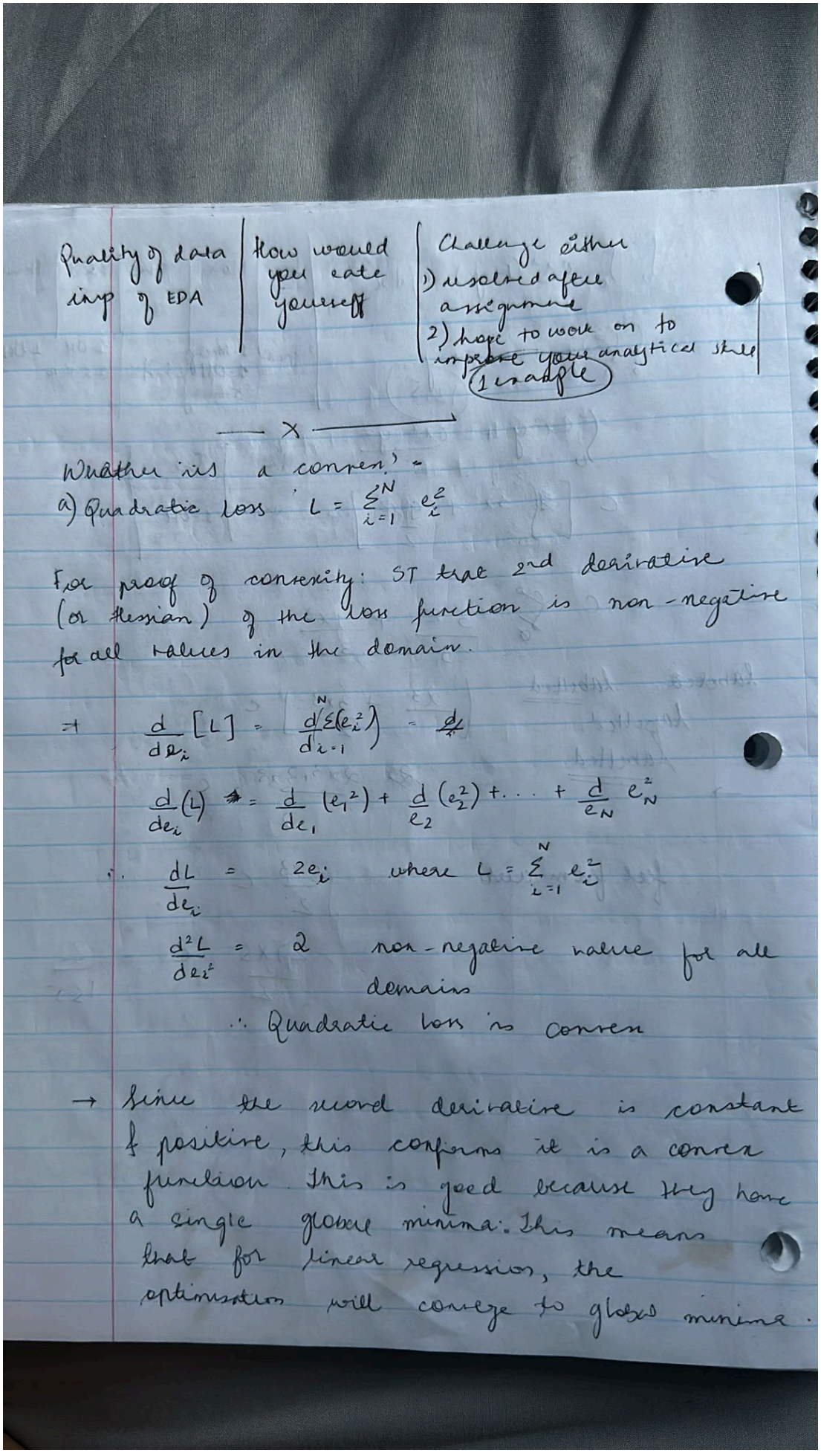
Questions 1-2 images



Quality of data imp of EDA	How would you rate yourself	Challenge either 1) resolved after assignment 2) hope to work on to improve your analytical skill (1 example)

Whether it's a convex:

a) Quadratic loss $L = \sum_{i=1}^{N} e_i^2$

For proof of convexity: ST that 2nd derivative (or Hessian) of the loss function is non-negative for all values in the domain.

$$\rightarrow \frac{d}{de_i}[L] = \frac{d}{de_i}\left(\sum_{i=1}^{N} e_i^2\right) = \frac{dL}{de_i}$$

$$\frac{d}{de_i}(L) = \frac{d}{de_1}(e_1^2) + \frac{d}{e_2}(e_2^2) + \ldots + \frac{d}{e_N} e_N^2$$

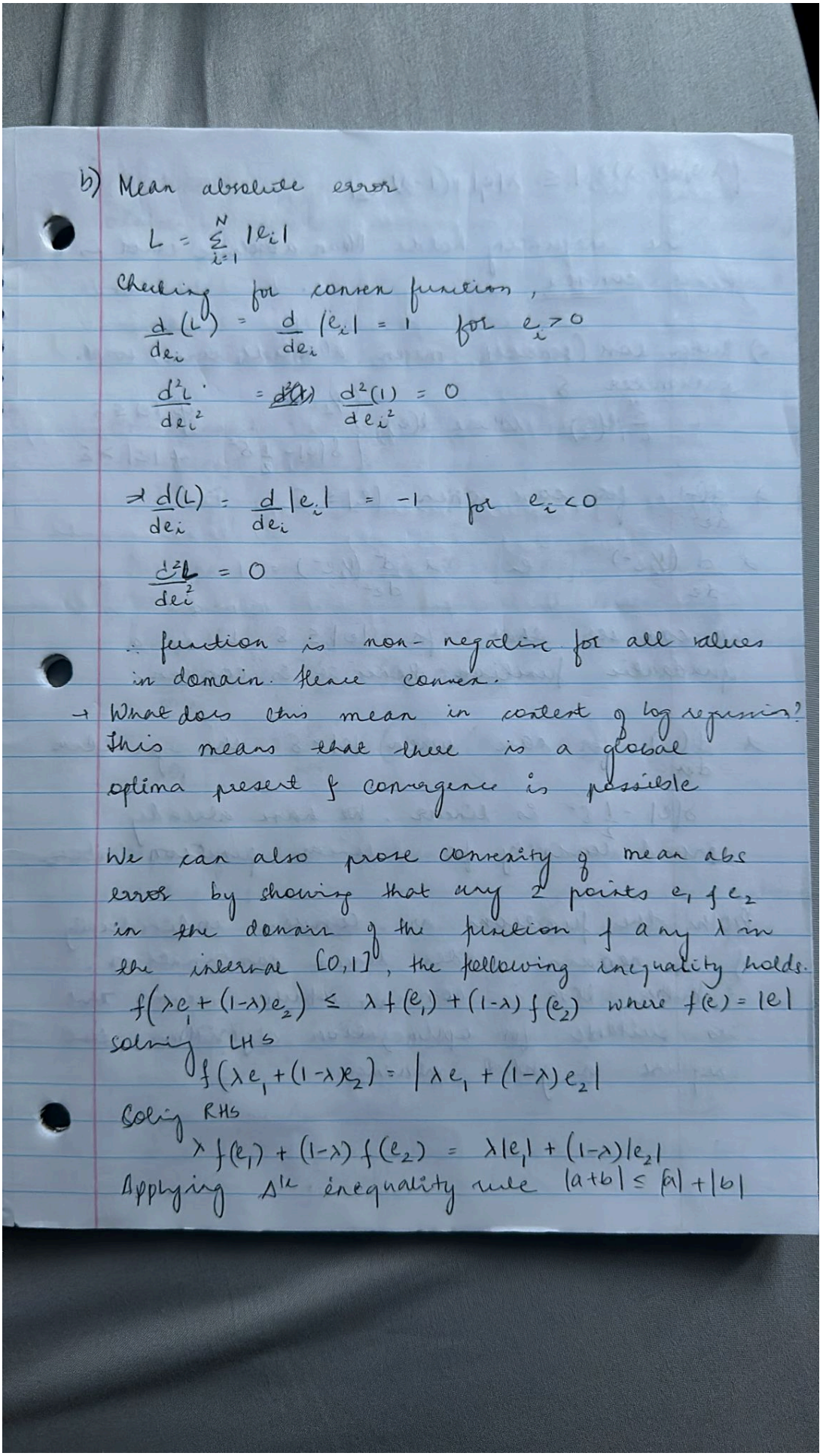
$$\therefore \frac{dL}{de_i} = 2e_i \quad \text{where } L = \sum_{i=1}^{N} e_i^2$$

$$\frac{d^2L}{de_i^2} = 2 \quad \text{non-negative value for all domains}$$

$\therefore$ Quadratic loss is convex

→ Since the second derivative is constant & positive, this confirms it is a convex function. This is good because they have a single global minima. This means that for linear regression, the optimization will converge to global minima.

Questions 1-2 images



b) Mean absolute error

$$L = \sum_{i=1}^{N} |e_i|$$

Checking for convex function,

$$\frac{d}{de_i}(L) = \frac{d}{de_i}|e_i| = 1 \quad \text{for } e_i > 0$$

$$\frac{d^2 L}{de_i^2} = \frac{d^2(1)}{de_i^2} = 0$$

$$\Rightarrow \frac{d(L)}{de_i} = \frac{d}{de_i}|e_i| = -1 \quad \text{for } e_i < 0$$

$$\frac{d^2 L}{de_i^2} = 0$$

∴ function is non-negative for all values in domain. Hence convex.

→ What does this mean in context of log regression?
This means that there is a global optima present & convergence is possible

We can also prove convexity of mean abs error by showing that any 2 points e_1 & e_2 in the domain of the function & any λ in the interval $[0,1]$, the following inequality holds.

$$f(\lambda e_1 + (1-\lambda) e_2) \leq \lambda f(e_1) + (1-\lambda) f(e_2) \quad \text{where } f(e) = |e|$$

Solving LHS

$$f(\lambda e_1 + (1-\lambda) e_2) = |\lambda e_1 + (1-\lambda) e_2|$$

Solving RHS

$$\lambda f(e_1) + (1-\lambda) f(e_2) = \lambda |e_1| + (1-\lambda)|e_2|$$

Applying Δ^{le} inequality rule $|a+b| \leq |a| + |b|$

Questions 1-2 images

$|\lambda e_1 + (1-\lambda) e_2| \leq \lambda|e_1| + (1-\lambda)|e_2|$

∴ The inequality holds. Mean absolute error is hence convex.

c) Huber loss (smooth mean absolute error) with parameter δ

$L = \sum_{i=1}^{N} l(e_i)$ where $l(e) = \begin{cases} \frac{1}{2}e^2 & \text{if } |e| \leq \delta \\ \delta|e| - \frac{1}{2}\delta^2 & \text{if } |e| > \delta \end{cases}$

⇒ $\frac{d(L)}{de^2}$ for case when $|e| \leq \delta$

⇒ $\frac{d}{de}(\frac{1}{2}e^2) = e$ ⇒ $\frac{d^2}{de^2}(\frac{1}{2}e^2) = 1$.

We can see that for $|e| \leq \delta$ L is a quadratic function. Hence it is convex.

⇒ $\frac{d(L)}{de}$ for case when $|e| > \delta$, the function $\delta|e| - \frac{1}{2}\delta^2$ is linear. We have already proved convexity for a linear function above.

Both the functions are convex individually & it remains under linear combination.

∴ Huber loss is convex, which means that its suitable for optimazation algorithms that require a global minima.

Questions 1-2 images

2) For linear regression $Y_i = \theta_0 + \theta_1 X_i + e_i$, $i = 1, \ldots N$ minimizing squared loss (e_i is random noise variable)

a) Calculate the gradient wrt parameter vector θ:
Given the linear regression model:

$$Y_i = \theta_0 + \theta_1 X_i + e_i$$

The squared loss func for linear regression is typically defined as sum of squared errors:

$$L = \sum_{i=1}^{N} (Y_i - (\theta_0 + \theta_1 X_i))^2$$

The parameter vector $\theta = [\theta_0 \; \theta_1]^T$
Calculating the gradient wrt θ vector by computing partial derivative wrt each param

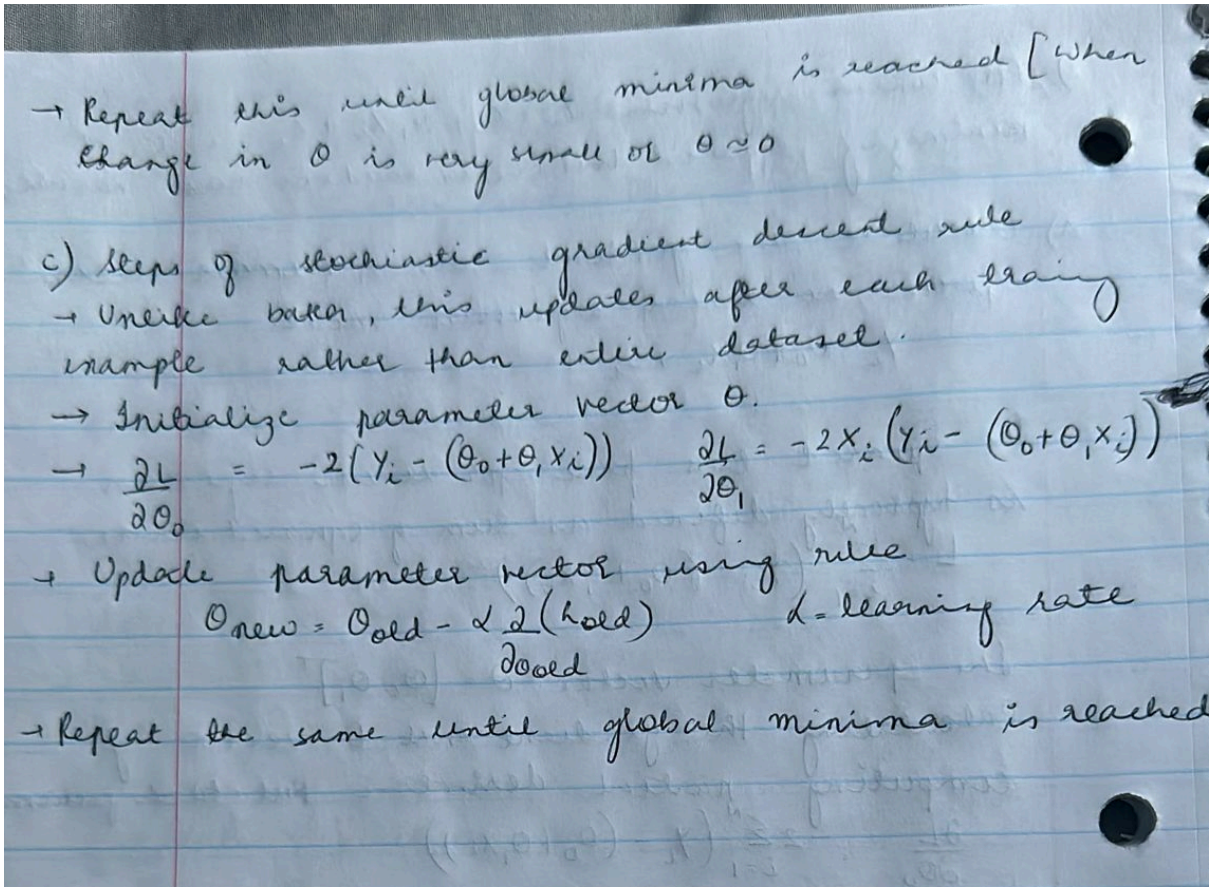
$$\frac{\partial L}{\partial \theta_0} = -2 \sum_{i=1}^{N} (Y_i - (\theta_0 + \theta_1 X_i))$$

$$\frac{\partial L}{\partial \theta_1} = -2 \sum_{i=1}^{N} X_i (Y_i - (\theta_0 + \theta_1 X_i))$$

b) Steps of batch gradient descent rule.
→ Initialize parameter vector θ to some initial values
→ Calculate the gradient [as shown in (a)] of the loss function wrt θ
→ Calculating gradient tells us if the curve/slope is decreasing/increasing. This helps us to know if we are reaching global minima.
→ Update parameters using gradient update rule

$$\theta_{new} = \theta_{old} - \alpha \frac{\partial}{\partial \theta_{old}} (L)_{old} \qquad \alpha = \text{learning rate}$$

Questions 1-2 images



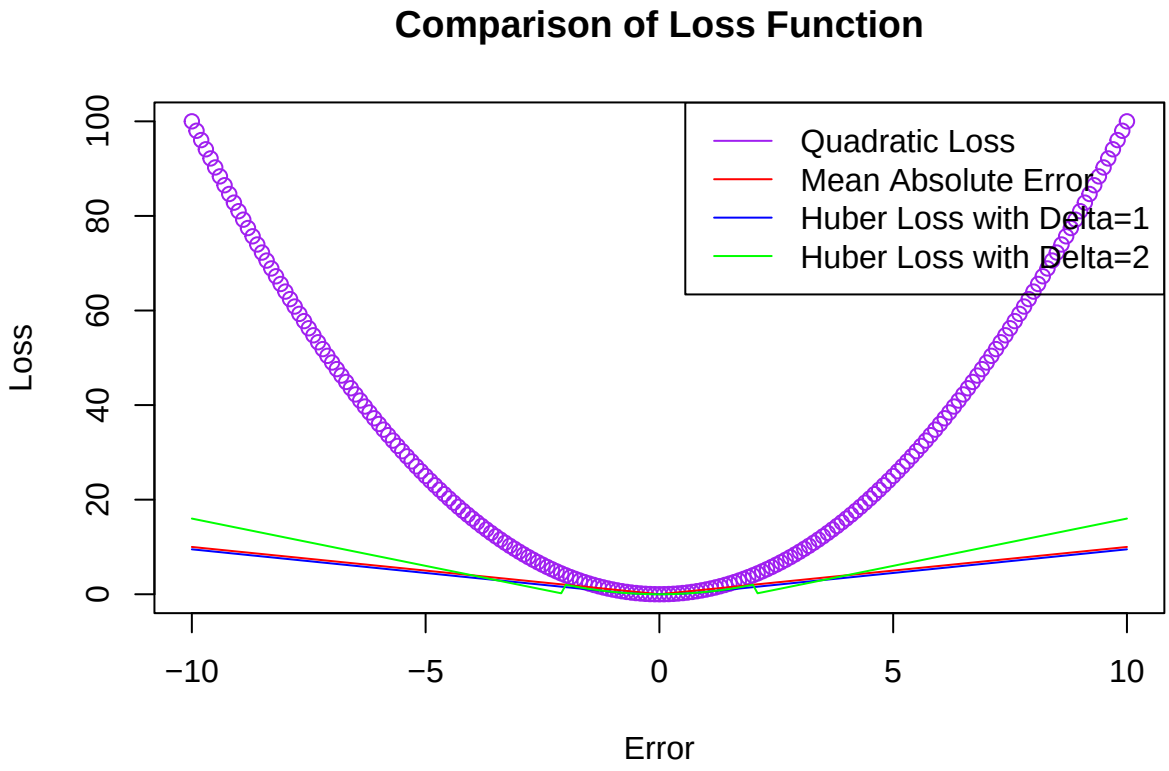
Assignment 02 R Notebook

Overlay graphs of the loss functions in question 1 for a range of (consider two different values of for Huber loss). Use the graph to discuss the relative advantages and disadvantages of these loss functions for linear regression

```
quadratic_loss <- function(err){
  result <- err^2
  return (result)
}

mean_abs_error <- function(err){
  result <- abs(err)
  return (result)
}

huber_loss <- function(err, delta){
  ifelse(abs(err) <= delta, 0.5 * err^2, ifelse(abs(err) > delta, (delta * (abs(err) - 0.5*delta^2)),NA
}
```

```
errs <- seq(-10,10,0.1)
```

```
plot(errs, sapply(errs,quadratic_loss), col = 'purple',xlab = 'Error',ylab = 'Loss',main = 'Comparison
lines(errs, sapply(errs,mean_abs_error), col = 'red')
lines(errs,sapply(errs, huber_loss,delta=1), col = 'blue')
lines(errs,sapply(errs, huber_loss,delta=2), col = 'green')
legend('topright', legend = c('Quadratic Loss', 'Mean Absolute Error', 'Huber Loss with Delta=1','Huber
```



Question 3b)

```r
quadratic_GD <- function(x,y, learningRate, epochs) {
  n <- length(y) #no of features
  m <- length(x) #training rows
  theta <- rep(0,m) #should this be n?

  for (err in 1:epochs){
    gradient <- 2/n * t(x) %*% (x%*% theta - y)
    theta <- theta - learningRate * gradient #Updating theta
  }
    return (theta)
}

MAE_GD <- function(x,y, learningRate, epochs){
  n <- length(y) #no of features
  m <- length(x) #training rows
  theta <- rep(0,m) #should this be n?

  for (err in 1:epochs){
  gradient <- 1/n * t(x) %*% sign(x%*% theta - y)
  theta <- theta - learningRate * gradient #Updating theta. This remains the same throughout.
  }
  return (theta)
}

huber_GD <- function(x,y, learningRate, epochs, delta){
```

```
  n <- length(y) #no of features
  m <- length(x) #training rows
  theta <- rep(0,m) #should this be n?

  for (err in 1:epochs){
  error <- x %*% theta - y
  gradient <- (1/n) * t(x) %*% ifelse(abs(error) <= delta, error, delta * sign(error))
  theta <- theta - learningRate * gradient #Updating theta. This remains the same throughout.
  }
  return (theta)
}
```

Question 3c)

```
quadratic_sto <- function(x,y,learningRate, epochs){
   n <- length(y) #no of features
  m <- length(x) #training rows
  theta <- rep(0,m) #should this be n?

  for (err in 1:epochs){
    for(i in 1:n){
      random <- sample(1:n,1)
      xi <- x[random,]
      yi <- y[random]
      gradient <- 2 * t(xi) %*% (xi %*% theta - yi)
      theta <- theta - learningRate * gradient
    }
  }
  return (theta)
}

MAE_sto <- function(x,y,learningRate, epochs){
   n <- length(y) #no of features
  m <- length(x) #training rows
  theta <- rep(0,m) #should this be n?

  for (err in 1:epochs){
    for(i in 1:n){
      random <- sample(1:n,1)
      xi <- x[random,]
      yi <- y[random]
      gradient <- 2 * t(xi) %*% sign(xi %*% theta - yi)
      theta <- theta - learningRate * gradient
    }
  }
  return (theta)
}

huber_sto <- function(x,y,learningRate, epochs){
   n <- length(y) #no of features
  m <- length(x) #training rows
  theta <- rep(0,m) #should this be n?
```

```
  for (err in 1:epochs){
    for(i in 1:n){
      random <- sample(1:n,1)
      xi <- x[random,]
      yi <- y[random]
      gradient <- (1/n) * t(x) %*% ifelse(abs(error) <= delta, error, delta * sign(error))
      theta <- theta - learningRate * gradient
    }
  }
  return (theta)
}
```

Question 4a)

```
#fabricate data
set.seed(0)
N <- 50
x <- runif(N, -2, 2)
epsilon <- rnorm(N, 0, 2)
y <- 3 + 2 * x + epsilon

x <- cbind(1,x)
analytical <- solve(t(x) %*% x) %*% t(x) %*% y #Analytical Solution


batch_GD <- function(x, y, alpha = 0.01, iterations = 1000){
  m <- nrow(x)
  theta <- matrix(0, ncol(x), 1)
  for(i in 1:iterations){
    gradient <- t(x) %*% (x %*% theta - y) / m
    theta <- theta - alpha * gradient
  }
  return(theta)
}

batch <- batch_GD(x, y)

#SGD
sto_GD <- function(X, y, learningRate = 0.01, iterations = 50 * N) {
  m <- nrow(X)
  theta <- matrix(0, ncol(X), 1) # Initial parameters
  for (i in 1:iterations) {
    idx <- sample(1:m, 1)
    X_i <- X[idx, , drop = FALSE] # Ensure X_i is always a matrix
    y_i <- y[idx]
    gradient <- t(X_i) %*% (X_i %*% theta - y_i)
    theta <- theta - learningRate * gradient
  }
  return(theta)
}

sto <- sto_GD(x, y)
```

```r
print(analytical)
```

```
##       [,1]
##   2.931669
## x 1.656737
```

```r
print(batch)
```

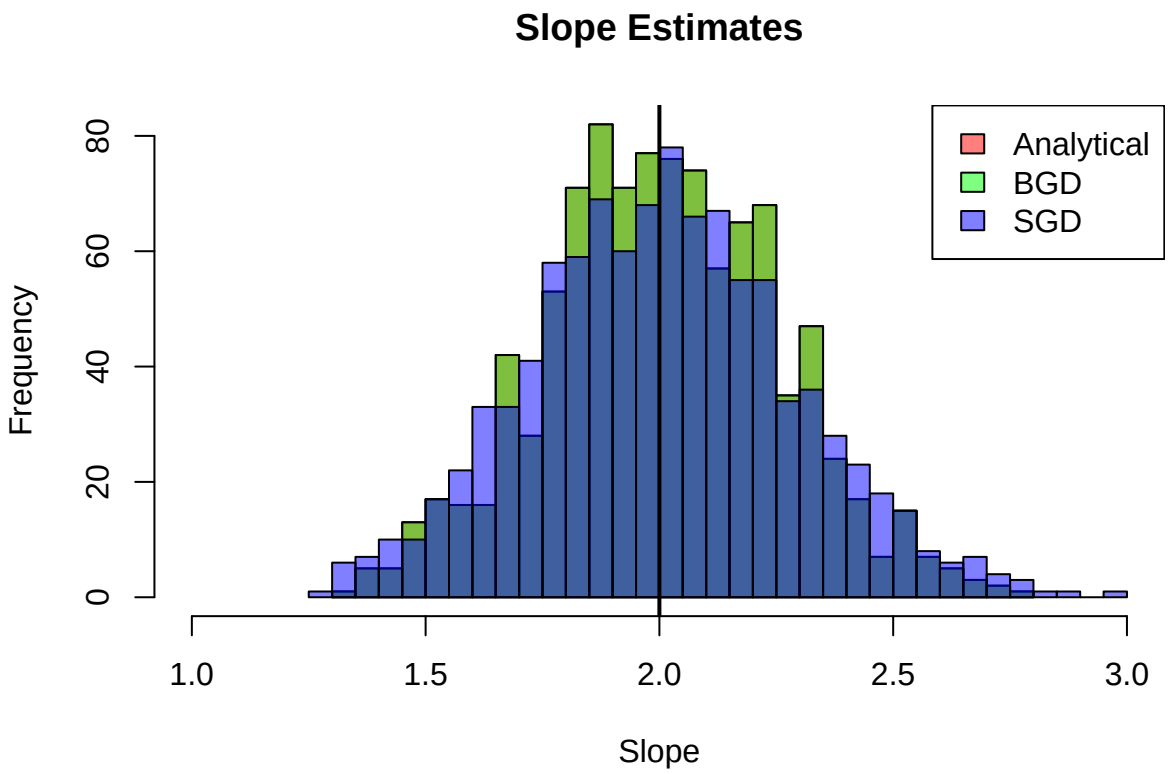
```
##       [,1]
##   2.931514
## x 1.656808
```

```r
print(sto)
```

```
##       [,1]
##   2.840583
## x 1.693194
```

4b)

```r
fit <- function() {
  x <- runif(N, -2, 2)
  epsilon <- rnorm(N, 0, sqrt(4))
  y <- 3 + 2*x + epsilon
  X <- cbind(1, x)

  # Analytical Soln
  analytical <- solve(t(X) %*% X) %*% t(X) %*% y

  # Batch Gradient Descent
  batch_GD <- batch_GD(X, y)

  # Stochastic Gradient Descent
  sto_GD <- sto_GD(X, y)

  c(analytical[2], batch_GD[2], sto_GD[2])
}

set.seed(0)
results <- replicate(1000, fit())

# Plot results
hist(results[1,], breaks = 30, col = rgb(1,0,0,0.5), xlim = c(1,3), main = "Slope Estimates", xlab = "S
hist(results[2,], breaks = 30, col = rgb(0,1,0,0.5), add = TRUE)
hist(results[3,], breaks = 30, col = rgb(0,0,1,0.5), add = TRUE)
abline(v = 2, col = "black", lwd = 2) # True value
legend("topright", legend = c("Analytical", "BGD", "SGD"), fill = c(rgb(1,0,0,0.5), rgb(0,1,0,0.5), rgb
```



Depending on the learning rate and number of epochs used, batch gradient descent and stochastic gradient descent may produce somewhat different estimates, even if the analytical solution offers objective and accurate estimates. SGD may offer improved generalization and faster convergence, however the estimates can vary somewhat. Batch gradient descent could yield more reliable estimates, but for best results, hyperparameters might need to be adjusted.

4c)

```
set.seed(0)
N <- 50
x <- runif(N, -2, 2)
epsilon <- rnorm(N, 0, sqrt(4))
y <- 3 + 2*x + epsilon
X <- cbind(1, x)

analytical <- solve(t(X) %*% X) %*% t(X) %*% y

gradient_l1_loss <- function(Y, X, O) { # MAE with Batch Gradient Descent
  gradient <- numeric(length(O))
  Y_hat <- O[1] + O[2]*X
  gradient[1] <- sum(ifelse(Y > Y_hat, 1, ifelse(Y < Y_hat, -1, 0)))
  gradient[2] <- sum(ifelse(Y > Y_hat, X, ifelse(Y < Y_hat, -X, 0)))
  return(gradient)
}

l1_gd <- function(Y, X, O, alpha = 0.01, num_iterations = 500) {
  for (i in 1:num_iterations) {
```

```
    gradient <- gradient_l1_loss(Y, X, O)
    O <- O - alpha * gradient
  }
  return(O)
}

b_mae = c()

b_hat_mae <- l1_gd(X, y, O = c(0.5,0.5))

gradient_huber_loss <- function(Y, X, O, delta){
  # Huber Loss with Batch Gradient Descent
  gradient <- numeric(length(O))
  Y_hat <- O[1] + O[2]*X
  is_small <- abs(Y - Y_hat) <= delta
  gradient[1] <- -X * ifelse(is_small, Y-Y_hat, delta * sign(Y-Y_hat))
  gradient[2] <- -1 * ifelse(is_small, Y-Y_hat, delta * sign(Y-Y_hat))
  return(gradient)
}

h_gd <- function(Y, X, O, alpha = 0.01, num_iterations = 500, delta = 1.5) {
  for (i in 1:num_iterations) {
    gradient <- gradient_huber_loss(Y, X, O, delta)
    O <- O - alpha * gradient
  }
  return(O)
}

suppressWarnings(b_hat_huber <- h_gd(X, y, O = c(0.5, 0.5), delta = 1.5))

print(analytical)
```

```
##       [,1]
##   2.931669
## x 1.656737
```

```
print(b_hat_mae)
```

```
## [1]  360.780 1731.747
```

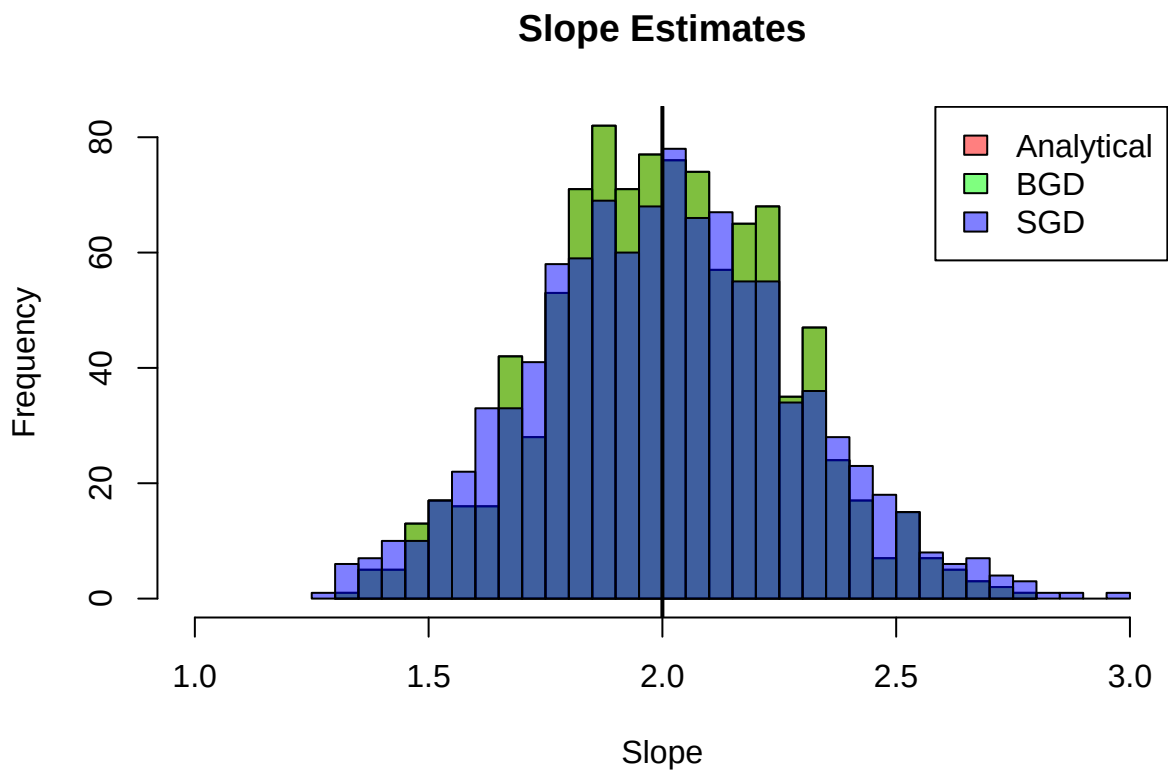
```
print(b_hat_huber)
```

```
## [1] -1.0451984  0.2848151
```

4d)

```
choice_loss <- function() {
  x <- runif(N, -2, 2)
  epsilon <- rnorm(N, 0, sqrt(4))
  y <- 3 + 2*x + epsilon
  X <- cbind(1, x)
```

```r
  beta_hat_analytical <- solve(t(X) %*% X) %*% t(X) %*% y

  beta_hat_mae <- l1_gd(X, y, 0 = c(0.5, 0.5)) #MAE

  suppressWarnings(beta_hat_huber <- h_gd(X, y, 0 = c(0.5, 0.5), delta = 1.5)) #Huber

  c(beta_hat_analytical[2], beta_hat_mae[2], beta_hat_huber[2])
}

set.seed(0)
results_part_c <- replicate(1000, choice_loss())

# Plotting hists
hist(results[1,], breaks = 30, col = rgb(1,0,0,0.5), xlim = c(1,3), main = "Slope Estimates", xlab = "Sl
hist(results[2,], breaks = 30, col = rgb(0,1,0,0.5), add = TRUE)
hist(results[3,], breaks = 30, col = rgb(0,0,1,0.5), add = TRUE)
abline(v = 2, col = "black", lwd = 2) # True value
legend("topright", legend = c("Analytical", "BGD", "SGD"), fill = c(rgb(1,0,0,0.5), rgb(0,1,0,0.5), rgb
```



The loss function used influences the slope parameter estimates in linear regression by changing the trade-off between resilience to outliers and optimization stability. Squared loss is sensitive to outliers but produces a convex optimization issue, whereas absolute loss is resilient to outliers but may result in non-convex optimization. Huber loss strikes a balance between these two extremes, providing resilience to outliers while retaining convergence features.

4e)

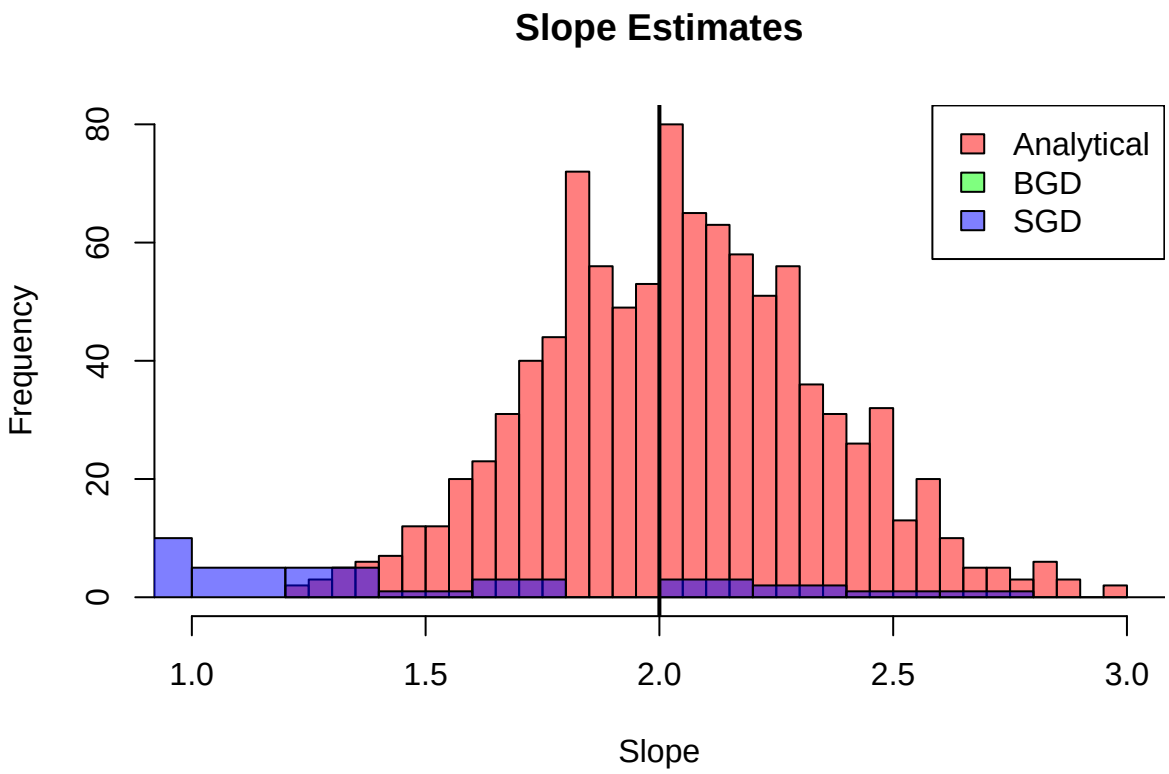
```r
outliers <- function(out_prop = 0.1) {
  x <- runif(N, -2, 2)
  epsilon <- rnorm(N, 0, sqrt(4))
  y <- 3 + 2*x + epsilon
  X <- cbind(1, x)

  #Outliers
  num_outliers <- ceiling(N * out_prop)
  outlier_indices <- sample(1:N, num_outliers, replace = FALSE)

  # Apply outlier effect
  for (i in outlier_indices) {
    # Randomly decide to add or subtract the magnitude
    if (runif(1) < 0.5) {
      y[i] <- 2 * y[i]
    } else {
      y[i] <- y[i] / 2
    }
  }

  #Analytical
  analytical <- solve(t(X) %*% X) %*% t(X) %*% y

  mae <- l1_gd(X, y, O = c(0.5, 0.5))  # MAE

  suppressWarnings(huber <- h_gd(X, y, O = c(0.5, 0.5), delta = 1.5)) #huber

  return(c(analytical[2], mae[2], huber[2]))
}

set.seed(0)
ans <- outliers()
print(ans)
```

```
## [1]    1.7316679 1832.7799157    0.2674075
```

4f)

```r
outliers <- function(out_prop = 0.1) {
  x <- runif(N, -2, 2)
  epsilon <- rnorm(N, 0, sqrt(4))
  y <- 3 + 2*x + epsilon
  X <- cbind(1, x)

  #Outliers
  num_outliers <- ceiling(N * out_prop)
  outlier_indices <- sample(1:N, num_outliers, replace = FALSE)

  # Apply outlier effect
  for (i in outlier_indices) {
    # Randomly decide to add or subtract the magnitude
    if (runif(1) < 0.5) {
      y[i] <- 2 * y[i]
```

```
    } else {
      y[i] <- y[i] / 2
    }
  }

  # Squared Loss Analytical
  beta_hat_analytical <- solve(t(X) %*% X) %*% t(X) %*% y

  # MAE with Batch Gradient Descent
  beta_hat_mae <- l1_gd(X, y, O = c(0.5, 0.5))

  # Huber Loss with Batch Gradient Descent
  suppressWarnings(beta_hat_huber <- h_gd(X, y, O = c(0.5, 0.5), delta = 1.5))

  return(c(beta_hat_analytical[2], beta_hat_mae[2], beta_hat_huber[2]))
}

set.seed(0)
results <- replicate(1000, outliers())

# Plot results
hist(results[1,], breaks = 30, col = rgb(1,0,0,0.5), xlim = c(1,3), main = "Slope Estimates", xlab = "S
hist(results[2,], breaks = 30, col = rgb(0,1,0,0.5), add = TRUE)
hist(results[3,], breaks = 30, col = rgb(0,0,1,0.5), add = TRUE)
abline(v = 2, col = "black", lwd = 2) # True value
legend("topright", legend = c("Analytical", "BGD", "SGD"), fill = c(rgb(1,0,0,0.5), rgb(0,1,0,0.5), rgb
```



Loss function selection in the presence of outliers influences the estimations of the slope parameter. Compared to mean absolute error and Huber loss, which handle outliers more leniently, squared loss is more resilient to outliers and may provide estimates that are skewed. When working with data that contains outliers, huber loss is a common option because it offers a flexible balance between the efficiency of squared loss and the robustness of absolute loss.