Statistical Computation Project Topic 1

Alexander Popolow, Griffin Lovato, Demi Zhuang, Aidan Kardan, Thomas Ladocsi

2024-12-06

Problem 1: Explaining Theory Behind Quasi-Newton Methods

Introducing Quasi-Newton Methods

Quasi-Newton methods are similar to steepest descent in that they only require the gradient of the objective function to be supplied at each iteration. By measuring the changes in gradients, they construct a model of the objective function that is good enough to produce superlinear convergence.

The general process for Quasi-Newton methods is that at each time t, when $x(t)$ is updated based on $x(t+1) = x(t) + h(t)$, an opportunity is available to learn about the curvature of ∇f in the direction of $h(t)$ near $x(t)$.

The matrix approximation of the Hessian, $M(t)$, can be efficiently and dynamically updated to incorporate this information.

Then, we can say that $M(t+1)$ satisfies the secant condition if $\nabla f(x(t+1)) - \nabla f(x(t)) = M(t+1)(x(t+1) - x(t))$.

This process is iterative, such that the Hessian approximation is not fully recalculated each time, but rather updated using the previous step, which is comparatively more computationally efficient. It allows us to learn about the curvature of ∇f, and initializing this approach is fairly easy as we can use $M(1) = I$ or the Fisher information matrix if that is known.

BFGS Updates

One popular Quasi-Newton method is the BFGS update method. In order to explain this, let $z(t) = x(t+1) - x(t)$ and $y(t) = \nabla f(t+1) - \nabla f(t)$. The update rule of $M(t)$ in this method is

$$M(t+1) = M(t) - \frac{M(t)z(t)[M(t)z(t)]^T}{[z(t)]^T M(t) z(t)} + \frac{y(t)[y(t)]^T}{[z(t)]^T y(t)}$$

Comparing Convergence Rates of Quasi-Newton with Gradient Descent and Newton's Method

The Quasi-Newton methods provide a fantastic middle ground between Newton's Method and Gradient Ascent and Descent methods in terms of convergence and computation. Newton's Method provides a faster convergence rate than Quasi-Newton methods as it has quadratic convergence near the optimum, however, calculating the Hessian can be very expensive depending on the model and it does not guarantee ascent either. As such, the Quasi-Newton methods are much easier to implement than Newton's Method, and they do not lose out too much in terms of convergence due to the super-linear convergence. On the other end of the convergence algorithms, there is the Gradient Ascent and Descent methods. These methods are

fairly simple, as they only require knowledge of the gradient, like Quasi-Newton. But these methods are even slower than Quasi-Newton, requiring small step sizes that can greatly slow down convergence unless there is already a very good guess of the optimum. These differences can be seen in the following example, comparing an optimization problem using all three methods.

In this case, we will use a quadratic function and will use the methods to find the minimum. This quadratic function is of the form $f(x) = \frac{1}{2}x^T Ax - b^T x$, where A is a symmetric matrix and b is a vector. Moreover, because A is nearly singular in this case, Newton's Method requires a special way of approximating the Hessian, further exemplifying its quality of not being that simple.

```
# Setting the seed for reproducibility
set.seed(123)

# Defining the quadratic function
f <- function(x,A,b){
  return(0.5*t(x)%*%A%*%x - t(b)%*%x)
}

# Defining the gradient of the quadratic function
grad_f <- function(x,A,b){
  return(A%*%x - b)
}

# Gradient Descent Algorithm
grad_descent <- function(f, grad_f, x_init, A,b, step_size=0.01, tol=1e-6, max_iter=1000){
   x <- x_init
   #creating vector to store values to measure convergence
   values <- c(rep(NA, max_iter))
  for (i in 1:max_iter) {
    grad <- grad_f(x, A, b)

    # Check for convergence
    if (sqrt(sum(grad^2)) < tol) {
      cat("Gradient descent converged after", i, "iterations.\n")
      return(list(x=x, convergence_values=values))
    }

    # Update step
    x <- x - step_size * grad
    values[i] <- f(x,A,b)
  }

  warning("Gradient descent did not converge")
  return(x)
}

# Finite difference approximation of the Hessian for Newton's Method
finite_difference_hessian <- function(f,x,A,b,h=1e-6){
  n <- length(x)
  H <- matrix(0, n, n)
  f_x <- f(x, A, b)  # Evaluate f at the current point

  for (i in 1:n) {
    for (j in 1:n) {
      x_ij <- x
```

```
      x_ij[i] <- x[i] + h
      x_ij[j] <- x[j] + h
      f_ij <- f(x_ij, A, b)

      H[i, j] <- (f_ij - f_x) / h^2
    }
  }

  return(H)
}

# Newton's method with finite difference Hessian approximation
newton_finite_difference <- function(f, grad_f, x_init, A, b, tol = 1e-10, max_iter = 1000) {
  x <- x_init
  # creating a list to store convergence values at each iteration for plotting
    convergence_values <- c(rep(NA, max_iter))
  for (i in 1:max_iter) {
    grad <- grad_f(x, A, b)

    # Check if gradient norm is below the tolerance
    if (sqrt(sum(grad^2)) < tol) {
      cat("Newton method with finite difference Hessian converged after", i, "iterations.\n")
      return(list(x=x, convergence_values=convergence_values))
    }

    # Approximate Hessian using finite difference
    H <- finite_difference_hessian(f, x, A, b)

    # Update step using the Hessian approximation
    step <- tryCatch({
      solve(H) %*% grad
    }, error = function(e) {
      cat("Hessian approximation is singular or ill-conditioned after", i, "iterations.\n")
      return(rep(NA, length(x)))  # Return NA if the Hessian is singular
    })

    if (all(is.na(step))) break

    # Update x
    x <- x - step
    convergence_values[i] <- f(x,A,b) # adding new x value to list
  }

  warning("Newton's method with finite difference Hessian did not converge")
  return(list(x=x, convergence_values=convergence_values))
}


# Using optim() to run BFGS method
BFGS_optim <- function(f, grad_f, A,b, x_init){
  # creating a list to store convergence values
  convergence_values <- list()
  obj_func <- function(x){ # recreating f to fit optim requirements and adding a function to trace the
```

```
    value <- 0.5*t(x)%*%A%*%x - t(b)%*%x
    convergence_values <<- c(convergence_values, value)
    return(value)
  }
  grad_func <- function(x){ # recreating grad_f to fit optim requirements
    return(A%*%x - b)
  }
  result <- optim(par=x_init, fn=obj_func, gr=grad_func, method="BFGS")
  #converting convergence values to numeric
  convergence_values <- unlist(convergence_values)
  return(list(x = result$par, convergence_values=convergence_values))
}

# Defining A
A <- matrix(c(1,0.999,0.999,1),2,2) # nearly singular matrix
b <- c(1,2)

# Initial guess
x_init <- c(-400,600)

# Run Gradient Descent
descent <- grad_descent(f,grad_f,x_init, A,b, step_size = 0.01)
```

```
## Gradient descent converged after 965 iterations.
```

```
time_descent <- system.time(grad_descent(f,grad_f,x_init, A,b, step_size = 0.01))
```

```
## Gradient descent converged after 965 iterations.
```

```
# Run Newton's Method
newton <- newton_finite_difference(f,grad_f, x_init, A,b)
```

```
## Warning in newton_finite_difference(f, grad_f, x_init, A, b): Newton's method
## with finite difference Hessian did not converge
```

```
time_newton <- system.time(newton_finite_difference(f,grad_f, x_init, A,b))
```

```
## Warning in newton_finite_difference(f, grad_f, x_init, A, b): Newton's method
## with finite difference Hessian did not converge
```

```
# Run BFGS Method
bfgs <- BFGS_optim(f,grad_f, A,b, x_init)
time_bfgs <- system.time(BFGS_optim(f,grad_f, A,b, x_init))

# Computing the true solution for comparison
x_true <- solve(A,b)

# Printing results
cat("\n\n True Solution:\n", x_true, "\n\n")
```

```
##
##
##  True Solution:
##  -499.2496 500.7504
```

```
cat("Results from each method:\n")
```

```
## Results from each method:
```

```
cat("Newton's Method:\n", newton$x, "\n\n")
```

```
## Newton's Method:
##  -400.0003 599.9997
```

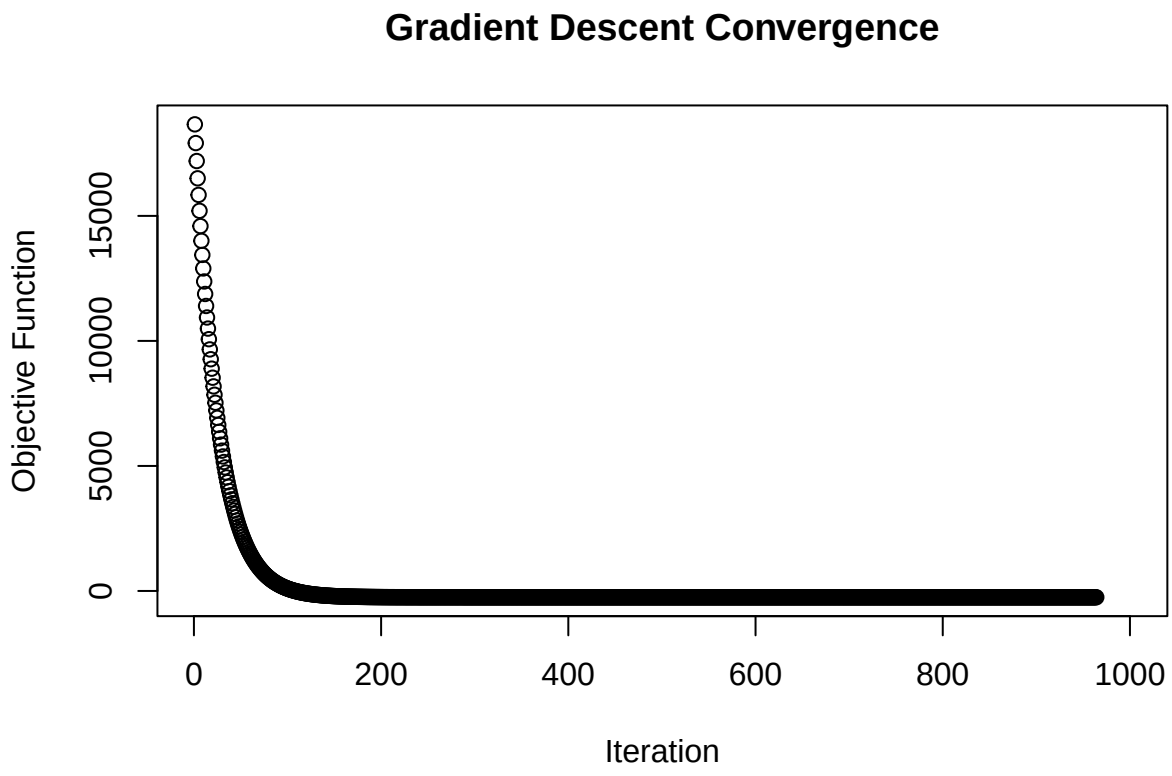
```
cat("Gradient Descent:\n", descent$x, "\n\n")
```

```
## Gradient Descent:
##  -499.2496 500.7504
```

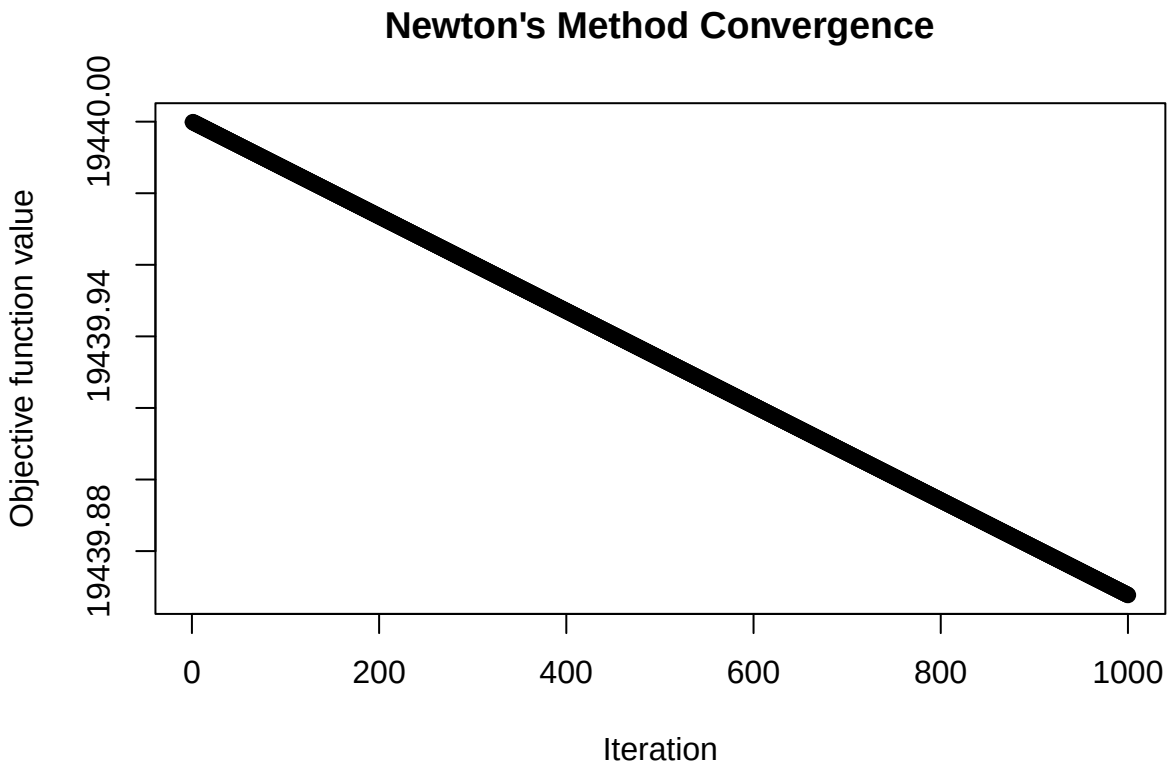
```
cat("BFGS via optim:\n", bfgs$x, "\n\n")
```

```
## BFGS via optim:
##  -499.2496 500.7504
```

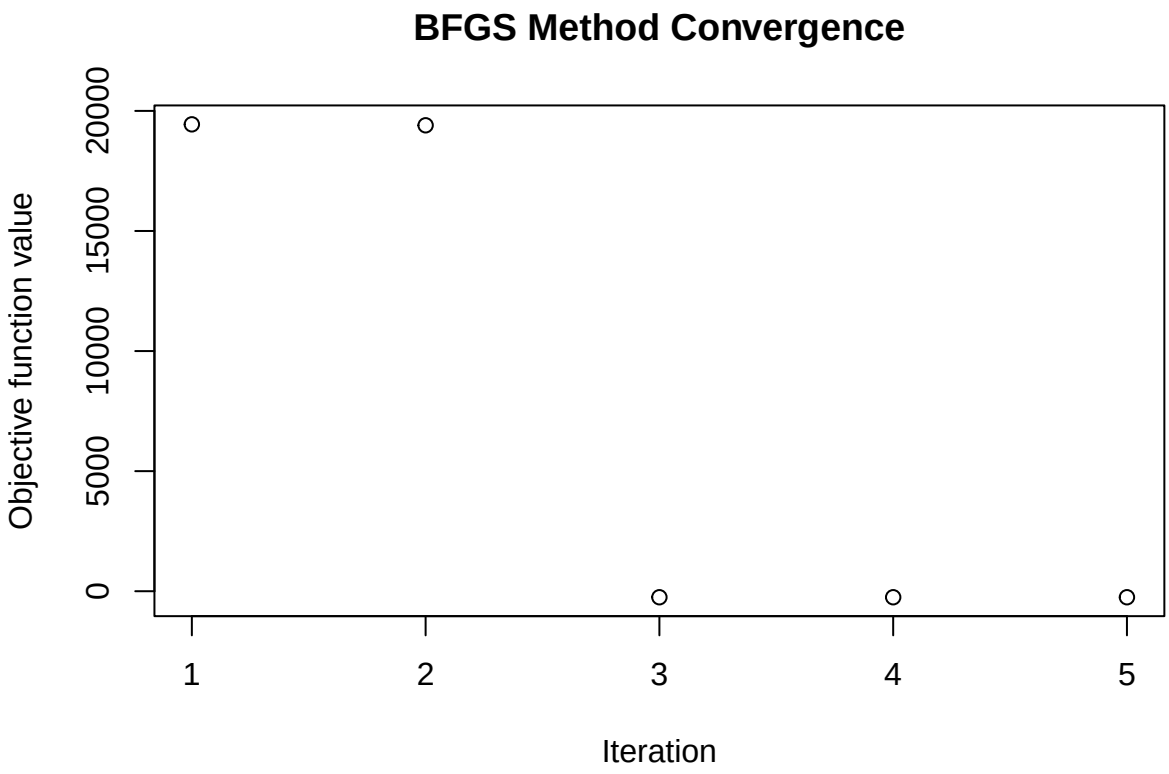
```
# Creating plots to show convergence
plot(descent$convergence_values, xlab="Iteration", ylab="Objective Function", main="Gradient Descent Co
```



```r
plot(newton$convergence_values, xlab="Iteration", ylab="Objective function value", main="Newton's Metho
```




```r
plot(bfgs$convergence_values, xlab="Iteration", ylab="Objective function value", main="BFGS Method Conv
```




```
# Printing time taken for results
cat("Time spent on convergence from each method:\n")
```

```
## Time spent on convergence from each method:
```

```
cat("Newton's Method:\n", time_newton[1:3], "\n\n")
```

```
## Newton's Method:
##  0.06 0 0.07
```

```
cat("Gradient Descent:\n", time_descent[1:3], "\n\n")
```

```
## Gradient Descent:
##  0 0 0.02
```

```
cat("BFGS via optim:\n", time_bfgs[1:3], "\n\n")
```

```
## BFGS via optim:
##  0 0 0
```

As can be seen by these results, the Gradient Descent and BFGS (Quasi-Newton) methods both obtained the true value of the function, however, the BFGS method was even faster than the Gradient Descent. Additionally, it can be seen that the BFGS method took only about 5 iterations to find the result, which

is an exceedingly faster convergence rate than the Gradient Descent method which took 965 iterations. Although the time difference is small here, in higher dimensions this could make the difference between minutes and hours. Additionally, I also have to mention Newton's Method, which we see that it was not even able to find an optimum within 1000 iterations. Despite its ability to converge quadratically in optimal conditions, in this case we see a definitely linear and extremely slow convergence, as seen by the extremely minimal decrease in the objective function value between the start and finish of the iterations. These results reinforce why Quasi-Newton methods are so highly adored, as they are relatively low-hassle to set up while also having great convergence, even as the number of dimensions increases, which we shall explore in the next part.

Implementing the BFGS algorithm and applying it to a high-dimensional logistic regression problem.

In this case, I will again use optim to implement BFGS, but this time it will be for a logistic regression problem that is high-dimensional. I have chosen the wine quality data from Homework 2, only this time I will use the 11 other variables to predict wine quality rather than just 4. The log-likelihood for logistic regression is given by:

$$l(\beta) = \sum_{i=1}^{n} [y_i log(p_i) + (1 - y_i) log(1 - p_i)]$$

The BFGS implementation here will work by minimizing the negative log-likelihood such that the best fitting coefficients of the parameters can be found. The wine data will be split into two parts so that the logistic regression can be trained on one part while its predictive ability can be measured using the other part.

```
wine_data <- fread("winequality-white.csv")
# logistic regression log-likelihood function
log_likelihood <- function(beta, X,y){
  p <- 1 / (1+exp(-(X%*%beta)))
  return(sum(y*log(p)+(1-y)*log(1-p)))
}

# Gradient of the log-likelihood
log_likelihood_gradient <- function(beta, X, y){
  p <- 1 / (1+exp(-(X%*%beta)))
  return(t(X)%*%(y-p))
}

# Setting the seed for reproducibility
set.seed(123)

# generating random sample of indices for sampling so that optimized method can be tested
sample_indices <- sample.int(nrow(wine_data), size=2000)

# Creating binary response variable for a binary logistic regression where quality >= 6: 1, else: 0
response <- ifelse(wine_data$quality>=6,1,0)

# creating X matrix (predictors)
predictors <- wine_data[,-12]

# normalizing the predictors
predictors <- scale(predictors)

# Creating design matrix with an intercept using the predictors
```

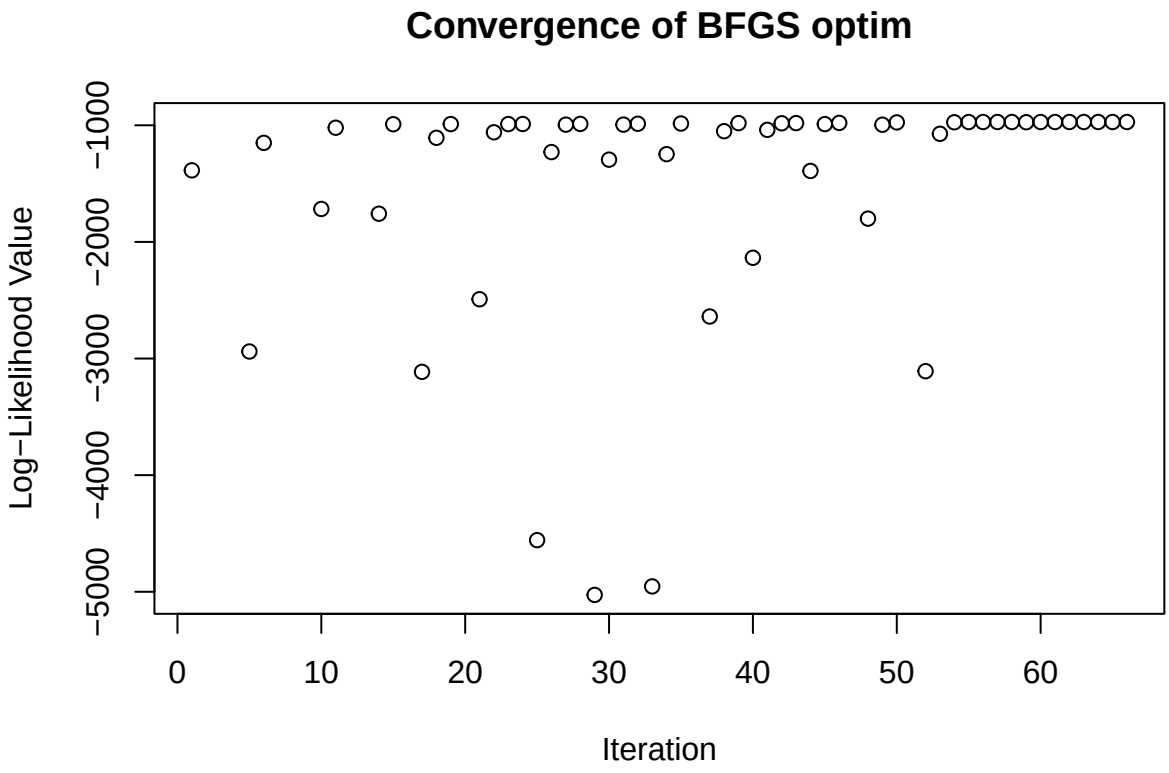
```r
X <- cbind(1, predictors)
variables <- c("Intercept", colnames(X)[-1])

# Taking sample from response and predictors
response_sample <- response[sample_indices]
predictor_sample <- X[sample_indices,]

# Creating initial guess for the regression coefficients
beta_init <- rep(0, ncol(X))

# Creating BFGS function for using optim to calculate coefficients while also plotting convergence
BFGS_logistic <- function(X, y, beta_init, log_likelihood, log_likelihood_gradient){
  # Creating list for convergence values
  convergence_values <- list()
  # creating larger wrapped function to also measure values for graphing convergence
  likelihood_wrapper <- function(X,y,beta){
    value <- log_likelihood(beta, X,y)
    convergence_values <<- c(convergence_values, value)
    return(value)
  }
  result <- optim(par=beta_init, fn=likelihood_wrapper,gr=log_likelihood_gradient, X=X,y=y, method="BFGS
  convergence_values <- unlist(convergence_values)
  return(list(results = result$par, convergence_values=convergence_values))

}

# Running BFGS for high-dimension dataset
results <- BFGS_logistic(predictor_sample,response_sample,beta_init,log_likelihood,log_likelihood_gradi
result <- results$results
time <- system.time(BFGS_logistic(X,response,beta_init,log_likelihood,log_likelihood_gradient))

# Printing the results
cat("The BFGS estimated regression coefficients are:\n\n")
```

```
## The BFGS estimated regression coefficients are:
```

```r
for(i in 1:ncol(X)){
  cat(variables[i],":", round(result[i],3), "\n")
}
```

```
## Intercept : 0.991
## fixed acidity : 0.175
## volatile acidity : -0.689
## citric acid : 0.07
## residual sugar : 1.336
## chlorides : 0.137
## free sulfur dioxide : 0.097
## total sulfur dioxide : -0.029
## density : -1.528
## pH : 0.286
## sulphates : 0.237
## alcohol : 0.629
```

```r
#Plotting the convergence rate
plot(results$convergence_values, xlab="Iteration", ylab="Log-Likelihood Value", main="Convergence of BF
```




```r
cat("\n\n The time used for the BFGS logistic regression is: \n")
```

```
##
##
##  The time used for the BFGS logistic regression is:
```

```r
time
```

```
##    user  system elapsed
##    0.06    0.00    0.07
```

In under a second and with around just 60 iterations, the BFGS algorithm managed to minimize the absolute value of the Log-Likelihood function. Now, we will show how this implementation of the logistic regression optimization compares to the built-in glm logistic regression implementation.

```r
# Calculating MSE of the optimized linear regression using the rest of the data
responses <- response[-sample_indices]
test_predictors <- X[-sample_indices,]
predictions <- test_predictors %*% result

# Running the same logistic regression using glm in order to compare to the BFGS implementation
```

```
model <- glm(formula= response_sample ~., data=as.data.frame(predictor_sample[,-1]), family=binomial(li
time_glm <- system.time(glm(formula= response_sample ~., data=as.data.frame(predictor_sample[,-1]), fami

# Using the glm model to make predictions for MSE calculation
model_predictions <- predict(model, newdata=as.data.frame(test_predictors[,-1]))


cat("The time used for the glm logistic regression is: \n")
```

```
## The time used for the glm logistic regression is:
```

```
time_glm
```

```
##    user  system elapsed
##       0       0       0
```

```
cat("\n\n The estimated logistic regression model using glm is:\n\n")
```

```
##
##
##  The estimated logistic regression model using glm is:
```

```
summary(model)
```

```
##
## Call:
## glm(formula = response_sample ~ ., family = binomial(link = "logit"),
##     data = as.data.frame(predictor_sample[, -1]))
##
## Coefficients:
##                         Estimate Std. Error z value Pr(>|z|)
## (Intercept)              0.99068    0.06099  16.244  < 2e-16 ***
## `fixed acidity`          0.17497    0.10277   1.702 0.088664 .
## `volatile acidity`      -0.68921    0.06850 -10.062  < 2e-16 ***
## `citric acid`            0.06967    0.06045   1.152 0.249136
## `residual sugar`         1.33633    0.23327   5.729 1.01e-08 ***
## chlorides                0.13659    0.05983   2.283 0.022434 *
## `free sulfur dioxide`    0.09688    0.07511   1.290 0.197117
## `total sulfur dioxide` -0.02850    0.08339  -0.342 0.732505
## density                 -1.52783    0.37081  -4.120 3.79e-05 ***
## pH                       0.28582    0.09101   3.140 0.001687 **
## sulphates                0.23718    0.06451   3.676 0.000237 ***
## alcohol                  0.62931    0.19530   3.222 0.001272 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2501.4  on 1999  degrees of freedom
## Residual deviance: 1943.9  on 1988  degrees of freedom
## AIC: 1967.9
##
## Number of Fisher Scoring iterations: 5
```

```r
# Calculating MSE of the two models
BFGS_MSE <- mean((responses - predictions)^2)
glm_MSE <- mean((responses-model_predictions)^2)

cat("\n\nThe MSE of the BFGS logistic regression is:", BFGS_MSE,"\n")
```

```
##
##
## The MSE of the BFGS logistic regression is: 1.689547
```

```r
cat("The MSE of the glm logistic regression is:", glm_MSE, "\n")
```

```
## The MSE of the glm logistic regression is: 1.689541
```

As can be seen, the two methods produced extremely similar results here, with the regression coefficients and the calculated MSEs being different only by small decimal values. Moreover, in this case, even the time for computation was fairly similar. This speaks to the level of accuracy of the BFGS optimization, as even though it did not directly calculate the logistic regression equation like the glm method did, the BFGS implementation still obtained almost exactly the best possible answer. Moreover, the fact that it was able to do so in just roughly 60 short iterations is important because this implies that given a much larger dataset in variables and/or observations, the BFGS method can still perform well and calculate a strong approximation of the regression model while the glm method may struggle as the calculations become increasingly complex. This becomes relevant especially in situations where models need to be calculated multiple times in a row, such as in a bootstrapping procedure. In this case, using BFGS to more quickly define a regression model can allow for bootstrap calculations of MSE for example. However, despite the strength and speed of Quasi-Newton methods, and especially of BFGS, they are not all-powerful. They require knowledge of the relevant objective function and its gradient for any implementation of an optimization problem. In situations where that is already known, then these methods are greatly useful with little downsides. But if these functions are not known, such as in the case of trial and error for determining a suitable regression model, then these methods can possibly be more work than they're worth, as they will require more extensive derivations and formulations to setup than other built-in and primitive functions. Nevertheless, these downsides are outweighed by their powerful and accessible convergence, such that Quasi-Newton methods are a powerful tool for practically any optimization problem.

Problem 2: Harnessing Stochasic Gradient Descent for Optimization

Objective: Explore how Stochastic Gradient Descent (SGD) helps overcome the computational cost of gradient evaluation in high dimensions.

Explain the differences between SGD and standard Gradient Descent

All gradient descent algorithms are methods of minimizing an objective function, in this case negative log-likelihood, parameterized by theta, in this case beta, the vector of our coefficients. It does this by updating theta in the opposite direction of the gradient of the objective function at the current position of theta, using η as the learning rate to determine the size of each step, effectively going "down" the slope of the objective function with the intent of reaching the absolute minimum, although some algorithms can get "caught" on a local minimum.

Standard, or batch Gradient Descent, computes the gradient with respect to the starting parameter theta for the entire dataset in one single move for each epoch:

$$\theta = \theta - \eta \cdot \nabla\theta \ J(\theta)$$

Although fast for small datasets, since this method requires the entire dataset to be in memory all at once, the method is impossible for some computers if the dataset does not fit into memory all at once. It is guaranteed to at least converge to a local minimum, depending on the objective function.

In contrast, Stochastic Gradient Descent, or SGD, performs one update at a time, on each pair of training example x(i) and label y(i), doing this for the entire dataset each epoch:

$$\theta = \theta - \eta \cdot \nabla\theta \ J(\theta; x(i); y(i))$$

SGD, given an appropriate learning rate hyperparameter, is guaranteed to converge to the absolute minimum of a dataset, as its frequent updates cause the objective function to fluctuate, allowing SGD to overcome local minima to continue to move towards the absolute minimum. One shortcoming of SGD is that, if the learning rate is too small, it may take a very long time to converge, and if it is too large, then it may oscillate around the absolute minimum without ever reaching it.

Review momentum-based methods and adaptive learning rages (e.g., Adam, RMSProp)

The Momentum method expands upon SGD by keeping the core concept but adding a velocity fraction γ that keeps track of the update vector from the previous update, adding it to the current update:

$$v_t = \gamma v_{t-1} + \eta \nabla\theta \ J(\theta)$$
$$\theta = \theta - v_t$$

This allows Momentum to overcome local minimum quicker, often leading to faster convergence and reduced oscillation once we get to the convergence points, as subesequent updates going in "opposite directions" will counteract each other.

Nesterov Accelerated Gradient, or NAG (not implemented), effectively looks at updates ahead as well as behind, accounting for the expected future value of theta via an estimate $\theta - \gamma \ vt - 1$:

$$v_t = \gamma v_{t-1} + \eta \nabla\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

This allows the "velocity" of the updates to slow down preemtively, further reducing the effects of oscillations.

Adagrad (not implemented) is a method for dealing with sparse data, and performs larger updates for infrequent (important) parameters. It does this by dividing the learning rate η by sqrt(Gt,ii + epsilon), where G is a diagonal matrix where each element i, i is the sum of squares of the gradients with respect to theta up to epoch t, while epsilon is a tiny term to avoid division by 0:

$$\theta_{t+1,i} = \theta_{t,i} - \eta/(\sqrt{(G_{t,ii+\epsilon})}) \cdot g_{t,i}$$

gt,i here is the gradient of the objective function with respect to parameter θ_i at epoch t. The problem with this is that as t grows, the denominator under the learning rate grows as well, never decreasing (the squares of the gradients can never be negative). This eventually causes our learning rate to vanish with enough iterations.

To fix this problem of Adagrad, we can utilize Root Mean Square Propagation (RMSProp) and Adaptive Moment Estimation (Adam). RMSProp instead divides the learning rate by an exponentially decaying average of squared gradients:

$$[g2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \eta/(sqrt(E[g^2]_t + \epsilon))g_t$$

Adam stores both an exponentially decaying average of past squared gradients vt like RMSProp, as well as an exponentially decaying average of past gradients mt like momentum. Adam's decay rates are hyperparameters β_1 and β_2). Because of this, Adam receives the benefits of both past methods. Note that Adam must correct for the bias towards 0 of both vt and mt by calculating the bias-corrected first and second moment estimates:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

$$m_{t-hat} = m_t/(1-\beta_1^t)$$

$$v_{t-hat} = v_t/(1-\beta_2^2)$$

$$\theta_{t+1} = \theta_t - \eta/(\sqrt{(}v_{t-hat} + \epsilon))m_{t-hat}$$

Implement SGD and Adam in R, apply them to a high-dimensional optimization problem, and compare the results

Note that, as stated in the presentation, we are maximizing the log-likelihood function by altering beta, the vector of our coefficients. This is equivalent to minimizing the negative log-likelihood, thus this is still Gradient DESCENT, despite the addition you see here.

Note that our hyperparameters:

```
learning_rate <- 0.0025
max_iter <- 1000
tol <- 1e-6
```

are the same for every function. This is to provide a level playing field in direct comparison of the 5 methods, and not a model to follow for actual implementation for a real-world problem. Momentum, RMSProp, and Adam would all benefit substantially from better-chosen hyperparameters, likely consistently outperforming regular SGD. Note that function-specific hyperparameters, such as β_1 and β_2 for Adam, use their recommended default values stated in Ruder's article.

Data Prep:

```
library(data.table)
wine_data <- fread("winequality-white.csv")

# Setting the seed for reproducibility
set.seed(123)

# generating random sample of indices for sampling so that optimized method can be tested
sample_indices <- sample.int(nrow(wine_data), size=2000)

# Creating binary response variable for a binary logistic regression where quality >= 6: 1, else: 0
response <- ifelse(wine_data$quality>=6,1,0)

# creating X matrix (predictors)
predictors <- wine_data[,-12]

# normalizing the predictors
predictors <- scale(predictors)
```

```r
# Creating design matrix with an intercept using the predictors
X <- cbind(1, predictors)
variables <- c("Intercept", colnames(X)[-1])

# Taking sample from response and predictors
response_sample <- response[sample_indices]
X_sample <- X[sample_indices,]

# Creating initial guess for the regression coefficients
beta_init <- rep(0, ncol(X))
```

Log-Likelihood:

```r
# Logistic regression log-likelihood function
log_likelihood <- function(beta, X, y) {
  p <- 1 / (1 + exp(-X %*% beta))
  p <- pmax(p, 1e-10)
  p <- pmin(p, 1 - 1e-10)
  return(sum(y * log(p) + (1 - y) * log(1 - p)))
}

# Gradient of the log-likelihood function
log_likelihood_gradient <- function(beta, X, y) {
  p <- 1 / (1 + exp(-X %*% beta))
  p <- pmax(p, 1e-10)
  p <- pmin(p, 1 - 1e-10)
  return(t(X) %*% (y - p))
}
```

GD:

```r
gd <- function(X, y, beta_init, learning_rate, max_iter, tol) {
  beta <- beta_init
  log_likelihood_values <- rep(NA, max_iter)  # Initialize with NA

  for (iter in 1:max_iter) {
    gradient <- log_likelihood_gradient(beta, X, y)

    # Check for gradient stability
    if (any(is.na(gradient))) stop("Gradient contains NA")

    # Update beta
    beta <- beta + learning_rate * gradient

    # Compute log-likelihood and store it
    ll <- log_likelihood(beta, X, y)
    if (is.na(ll)) stop("Log-likelihood returned NA")
    log_likelihood_values[iter] <- ll

    # Check convergence
    if (iter > 1 &&
        !is.na(log_likelihood_values[iter]) &&
        !is.na(log_likelihood_values[iter - 1]) &&
```

```
        abs(log_likelihood_values[iter] - log_likelihood_values[iter - 1]) < tol) {
      log_likelihood_values <- log_likelihood_values[1:iter]
      print(log_likelihood_values[iter])
      break
    }
  }
  print(log_likelihood_values[iter])
  list(beta = beta, log_likelihood_values = log_likelihood_values)
}
```

SGD:

```
# SGD Implementation
sgd <- function(X, y, beta_init, learning_rate, max_iter, tol) {
  beta <- beta_init
  log_likelihood_values <- numeric(max_iter)
  n <- nrow(X)

  for (iter in 1:max_iter) {
    for (i in 1:n) {
      xi <- matrix(X[i, ], nrow = 1)
      yi <- y[i]
      gradient <- log_likelihood_gradient(beta, xi, yi)
      beta <- beta + learning_rate * gradient
    }

    # Store log-likelihood
    ll <- log_likelihood(beta, X, y)
    log_likelihood_values[iter] <- ll

    # Check convergence
    if (iter > 1 && abs(log_likelihood_values[iter] - log_likelihood_values[iter - 1]) < tol) {
      log_likelihood_values <- log_likelihood_values[1:iter]
      print(log_likelihood_values[iter])
      print(iter)
      break
    }
  }
  print(log_likelihood_values[iter])
  list(beta = beta, log_likelihood_values = log_likelihood_values)
}
```

Momentum:

```
momentum <- function(X, y, beta_init, learning_rate, max_iter, tol, momentum_factor = 0.8) {
  beta <- beta_init
  velocity <- rep(0, length(beta))
  log_likelihood_values <- numeric(max_iter)
  n <- nrow(X)

  for (iter in 1:max_iter) {
    for (i in 1:n) {
      xi <- matrix(X[i, ], nrow = 1)
```

```
      yi <- y[i]
      gradient <- log_likelihood_gradient(beta, xi, yi)
      velocity <- momentum_factor * velocity + learning_rate * gradient
      beta <- beta + velocity
    }

    # Store log-likelihood
    ll <- log_likelihood(beta, X, y)
    log_likelihood_values[iter] <- ll

    # Check convergence
    if (iter > 1 && abs(log_likelihood_values[iter] - log_likelihood_values[iter - 1]) < tol) {
      log_likelihood_values <- log_likelihood_values[1:iter]
      print(log_likelihood_values[iter])
      print(iter)
      break
    }
  }
  print(log_likelihood_values[iter])
  list(beta = beta, log_likelihood_values = log_likelihood_values)
}
```

RMSProp:

```
rmsprop <- function(X, y, beta_init, learning_rate, max_iter, tol, rho = 0.9, epsilon = 1e-8) {
  beta <- beta_init
  avg_squared_gradient <- rep(0, length(beta))
  log_likelihood_values <- numeric(max_iter)
  n <- nrow(X)

  for (iter in 1:max_iter) {
    for (i in 1:n) {
      xi <- matrix(X[i, ], nrow = 1)
      yi <- y[i]
      gradient <- log_likelihood_gradient(beta, xi, yi)
      avg_squared_gradient <- rho * avg_squared_gradient + (1 - rho) * (gradient^2)
      beta <- beta + learning_rate * gradient / (sqrt(avg_squared_gradient) + epsilon)
    }

    # Store log-likelihood
    ll <- log_likelihood(beta, X, y)
    log_likelihood_values[iter] <- ll

    # Check convergence
    if (iter > 1 && abs(log_likelihood_values[iter] - log_likelihood_values[iter - 1]) < tol) {
      log_likelihood_values <- log_likelihood_values[1:iter]
      print(log_likelihood_values[iter])
      print(iter)
      break
    }
  }
  print(log_likelihood_values[iter])
  list(beta = beta, log_likelihood_values = log_likelihood_values)
```

```
}
```

Adam:

```
# Adam Implementation
adam <- function(X, y, beta_init, learning_rate, max_iter, tol, beta1 = 0.9, beta2 = 0.999, epsilon = 1e
  beta <- beta_init
  m <- rep(0, length(beta))
  v <- rep(0, length(beta))
  log_likelihood_values <- numeric(max_iter)
  n <- nrow(X)

  for (iter in 1:max_iter) {
    for (i in 1:n) {
      xi <- matrix(X[i, ], nrow = 1)
      yi <- y[i]
      gradient <- log_likelihood_gradient(beta, xi, yi)
      m <- beta1 * m + (1 - beta1) * gradient
      v <- beta2 * v + (1 - beta2) * (gradient^2)

      m_hat <- m / (1 - beta1^iter)
      v_hat <- v / (1 - beta2^iter)

      beta <- beta + learning_rate * m_hat / (sqrt(v_hat) + epsilon)
    }

    # Store log-likelihood
    ll <- log_likelihood(beta, X, y)
    log_likelihood_values[iter] <- ll

    # Check convergence
    if (iter > 1 && abs(log_likelihood_values[iter] - log_likelihood_values[iter - 1]) < tol) {
      log_likelihood_values <- log_likelihood_values[1:iter]
      print(log_likelihood_values[iter])
      print(iter)
      break
    }
  }
  print(log_likelihood_values[iter])
  list(beta = beta, log_likelihood_values = log_likelihood_values)
}
```

Hyperparameters:

```
# Hyperparameters
learning_rate <- 0.0025
max_iter <- 1000
tol <- 1e-6
```

SGD vs. GD:

```r
# Run GD and SGD
system.time(
  gd_result <- gd(X_sample, response_sample, beta_init, learning_rate, max_iter, tol)
)
```

```
## [1] -1167.102
```

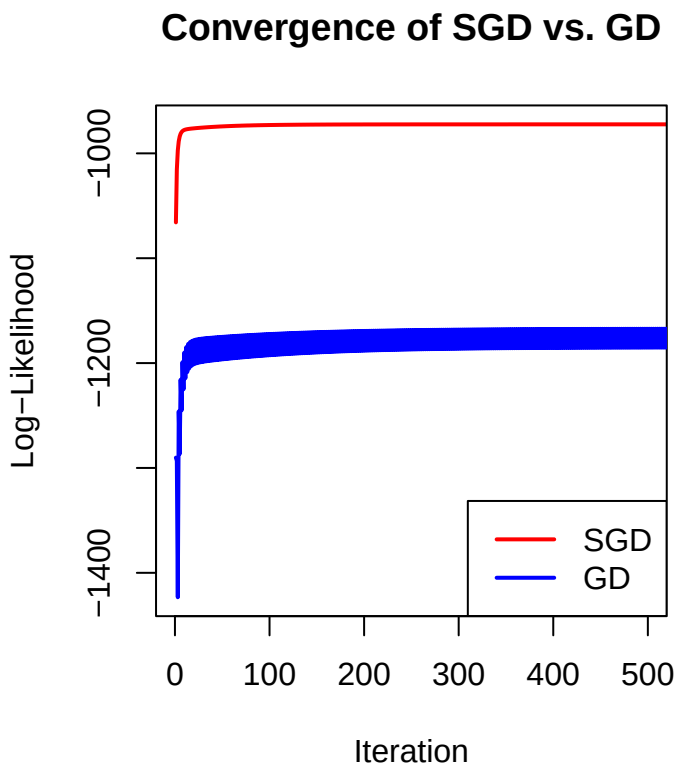
```
##    user  system elapsed
##    0.53    0.14    0.66
```

```r
system.time(
  sgd_result <- sgd(X_sample, response_sample, beta_init, learning_rate, max_iter, tol)
)
```

```
## [1] -972.3541
## [1] 637
## [1] -972.3541
```

```
##    user  system elapsed
##   25.00    0.57   25.99
```

```r
# Extract log-likelihood values
sgd_ll <- sgd_result$log_likelihood_values
gd_ll <- gd_result$log_likelihood_values

# Plot the convergence of log-likelihood values
par(pty = "s")
plot(sgd_ll, type = "l", col = "red", lwd = 2, xlim = range(c(0, 500)), ylim = range(c(sgd_ll, gd_ll)),
     xlab = "Iteration", ylab = "Log-Likelihood", main = "Convergence of SGD vs. GD")
lines(gd_ll, col = "blue", lwd = 2)
legend("bottomright", legend = c("SGD", "GD"), col = c("red", "blue"), lty = 1, lwd = 2)
```



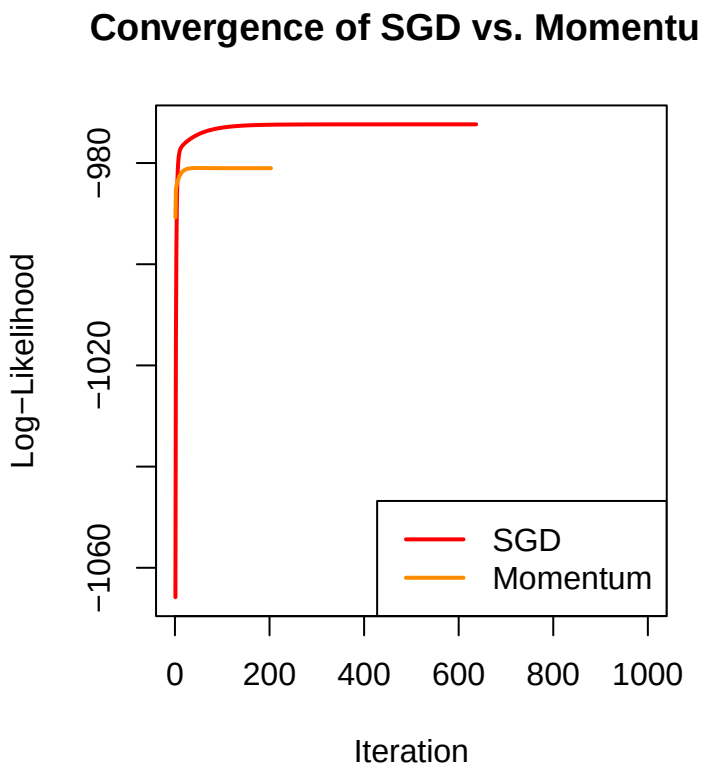
SGD vs. Momentum:

```
# Run Momentum and SGD
system.time(
  momentum_result <- momentum(X_sample, response_sample, beta_init, learning_rate, max_iter, tol)
)
```

```
## [1] -981.0177
## [1] 203
## [1] -981.0177
```

```
##    user  system elapsed
##    8.25    0.17    8.50
```

```
# Extract log-likelihood values
sgd_ll <- sgd_result$log_likelihood_values
momentum_ll <- momentum_result$log_likelihood_values

# Plot the convergence of log-likelihood values
par(pty = "s")
plot(sgd_ll, type = "l", col = "red", lwd = 2, xlim = range(c(0, 1000)), ylim = range(c(sgd_ll, momentu
     xlab = "Iteration", ylab = "Log-Likelihood", main = "Convergence of SGD vs. Momentum")
lines(momentum_ll, col = "darkorange", lwd = 2)
legend("bottomright", legend = c("SGD", "Momentum"), col = c("red", "darkorange"), lty = 1, lwd = 2)
```



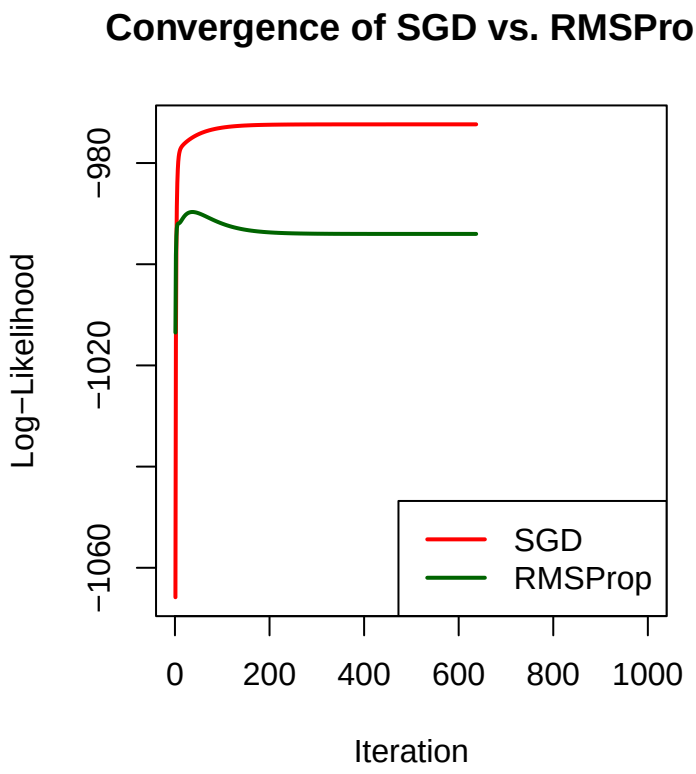
SGD vs. RMSProp:

```
# Run SGD and RMSProp
system.time(
  rmsprop_result <- rmsprop(X_sample, response_sample, beta_init, learning_rate, max_iter, tol)
)
```

```
## [1] -994.0116
## [1] 637
## [1] -994.0116
```

```
##    user  system elapsed
##   26.42    0.40   27.26
```

```
# Extract log-likelihood values
sgd_ll <- sgd_result$log_likelihood_values
rmsprop_ll <- rmsprop_result$log_likelihood_values

# Plot the convergence of log-likelihood values
par(pty = "s")
plot(sgd_ll, type = "l", col = "red", lwd = 2, xlim = range(c(0, 1000)), ylim = range(c(sgd_ll, rmsprop_
     xlab = "Iteration", ylab = "Log-Likelihood", main = "Convergence of SGD vs. RMSProp")
lines(rmsprop_ll, col = "darkgreen", lwd = 2)
legend("bottomright", legend = c("SGD", "RMSProp"), col = c("red", "darkgreen"), lty = 1, lwd = 2)
```



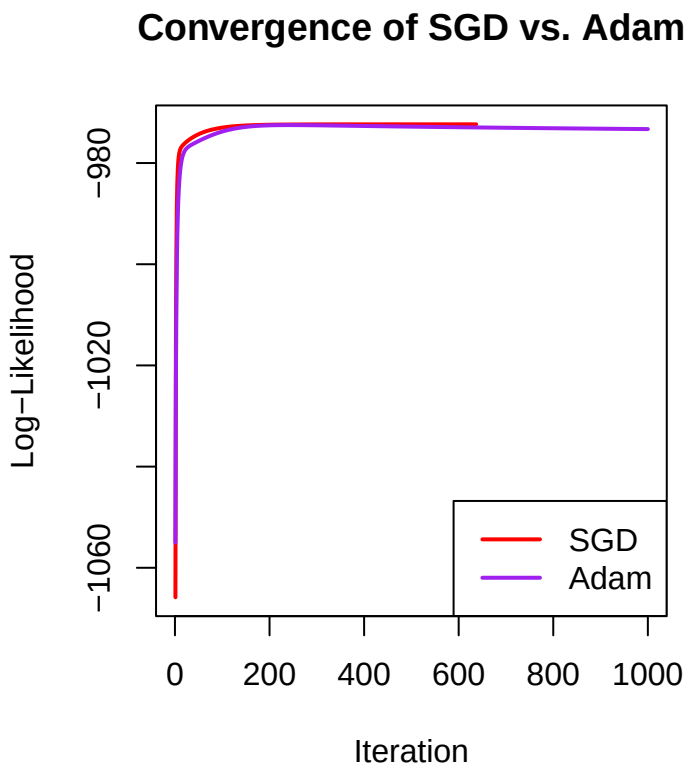
SGD vs. Adam:

```
# Run SGD and Adam
system.time(
  adam_result <- adam(X_sample, response_sample, beta_init, learning_rate, max_iter, tol)
)
```

```
## [1] -973.2914
```

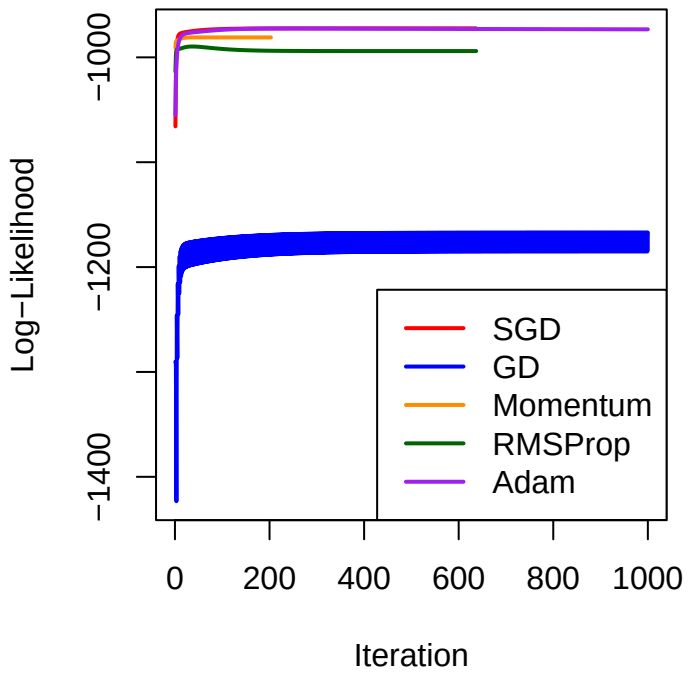
```
##    user  system elapsed
##   43.01    0.89   44.67
```

```
# Extract log-likelihood values
sgd_ll <- sgd_result$log_likelihood_values
adam_ll <- adam_result$log_likelihood_values

# Plot the convergence of log-likelihood values
par(pty = "s")
plot(sgd_ll, type = "l", col = "red", lwd = 2, xlim = range(c(0, 1000)), ylim = range(c(sgd_ll, adam_ll
     xlab = "Iteration", ylab = "Log-Likelihood", main = "Convergence of SGD vs. Adam")
lines(adam_ll, col = "purple", lwd = 2)
legend("bottomright", legend = c("SGD", "Adam"), col = c("red", "purple"), lty = 1, lwd = 2)
```



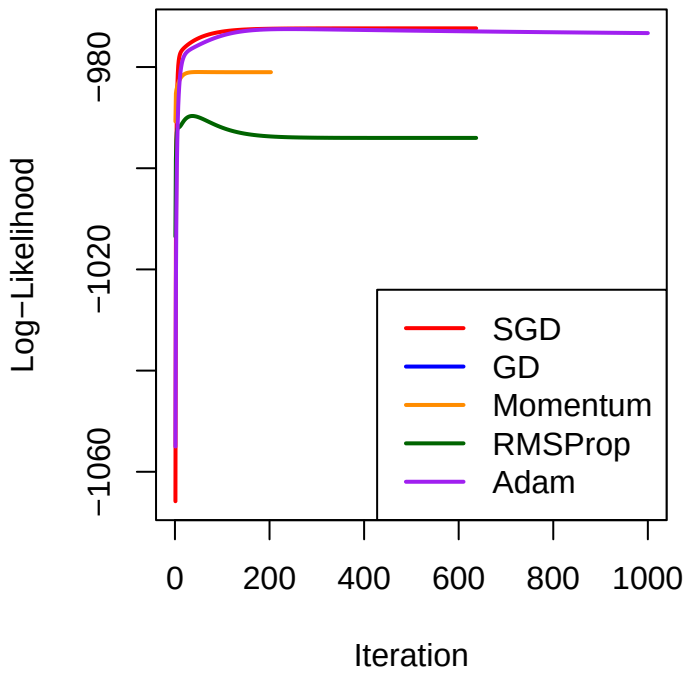
All at once:

```
par(pty = "s")

plot(sgd_ll, type = "l", col = "red", lwd = 2, xlim = range(c(0, 1000)), ylim = range(c(sgd_ll, gd_ll, r
     xlab = "Iteration", ylab = "Log-Likelihood", main = "Convergence of SGD, GD, Momentum, RMSProp, and
lines(gd_ll, col = "blue", lwd = 2)
lines(momentum_ll, col = "darkorange", lwd = 2)
lines(rmsprop_ll, col = "darkgreen", lwd = 2)
lines(adam_ll, col = "purple", lwd = 2)
legend("bottomright", legend = c("SGD", "GD", "Momentum", "RMSProp", "Adam"), col = c("red", "blue", "da
```



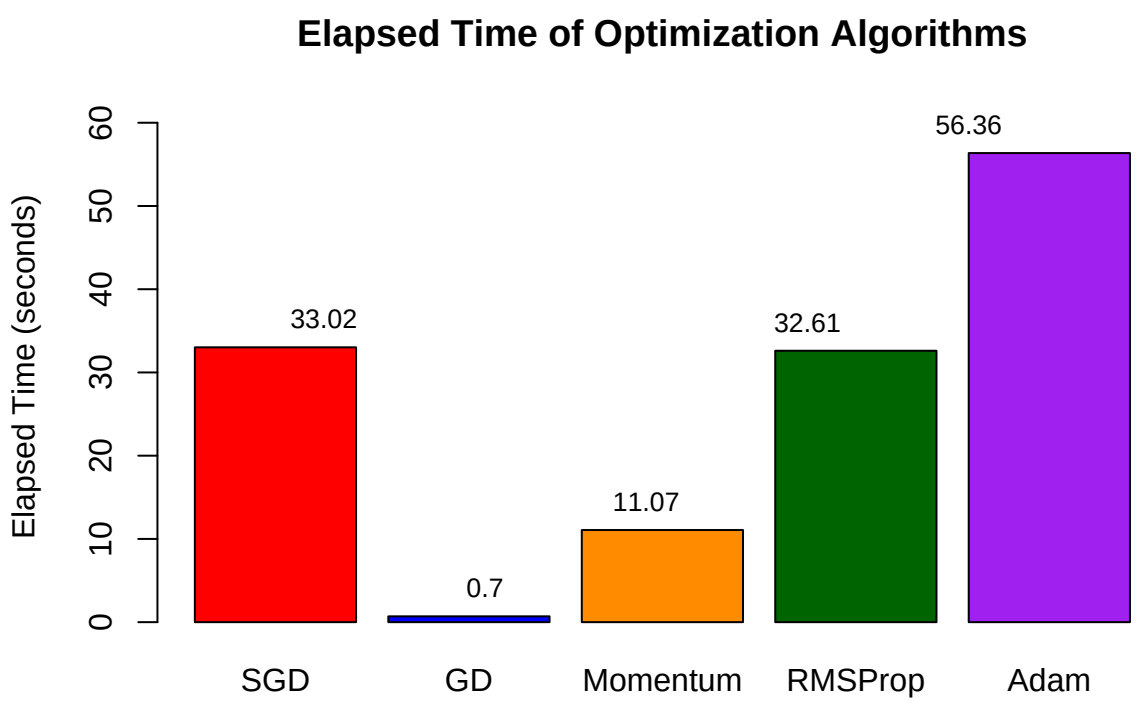
All at once (zoomed)

```
par(pty = "s")

plot(sgd_ll, type = "l", col = "red", lwd = 2, xlim = range(c(0, 1000)), ylim = range(c(sgd_ll, momentu
     xlab = "Iteration", ylab = "Log-Likelihood", main = "Convergence of SGD, GD, Momentum, RMSProp, an
lines(gd_ll, col = "blue", lwd = 2)
lines(momentum_ll, col = "darkorange", lwd = 2)
lines(rmsprop_ll, col = "darkgreen", lwd = 2)
lines(adam_ll, col = "purple", lwd = 2)
legend("bottomright", legend = c("SGD", "GD", "Momentum", "RMSProp", "Adam"), col = c("red", "blue", "d
```



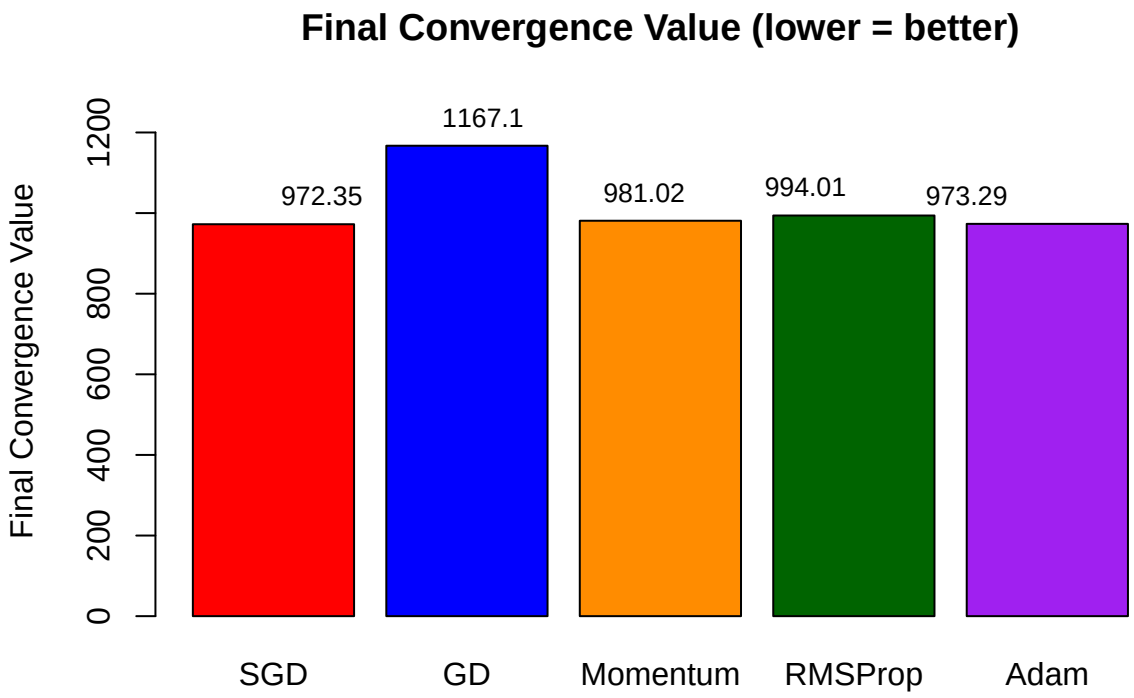
Elapsed Time Bar Plot:

```r
elapsed_times <- c(SGD = 33.02, GD = 0.70, Momentum = 11.07, RMSProp = 32.61, Adam = 56.36)

# Create a bar plot
barplot(
  elapsed_times,
  main = "Elapsed Time of Optimization Algorithms",
  ylab = "Elapsed Time (seconds)",
  col = c("red", "blue", "darkorange", "darkgreen", "purple"),
  ylim = c(0, max(elapsed_times) + 5)  # Add space above the tallest bar
)

# Add text labels to each bar
text(
  x = seq_along(elapsed_times),
  y = elapsed_times,
  labels = round(elapsed_times, 2),
  pos = 3,  # Position text above the bars
  cex = 0.8  # Text size
)
```



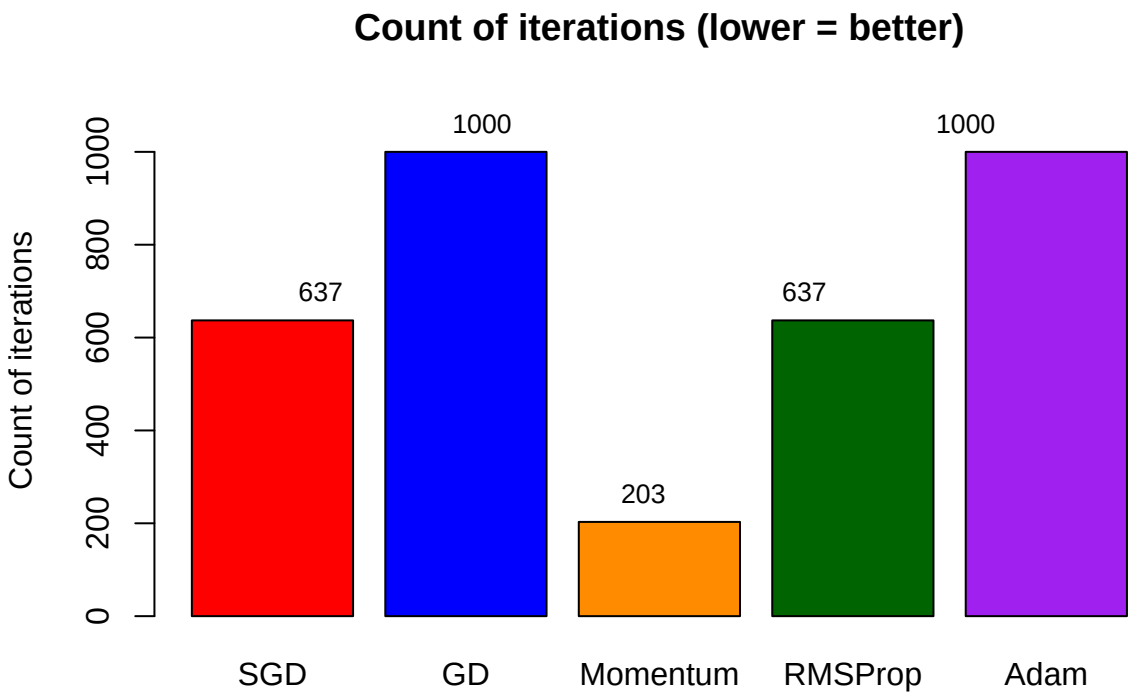
Final Convergence Point Bar Plot:

```r
convergence <- c(SGD = 972.3541, GD = 1167.102, Momentum = 981.0177, RMSProp = 994.0116, Adam = 973.2914

# Create a bar plot
barplot(
  convergence,
  main = "Final Convergence Value (lower = better)",
  ylab = "Final Convergence Value",
  col = c("red", "blue", "darkorange", "darkgreen", "purple"),
  ylim = c(0, max(convergence) + 100)  # Add space above the tallest bar
)

# Add text labels to each bar
text(
  x = seq_along(convergence),
  y = convergence,
  labels = round(convergence, 2),
  pos = 3,  # Position text above the bars
  cex = 0.8  # Text size
)
```



Iteration Count Bar Plot:

```r
iterations <- c(SGD = 637, GD = 1000, Momentum = 203, RMSProp = 637, Adam = 1000)

# Create a bar plot
barplot(
  iterations,
  main = "Count of iterations (lower = better)",
  ylab = "Count of iterations",
  col = c("red", "blue", "darkorange", "darkgreen", "purple"),
  ylim = c(0, max(iterations) + 100)  # Add space above the tallest bar
)

# Add text labels to each bar
text(
  x = seq_along(iterations),
  y = iterations,
  labels = round(iterations, 2),
  pos = 3,  # Position text above the bars
  cex = 0.8  # Text size
)
```



Problem 3: The Curse of Dimensionality in Optimization

Objective: Understand how the curse of dimensionality affects optimization in high-dimensional spaces and explore strategies to mitigate its effects.

Explain Curse of Dimensionality

Curse of dimensionality refers to the various challenges and problems that arise when working in higher dimensions (more variables). As the number of dimensions increases, the volume of the space grows exponentially and the data points within the space become more sparse, resulting in "blind spots" in data. Increased sparsity exponentially increases the "blind spots", making it difficult to find meaningful relationships. Oftentimes machine learning algorithms are affected by the curse of dimensionality as some of it (i.e. the k-th nearest neighbor algorithm) rely on using the distances between points to classify or predict new points. In higher dimensions, the distances between different pairs of points could be similar, resulting in low accuracy and poor performance of the model.

Implications for optimization

Curse of dimensionality has huge implications in optimization and model development since it often explains the poor generalization performance of a model on complex data sets with many variables.

A real-world application where curse of dimensionality may affect the model performance and result in catastrophic failures is in the field of digital medicine. Many types of data and parameters that capture one's health state can be stored as a digital health data, so the dimension of these data sets are enormous

and will continuously grow. Although cross validation can be used to develop a model and avoid overfitting, a large training data set is required, especially for higher dimensions when multiple variables are included in the study, which may be difficult in a clinical setting. Models for higher dimensions also tend to learn overly the complex relationships within the training data and fail to generalize to unseen new data. This has dangerous implications as the developers may not be able to fully evaluate the model's true performance and inaccurate treatment recommendations may be given.

When the data has high intrinsic dimensionality, curse of dimensionality imposes more challenges to the bias-variance dilemma. The bias-variance dilemma describes the trade off between the two types of model error (bias and variance), where decreasing one will increase the other. This makes it difficult to find the balance that minimize the total error, and more difficult for higher dimension problems. Classical non-parametric local learning algorithms and kernel estimators are examples that suffer from the curse of dimensionality.

According to (Härdle et al., 2004), the expected error of a model can be written as follow:

$$error = \frac{C_1}{n\sigma^d} + C_2\sigma^4$$

If n_1 is the number of samples required to get a certain level of error, then $n_1^{(4+d)/5}$ samples are required to get the same level of error in d dimensions, meaning the number of samples increases exponentially with dimensions to keep the error constant. The complexity of an estimator converging to the target function also increases dramatically as dimension increases.

Solutions: PCA & Random Search

Principal component analysis (PCA) is a dimensionality-reduction technique that identifies the most important direction (principal components) in the data set where data varies the most and only uses these covariates to predict the response variable to reduce dimensionality. To perform PCA, we proceed with the following steps: standardize the data, compute the covariance matrix, calculate the eigenvalues and eigenvectors of the covariance matrix, select the top k eigenvectors corresponding to the largest eigenvalues, and project the original data onto the new k-dimensional space. The process is complicated but fortunately there is a built-in R function `prcomp()` that helps us perform PCA.

Random search is a hyperparameter optimization technique that involves randomly selecting hyperparameter values form a predefined range and finding the best-performing set. Hyperparameters are parameters that have a set value before training a machine learning model. Some examples include learning rate, step size in gradient descent, and regularization parameters. The process of random search is the following: we scale numerical variables and standardize the range, then we randomly select hyperparameters and fit to a certain model. We then evaluate the model's performance through metrics such as R-squared or RMSE to determine the best-performing set of hyperparameters.

Dimensionality Reduction Method Implementation

I implement the two methods on all 11 variables in the `wine-quality` data set from a previous assignment. Specifically, the `quality` covariate is used as the target or response variable when fitting a linear regression model onto the data set.

PCA

```
# load the data set
data <- fread("winequality-white.csv"); data <- as.data.frame(data)

# Separate features and target
```

```r
data_numeric <- subset(data, select = -quality)
target <- data$quality

# Standardize features
data_scaled <- scale(data_numeric); data_scaled <- as.data.frame(data_scaled)

# Perform PCA
pca_result <- prcomp(data_scaled, center = TRUE, scale. = TRUE)
summary(pca_result)
```

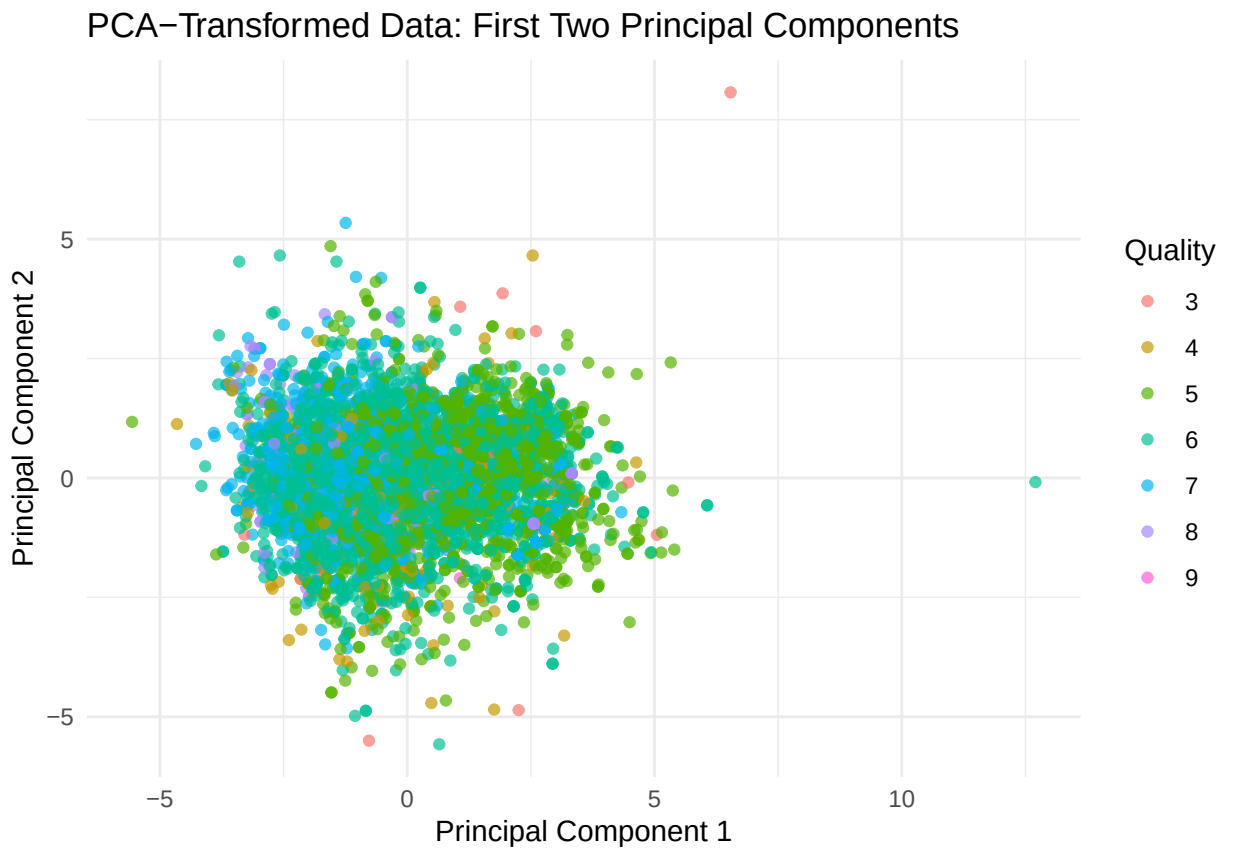
```
## Importance of components:
##                           PC1    PC2    PC3     PC4     PC5     PC6     PC7
## Standard deviation     1.7951 1.2551 1.1053 1.00922 0.98658 0.96889 0.85241
## Proportion of Variance 0.2929 0.1432 0.1111 0.09259 0.08848 0.08534 0.06605
## Cumulative Proportion  0.2929 0.4361 0.5472 0.63979 0.72827 0.81361 0.87967
##                           PC8     PC9    PC10    PC11
## Standard deviation     0.77418 0.64354 0.53804 0.14370
## Proportion of Variance 0.05449 0.03765 0.02632 0.00188
## Cumulative Proportion  0.93416 0.97181 0.99812 1.00000
```

```r
# Determine the number of components to explain 95% variance
explained_variance <- cumsum(pca_result$sdev^2 / sum(pca_result$sdev^2))
num_components <- which(explained_variance >= 0.95)[1]
cat("Number of components needed to retain 95% variance:", num_components, "\n")
```

```
## Number of components needed to retain 95% variance: 9
```

```r
# Transform data into the reduced dimensions and add target variable back
data_pca <- as.data.frame(pca_result$x[, 1:num_components])
data_pca$target <- target

# Plot before dimensionality reduction
original_data <- as.data.frame(data_scaled)
original_data$target <- target
ggplot(original_data, aes(x = "fixed acidity" , y = "volatile acidity",
                          color = as.factor(target))) +
  geom_point(alpha = 0.7) +
  labs(title = "Original Data: First Two Features",
       x = "fixed acidity", y = "volatile acidity", color = "Quality") +
  theme_minimal()
```


```r
# Plot PCA-transformed data using the first two principal components
ggplot(data_pca, aes(x = PC1, y = PC2, color = as.factor(target))) +
  geom_point(alpha = 0.7) +
  labs(title = "PCA-Transformed Data: First Two Principal Components",
       x = "Principal Component 1", y = "Principal Component 2", color = "Quality") +
  theme_minimal()
```



From the plot before dimensionality reduction, we can observe that all the data points are concentrated in the center. This indicates that the two features, `volatile acidity` and `fixed acidity`, alone are not sufficient to explain the variations in the response variable, which is `wine quality`. When implementing PCA as a dimensionality reduction technique, we observe that only 9 out of the 11 variables are required to explain 95% variance in the response. This way, we successfully reduces the dimensionality of the data set by identifying the principal components. In the plot after dimensionality reduction, more data points appear when the first two principal components were plotted, suggesting that more variations could be explained by the two features, `volatile acidity` and `fixed acidity`, after dimensionality reduction.

Random Search

```
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.4.2
```

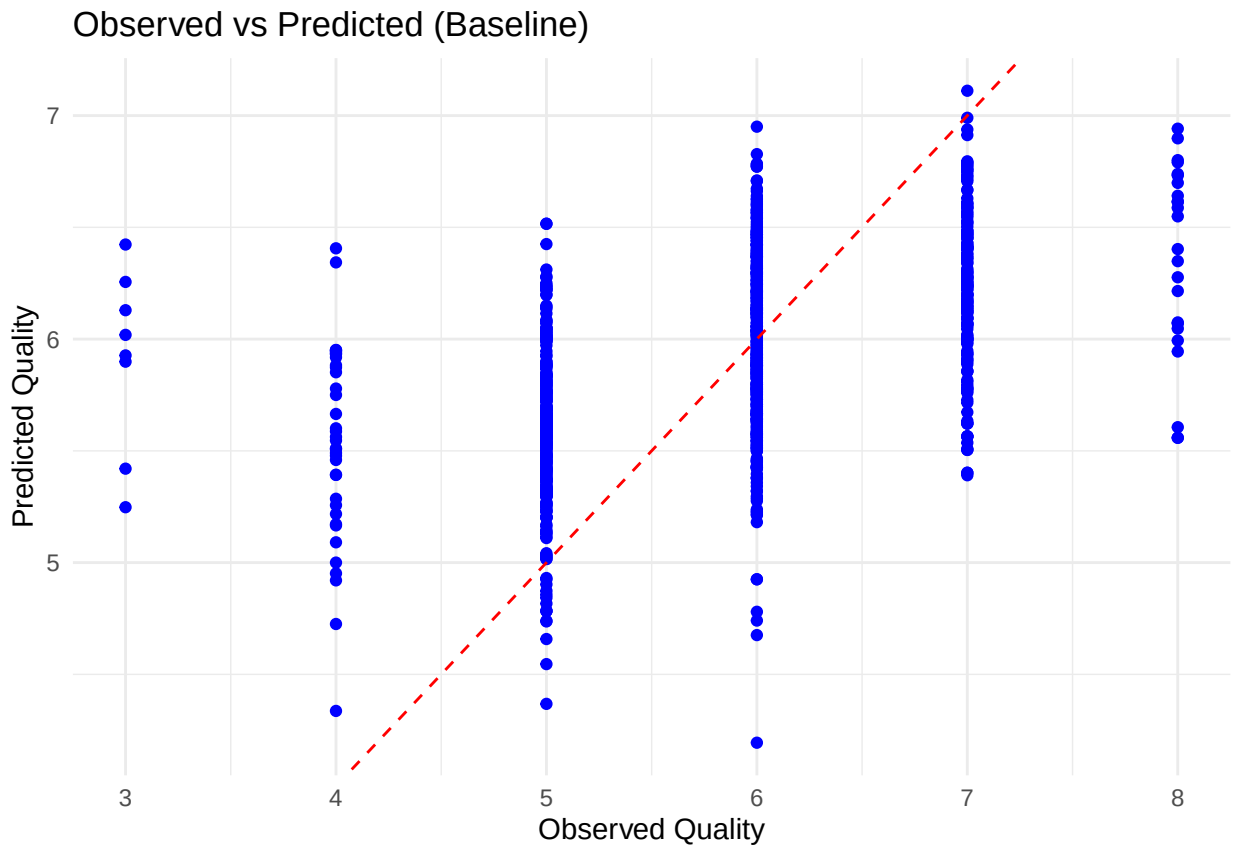
```
## Loading required package: Matrix
```

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

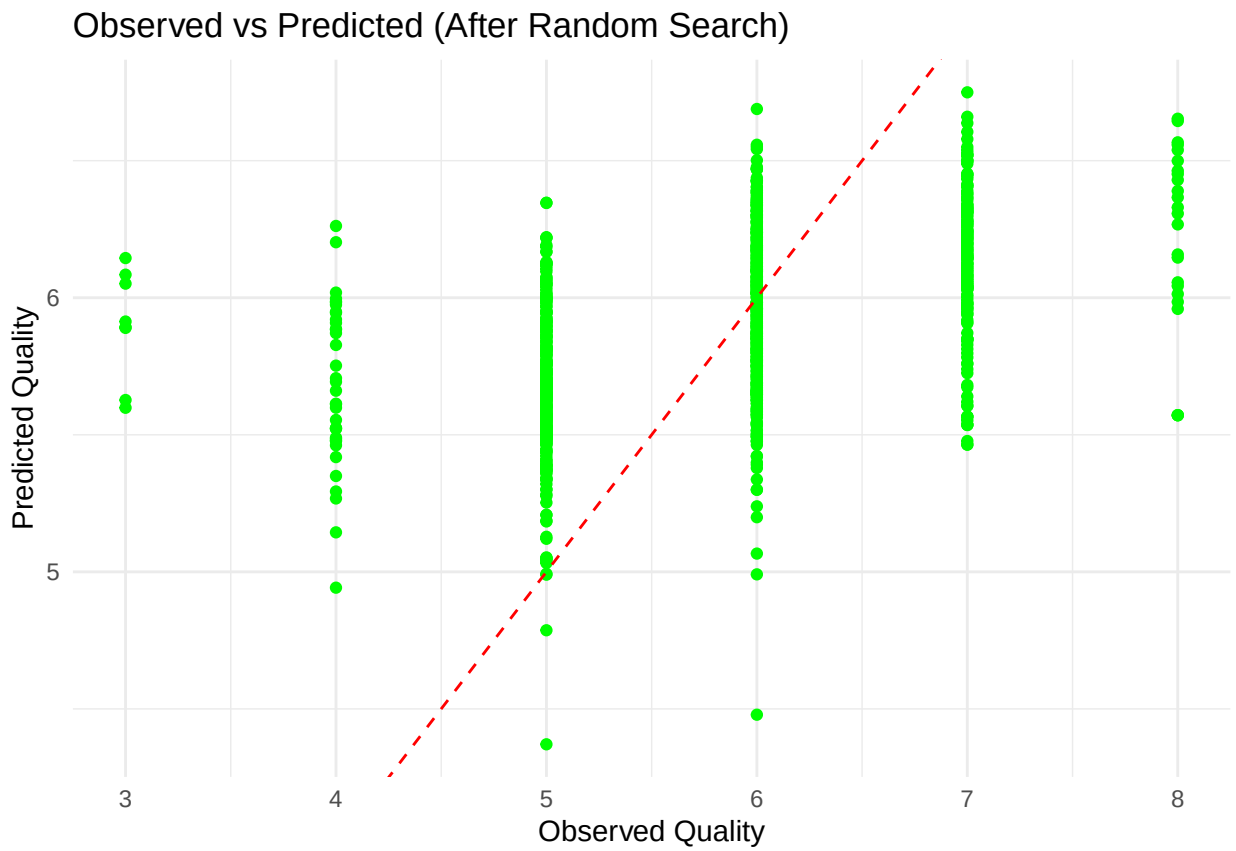
```
library(ggplot2)

# Load and prepare the data
wine <- fread("winequality-white.csv"); wine <- as.data.frame(wine)
```

```r
X <- as.matrix(wine[, -ncol(wine)])
y <- wine$quality
X_scaled <- scale(X) # standardize

# Split data into training and test sets
set.seed(42)
train_index <- sample(1:nrow(X_scaled), 0.8 * nrow(X_scaled))
X_train <- X_scaled[train_index, ]
y_train <- y[train_index]
X_test <- X_scaled[-train_index, ]
y_test <- y[-train_index]

# Baseline model (with default lambda)
baseline_model <- glmnet(X_train, y_train, alpha = 0)
baseline_predictions <- predict(baseline_model, s = 0.01, newx = X_test)
baseline_predictions <- as.vector(baseline_predictions)
baseline_rmse <- sqrt(mean((y_test - baseline_predictions)^2))

# Plot observed vs predicted for baseline model
df_baseline <- data.frame(Observed = y_test, Predicted = baseline_predictions)
plot_baseline <- ggplot(data = df_baseline, aes(x = Observed, y = Predicted)) +
  geom_point(color = "blue") +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  labs(title = "Observed vs Predicted (Baseline)",
       x = "Observed Quality", y = "Predicted Quality") +
  theme_minimal()

# Random Search
set.seed(123)
num_random_search <- 10
random_lambdas <- runif(num_random_search, min = 0.001, max = 10)
results <- data.frame(Lambda = numeric(), RMSE = numeric())

for (lambda in random_lambdas) {
  model <- glmnet(X_train, y_train, alpha = 0, lambda = lambda)
  predictions <- predict(model, newx = X_test) # prediction on testing data set
  rmse <- sqrt(mean((y_test - predictions)^2))
  results <- rbind(results, data.frame(Lambda = lambda, RMSE = rmse))
}

# Best Lambda
best_lambda <- results[which.min(results$RMSE), "Lambda"] # find minimum RMSE
best_model <- glmnet(X_train, y_train, alpha = 0, lambda = best_lambda)
best_predictions <- predict(best_model, newx = X_test)
best_predictions <- as.vector(best_predictions)

# Plot observed vs predicted for the model after random search
df_random <- data.frame(Observed = y_test, Predicted = best_predictions)
plot_after <- ggplot(df_random, aes(x = Observed, y = Predicted)) +
  geom_point(color = "green") +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  labs(title = "Observed vs Predicted (After Random Search)",
       x = "Observed Quality", y = "Predicted Quality") +
```

```
  theme_minimal()

# Display both plots
print(plot_baseline)
```




```
print(plot_after)
```



For this data set, I chose *lambda* that measures the regularization strength in Ridge regression models as my hyperparameter to optimize and perform random search on. I separated the data set into training and testing parts, randomly choose a *lambda* value, fit a Ridge regression model on the training set, and assessed the model's performance by analyzing its predictions on the testing set. I repeated this process multiple times and identified the *lambda* value that resulted in the best model performance, which has the lowest RMSE value.

The two plots before and after random search look similar, potentially because the `wine-quality` data set only includes 11 variables and isn't as complex or high dimensional as others. It may also be due to the fact that a Ridge regression model doesn't fit the data set well. However, there are still some differences between the two plots. For example, data points are more clustered after performing random search, indicating smaller variances and better performance of the model.

Problem 4: Regularization Techniques for High-Dimensional Optimization

Theory Behind Lasso

Lasso (Least Absolute Shrinkage and Selection Operator) is a linear regression method that adds a penalty to the loss function to constrain the magnitude of the regression coefficients.

By penalizing the absolute values of coefficients, Lasso encourages sparsity, driving some coefficients to zero. This makes it an effective tool for both feature selection and regularization, improving model simplicity and interpretability. Lasso helps reduce overfitting by shrinking large coefficients, especially when predictors are highly correlated or the number of predictors exceeds the number of observations.

Why Use Lasso?

Lasso is widely used in scenarios where feature selection is critical, such as high-dimensional datasets where many predictors are irrelevant or redundant. It is particularly effective in addressing multicollinearity, as it selects one variable from a group of correlated variables and discards the others. Moreover, Lasso enhances model performance by balancing bias and variance, making it a powerful method for building interpretable and robust models.

Lasso Analysis

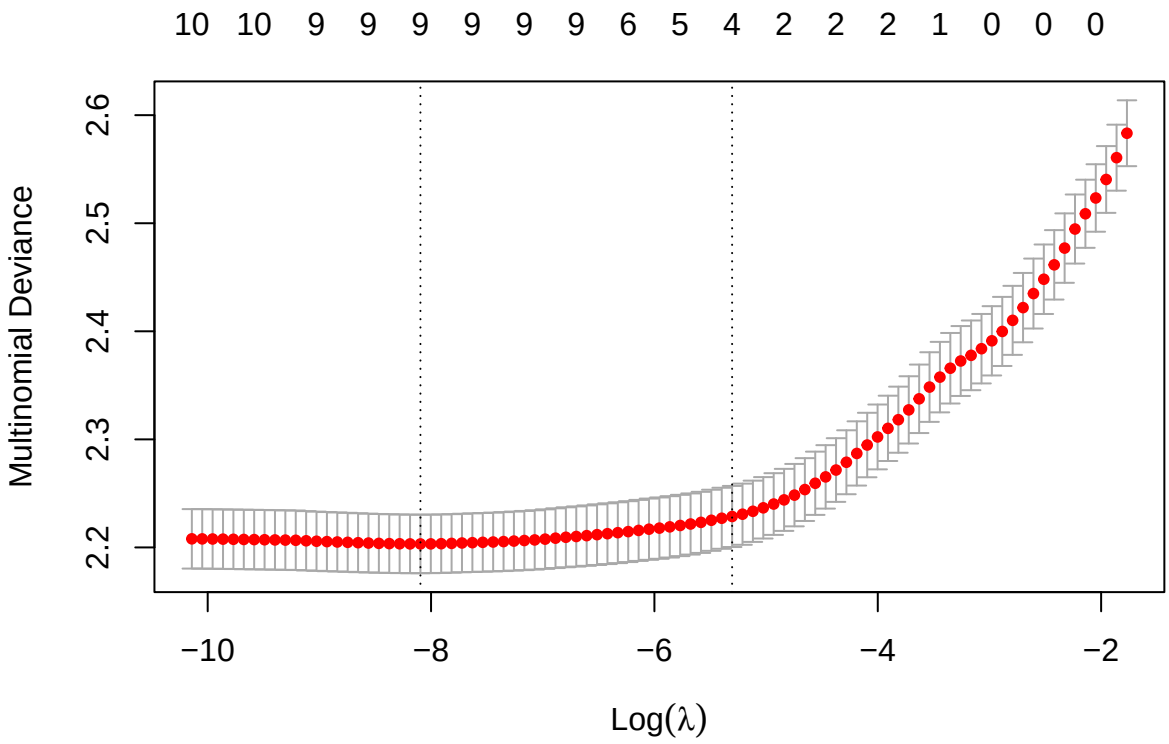
Load and Clean Data

We need to consider that the response is a discrete variable, taking integer values. Therefore, we need to adjust Lasso for a classification problem, since we are dealing with more than 2 classes, instead of logistic regression, we need to consider multinomial regression.

```r
library(data.table)
library(glmnet)

# Load the dataset
data <- fread("winequality-white.csv")

# Separate predictors and response
predictors <- as.matrix(data[, !'quality', with = FALSE])  # Exclude 'quality' and convert to matrix
response <- as.factor(data$quality)  # Convert response to a factor for classification

# Lasso regression with multinomial family
lasso_model <- cv.glmnet(predictors, response, alpha = 1, family = "multinomial")
```

```
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
```

```r
# Plot the cross-validation curve
plot(lasso_model)
```



This plot helps in selecting the optimal regularization strength for the model, balancing bias and variance.

The left dashed line suggests the lambda where deviance is minimized, while the right dashed line provides a more regularized, simpler model that is robust to overfitting.

Now we will find the optimal lambda.

```r
# Get the best lambda
best_lambda <- lasso_model$lambda.min
print(paste("Optimal lambda:", best_lambda))
```

```
## [1] "Optimal lambda: 0.000305026976143207"
```

The next step is to fit the final Lasso model

```r
# Fit the final Lasso model
final_model <- glmnet(predictors, response, alpha = 1, family = "multinomial", lambda = best_lambda)
```

```
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
```

```r
# Coefficients of the final model
coef(final_model)
```

```
## $`3`
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                                s0
##                      -9.690301e+01
## fixed acidity         6.678285e-01
## volatile acidity      5.768038e+00
## citric acid          -2.835976e-01
## residual sugar       -2.239335e-02
## chlorides             9.777827e+00
## free sulfur dioxide   2.440622e-02
## total sulfur dioxide  9.453183e-04
## density               .
## pH                    .
## sulphates            -8.260432e-01
## alcohol              -7.208731e-02
##
## $`4`
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                                s0
##                      -3.082025e+02
## fixed acidity        -1.199144e-01
## volatile acidity      7.857760e+00
## citric acid          -2.329022e-01
## residual sugar       -2.005604e-01
## chlorides             2.384575e-01
## free sulfur dioxide  -4.610007e-02
## total sulfur dioxide -2.772333e-03
## density               2.368408e+02
## pH                   -2.452915e+00
## sulphates             .
## alcohol              -5.914184e-01
##
## $`5`
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                                s0
##                      -1.639598e+02
## fixed acidity        -3.407229e-01
## volatile acidity      4.237268e+00
## citric acid           3.118658e-01
## residual sugar       -8.744076e-02
## chlorides             .
## free sulfur dioxide  -1.245853e-02
## total sulfur dioxide  1.994911e-03
## density               9.815786e+01
## pH                   -2.988536e+00
## sulphates            -3.061070e-01
## alcohol              -7.884211e-01
##
## $`6`
## 12 x 1 sparse Matrix of class "dgCMatrix"
```

```
##                               s0
##                     -87.709912824
## fixed acidity         -0.403362858
## volatile acidity      -1.371937489
## citric acid            0.415760372
## residual sugar         .
## chlorides              1.327931961
## free sulfur dioxide   -0.004863345
## total sulfur dioxide   .
## density               13.336424121
## pH                    -2.619348647
## sulphates              1.009521996
## alcohol                .
##
## $`7`
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                                s0
##                       395.75512727
## fixed acidity           0.06077696
## volatile acidity       -3.26573059
## citric acid            -0.41119551
## residual sugar          0.20884825
## chlorides             -13.03083147
## free sulfur dioxide     .
## total sulfur dioxide    .
## density              -487.03676254
## pH                      .
## sulphates               2.80333428
## alcohol                 0.01260807
##
## $`8`
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                                s0
##                       377.70045833
## fixed acidity           .
## volatile acidity       -2.37206525
## citric acid             .
## residual sugar          0.24827775
## chlorides              -1.89686193
## free sulfur dioxide     0.01381752
## total sulfur dioxide   -0.00162014
## density              -475.55967628
## pH                      0.32590880
## sulphates               1.82045197
## alcohol                 0.32541180
##
## $`9`
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                               s0
##                     -116.6803433
## fixed acidity          0.6724758
## volatile acidity       .
## citric acid            0.3278265
## residual sugar         .
```

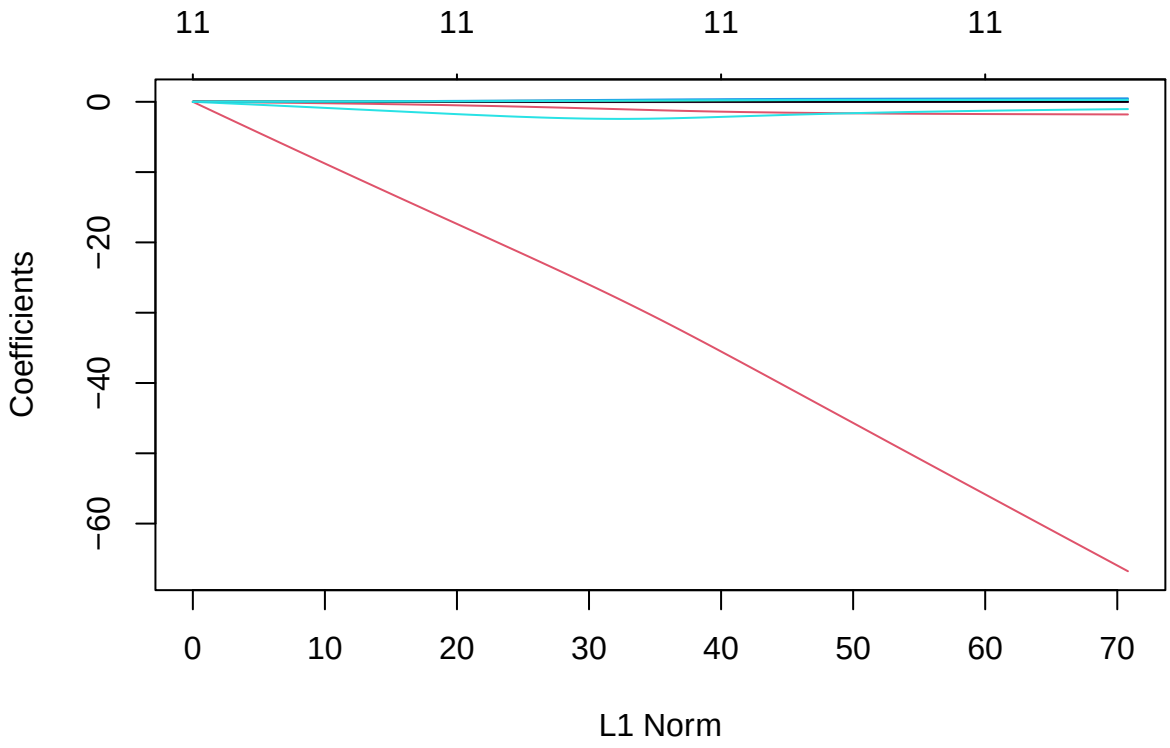
```
## chlorides             -69.3322973
## free sulfur dioxide     .
## total sulfur dioxide    .
## density                 .
## pH                      4.2854805
## sulphates               .
## alcohol                 0.7969466
```

There are 7 different categorical values that the response can take, so we see 7 sets of outputs for the coefficients in the model. As expected, lasso performs feature selection by eliminating predictors that are redundant or irrelevant.

Theory behind Ridge Regression

Ridge regression is a technique developed in the late 1960s by Arthur E. Hoerl and Robert W. Kennard. The goal of ridge regression is to account for overfitting issues that arise with traditional regression techniques. Ridge regression works by adding a penalty term to a regression, training some bias for a greatly reduced variance. Ridge minimizes the squared magnitude of the regression coefficients, and will never set any coefficient to 0. Below is an implementation of a ridge regression using the glmnet library.

```r
# Separate predictors and response
Wine_Quality_1 <- fread("winequality-white.csv")
predictors <- as.matrix(Wine_Quality_1[, -12])
response <- Wine_Quality_1$quality

# Perform Ridge Regression (alpha = 0 for Ridge)
ridge_model <- glmnet(predictors, response, alpha = 0)

#plot coefficient results
plot(ridge_model)
```




```
# Cross-validation to select the best lambda
cv_ridge <- cv.glmnet(predictors, response, alpha = 0)
best_lambda <- cv_ridge$lambda.min

# Coefficients for the best model
ridge_coefficients <- coef(cv_ridge, s = "lambda.min")

# Print the best lambda and coefficients
print(paste("Best Lambda:", best_lambda))
```

```
## [1] "Best Lambda: 0.0385722388763823"
```

```
print(ridge_coefficients)
```

```
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                                s1
## (Intercept)          6.828120e+01
## fixed acidity       -3.370582e-03
## volatile acidity    -1.795520e+00
## citric acid          2.198816e-02
## residual sugar       4.579367e-02
## chlorides           -1.036223e+00
## free sulfur dioxide  4.239866e-03
## total sulfur dioxide -6.309719e-04
## density             -6.677847e+01
```

```
## pH                   3.665307e-01
## sulphates            4.926943e-01
## alcohol              2.638397e-01
```

As shown by the results, the ridge regression is able to calculate both a best lambda through cross-validation and new coefficients for the regressors. This technique is able to handle the increased number of regressors in the dataset, as when we initially worked on it in a previous assignment. The regression uses the lambda term to asses how to control the bias-variance tradeoff. Another key point is that the intercept is not included in the penalty term (lambda). This is to eusure that the procedure does not depend on the origin. Use cases for this type of regression include climate modeling, stock price forecasting, and genetic predictions. Ridge generally performs well in situations with many predictors and high collinearity. It is also less computationally expensive than lasso.

References

Berisha, V., Krantsevich, C., Hahn, P.R. et al. Digital medicine and the curse of dimensionality. npj Digit. Med. 4, 153 (2021). https://doi.org/10.1038/s41746-021-00521-5

Bengio, Y., Delalleau, O., & Le Roux, N. (2005). "The Curse of Dimensionality for Local Kernel Machines." Ruder, S. (2016). "An Overview of Gradient Descent Optimization Algorithms." https://arxiv.org/pdf/1609.04747 Zheyuan Hu, Khemraj Shukla, George Em Karniadakis, Kenji Kawaguchi, Tackling the curse of dimensionality with physics-informed neural networks, Neural Networks, Volume 176, 2024, 106369, ISSN 0893-6080, https://doi.org/10.1016/j.neunet.2024.106369