DS5220 Homework-2

Ruchik Jani (NUID - 002825482)

23rd February, 2024

Question-3

a

Based on the graphs generated below, let's discuss the relative advantages and disadvantages of these loss functions for linear regression:

Quadratic loss (mean squared error or MSE):

Advantages: From the graph, we can see that it is differentiable everywhere, this allows us to calculate the slop at each point which helps in convergence when using gradient descent optimization. It also has exactly one local and one global minima, which are one and the same. This property is also useful in gradient based optimization since the convergence doesn't get stuck at a local minima, and as a result, it converges faster.

Disadvantages: From the graph, we can see that the loss function is heavily penalized for larger errors. This makes it sensitive to outliers and disrupts our best-fit line leading to inaccurate predictions.

Mean absolute error (MAE)::

Advantages: From the graph, we can see that the value of loss function is not very high for higher errors. This makes it robust to outliers since it does not heavily penalize large errors compared to quadratic loss. It is also pretty straightforward to interpret.

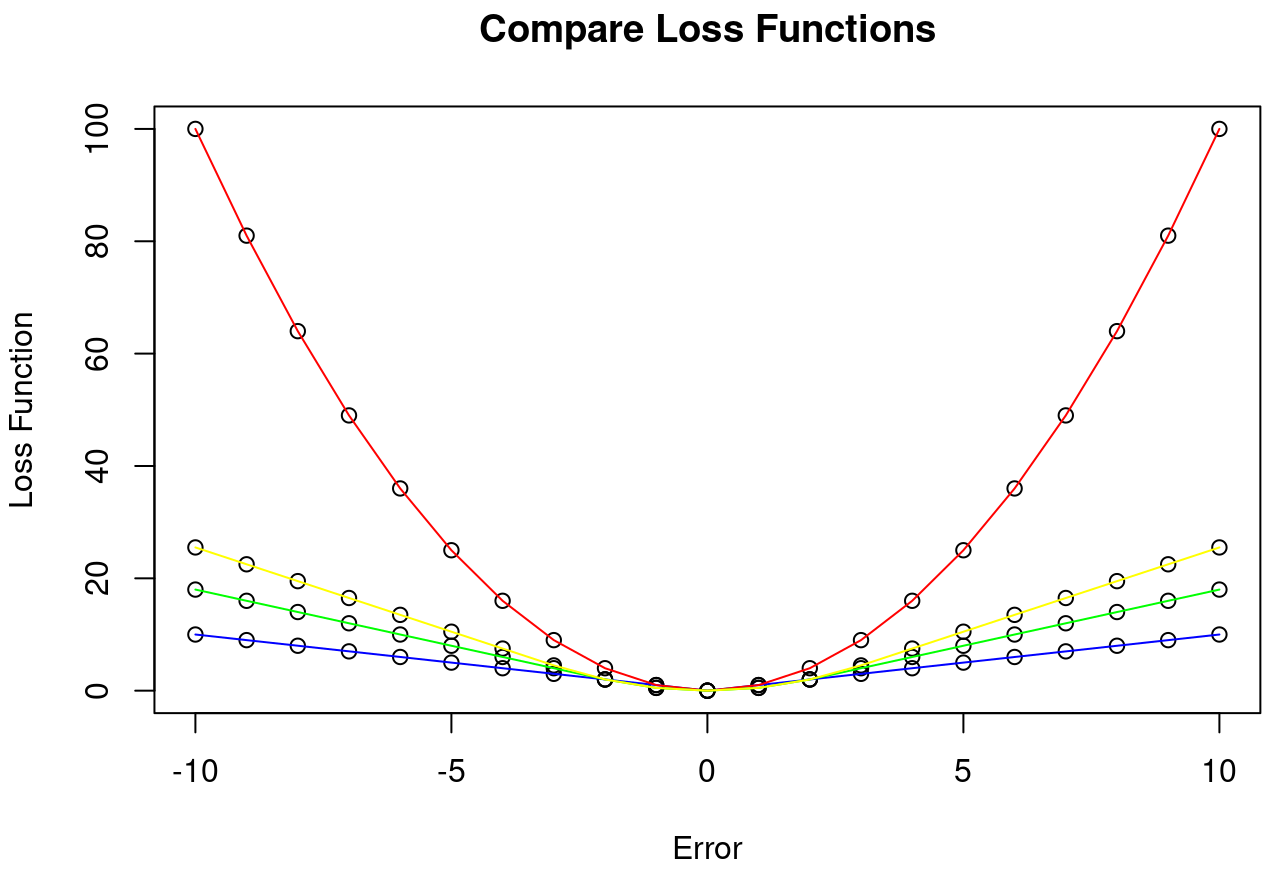
Disadvantages: From the graph, we can see that this function is non-differentiable at e=0, which can complicate optimization, especially when using gradient-based methods. As a result, the convergence will take more time.

Huber loss (smooth mean absolute error or SMAE)::

Advantages: From the graph, we can see that this function is differentiable everywhere and it doesn't penalize large errors as heavily as the quadratic loss function. This makes it robust to outliers by combining the best properties of both the MSE and MAE functions. It also has a smooth transition from quadratic form for small errors to the linear form for large errors, which is controlled by changing the values of parameter δ. Here, we have plotted for δ equal to 2 and 3. The loss function shows a more pronounced transition for δ = 2 as compared to δ = 3. For δ = 3, the loss function imposes a quadratic penalty for a wider range of errors before changing to a linear penalty as compared to δ = 2.

Disadvantages: From the graph, we can see that the choice of δ can be subjective and may require tuning. Unlike the quadratic loss function, which directly penalizes the square of errors, the Huber loss function's interpretation is less intuitive, especially when different values of δ are used.

```
# implement huber function
huber <- function(e, sig) {
quad <- 0.5*e^2
linr <- sig*abs(e) - 0.5*sig^2
return(if(abs(e) <= sig) quad else linr)
}

# generate range of e for plotting the losses
e_list <- -10:10
x <- c(e_list, e_list, e_list, e_list)
y <- c(abs(e_list), e_list^2, sapply(e_list, huber, 2), sapply(e_list, huber, 3))

plot(x, y, main="Compare Loss Functions", xlab="Error", ylab="Loss Function")

lines(e_list, abs(e_list), col="blue") # Absolute loss
lines(e_list, e_list^2, col="red") # Square loss
lines(e_list, sapply(e_list, huber, 2), col="green") # Huber-2
lines(e_list, sapply(e_list, huber, 3), col="yellow") # Huber-3
```



b & c

As per the hints given in the assignment, the implementation of gradient descent optimizer will be divided into 3 parts for code reusability:

• gradient step for each loss • optimizing (batch and stochastic) • full gradient descent algorithm: combine a gradient descent step and optimizing

When we implement the batch and stochastic optimizer, each optimizer shall take the initialized parameters and a gradient descent step as inputs. The optimizer will use the step to update parameters for each iteration.

```
# gradient update step for Huber loss
gradient_descent_huber <- function(a, b, X, Y, sig, alpha=0.01) {
  E <- a + b*X - Y
 sign <- (2*(E>0)-1)

 a_step <- sum((abs(E) <= sig)*(a + b*X - Y) + (abs(E) > sig)*sign)
 b_step <- sum((abs(E) <= sig)*(a + b*X - Y)*X + (abs(E) > sig)*sign*X)

 return(c(
a - alpha*a_step,
b - alpha*b_step
))
}

# gradient update step for Squared loss
gradient_descent_squared <- function(a, b, X, Y, alpha=0.01) {
E <- a + b*X - Y
a_step <- sum(2*(a + b*X - Y))
b_step <- sum(2*(a + b*X - Y)*X)

return(c(
a - alpha*a_step,
b - alpha*b_step
))
}

# gradient update step for Absolute loss
gradient_descent_absolute <- function(a, b, X, Y, alpha=0.01) {
E <- a + b*X - Y
sign <- (2*(E>0)-1)

a_step <- sum(sign)
b_step <- sum(sign*X)

return(c(
a - alpha*a_step,
b - alpha*b_step
))
}



# batch optimizer
optimize <- function(a, b, step) {
for  (iter  in  1:1000)  {
  pars <- step(a, b)
if (max(abs(c(a, b) - pars)) < 10^-4) {
return(pars)
}
a <- pars[1]
b <- pars[2]
}
return(c(a, b))
```

```
}

# stochastic optimizer
optimize_stochastic <- function(a, b, X, Y, step) {
for (iter in 1:1000) {
for (i in 1:length(X)) {
pars <- step(a, b, X[i], Y[i])
if (max(abs(c(a, b) - pars)) < 10^-4) {
  return(pars)
}

      a <- pars[1]
      b <- pars[2]
    }
  }

  return(c(a, b))
}




# analytical solution for OLS
obtain_ols_analytical <- function(X, Y) {
b <- sum((X-mean(X))*(Y-mean(Y)))/sum((X-mean(X))^2)
a <- mean(Y) - b*mean(X)

return(c(a,b))
}

# batch gradient descent for OLS
obtain_ols_gradient_descent <- function(X, Y) {
pars <- optimize(0, 0, step=function(a, b) gradient_descent_squared(a, b, X, Y))

return(pars)
}

# stochastic gradient descent for OLS
obtain_ols_stochastic_gradient_descent <- function(X, Y) {
pars <- optimize_stochastic(0, 0, X, Y, step=function(a, b, x, y) gradient_descent_squar
ed(a, b, x, y))

return(pars)
}

# batch gradient descent for linear regression with absolute loss
obtain_absolute_gradient_descent <- function(X, Y) {
pars <- optimize(0, 0, step=function(a, b) gradient_descent_absolute(a, b, X, Y))

return(pars)
```

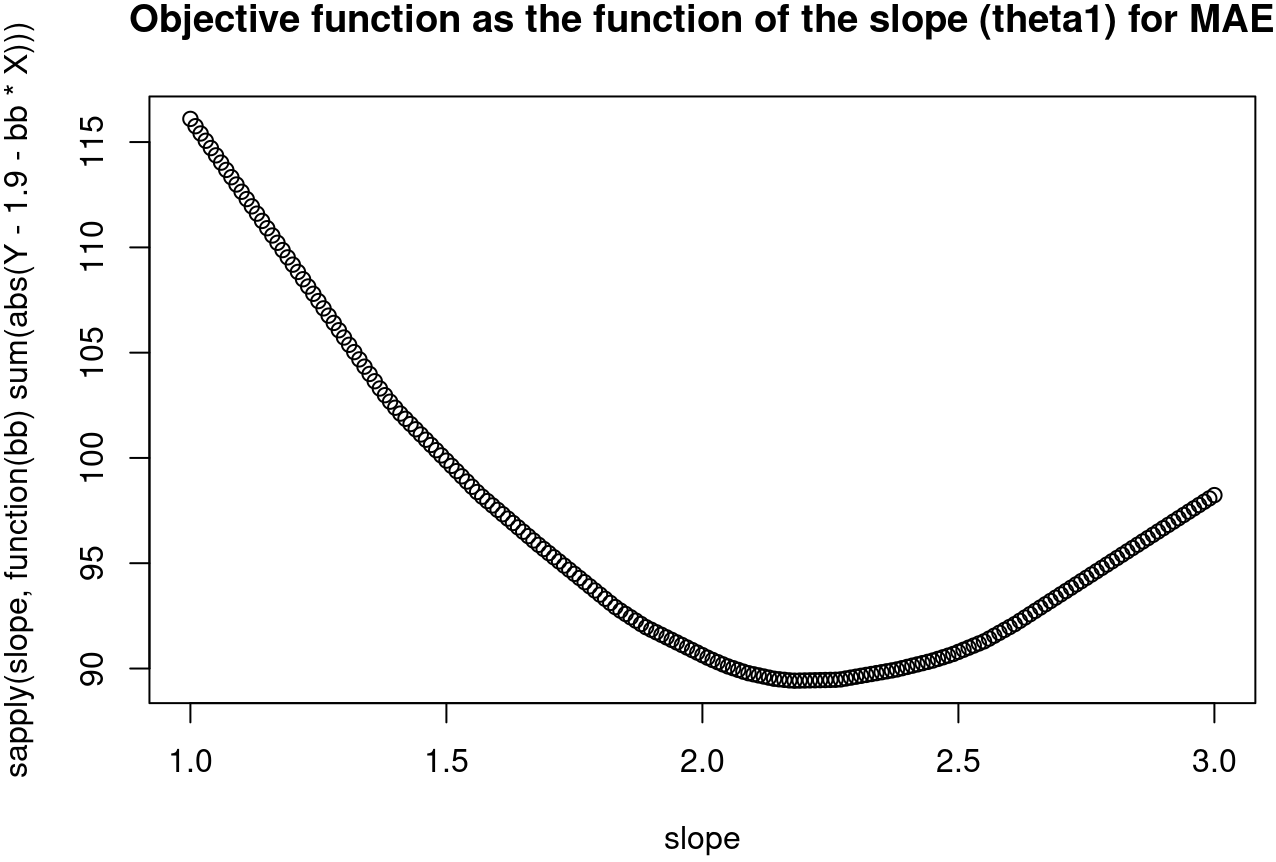
```
}

# batch gradient descent for linear regression with Huber loss
obtain_huber_gradient_descent <- function(X, Y, sig) {

  pars <- optimize(0, 0, step=function(a, b) gradient_descent_huber(a, b, X, Y, sig = si
g))

  return(pars)
}
```

Question-4

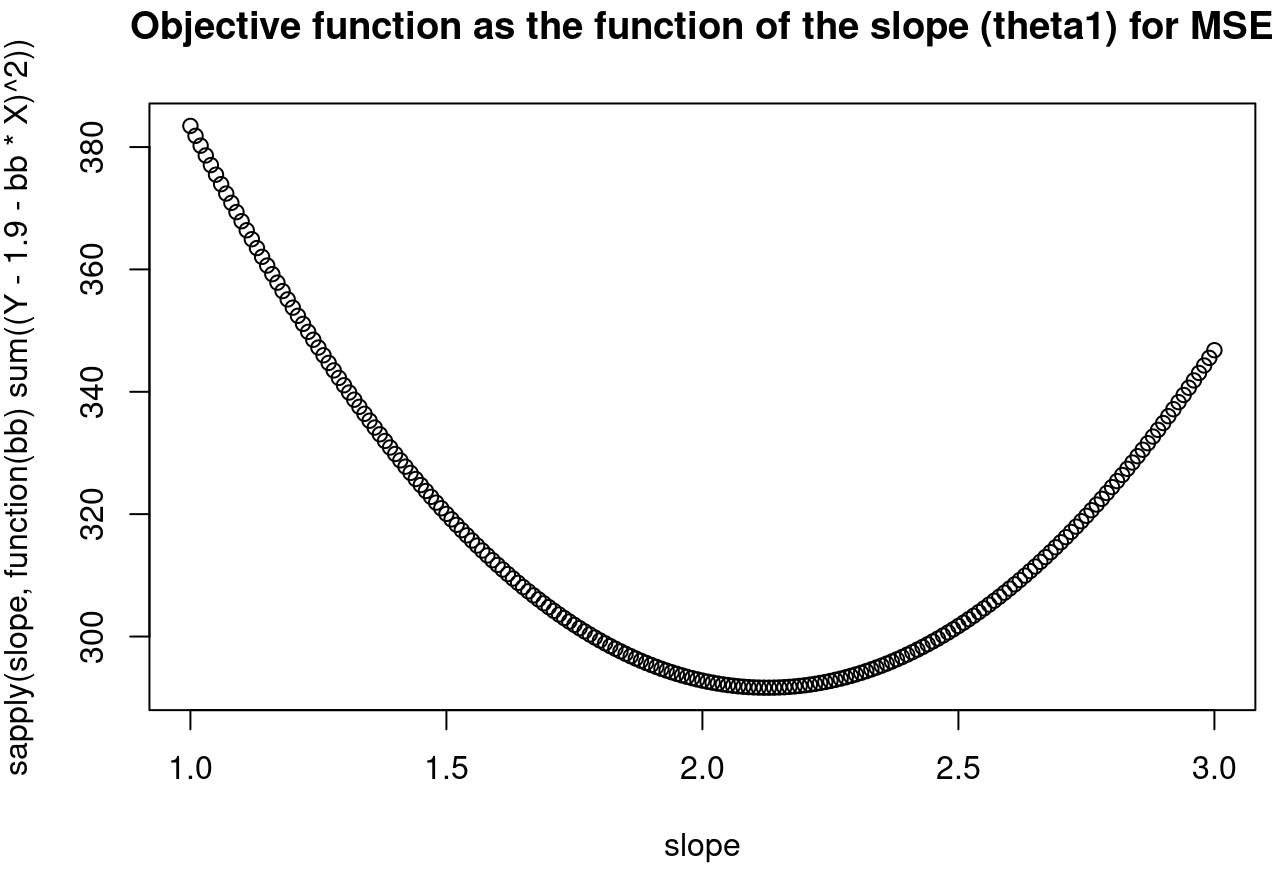
a & c

This code simulates the data generator. We have then plotted the objective function as a function of the slope (Theta-1) of the loss function. From the below graphs, it is evident that the curves are convex. The optimization will always occur since the functions are convex, and we will be able to optimize the objective fucntion. We then calculate the parameter estimates.

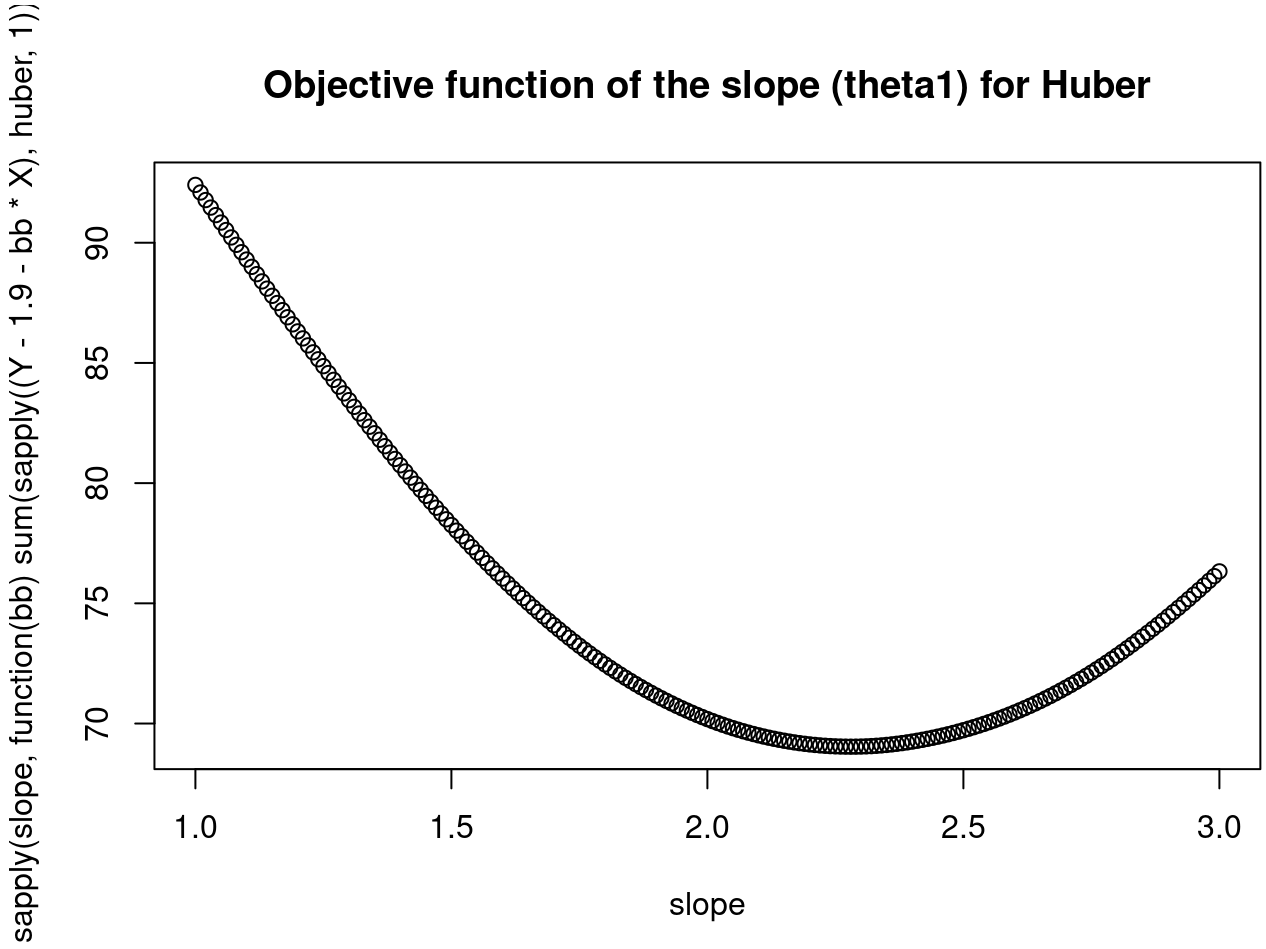
```
data_simulator <- function() {
X <- runif(50, -2, 2)
Y <- 3 + 2*X + rnorm(50, 0, 2)
 return(list(X, Y))
}

d <- data_simulator()
X <- d[[1]]
Y <- d[[2]]

slope <- (100:300)/100
plot(slope, sapply(slope, function(bb) sum(abs(Y-1.9-bb*X)) ), main="Objective function
as the function of the slope (theta1) for MAE")
```




```
plot(slope, sapply(slope, function(bb) sum((Y-1.9-bb*X)^2) ), main="Objective function a
s the function of the slope (theta1) for MSE")
```




```
plot(slope, sapply(slope, function(bb) sum(sapply((Y-1.9-bb*X), huber, 1)) ), main="Obje
ctive function of the slope (theta1) for Huber")
```




```
obtain_ols_analytical(X, Y)
```

```
## [1] 2.951999 2.070155
```

```
obtain_ols_gradient_descent(X, Y)
```

```
## [1] 2.952003 2.070177
```

```
obtain_ols_stochastic_gradient_descent(X, Y)
```

```
## [1] 2.926098 2.047281
```

```
obtain_absolute_gradient_descent(X, Y)
```

```
## [1] 2.420000 2.511272
```

```
obtain_huber_gradient_descent(X, Y, sig=4)
```

```
## [1] 2.760804 2.113374
```

b & d

This code derives 1000 estimates for 1000 runs of the simulator for the given optimization algorithm. After obtaining the simulated fits, we plot the histogram of the slopes of all algorithms. From the histograms, we can seee that the output of gradient descent and analytical solution is almost the same. The Huber loss function provides a precise and stable result by combining the properties of MAE and MSE loss fucntions.

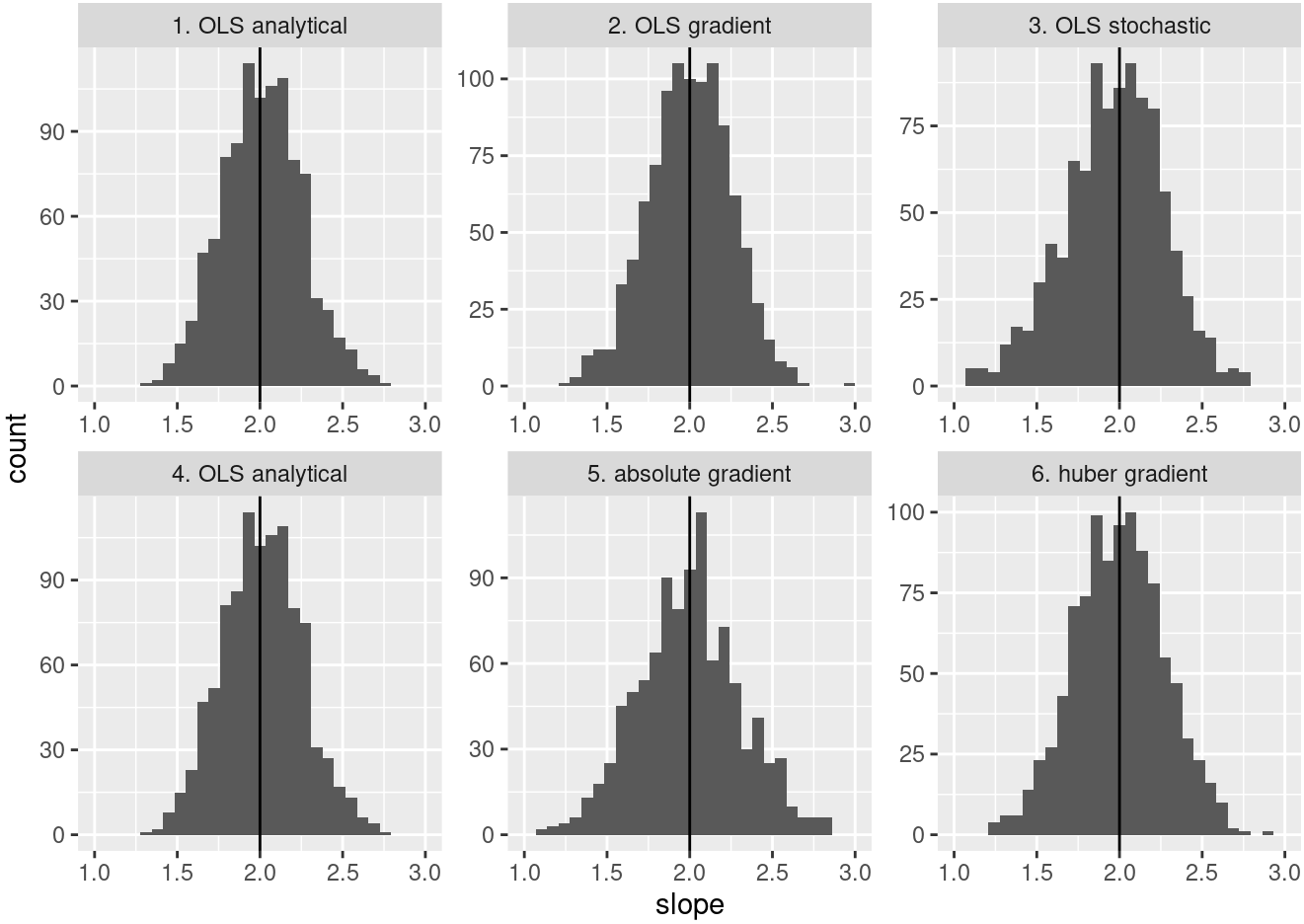
```
estimates_simulated <- function(optimizer, simulator) {
# optimizer: the algorithm that obtain the fit
# simulator: this generates the data for each simulation

res <-data.frame(lapply(1:1000, function(i) {
d <- data_simulator()
X <- d[[1]]
Y <- d[[2]]

return(optimizer(X, Y))
}))

names(res) <- 1:1000

return(data.frame(t(res)))
}

estimates_sim_ols_ana <- estimates_simulated(obtain_ols_analytical, data_simulator)
estimates_sim_ols_grad <- estimates_simulated(obtain_ols_gradient_descent, data_simulato
r)
estimates_sim_ols_sto <- estimates_simulated(obtain_ols_stochastic_gradient_descent, dat
a_simulator)

estimates_sim_abs <- estimates_simulated(obtain_absolute_gradient_descent, data_simulato
r)
estimates_sim_huber <- estimates_simulated(function(X, Y) obtain_huber_gradient_descent
(X, Y, sig=4), data_simulator())

library(ggplot2)


dp <- data.frame(
slope=c(
estimates_sim_ols_ana[,2],
estimates_sim_ols_grad[,2],
estimates_sim_ols_sto[,2],
estimates_sim_ols_ana[,2],
estimates_sim_abs[,2],
estimates_sim_huber[,2]
),

optimizer=c(
rep("1. OLS analytical", 1000),
rep("2. OLS gradient", 1000),
rep("3. OLS stochastic", 1000),
rep("4. OLS analytical", 1000),
rep("5. absolute gradient", 1000),
rep("6. huber gradient", 1000)
))
```

```
ggplot(data=dp, mapping=aes(x=slope)) + geom_histogram() + geom_vline(xintercept=2) + xl
im(c(1, 3)) + facet_wrap(~optimizer, scales="free")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 26 rows containing non-finite values (`stat_bin()`).
```

```
## Warning: Removed 6 rows containing missing values (`geom_bar()`).
```



e & f

From the below histograms, it is evident that addition of outliers causes an increase in the width of slope estimate histograms. Outliers decrease the fitting accuracy. With its robustness to outliers and stability, Huber loss function is the best choice for linear regression

```
data_outliers_simulator <- function() {
d <- data_simulator()
Y <- d[[2]]
outliers_selected <- sample(0:1, 50, replace=T, prob=c(0.9,0.1))
increase_selected <- sample(0:1, 50, replace=T, prob=c(0.5,0.5))
YY <- Y + 2*outliers_selected*(increase_selected - (1-increase_selected))*Y

d[[2]] <- YY

return(d)
}

d <- data_outliers_simulator()
X <- d[[1]]
YY <- d[[2]]

obtain_ols_analytical(X, YY)
```

```
## [1] 2.269637 1.005644
```

```
obtain_absolute_gradient_descent(X, YY)
```

```
## [1] 2.200000 1.452037
```

```
obtain_huber_gradient_descent(X, YY, sig=4)
```

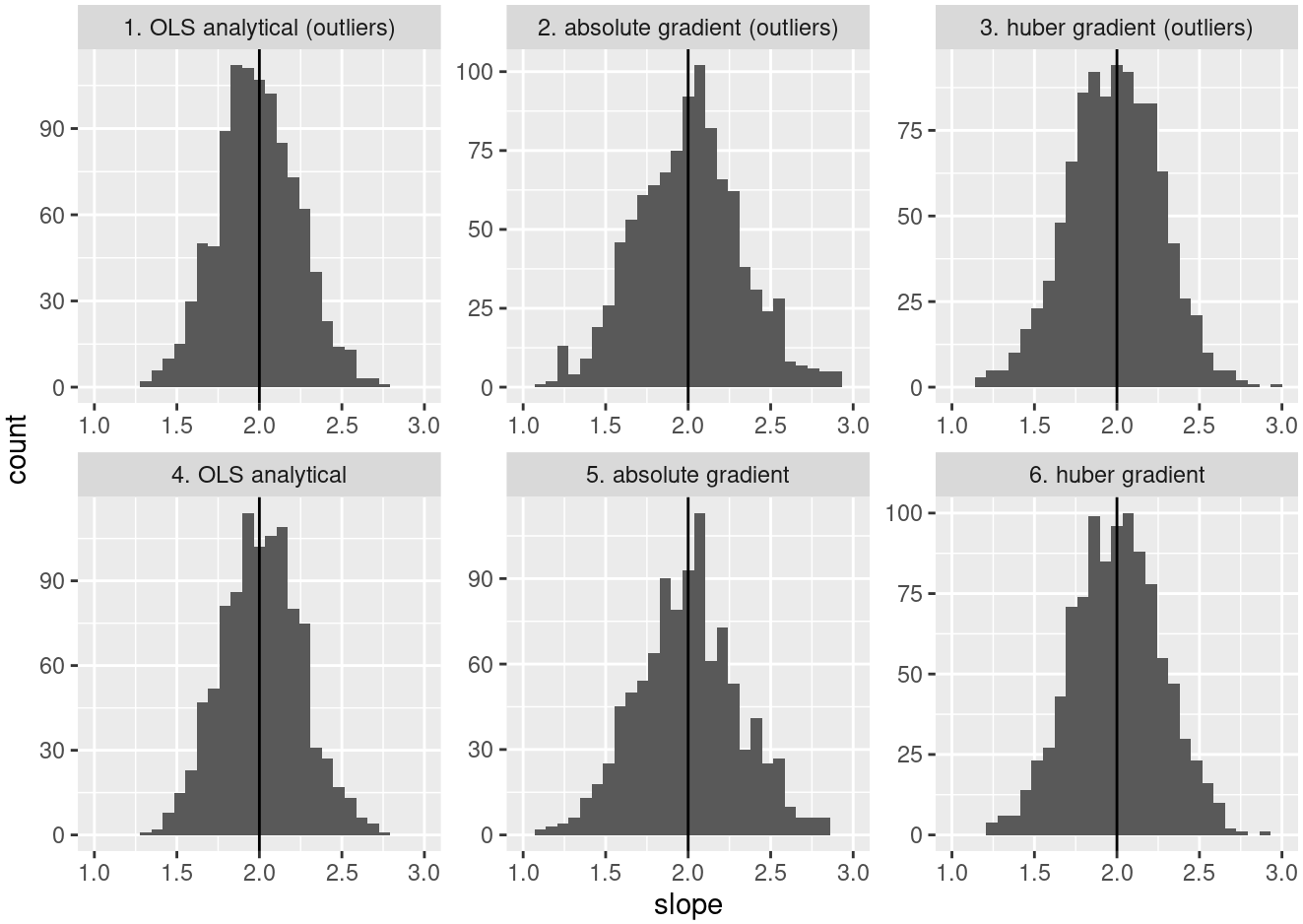
```
## [1] 2.421214 1.282564
```

```
estimates_sim_outl_ols_ana <- estimates_simulated(obtain_ols_analytical, data_outliers_s
imulator)
estimates_sim_outl_ols_grad <- estimates_simulated(obtain_ols_gradient_descent, data_out
liers_simulator)
estimates_sim_outl_abs <- estimates_simulated(obtain_absolute_gradient_descent, data_out
liers_simulator)
estimates_sim_outl_huber <- estimates_simulated(function(X, Y) obtain_huber_gradient_des
cent(X, Y, sig=4), data_outliers_simulator())


dp <- data.frame(
slope=c(
estimates_sim_outl_ols_ana[,2],
estimates_sim_outl_abs[,2],
estimates_sim_outl_huber[,2],
estimates_sim_ols_ana[,2],
estimates_sim_abs[,2],
estimates_sim_huber[,2]
),
optimizer=c(
rep("1. OLS analytical (outliers)", 1000),
rep("2. absolute gradient (outliers)", 1000),
rep("3. huber gradient (outliers)", 1000),
rep("4. OLS analytical", 1000),
rep("5. absolute gradient", 1000),
rep("6. huber gradient", 1000)
))

ggplot(data=dp, mapping=aes(x=slope)) + geom_histogram() + geom_vline(xintercept=2) + xl
im(c(1, 3)) + facet_wrap(~optimizer, scales="free")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 3 rows containing non-finite values (`stat_bin()`).
```

```
## Warning: Removed 6 rows containing missing values (`geom_bar()`).
```


1. OLS analytical (outliers)
2. absolute gradient (outliers)
3. huber gradient (outliers)
4. OLS analytical
5. absolute gradient
6. huber gradient
count
slope
0
30
60
90
25
50
75
100
1.0
1.5
2.0
2.5
3.0