it gets stuck when the curvature of the graph changes. I think that the graph goes from concave up to concave down at this point, which will mess up the gradient descent calculation. The Hessian is the second derivative, or second order gradient, meaning the concavity of the function will affect the sign. Given the update step, $x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$, we can see that if the Hessian's sign goes from negative to positive(concave down to concave up) or vice versa then it will change the sign of the update step to be gradient ascent(away from the minimum) rather than towards it

Part (ii) How can you fix the issue reported in (i)?

I believe that the issue is normally fixed by drawing the graph and getting and understanding of the shape, and therefore choosing one of the starting points that does end up converging. However, in cases where that isn't possible(if the function cannot be plotted or is very complex), I would try to build a vector of possible starting points and perform a grid search where we run the algorithm on multiple starting points and various parameters. This would allow us to find a starting point where the algorithm would converge. Although the grid search is an easy fix, switching algorithms is a better fix. From the lecture slides, there are alternative Newton's algorithms like the Levenberg-Marquard algorithm, which is a fix for when the Hessian is not positive definite by essentially translating the Hessian to positive definite through transforming the eigenvalues of the matrix, which I would implement if there is too much trouble with the pure Newton's algorithm.

Problem 2

Consider the data in the train data.csv file. The first 600 columns correspond to the predictors and the last column to the response y.

Part (i) Implement that proximal gradient algorithm for Lasso regression, by modifying appropriately your code from Homework 1.

Proximal Gradient Descent is good for problems in the form:

$$\min_x F(x) = f(x) + g(x), x \in \mathbb{R}^n$$

Where f and g are functions with a global minimum, f is differentiable, and g is not differentiable. In this case, we have $MSE = f(x) = \frac{1}{n}||y - X\beta||_2^2$ and $g(x) = \lambda||\beta||_1$

The Proximal Gradient Descent Algorithm(General):

1. Select $x_0 \in \mathbb{R}^n$
2. While stopping criterion > tolerance do:
    1. $y_k = x_k + \eta_k \nabla f(x_k)$
    2. $x_{k+1} = \text{prox}_{\eta_k, g}(y_k)$
    3. Calculate value of stopping criterion($|f(x_{k+1}) - f(x_k)| \leq \epsilon$)

Proximal Operator: $\text{prox}_{\eta_k, g}(y_k) = \underset{x}{\text{argmin}}\{g(x) + \frac{1}{2\eta_k}||x - y_k||_2^2\}$, where,

- η_k is the step size at the previous iteration,
- y_k is the gradient step at the previous iteration,
- argmin is the minimum value of the function

The Proximal Function for L1-Regularization(LASSO), as shown in class:

$$\text{prox}_{t, \lambda||\cdot||_1}(z) = S_{t,\lambda}(z) = \text{sign}(z) \cdot \max(|z| - t\lambda, 0)$$

```
# Params
X_train <- train %>%
  select(-y) %>%
  as.matrix()
y_train <- train %>%
  select(y) %>%
  as.matrix()
X_val <- val %>%
  select(-y) %>%
  as.matrix()
y_val <- val %>%
  select(y) %>%
  as.matrix()


# Implement Proximal Gradient Descent
proximal_gradient_descent_lasso <- function(X, y, X_val, y_val, eta, lambda, tol, max_iter) {
  n <- nrow(X)
  p <- ncol(X)
  beta <- rep(0, p)
```

```r
obj_values <- numeric(max_iter) # obj values result
eta_values <- numeric(max_iter)  # eta values result
sse_train_loss <- numeric(max_iter)  # SSE train loss result
sse_val_loss <- numeric(max_iter)  # SSE test loss result
beta_values <- matrix(0, nrow = max_iter, ncol = p) # betas result


# Objective function: Mean Squared Error (MSE)
obj_function <- function(beta) {
  sum((X %*% beta - y)^2) / (2 * n) + lambda * sum(abs(beta))
}

# Gradient function
gradient <- function(beta) {
  t(X) %*% (X %*% beta - y) / n
}

# Proximal Function
prox <- function(eta, lambda, y_k) {
  sign(y_k) * pmax(abs(y_k) - eta * lambda, 0)
}

# Grad Descent Step
for (iter in 1:max_iter) {
  grad <- gradient(beta)

  y_k <- beta - eta * grad
  beta_new <- prox(eta, lambda, y_k)

  # Storing the values for result
  eta_values[iter] <- eta
  beta_values[iter, ] <- beta_new
  obj_values[iter] <- obj_function(beta_new)
  # Storing the SSE losses
  # get prediction and calculate SSE-train-loss
  # y = XB
  y_pred <- X %*% beta_new
  sse_train_loss[iter] <- sum((y - y_pred)^2)

  # get prediction and calculate SSE-val-loss
  # y = XB
  y_pred <- X_val %*% beta_new
  sse_val_loss[iter] <- sum((y_val - y_pred)^2)

  # Stop crit
  if (norm(beta_new - beta, type = "2") < tol) {break}

  beta <- beta_new
}

beta_values <- beta_values[1:iter, ]
obj_values <- obj_values[1:iter]
sse_train_loss <- sse_train_loss[1:iter]
sse_val_loss <- sse_val_loss[1:iter]
```

```
  return(list(iterations=iter, beta_fin=beta, beta_values=beta_values, obj_values=obj_values,
   ↪  eta_values=eta_values, eta=eta, lambda=lambda, sse_train_loss=sse_train_loss,
   ↪  sse_val_loss=sse_val_loss))
}
```

```
m1 <- proximal_gradient_descent_lasso(X_train, y_train, X_val, y_val, 0.01, 0.1, 1e-6, 10000)
```

```
cat("Iterations:", m1$iterations, "\n")
```

```
Iterations: 6435
```

```
cat("Obj Values(Last 10):", "\n")
```

```
Obj Values(Last 10):
```

```
tail(m1$obj_values, n = 10)
```

```
 [1] 10.55455 10.55455 10.55455 10.55455 10.55455 10.55455 10.55455 10.55455
 [9] 10.55455 10.55455
```

```
cat("Final Beta(first 10):", "\n")
```

```
Final Beta(first 10):
```

```
print(m1$beta_fin[1:10])
```

```
 [1]  0.00000000  0.00000000  0.02960429  0.00000000  0.15895370  0.00000000
 [7]  0.00000000  0.00000000 -0.02223592  0.00000000
```

```
cat("SSE Train Loss(last 10):", "\n")
```

```
SSE Train Loss(last 10):
```

```
tail(m1$sse_train_loss, n=10)
```

```
 [1] 5923.782 5923.782 5923.782 5923.782 5923.782 5923.782 5923.782 5923.782
 [9] 5923.782 5923.782
```

```
cat("SSE Val Loss(last 10):", "\n")
```

```
SSE Val Loss(last 10):
```

```
tail(m1$sse_val_loss, n=10)
```

```
 [1] 3995.004 3995.003 3995.003 3995.003 3995.003 3995.003 3995.003 3995.003
 [9] 3995.003 3995.003
```

Part (ii) To select a good value for the regularization parameter λ use the data in the validation data.csv to calculate the sum-of-squares error validation loss.

```
# Grid Search Params
tol <- 1e-6
max_iter <- 10000
eta <- 0.01
lambdas <- c(
  0.001,
  0.01, 0.02, 0.05, 0.08,
  0.1, 0.2, 0.5,
  1, 2, 5, 10, 20, 50
)
#lambdas <- c(0.001, 0.01, 0.1)

# Initialize storing structures
models_df <- data.frame(
  Model = integer(),
  Lambda = numeric(),
  Iterations = integer(),
  Converged = logical(),
  SSE_Train_Loss = numeric(),
  SSE_Val_Loss = numeric()
)
models <- list()

# Running the Code
for (i in 1:length(lambdas)) {
    # train the model on specific lambda
    lambda <- lambdas[i]
    result <- proximal_gradient_descent_lasso(X_train, y_train, X_val, y_val, eta, lambda, tol,
 ↪  max_iter)

    # get prediction and calculate SSE-train-loss
    # y = XB
    y_pred <- X_train %*% result$beta_fin
    sse_train_loss <- sum((y_train - y_pred)^2)

    # get prediction and calculate SSE-val-loss
    # y = XB
    y_pred <- X_val %*% result$beta_fin
    sse_val_loss <- sum((y_val - y_pred)^2)

    # Output the model
    cat("\nModel:", i, "| Lambda:", result$lambda, "| Iterations:", result$iterations, "|
     ↪  Converged?:", result$iterations < max_iter, "| SSE Train Loss:", sse_train_loss, "| SSE Val
     ↪  Loss:", sse_val_loss, "\n")
```

```r
    # Store row in the results data frame
    models_df <- rbind(models_df, data.frame(
      Model = i,
      Lambda = result$lambda,
      Iterations = result$iterations,
      Converged = result$iterations < max_iter,
      SSE_Train_Loss = sse_train_loss,
      SSE_Val_Loss = sse_val_loss
    ))

    # Store the model params in a list for later retrieval
    models[[i]] <- result
}
```

```
Model: 1 | Lambda: 0.001 | Iterations: 10000 | Converged?: FALSE | SSE Train Loss: 523.841 | SSE Val Lo

Model: 2 | Lambda: 0.01 | Iterations: 10000 | Converged?: FALSE | SSE Train Loss: 1128.191 | SSE Val Lo

Model: 3 | Lambda: 0.02 | Iterations: 10000 | Converged?: FALSE | SSE Train Loss: 1881.458 | SSE Val Lo

Model: 4 | Lambda: 0.05 | Iterations: 10000 | Converged?: FALSE | SSE Train Loss: 3670.824 | SSE Val Lo

Model: 5 | Lambda: 0.08 | Iterations: 7629 | Converged?: TRUE | SSE Train Loss: 5112.683 | SSE Val Loss

Model: 6 | Lambda: 0.1 | Iterations: 6435 | Converged?: TRUE | SSE Train Loss: 5923.782 | SSE Val Loss:

Model: 7 | Lambda: 0.2 | Iterations: 3440 | Converged?: TRUE | SSE Train Loss: 8755.456 | SSE Val Loss:

Model: 8 | Lambda: 0.5 | Iterations: 2045 | Converged?: TRUE | SSE Train Loss: 15849.07 | SSE Val Loss:

Model: 9 | Lambda: 1 | Iterations: 1099 | Converged?: TRUE | SSE Train Loss: 27946.54 | SSE Val Loss: 1

Model: 10 | Lambda: 2 | Iterations: 809 | Converged?: TRUE | SSE Train Loss: 39196.53 | SSE Val Loss: 1

Model: 11 | Lambda: 5 | Iterations: 1 | Converged?: TRUE | SSE Train Loss: 39951.34 | SSE Val Loss: 145

Model: 12 | Lambda: 10 | Iterations: 1 | Converged?: TRUE | SSE Train Loss: 39951.34 | SSE Val Loss: 14

Model: 13 | Lambda: 20 | Iterations: 1 | Converged?: TRUE | SSE Train Loss: 39951.34 | SSE Val Loss: 14

Model: 14 | Lambda: 50 | Iterations: 1 | Converged?: TRUE | SSE Train Loss: 39951.34 | SSE Val Loss: 14
```

```r
models_df
```

Model Lambda Iterations Converged SSE_Train_Loss SSE_Val_Loss 1 1 1e-03 10000 FALSE 523.841 18612.824 2 2 1e-02 10000 FALSE 1128.191 10146.815 3 3 2e-02 10000 FALSE 1881.458 6926.871 4 4 5e-02 10000 FALSE 3670.824 4674.736 5 5 8e-02 7629 TRUE 5112.683 4118.298 6 6 1e-01 6435 TRUE 5923.782 3995.003 7 7 2e-01 3440 TRUE 8755.456 4062.508 8 8 5e-01 2045 TRUE 15849.066 5859.498 9 9 1e+00 1099 TRUE 27946.536 10324.959 10 10 2e+00 809 TRUE 39196.533 14343.601 11 11 5e+00 1 TRUE 39951.338 14548.269 12 12 1e+01 1 TRUE 39951.338 14548.269 13 13 2e+01 1 TRUE 39951.338 14548.269 14 14 5e+01 1 TRUE 39951.338 14548.269

```
latex_table <- xtable(models_df, digits = 4, caption = "Model Results For Different Lambdas")
print(latex_table, include.rownames = FALSE)
```

% latex table generated in R 4.5.0 by xtable 1.8-4 package % Wed Jun 4 21:48:25 2025

Model	Lambda	Iterations	Converged	SSE_Train_Loss	SSE_Val_Loss
1	0.0010	10000	FALSE	523.8410	18612.8235
2	0.0100	10000	FALSE	1128.1908	10146.8151
3	0.0200	10000	FALSE	1881.4576	6926.8709
4	0.0500	10000	FALSE	3670.8241	4674.7357
5	0.0800	7629	TRUE	5112.6825	4118.2976
6	0.1000	6435	TRUE	5923.7823	3995.0029
7	0.2000	3440	TRUE	8755.4563	4062.5080
8	0.5000	2045	TRUE	15849.0656	5859.4978
9	1.0000	1099	TRUE	27946.5359	10324.9595
10	2.0000	809	TRUE	39196.5329	14343.6008
11	5.0000	1	TRUE	39951.3385	14548.2692
12	10.0000	1	TRUE	39951.3385	14548.2692
13	20.0000	1	TRUE	39951.3385	14548.2692
14	50.0000	1	TRUE	39951.3385	14548.2692

Table 1: Model Results For Different Lambdas

Part(iii) Show a plot of the training and validation loss as a function of iterations.

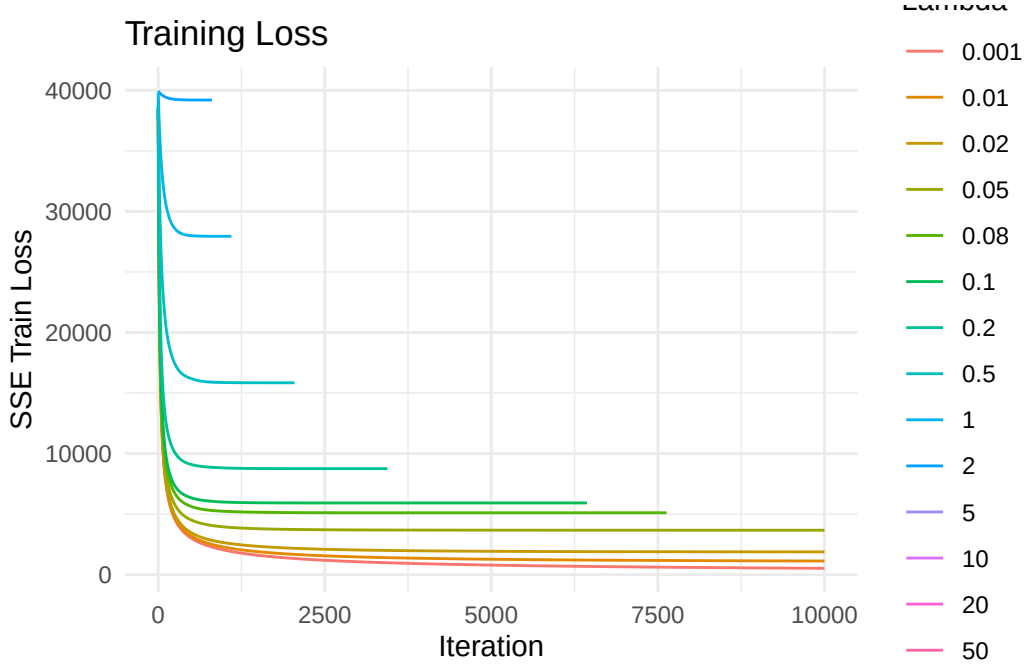
Getting the Data:

```
# build a big data frame
big_df <- data.frame(
  Lambda = numeric(),
  Iterations = integer(),
  SSE_Train_Loss = numeric(),
  SSE_Val_Loss = numeric()
)
# get the SSE data and put it in the data frame
for(model in models) {
  sub_df <- cbind(Lambda = model$lambda, Iteration = 1:model$iterations, SSE_Train_Loss =
 ↪  model$sse_train_loss, SSE_Val_Loss = model$sse_val_loss)
  big_df <- rbind(big_df, sub_df)
}

big_df <- big_df %>%
  mutate(Lambda = factor(Lambda))

head(big_df)
```

```
  Lambda Iteration SSE_Train_Loss SSE_Val_Loss
1  0.001         1       38360.91     14184.63
2  0.001         2       36873.68     13842.38
3  0.001         3       35481.71     13519.99
4  0.001         4       34177.75     13216.04
5  0.001         5       32955.15     12929.25
6  0.001         6       31807.80     12658.40
```

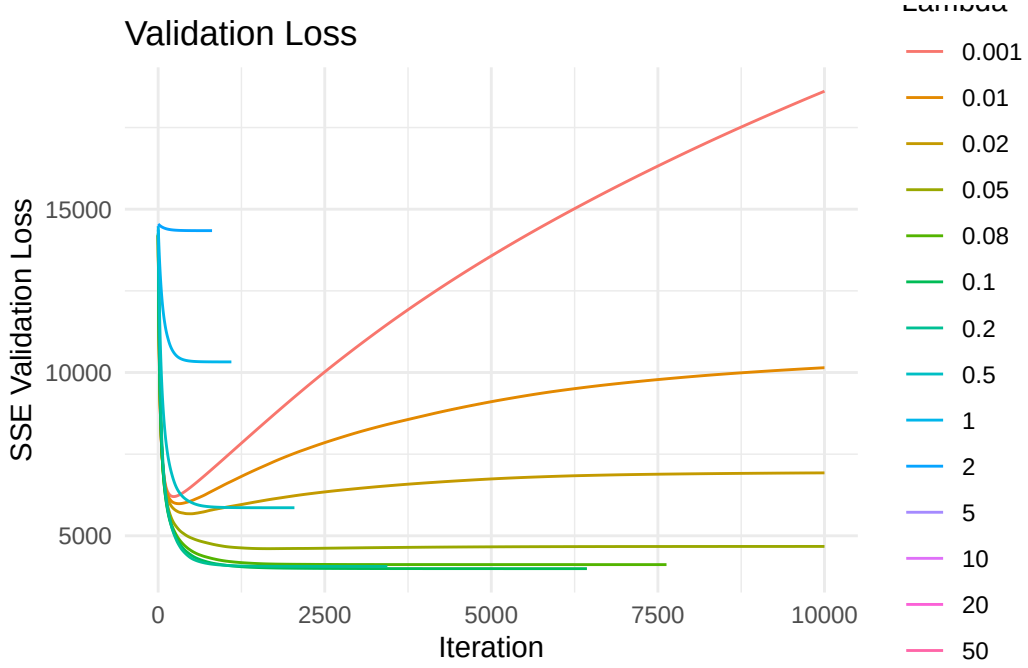
Plotting Training Loss:

```
big_df %>%
  ggplot(aes(x=Iteration, y=SSE_Train_Loss, color=Lambda)) +
    geom_line() +
    theme_minimal() +
    ggtitle("Training Loss") +
    ylab("SSE Train Loss")
```



Plotting Validation Loss:

```
big_df %>%
  ggplot(aes(x=Iteration, y=SSE_Val_Loss, color=Lambda)) +
    geom_line() +
    theme_minimal() +
    ggtitle("Validation Loss") +
    ylab("SSE Validation Loss")
```



Part(iv) Report the number of regression coefficients estimated as zero based on the best value of λ you selected.

Output the table of the best model

```
models_df %>%
  arrange(SSE_Val_Loss)
```

```
   Model Lambda Iterations Converged SSE_Train_Loss SSE_Val_Loss
1      6  1e-01       6435      TRUE       5923.782     3995.003
2      7  2e-01       3440      TRUE       8755.456     4062.508
3      5  8e-02       7629      TRUE       5112.683     4118.298
4      4  5e-02      10000     FALSE       3670.824     4674.736
5      8  5e-01       2045      TRUE      15849.066     5859.498
6      3  2e-02      10000     FALSE       1881.458     6926.871
7      2  1e-02      10000     FALSE       1128.191    10146.815
8      9  1e+00       1099      TRUE      27946.536    10324.959
9     10  2e+00        809      TRUE      39196.533    14343.601
10    11  5e+00          1      TRUE      39951.338    14548.269
11    12  1e+01          1      TRUE      39951.338    14548.269
12    13  2e+01          1      TRUE      39951.338    14548.269
13    14  5e+01          1      TRUE      39951.338    14548.269
14     1  1e-03      10000     FALSE        523.841    18612.824
```

The best lambda that we found was 0.1

```
# print(models[[6]]$beta_fin)
models[[6]]$lambda
```

```
[1] 0.1
```

```
sum(abs(models[[6]]$beta_fin) < 1e-6)
```

```
[1] 380
```

There were 380 coefficients that were shrunk to 0 for the model that had a lambda of 0.1